



Universidade de Brasília

Instituto de Ciências Exatas
Departamento de Ciência da Computação

**Algoritmos Paralelos Exatos e Otimizações
para Alinhamento de Sequências Biológicas Longas
em Plataformas de Alto Desempenho**

Edans Flávius de Oliveira Sandes

Brasília
2015



Universidade de Brasília

Instituto de Ciências Exatas
Departamento de Ciência da Computação

**Algoritmos Paralelos Exatos e Otimizações
para Alinhamento de Sequências Biológicas Longas
em Plataformas de Alto Desempenho**

Edans Flávio de Oliveira Sandes

Monografia apresentada como requisito parcial
para conclusão do Doutorado em Informática

Orientadora

Prof.^a Dr.^a Alba Cristina M. A. de Melo

Brasília

2015

Universidade de Brasília — UnB
Instituto de Ciências Exatas
Departamento de Ciência da Computação
Doutorado em Informática

Coordenadora: Prof.^a Dr.^a Alba Cristina M. A. de Melo

Banca examinadora composta por:

Prof.^a Dr.^a Alba Cristina M. A. de Melo (Orientadora) — CIC/UnB
Prof. Dr. Philippe O. A. Navaux — UFRGS
Prof. Dr. Edson N. Cáceres — UFMS
Prof.^a Dr.^a Maria Emília M. T. Walter — CIC/UnB
Prof. Dr. Ricardo P. Jacobi — CIC/UnB
Prof. Dr. George L. M. Teodoro (membro convidado) — CIC/UnB

CIP — Catalogação Internacional na Publicação

Sandes, Edans Flávio de Oliveira.

Algoritmos Paralelos Exatos e Otimizações
para Alinhamento de Sequências Biológicas Longas
em Plataformas de Alto Desempenho / Edans Flávio de Oli-
veira Sandes. Brasília : UnB, 2015.

227 p. : il. ; 29,5 cm.

Tese (Doutorado) — Universidade de Brasília, Brasília, 2015.

1. Bioinformática, 2. GPU, 3. Comparação de Sequências

CDU 004

Endereço: Universidade de Brasília
Campus Universitário Darcy Ribeiro — Asa Norte
CEP 70910-900
Brasília-DF — Brasil



Universidade de Brasília

Instituto de Ciências Exatas
Departamento de Ciência da Computação

Algoritmos Paralelos Exatos e Otimizações para Alinhamento de Sequências Biológicas Longas em Plataformas de Alto Desempenho

Edans Flávio de Oliveira Sandes

Monografia apresentada como requisito parcial
para conclusão do Doutorado em Informática

Prof.^a Dr.^a Alba Cristina M. A. de Melo (Orientadora)
CIC/UnB

Prof. Dr. Philippe O. A. Navaux Prof. Dr. Edson N. Cáceres
UFRGS UFMS

Prof.^a Dr.^a Maria Emília M. T. Walter Prof. Dr. Ricardo P. Jacobi
CIC/UnB CIC/UnB

Prof. Dr. George L. M. Teodoro (membro convidado)
CIC/UnB

Prof.^a Dr.^a Alba Cristina M. A. de Melo
Coordenadora do Programa de Pós-Graduação em Informática

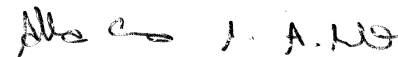
Brasília, 9 de Setembro de 2015

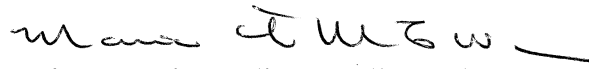
EDANS FLÁVIUS DE OLIVEIRA SANDES


Algoritmos Paralelos Exatos e Otimizações para Alinhamento de Sequências Biológicas Longas em Plataformas de Alto Desempenho


Tese aprovada como requisito parcial para obtenção do grau de Doutor no Curso de Pós-graduação em Informática da Universidade de Brasília, pela Comissão formada pelos professores:

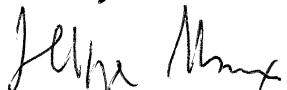
Orientador:

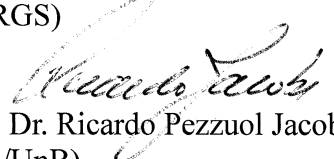

Prof. Dr. Alba Cristina M. A. de Melo
(CIC/UnB)


Prof. Dr. Maria Emília M. Telles Walter
(CIC/UnB)


Prof. Dr. George Luís Médeiros Teodoro
(CIC/UnB)


Prof. Dr. Edson Norberto Caceres
(UFMS)


Prof. Dr. Philippe Olivier Alexandre Navaux
(UFRGS)


Prof. Dr. Ricardo Pezzuol Jacobi
(CIC/UnB)

Vista e permitida a impressão.
Brasília, 09 de setembro de 2015.

Prof.^a Dr.^a Alba Cristina Magalhães Alves de Melo
Programa de Pós-Graduação em Informática
Departamento de Ciência da Computação
Universidade de Brasília

Lista de Figuras

| | | |
|------|--|----|
| 1.1 | Exemplo de alinhamento entre duas sequências. | 2 |
| 2.1 | Exemplo de alinhamento entre duas sequências. | 8 |
| 2.2 | Escore de cada coluna de um alinhamento. | 12 |
| 2.3 | Alinhamento global ótimo entre duas sequências. | 13 |
| 2.4 | Plot de coordenadas de um alinhamento. | 14 |
| 2.5 | Tipos de alinhamento | 16 |
| 2.6 | Conjuntos formados pelos alinhamentos de tipo 1/1, 2/2, 3/3, ++ e */*. | 16 |
| 2.7 | Alinhamento global ótimo | 17 |
| 2.8 | Alinhamento local ótimo | 19 |
| 2.9 | Alinhamento semi-global ótimo | 22 |
| 2.10 | Dois níveis de recursão do algoritmo de Myers e Miller | 23 |
| 2.11 | FastLSA | 23 |
| 2.12 | Representação do algoritmo de Fickett | 24 |
| 2.13 | Representação do algoritmo de Chao-Pearson-Miller | 25 |
| 3.1 | Arquitetura CUDA Fermi | 29 |
| 3.2 | Arquitetura CUDA Kepler | 30 |
| 3.3 | Arquitetura CUDA Maxwell | 31 |
| 3.4 | Arquitetura CUDA - Componentes de <i>software</i> | 33 |
| 3.5 | Arquitetura CUDA - Hierarquia de memória | 34 |
| 4.1 | Paralelismo <i>Fine-grained</i> | 37 |
| 4.2 | Soluções com instruções vetoriais em CPU | 39 |
| 5.1 | Paralelismo Externo - Divisão da matriz em blocos | 51 |
| 5.2 | Bloco retangular com 3 <i>threads</i> | 52 |
| 5.3 | Barramentos de comunicação | 53 |
| 5.4 | Bloco não retangular com 3 <i>threads</i> | 54 |
| 5.5 | Cadeia de delegação de células entre os blocos | 55 |
| 5.6 | Problema de dependência de dados entre blocos | 56 |
| 5.7 | Execução do bloco dividida em duas fases | 57 |
| 6.1 | Execução do CUDAlign 2.0 dividida em seis estágios. | 60 |
| 6.2 | Procedimentos de <i>Matching</i> | 61 |
| 6.3 | Execução ortogonal | 62 |
| 6.4 | Detalhe no procedimento de <i>matching</i> ortogonal. | 62 |
| 6.5 | Divisão Balanceada | 64 |

| | | |
|------|---|-----|
| 6.6 | Visualizador de alinhamentos do CUDAlign | 66 |
| 6.7 | Alinhamento entre cromossomos do homem e do chimpanzé (gráfico) . . . | 69 |
| 6.8 | Alinhamento entre cromossomos do homem e do chimpanzé (textual) . . . | 69 |
| 7.1 | Definições geométricas do <i>Block Pruning</i> | 71 |
| 7.2 | Áreas prunable (em cinza). | 73 |
| 7.3 | Tempo de execução da comparação de 1044K×1073K | 75 |
| 7.4 | Tempo de execução × tamanho da matriz (em células) | 77 |
| 7.5 | Gráficos de alguns alinhamentos, com a área <i>pruned</i> em cinza. | 78 |
| 8.1 | Grafo de <i>Traceback</i> | 82 |
| 8.2 | Casos de dependência de $H_{i,j}$ | 83 |
| 8.3 | Diferença $\bar{\delta}$ entre células adjacentes conectadas por arestas. | 84 |
| 8.4 | Limites superiores das células adjacentes | 85 |
| 8.5 | Possíveis valores nas duas primeiras linhas e colunas. | 86 |
| 8.6 | Limites das células adjacentes a $H_{1,j}$ | 86 |
| 8.7 | Limites inferiores das células adjacentes | 88 |
| 8.8 | Representações geométricas de $H_{i,j}^{max}$ e $H_{i,j}^{min}$ | 91 |
| 8.9 | Visualização das induções na condição de <i>pruning</i> | 92 |
| 8.10 | Algoritmos de <i>Pruning</i> | 93 |
| 8.11 | Diferentes formatos de blocos e suas coordenadas máximas mínimas. | 98 |
| 8.12 | Dependências entre blocos de um grid | 98 |
| 8.13 | Formas de processamento. | 101 |
| 8.14 | Maior escore provisório ($best_{i,j}$) em diversas formas de processamento. . . . | 102 |
| 8.15 | Representações geométricas da área de <i>Block Pruning</i> | 103 |
| 8.16 | Código para resolução das equações de área - Ferramenta Maxima | 106 |
| 8.17 | Eficácia de <i>Pruning</i> vs. ângulo de processamento | 108 |
| 8.18 | Eficácia de <i>Pruning</i> vs. penalidade de <i>gap</i> | 108 |
| 8.19 | Eficácia de <i>Pruning</i> vs. o valor de ψ (ou p) | 109 |
| 8.20 | Eficácia de <i>Pruning</i> vs. pontuação de <i>match</i> ou <i>mismatch</i> | 110 |
| 8.21 | Eficácia de <i>pruning</i> em diferentes formas de processamento | 112 |
| 8.22 | Variação do ângulo de processamento | 113 |
| 8.23 | Variação da taxa p de crescimento do escore ótimo | 114 |
| 8.24 | Variação da penalidade de <i>gap</i> | 115 |
| 8.25 | Variação da proporção do tamanho das sequências | 116 |
| 9.1 | Divisão de colunas entre múltiplas GPUs. | 119 |
| 9.2 | Encadeamento de <i>threads</i> em ambiente Multi-GPU. | 119 |
| 9.3 | <i>Buffers</i> de comunicação Multi-GPU. | 120 |
| 9.4 | Previsões de tempo e GCUPS no Minotauro (1×M2090) | 125 |
| 9.5 | <i>Speedup</i> previsto no Minotauro. | 126 |
| 9.6 | <i>Speedups</i> obtidos no Minotauro. | 127 |
| 9.7 | Desempenho obtido quando variamos os tamanhos das sequências | 127 |
| 9.8 | Tempos de execução em escala logarítmica. | 128 |
| 10.1 | Linhas de tempo para o <i>Pipelined Traceback</i> (PT) | 132 |
| 10.2 | Linhas de tempo para o <i>Incremental Speculative Traceback</i> (IST) | 135 |

| | | |
|-------|---|-----|
| 10.3 | Alinhamentos ótimos entre cromossomos do homem e do chimpanzé | 139 |
| 10.4 | Linhas de tempo dos estágios | 143 |
| 10.5 | Regiões não sequenciadas nos cromossomos homólogos | 144 |
| 11.1 | Arquitetura do sistema multiagentes proposto. | 147 |
| 11.2 | Negociação do Balanceamento. | 153 |
| 11.3 | Diagrama de transição de estados. | 153 |
| 11.4 | Gráfico de balanceamento para <i>wavefront</i> estático. | 155 |
| 11.5 | Balanceamento com três mudanças no poder computacional. | 157 |
| 12.1 | Arquitetura MASA | 161 |
| 12.2 | Processamento em partições. | 162 |
| 12.3 | Estratégias de <i>Block Pruning</i> implementadas no MASA | 163 |
| 12.4 | MASA-API: Diagrama de Classes. | 164 |
| 12.5 | Extensões MASA | 168 |
| 12.6 | Extensões MASA - Hierarquia de Classes | 169 |
| 12.7 | Prioridades e ordem de criação das tarefas no OmpSs. | 169 |
| 12.8 | Padrão de processamento do MASA-OmpSs/CPU. | 173 |
| 12.9 | Área de <i>pruning</i> comparando a sequência CP000051.1 com ela mesma . . . | 174 |
| 12.10 | Outras áreas de <i>pruning</i> (cinza) com <i>perfect match</i> , utilizando o MASA-CUDAlign. . . | 174 |
| 12.11 | Alinhamentos ótimos da comparação de 1M (CP000051.1 e AE002160.2) . | 175 |
| 12.12 | Alinhamento local ótimo de algumas comparações | 177 |
| 12.13 | Alinhamento textual entre os cromossomos 21 do homem e chimpanzé. . . | 178 |
| 12.14 | Frequências de N, A+T e C+G nos cromossomos 21 | 178 |
| 12.15 | Maiores regiões <i>unmatched</i> entre os cromossomos 21 do homem e chimpanzé. | 179 |
| I.1 | Captura de tela da página do repositório do MASA-Core (GitHub) | 198 |

Lista de Tabelas

| | | |
|------|---|-----|
| 2.1 | Definições | 9 |
| 2.2 | Códigos IUPAC | 10 |
| 2.3 | Exemplo de matriz de substituição para DNA. | 12 |
| 2.4 | Definições | 13 |
| 2.5 | Classificação dos tipos de coordenada. | 15 |
| 2.6 | Matriz de programação dinâmica do algoritmo Needleman-Wunsch | 18 |
| 2.7 | Matriz de programação dinâmica do algoritmo Smith-Waterman | 19 |
| 2.8 | Modificações no algoritmo NW para cada tipo de alinhamento. | 21 |
| 2.9 | Matriz de programação dinâmica para alinhamento semi-global | 21 |
| 3.1 | Comparação entre placas NVIDIA | 32 |
| 3.2 | Comparação entre alguns modelos da linha TESLA | 32 |
| 4.1 | Artigos de comparação paralela com Smith-Waterman. | 49 |
| 5.1 | Tamanho e tipo de memória utilizada no CUDAalign 1.0 | 53 |
| 6.1 | Detalhes das sequências reais | 67 |
| 6.2 | Resultados dos alinhamentos ótimos | 67 |
| 6.3 | Tempo de Execução do Estágio 1 - CUDAalign 2.0 | 68 |
| 6.4 | Tempo de execução de cada estágios do CUDAalign 2.0 | 68 |
| 7.1 | Tempo de Execução do Estágio 1 - CUDAalign 2.1 | 76 |
| 7.2 | Tempo de execução de cada estágios do CUDAalign 2.1 | 76 |
| 8.1 | Definições | 81 |
| 8.2 | Diferença máxima $\bar{\delta}$ entre células conectadas. | 85 |
| 8.3 | Diferença máxima $\delta_{i+\Delta_i, j+\Delta_j}$ entre células. | 88 |
| 8.4 | Eficácia de <i>Pruning</i> | 107 |
| 9.1 | Sequências utilizadas nos testes. | 122 |
| 9.2 | Tempo de Execução e GCUPS no <i>cluster</i> Minotauro. | 123 |
| 9.3 | Tempo de Execução e GCUPS no Panoramax e no Laico | 123 |
| 9.4 | Tempo de Execução e GCUPS do cromossomo 1 | 124 |
| 9.5 | Constantes para previsão de desempenho. | 124 |
| 9.6 | <i>Speedup</i> previsto no Minotauro. | 125 |
| 9.7 | Erros da previsão de <i>speedup</i> no Minotauro | 126 |
| 10.1 | Cromossomos usados nos testes do CUDAalign 4.0. | 138 |

| | | |
|------|--|-----|
| 10.2 | Tempos de execução do chr22 e do chr16 (Estratégia PT). | 140 |
| 10.3 | Tempo de Execução do PT e IST | 141 |
| 11.1 | Percepções, ambiente, objetivos e ações (PAGE) dos agentes propostos | 147 |
| 12.1 | Alguns métodos do IManager | 164 |
| 12.2 | Sequências utilizadas nos testes. | 170 |
| 12.3 | Desempenho das extensões do MASA | 171 |
| 12.4 | Desempenho do MASA-CUDAlign sem <i>Block Pruning</i> | 176 |
| 12.5 | Tamanhos e localização das maiores regiões <i>unmatched</i> | 180 |
| 13.1 | Desempenho das versões do CUDAlign propostas nesta tese | 182 |
| 13.2 | Comparação do CUDAlign 4.0 com outras abordagens | 184 |

Sumário

| | |
|---|-------------|
| Lista de Figuras | v |
| Lista de Tabelas | viii |
| 1 Introdução | 1 |
| 1.1 Alinhamento de Sequências | 1 |
| 1.2 Motivação | 2 |
| 1.3 Objetivos | 4 |
| 1.4 Contribuições | 5 |
| 1.5 Sumário da tese | 6 |
| I <i>Background</i>/Contextualização | 7 |
| 2 Comparação de Sequências Biológicas | 8 |
| 2.1 Definições | 9 |
| 2.1.1 Definições Constantes na Literatura | 9 |
| 2.1.2 Definições Propostas nesta tese | 13 |
| 2.2 Algoritmos Exatos | 16 |
| 2.2.1 Needleman-Wunsch (NW) | 17 |
| 2.2.2 Smith-Waterman (SW) | 18 |
| 2.2.3 Gotoh | 20 |
| 2.2.4 Alinhamento Semi-Global Ótimo | 20 |
| 2.2.5 Myers e Miller (MM) | 21 |
| 2.2.6 FastLSA | 22 |
| 2.2.7 Fickett | 23 |
| 2.2.8 Chao-Pearson-Miller | 25 |
| 3 Arquiteturas de Unidades de Processamento Gráfico (GPUs) | 26 |
| 3.1 Evolução das Placas Gráficas | 26 |
| 3.2 CUDA | 28 |
| 3.2.1 Componentes de <i>Hardware</i> | 28 |
| 3.2.2 Linhas de Produtos da NVIDIA | 30 |
| 3.2.3 Componentes de <i>Software</i> | 31 |
| 3.2.4 Programação em CUDA | 33 |

| | | |
|-----------|---|-----------|
| 4 | Comparação Paralela de Sequências Biológicas | 36 |
| 4.1 | Estratégias de Paralelização | 36 |
| 4.1.1 | Paralelismo <i>Coarse-grained</i> | 36 |
| 4.1.2 | Paralelismo <i>Fine-grained</i> | 37 |
| 4.1.3 | Métrica de desempenho (CUPS) | 38 |
| 4.2 | Soluções em CPU (SIMD) | 38 |
| 4.2.1 | SWMMX (Rognes e Seeberg) | 38 |
| 4.2.2 | StripSW (Farrar) | 39 |
| 4.2.3 | SWIPE (Rognes) | 40 |
| 4.2.4 | Parallel-LTDP (Maleki) | 40 |
| 4.3 | Soluções em GPU | 41 |
| 4.3.1 | DASW (Y. Liu et al.) | 41 |
| 4.3.2 | GPU-SW (W. Liu et al.) | 41 |
| 4.3.3 | SWCuda (Manavsky e Valle) | 42 |
| 4.3.4 | LR-SW (L. Ligowski et al.) | 42 |
| 4.3.5 | CUDA-SSCA (A. Khajeh-Saeed et al.) | 43 |
| 4.3.6 | CudaSW++ (Y. Liu et al.) | 43 |
| 4.3.7 | CUDAlign 1.0 (Sandes et al.) | 45 |
| 4.3.8 | FGCS (Ino et al.) | 46 |
| 4.3.9 | SW# (Korpar e Sikic) | 46 |
| 4.3.10 | GSWABE (Liu e Schmidt) | 46 |
| 4.4 | Soluções em outras plataformas | 46 |
| 4.4.1 | CBESW - Cell BE (A. Wirawan) | 46 |
| 4.4.2 | PP-Cell - Cell BE (A. Sarje) | 47 |
| 4.4.3 | NoC-SW - Asic (Sarkar) | 47 |
| 4.4.4 | SW-Rivyera - FPGA (L. Wienbrandt) | 47 |
| 4.4.5 | XSW (Wang et al.) | 48 |
| 4.4.6 | SWAPHI-LS (Liu et al.) | 48 |
| 4.5 | Comparação dos artigos | 48 |
| 5 | Visão Geral do CUDAlign 1.0 | 50 |
| 5.1 | Técnicas de Paralelismo | 50 |
| 5.1.1 | Paralelismo externo | 50 |
| 5.1.2 | Paralelismo interno | 51 |
| 5.2 | Estruturas em memória | 52 |
| 5.3 | Otimizações | 54 |
| 5.3.1 | Delegação de Células | 54 |
| 5.3.2 | Divisão de Fases | 55 |
| 5.4 | Previsão de desempenho | 57 |
| II | Contribuições | 58 |
| 6 | CUDAlign 2.0: Obtenção de alinhamentos longos em uma GPU | 59 |
| 6.1 | Projeto do CUDAlign 2.0 | 59 |
| 6.1.1 | Estágio 1 - Obtenção do Escore Ótimo | 59 |
| 6.1.2 | Estágio 2 - <i>Traceback</i> Parcial | 61 |

| | | |
|----------|---|------------|
| 6.1.3 | Estágio 3 - Divisão de Partições | 63 |
| 6.1.4 | Estágio 4 - Myers-Miller otimizado | 63 |
| 6.1.5 | Estágio 5 - Obtenção do alinhamento completo | 64 |
| 6.1.6 | Estágio 6 - Visualização | 65 |
| 6.2 | Resultados Experimentais | 65 |
| 6.3 | Conclusão do Capítulo | 66 |
| 7 | CUDAlign 2.1: Obtenção de alinhamentos em uma GPU com descarte de blocos | 70 |
| 7.1 | Otimização Block Pruning | 70 |
| 7.1.1 | Definições | 70 |
| 7.1.2 | Procedimento de <i>Pruning</i> | 71 |
| 7.1.3 | Teoremas | 72 |
| 7.2 | Resultados Experimentais | 74 |
| 7.3 | Conclusão do Capítulo | 75 |
| 8 | Formalização e Generalização do Descarte de Blocos (<i>Pruning</i>) | 79 |
| 8.1 | Trabalhos relacionados | 80 |
| 8.2 | Definições | 81 |
| 8.3 | Diferença de valores entre células da matriz | 84 |
| 8.3.1 | Limites das diferenças entre células conectadas ($\bar{\delta}$) | 84 |
| 8.3.2 | Limites das diferenças entre células quaisquer (δ) | 85 |
| 8.3.3 | Outras equações de recorrência | 89 |
| 8.3.4 | Escore derivado de uma célula ($H_{i,j}^{max}$ e $H_{i,j}^{min}$) | 89 |
| 8.4 | Método de <i>Pruning</i> | 90 |
| 8.4.1 | Definições | 91 |
| 8.4.2 | Algoritmos de <i>Pruning</i> | 93 |
| 8.4.3 | <i>Block Pruning</i> | 97 |
| 8.5 | Método de Avaliação Teórica | 99 |
| 8.5.1 | Eficácia do método de <i>Pruning</i> | 102 |
| 8.5.2 | Análise das fórmulas de eficácia de <i>pruning</i> | 107 |
| 8.6 | Simulação do <i>Pruning</i> | 111 |
| 8.6.1 | Formas de processamento da matriz | 111 |
| 8.6.2 | Ângulos de processamento da Matriz | 113 |
| 8.6.3 | Similaridade entre as sequências | 114 |
| 8.6.4 | Penalidade de Gaps | 115 |
| 8.6.5 | Sequências de tamanhos diferentes | 116 |
| 8.7 | Conclusão do Capítulo | 116 |
| 9 | CUDAlign 3.0: Comparação de sequências em Múltiplas GPUs | 118 |
| 9.1 | Arquitetura Multi-GPU | 118 |
| 9.2 | Buffers de Comunicação | 119 |
| 9.3 | Método de Previsão de desempenho | 120 |
| 9.4 | Resultados Experimentais | 121 |
| 9.4.1 | Tempos de Execução e GCUPS | 122 |
| 9.4.2 | Comparação do Cromossomo 1 | 123 |
| 9.4.3 | <i>Overhead</i> de comunicação | 124 |

| | | |
|-----------|---|------------|
| 9.4.4 | Previsão de Desempenho | 124 |
| 9.4.5 | <i>Speedups</i> obtidos | 126 |
| 9.5 | Conclusão do Capítulo | 128 |
| 10 | CUDAlign 4.0: Obtenção de alinhamentos em múltiplas GPUs | 130 |
| 10.1 | Estratégia Multi-GPU para o Estágio 1 | 130 |
| 10.2 | <i>Traceback</i> em Múltiplas GPUs | 131 |
| 10.2.1 | <i>Pipelined Traceback</i> (PT) - Estratégia de <i>baseline</i> | 131 |
| 10.2.2 | <i>Incremental Speculative Traceback</i> (IST) | 133 |
| 10.3 | Resultados Experimentais | 137 |
| 10.3.1 | Sequências utilizadas nos testes | 138 |
| 10.3.2 | Desempenho do <i>Pipelined Traceback</i> | 139 |
| 10.3.3 | Impacto do <i>Incremental Speculative Traceback</i> | 140 |
| 10.3.4 | Efeitos das características do alinhamento no IST | 141 |
| 10.3.5 | Análise do uso do <i>Buffer</i> (Estágio 1) | 144 |
| 10.4 | Conclusão do Capítulo | 144 |
| 11 | Balanceamento de <i>Wavefront</i> em Múltiplas GPUs | 146 |
| 11.1 | Projeto do Sistema Multiagentes | 146 |
| 11.2 | Métricas de Execução | 148 |
| 11.3 | Métricas de Balanceamento Global | 148 |
| 11.4 | Métricas de Balanceamento Local | 150 |
| 11.4.1 | Pesos de Balanceamento | 151 |
| 11.4.2 | Negociação de Balanceamento | 152 |
| 11.5 | Resultados Experimentais | 154 |
| 11.5.1 | Gráfico de Balanceamento | 154 |
| 11.5.2 | <i>Wavefront</i> multinodo estático | 155 |
| 11.5.3 | Balanceamento dinâmico no <i>Wavefront</i> multinodo | 156 |
| 11.6 | Conclusão do Capítulo | 158 |
| 12 | Multi-platform Architecture for Sequence Aligners (MASA) | 159 |
| 12.1 | Análise do Código do CUDAlign | 159 |
| 12.2 | Arquitetura MASA | 160 |
| 12.3 | MASA-API | 163 |
| 12.4 | Criação de uma extensão MASA | 165 |
| 12.5 | Extensões MASA | 166 |
| 12.5.1 | MASA-OpenMP/CPU | 167 |
| 12.5.2 | MASA-OpenMP/Phi | 167 |
| 12.5.3 | MASA-OmpSs/CPU | 168 |
| 12.5.4 | MASA-CUDAlign | 170 |
| 12.6 | Resultados Experimentais | 170 |
| 12.6.1 | Sequências utilizadas nos testes | 170 |
| 12.6.2 | Tempo de execução para alinhamentos locais | 171 |
| 12.6.3 | Resultados de pruning para <i>perfect match</i> | 173 |
| 12.6.4 | Tempo de execução para alinhamentos globais e semiglobais | 174 |
| 12.6.5 | Resultados dos alinhamentos | 177 |
| 12.7 | Conclusão do Capítulo | 179 |

| | |
|---|------------|
| III Conclusão | 181 |
| 13 Conclusões e Trabalhos Futuros | 182 |
| 13.1 Resultados Obtidos | 182 |
| 13.2 Trabalhos Futuros | 185 |
| Referências | 187 |
| Referências | 187 |
| I Repositórios com o código fonte do MASA | 198 |
| II <i>Script</i> de simulação do <i>Block Pruning</i> (Gnuplot) | 199 |
| III Artigos decorrentes desta tese | 201 |
| III.1 Artigos completos publicados em periódicos internacionais | 201 |
| III.2 Artigos completos publicados em conferências internacionais | 201 |
| III.3 Artigos resumidos publicados em conferências internacionais | 202 |
| III.4 Artigos completos submetidos a periódicos internacionais (em revisão) . . . | 202 |
| III.5 Primeira página dos artigos | 202 |

Agradecimentos

Primeiramente agradeço a Deus por me permitir realizar mais esse sonho e por me dar forças para nunca desistir. Sei que o homem não pode receber coisa alguma se antes não lhe for dada do céu.

À professora Alba, agradeço pela orientação e por todos os ensinamentos e conselhos. O seu apoio propiciou-me inúmeras oportunidades na vida, as quais fiz questão de aproveitá-las ao máximo. Obrigado pelo tempo investido, sempre direcionando meus estudos e colocando ordem em minhas ideias, pois sei que sem sua grande competência e amizade eu não teria chegado até aqui. Aos professores deste departamento, os quais me ensinaram por vários anos, agradeço por toda a minha formação. As lições que aprendi não ficaram restritas à universidade, pois pude aplicá-las em várias outras situações da minha vida.

Agradeço também à minha família, que nunca me abandonou e sempre me apoiou nos sonhos que eu desejei realizar. À minha noiva Luciana, agradeço pelo carinho, amor, fé e paciência. Sua companhia foi fundamental para que eu alcançasse mais um objetivo em minha vida. Aos amigos que ganhei durante os anos, obrigado por me conceder momentos alegres e descontraídos. Isso com certeza tornou minha caminhada até aqui muito mais fácil. E a todos que contribuíram direta ou indiretamente para este trabalho, registro aqui a minha gratidão.

Abstract

Biological sequence alignment is one of the most important operations in Bioinformatics, executing thousands of times every day around the world. The exact algorithms for this purpose have quadratic time complexity. So when the comparison involves very long sequences, such as in the human genome, matrices with petabytes must be calculated, and this is still considered unfeasible by most researchers. The main objective of this Thesis is to propose and evaluate algorithms and optimizations that produce the optimal alignment of very long DNA sequences in a short time using high-performance computing platforms. The proposed algorithms use parallel divide-and-conquer techniques with reduced memory complexity, whilst with quadratic time complexity. CUDAlign, in its versions 2.0, 2.1, 3.0 and 4.0, is the main contribution of this Thesis. The proposed algorithms are integrated into the same tool, allowing efficient retrieval of the optimal alignment between two long DNA sequences using multiple GPUs (Graphics Processing Unit) from NVIDIA. The proposed optimizations maintain the maximum parallelism during most of the processing time. To accelerate the matrix calculation in a single GPU, the Orthogonal Execution, Balanced Partition and Block Pruning optimizations were proposed, increasing the performance of the matrix computation and discarding areas that do not contribute to the optimal alignment. The formal analysis of Block Pruning shows that its effectiveness depends on factors such as the sequences similarity and the matrix processing order. During the alignment computation with multiple GPUs, the Incremental Speculative Traceback optimization is proposed to accelerate the alignment retrieval, using speculated values with high accuracy rate. A dynamic load balancing method has also been proposed and its effectiveness has been shown in simulated environments. Finally, the software architecture called Multi-Platform Architecture for Sequence aligners (MASA) was proposed to simplify the portability of CUDAlign to different hardware and software platforms. With this architecture, it was possible to port CUDAlign to hardware platforms such as CPU and Intel Phi, and using software platforms such as OpenMP and OmpSs. In this Thesis, real sequences are used to validate the effectiveness of the proposed algorithms and optimizations in several supported architectures. Our proposed tools were able to advance the state-of-the-art of sequence alignment algorithms, allowing a fast retrieval of all human and chimpanzee homologous chromosomes, using exact algorithms at an unprecedented rate of up to 10.35 TCUPS (Trillions of Cells Updated Per Second). As far as we know, this was the first time that this type of comparison was carried out with exact sequence comparison algorithms.

Keywords: Bioinformatics, GPU, Sequence Comparison

Resumo

O alinhamento de sequências biológicas é uma das operações mais importantes em Bioinformática, sendo executado milhares de vezes a cada dia ao redor do mundo. Os algoritmos exatos existentes para este fim possuem complexidade quadrática de tempo. Logo, quando a comparação é realizada com sequências muito longas, tais como no escopo do genoma humano, matrizes na ordem de petabytes devem ser calculadas, algo considerado inviável pela maioria dos pesquisadores. O principal objetivo desta tese de Doutorado é propor e avaliar algoritmos e otimizações que permitam que o alinhamento ótimo de sequências muito longas de DNA seja obtido em tempo reduzido em plataformas de alto desempenho. Os algoritmos propostos utilizam técnicas paralelas de dividir e conquistar com complexidade de memória reduzida mantendo a complexidade quadrática do tempo de execução. O CUDAlign, em suas versões 2.0, 2.1, 3.0 e 4.0, é a principal contribuição desta tese, onde os algoritmos propostos estão integrados na mesma ferramenta, permitindo a recuperação eficiente do alinhamento ótimo entre duas sequências longas de DNA em múltiplas GPUs (*Graphics Processing Unit*) da NVIDIA. As otimizações propostas neste trabalho permitem que o nível máximo de paralelismo seja mantido durante quase todo o processamento. No cálculo do alinhamento em uma GPU, as otimizações *Orthogonal Execution*, *Balanced Partition* e *Block Pruning* foram propostas, aumentando o desempenho no cálculo da matriz e descartando áreas que não contribuem para o alinhamento ótimo. A análise formal do *Block Pruning* mostra que sua eficácia depende de vários fatores, tais como a similaridade entre as sequências e a forma de processamento da matriz. No cálculo do alinhamento com várias GPUs, a otimização *Incremental Speculative Traceback* é proposta para acelerar a obtenção do alinhamento utilizando valores especulados com alta taxa de acerto. Também são propostos métodos de balanceamento dinâmico de carga que se mostraram eficientes em ambientes simulados. A arquitetura de software chamada de *Multi-Platform Architecture for Sequence Aligners* (MASA) foi proposta para facilitar a portabilidade do CUDAlign para diferentes plataformas de *hardware* ou *software*. Com esta arquitetura, foi possível portar o CUDAlign para plataformas de *hardware* como CPUs e Intel Phi e utilizando plataformas de *software* como OpenMP e OmpSs. Nesta tese, sequências reais são utilizadas para validar a eficácia dos algoritmos e otimizações nas várias arquiteturas suportadas. Por meio do desempenho das ferramentas implementadas, avançou-se o estado da arte para permitir o alinhamento, em tempo viável, de todos os cromossomos homólogos do homem e do chimpanzé, utilizando algoritmos exatos de comparação de sequências com um desempenho de até 10,35 TCUPS (Trilhões de Células Atualizadas por Segundo). Até onde sabemos, esta foi a primeira vez que tal tipo de comparação foi realizada com métodos exatos.

Palavras-chave: Bioinformática, GPU, Comparação de Sequências

Capítulo 1

Introdução

Ao longo das últimas décadas, observamos inúmeros avanços nas áreas de Biologia Molecular e Genética. Novas tecnologias permitiram que os genomas de vários organismos fossem sequenciados e armazenados em diversas bases genômicas públicas, que crescem de tamanho com uma velocidade exponencial [1]. A análise do conteúdo dessas sequências biológicas é de extrema importância, pois permite: (a) identificar genes que causam doenças, (b) determinar eventos evolucionários em uma análise comparativa das espécies, (c) identificar regiões conservadas entre organismos relacionados filogeneticamente, (d) reconstruir redes metabólicas e comparar metabolismos entre diferentes espécies, entre outros. Os avanços que observamos atualmente não poderiam ter ocorrido sem a existência de ferramentas e algoritmos desenvolvidos na área de Bioinformática. Essa área visa criar ferramentas para organizar e avaliar os dados genômicos, possibilitando que estes dados sejam transformados em informações relevantes do ponto de vista biológico [2][3].

A comparação de sequências biológicas é uma das operações mais básicas e importantes na Bioinformática, sendo amplamente utilizada para determinar o grau de similaridade entre as sequências [4]. Logo, ao compararmos a sequência de DNA de um organismo com a sequência de outro cujas funcionalidades já são conhecidas, é possível estimar o nível de similaridade e inferir características comuns entre eles [5]. O resultado de uma operação de comparação de sequências biológicas pode ser (a) um escore que indica a similaridade entre as mesmas ou (b) o escore e o alinhamento, onde uma sequência (ou parte dela) é colocada sobre a outra (ou parte dela), de maneira a evidenciar as regiões de similaridades/diferenças [4].

Este capítulo apresenta a introdução desta tese, que propõe algoritmos e otimizações para alinhamento de sequências longas de DNA em arquiteturas de alto desempenho. Na Seção 1.1, o problema de alinhamento de sequências é apresentado. Na Seção 1.2, a motivação deste trabalho é descrita. Os objetivos principais e específicos estão listados na Seção 1.3 e as contribuições são elencadas na Seção 1.4. Finalmente, a Seção 1.5 apresenta a estrutura dos demais capítulos desta tese.

1.1 Alinhamento de Sequências

Em Bioinformática, as sequências biológicas, sejam elas compostas por nucleotídeos ou aminoácidos, são representadas como uma cadeia de caracteres (*strings*) [6, 7]. O alinhamento de duas sequências biológicas é então definido como um pareamento entre os

caracteres das sequências, com a possível inserção de espaços (*gaps*) entre os caracteres. Chamamos de *matches* os pareamentos entre caracteres iguais e de *mismatches* os pareamentos entre caracteres diferentes. A Figura 1.1 apresenta um exemplo de alinhamento entre duas sequências, onde os *matches* estão representados pelo símbolo ‘:’, *mismatches* pelo símbolo ‘.’ e *gaps* pelo símbolo ‘-’.

```
GCTCACGCCGGTAGTCCCAGCACAGAGGGAG---GAGGCCAACGTATCACCTGAGGTC-----
      :::::::::::::::::::::::::::::  :::::::::::::::::::::  :::::
-----GCCTGTAATCCC-GCACTTTGGGAGGCCGAGGTGGGCGCATCAC--GAGGTCAGCGCGAAG
```

Figura 1.1: Exemplo de alinhamento entre duas sequências.

O problema do alinhamento ótimo de sequências visa encontrar, entre todas as possibilidades de pareamento entre duas sequências, um alinhamento cujo score seja o maior possível. O score de um alinhamento é definido de acordo com o tipo de alinhamento e os parâmetros de pontuação para os *matches*, *mismatches* e *gaps*. Dentre os tipos de alinhamento, o alinhamento *global* considera todos os caracteres de ambas as sequências, o alinhamento *semi-global* permite ignorar o início ou o final de no mínimo uma das sequências e o alinhamento *local* considera *substrings* das sequências. Para a pontuação de *gap*, os modelos mais comuns são o *linear gap* (penalidade proporcional ao comprimento do *gap*) e o *affine gap* (a abertura do *gap* possui uma penalidade adicional).

Desde 1970, vários algoritmos exatos foram propostos para resolver o problema de alinhamento ótimo de duas sequências biológicas. Entre eles, podemos citar o Needleman-Wunsch [8] para alinhamento global ótimo, Smith-Waterman [9] para alinhamento local ótimo, Gotoh [10] para alinhamento global ótimo com o modelo de *affine gap* e Myers-Miller [11], que utiliza as ideias de Hirschberg [12] para reduzir o uso de memória na obtenção do alinhamento global ótimo com *affine gap*. O Capítulo 2 descreverá cada um destes algoritmos em detalhes.

1.2 Motivação

Os algoritmos exatos que computam o alinhamento ótimo de sequências biológicas calculam, em geral, uma ou mais matrizes de programação dinâmica de tamanho $m \times n$, onde m e n são os tamanhos das sequências comparadas. Sendo assim, esses algoritmos demandam alto poder de processamento e uma grande quantidade de memória. Por este motivo, o uso de algoritmos exatos foi considerado inviável por muito tempo e, com isso, algoritmos heurísticos surgiram para acelerar o procedimento de comparação de sequências, embora sem garantir a produção do resultado ótimo. Dentre esses algoritmos heurísticos podemos citar o FASTA [13] e o BLAST [14]. Apesar de haver intensas pesquisas em algoritmos heurísticos para comparação de sequências, a presente tese se concentra em algoritmos exatos, deixando os algoritmos heurísticos fora do escopo deste trabalho.

O uso de memória dos algoritmos exatos também é um fator limitante, pois a complexidade quadrática de espaço ($O(mn)$) restringe bastante o tamanho das sequências

comparadas. Este fato fica evidente quando comparamos seqüências muito longas. Por exemplo, para alinhar seqüências de um milhão de pares de bases (MBP), precisaríamos de mais de um terabyte de memória. Myers e Miller[11] utilizaram as ideias de Hirschberg [12] para reduzir o uso de memória por meio de técnicas de dividir para conquistar, permitindo a redução do uso de memória para complexidade linear. Entretanto, ao se utilizar esse tipo de técnica, o tempo de processamento duplica, no pior caso.

Para aumentar o desempenho dos algoritmos exatos e, conseqüentemente, reduzir o tempo necessário para encontrar alinhamentos ótimos, utilizam-se de técnicas de paralelismo. Dentre as arquiteturas que permitem executar variantes paralelas desses algoritmos, podemos citar FPGAs [15] (*Field Programmable Gate Arrays*), *Clusters* de CPUs, multicóres com ou sem o uso de instruções vetoriais, Cell BEs [16] (*Cell Broadband Engine*), GPUs (*Graphics Processing Units*) [17] e, mais recentemente, Intel Phi [18]. O Capítulo 4 desta tese abordará várias implementações paralelas que utilizam essas arquiteturas. Em especial, as GPUs [19] possuem uma boa razão custo-benefício para resolver problemas paralelizáveis, ganhando popularidade nos últimos anos. Este tipo de processamento genérico em uma unidade de processamento gráfico é chamado de GPGPU (*General-purpose computing on graphics processing units*) [20]. Para comparar o desempenho das implementações paralelas de algoritmos de comparação de seqüências, convencionou-se o uso da métrica CUPS (*Cells Updated Per Second*), que é calculada dividindo-se o tamanho da matriz de programação dinâmica ($m \times n$) pelo tempo de execução em segundos.

A pesquisa em soluções paralelas para alinhamento ótimo de seqüências biológicas iniciou-se na década de 1980. Em 1985, surgiu uma das primeiras implementações paralelas do algoritmo exato Smith-Waterman, que usava a abordagem de array sistólico em FPGA [21]. A seguir, nas décadas de 1990 e 2000, *clusters* de CPUs foram intensamente utilizados para implementar variantes do algoritmo Smith-Waterman [22–25]. A maior parte desses algoritmos comparava duas seqüências de DNA sendo que o tamanho máximo das seqüências comparadas chegou a 3 MBP (Milhões de Pares de Bases). Nesse caso, a comparação 3 MBP \times 3 MBP demorou 13 horas e 32 minutos (0,18 GCUPS) em um *cluster* de 16 CPUs. Paralelamente, vários pesquisadores desenvolveram propostas para comparar uma seqüência de busca com um banco de dados genômico, composto de um grande número de seqüências de proteínas. Para resolver esse tipo de problema, uma das primeiras abordagens a usar instruções vetoriais de CPU foi a de Wozniak [26], que atingiu 0,19 GCUPS (Bilhões de CUPS) no ano de 1997. Em 2011, outra abordagem atingiu 106 GCUPS [27] utilizando 12 núcleos de CPU.

O advento das GPGPUs permitiu que diversas variantes do algoritmo de Smith-Waterman fossem propostas para tais plataformas tanto para a comparação de seqüências de proteínas contra bancos genômicos como para a comparação de seqüências longas de DNA. Em 2008, atingiu-se um desempenho de 3,6 GCUPS com uma das primeiras implementações paralelas em GPU [28] e em 2010 atingiu-se o desempenho de 29,7 GCUPS [29].

Desde o artigo seminal de Lipton e Lopresti [21], a pesquisa em *hardware* reconfigurável avançou bastante, atingindo 243 GCUPS em 2010 com um projeto ASIC (*Application Specific Integrated Circuit*) para execução paralela de variantes do Smith-Waterman [30].

Portanto, ao iniciar a presente tese em 2011, o melhor desempenho para comparação de seqüências biológicas com algoritmos exatos era de 243 GCUPS em ASIC e o desempenho máximo em GPUs era 29,7 GCUPS. Com 29,7 GCUPS, é impraticável comparar

sequências de DNA muito longas, tais como $230 \text{ MBP} \times 230 \text{ MBP}$, pois a comparação demoraria cerca de 20 dias. Com o desempenho de 243 GCUPS em projeto ASIC, a mesma comparação demoraria menos tempo, mas ainda levaria cerca de 2 dias e meio.

Embora o desempenho das implementações paralelas atuais consiga atingir bilhões de células processadas por segundo (GCUPS), a maioria das implementações impõe um limite para o tamanho da sequência de busca, o que significa que elas não são capazes de alinhar sequências muito longas. Por exemplo, a grande maioria destas implementações não aceita sequências maiores que 60 mil caracteres, sendo que alguns cromossomos humanos possuem mais de 200 milhões de caracteres. Com o surgimento de tecnologias mais recentes de sequenciamento, tais como no PacBio SingleMolecule Real-Time (SMRT) [31], existe uma expectativa que os fragmentos genômicos obtidos fiquem ainda maiores e sejam gerados com melhor qualidade. Desta forma, o processo de geração de sequências cromossômicas tende a ficar cada vez mais simples e o alinhamento de sequências com vários milhões de pares de bases tende a ganhar maior importância, aumentando a demanda por ferramentas que alinhem sequências em escala cromossômica com algoritmos exatos.

A principal motivação desta tese é, então, de evoluir o estado da arte de forma que o alinhamento ótimo de duas sequências longas de DNA possa ser executado em tempo viável, permitindo que cromossomos completos sejam comparados em poucas horas ou até mesmo em minutos. Visto que as ferramentas existentes não são capazes de produzir em pouco tempo os alinhamentos ótimos de sequências com mais de 200 milhões de bases, os biólogos ficam limitados ao uso de métodos heurísticos tanto para comparação como para a geração do alinhamento de sequências longas. As ferramentas propostas por meio dessa tese poderão ser utilizadas por pesquisadores para complementar as análises já efetuadas na literatura, mas considerando métodos exatos em vez de heurísticos. Embora o escopo desta tese não inclua a análise biológica dos alinhamentos obtidos, espera-se que as ferramentas e os resultados divulgados sejam analisados por especialistas de todo mundo em busca de novas descobertas, tais como a identificação de doenças e o estudo comparativo das espécies.

1.3 Objetivos

A presente tese de Doutorado possui o objetivo geral de desenvolver algoritmos e otimizações que permitam que o alinhamento ótimo de sequências muito longas de DNA seja obtido em tempo reduzido em plataformas de alto desempenho. Consideramos que o processamento é realizado em tempo reduzido se duas sequências na ordem de 200 milhões de pares de base forem alinhadas em menos de duas horas, algo ainda não visto na literatura com métodos exatos.

A principal plataforma escolhida para o desenvolvimento desta tese foi a arquitetura CUDA (*Compute Unified Device Architecture*) [32], das placas de processamento gráfico da NVIDIA. Entretanto, ao longo da pesquisa, o requisito de heterogeneidade foi levado em consideração, de forma que outras plataformas pudessem ser utilizadas para acelerar ainda mais o processamento dos algoritmos envolvidos.

Os objetivos específicos da tese estão elencados a seguir:

- Desenvolver e avaliar de algoritmos paralelos que permitam a recuperação eficiente de alinhamentos ótimos de sequências de DNA longas em GPUs;
- Propor e avaliar teoricamente de método para redução do espaço de busca dos algoritmos propostos, sem prejuízo do resultado ótimo;
- Propor, implementar e avaliar estratégia para execução com múltiplas GPUs dos algoritmos propostos;
- Propor uma arquitetura de *software* que permita a execução dos algoritmos propostos em plataformas homogêneas ou heterogêneas compostas por *multicores* ou GPUs, recuperando alinhamentos ótimos locais, globais e semiglobais;

1.4 Contribuições

Como contribuição desta tese, o *software* CUDAAlign foi evoluído em diversas versões incrementais. A primeira versão, CUDAAlign 1.0, foi desenvolvida na dissertação de mestrado [33] do mesmo autor desta tese. Nesta versão, apenas o escore era obtido utilizando apenas uma GPU. A seguir listamos as evoluções de cada uma das versões propostas no escopo desta tese:

- CUDAAlign 2.0 [34]: versão capaz de recuperar alinhamentos locais ótimos entre cromossomos completos em uma GPU. Isso foi possível devido às otimizações propostas nessa tese (*matching* baseado em objetivo, execução ortogonal e divisão balanceada). Atingiu-se 23,1 GCUPS em uma GPU;
- CUDAAlign 2.1 [35]: versão que propõe a otimização *Block Pruning* para alinhamentos locais ótimos, permitindo acelerar o cálculo da matriz de programação dinâmica em mais de 50%, para sequências similares. Nesta otimização, descartam-se áreas da matriz de programação dinâmica que não contribuem para a obtenção do alinhamento ótimo, sem prejuízo do resultado ótimo. Atingiu-se 50,7 GCUPS em uma GPU;
- CUDAAlign 3.0 [36][37]: versão capaz de comparar sequências longas de DNA em *clusters* com múltiplas GPUs (homogêneas e heterogêneas), permitindo comparar sequências de até 249 milhões de pares de base (MBP). Atingiu-se 1,73 TCUPS (Trilhões de CUPS) com 64 GPUS;
- CUDAAlign 4.0 [38]: versão que obtém o alinhamento completo em múltiplas GPUs de maneira eficiente. Por meio dos mecanismo de *traceback* proposto para múltiplos nós, foi possível obter o alinhamento completo de todos os cromossomos homólogos entre o homem e o chimpanzé, totalizando mais que 500 trilhões de células processadas. Atingiu-se 10,37 TCUPS com 384 GPUS.

Adicionalmente, as seguintes contribuições foram apresentadas nesta tese.

- Análise teórica da otimização *Block Pruning*: a análise matemática do *Block Pruning* foi feita para identificar quais os aspectos que contribuem ou prejudicam a eficácia deste método;
- Método de balanceamento dinâmico de carga [39]: para que ambientes distribuídos e não dedicados sejam utilizados para execução do CUDAAlign, é necessário que haja um balanceamento dinâmico de carga eficiente. Nesta tese, propusemos um método

de balanceamento de carga baseado em agentes, sem necessidade de um elemento central coordenador. Este método foi testado em ambiente simulado, mostrando ser bastante efetivo para o uso proposto;

- Arquitetura MASA [40]: A arquitetura MASA (*Multi-platform Architecture for Sequence Aligners*) foi projetada para simplificar a portabilidade do CUDAlign para outras arquiteturas (tais como CPU, Intel Phi, GPUs de diversos fabricantes, entre outros). O MASA suporta alinhamentos ótimos locais, globais e semi-globais. Nesta tese, a arquitetura MASA foi aplicada em 4 plataformas distintas: GPU (CUDA), CPU (OpenMP e OmpSs) e Intel Phi (OpenMP). Como contribuição indireta desta tese, a arquitetura MASA foi aplicada na arquitetura OpenCL no escopo da dissertação de mestrado do aluno Marco Antônio C. de Figueiredo Jr. [41], do mesmo grupo de trabalho do autor desta tese. Neste trabalho, atingiu-se 179,2 GCUPS em uma única GPU da AMD [42].

1.5 Sumário da tese

A presente tese está dividida em três partes. A primeira parte (*Background/Contextualização*) apresenta os conceitos, algoritmos e arquiteturas abordados ao longo da tese. Esta parte consiste em quatro capítulos. O Capítulo 2 descreve os algoritmos básicos de comparação de sequências, assim como uma série de definições que serão utilizadas ao longo desta tese. O Capítulo 3 descreve as arquiteturas de GPU, em especial das arquiteturas produzidas pela NVIDIA, que foram extensamente utilizadas ao longo deste trabalho. O Capítulo 4 apresenta vários algoritmos paralelos para comparação de sequências, formando o estado da arte desta linha de pesquisa. O trabalho inicialmente desenvolvido pelo autor desta tese no escopo da sua dissertação de mestrado (CUDAlign 1.0) está descrito detalhadamente no Capítulo 5 e será utilizado como base das contribuições propostas neste documento.

A segunda parte da tese (Contribuições) descreve as contribuições resultantes das propostas feitas neste trabalho. Esta parte foi desenvolvida em sete capítulos. As versões 2.0, 2.1, 3.0 e 4.0 do CUDAlign são relatadas nos Capítulos 6, 7, 9 e 10, respectivamente. O Capítulo 8 apresenta uma análise teórica e simulada da otimização *Block Pruning*. A proposta do mecanismo de balanceamento dinâmico de carga é apresentada no Capítulo 11. A arquitetura MASA está descrita no Capítulo 12.

Por fim, a conclusão e os trabalhos futuros estão descritos na última parte da tese, que contém o Capítulo 13. Ao final, serão apresentadas as referências bibliográficas e três anexos. O Anexo I apresenta os repositórios com o código fonte da arquitetura MASA e de suas extensões (incluindo o MASA-CUDAlign). O Anexo II apresenta o *script* do Gnuplot utilizado para simular as áreas descartadas com a otimização *Block Pruning* discutida no Capítulo 8. Por fim, o Anexo III apresenta a produção científica derivada desta tese, compreendendo 3 artigos publicados em periódicos internacionais [35][39][40], 2 artigos publicados em conferências internacionais [34][36] e um artigo resumido publicado em conferência internacional [37]. Além disso, dois artigos estão em processo de revisão em periódicos internacionais [38][43].

Parte I

Background / Contextualização

Capítulo 2

Comparação de Sequências Biológicas

Uma das operações mais básicas da Bioinformática é a comparação de sequências biológicas [4]. Na Ciência da Computação, existem vários algoritmos e métodos que comparam sequências biológicas, subsidiando a análise de diversos problemas da Biologia Molecular.

Para a Bioinformática, as sequências biológicas são representadas como uma sequência ordenada de caracteres. Um alinhamento é um pareamento entre os caracteres de duas sequências com a possível inserção de espaços (*gaps*) nas sequências. Chamamos de *matches* os pareamentos entre caracteres iguais e de *mismatches* os pareamentos entre caracteres diferentes [4]. A Figura 2.1 apresenta um exemplo de alinhamento entre duas sequências S_0 (acima) e S_1 (abaixo), onde os *matches* estão representados pelo símbolo ‘:’, *mismatches* pelo símbolo ‘.’ e *gaps* pelo símbolo ‘-’.

```
1  AAATT---GTAGCGA- 12
   :::  ::  ::::
1  --AATGGCGT-GCTAC 13
```

Figura 2.1: Exemplo de alinhamento entre duas sequências.

Um alinhamento é dito *global* caso ele englobe todos os caracteres de ambas as sequências, *semi-global* caso ele ignore o início ou o final de no mínimo uma das sequências e *local* caso ele englobe *substrings* das sequências. Cada alinhamento possui um escore associado, de acordo com um sistema de pontuação para os *matches*, *mismatches* e *gaps* [4]. A pontuação dos *gaps* é definida por uma função de penalidade dependente do número de *gaps* consecutivos. Dentre as funções mais comuns encontramos o *linear gap* (penalidade proporcional ao comprimento de uma sequência de *gaps*) e o *affine gap* (a abertura do *gap* possui uma penalidade adicional).

A comparação de sequências biológicas visa encontrar o escore ótimo entre todos os possíveis alinhamentos de duas sequências, considerando o sistema de pontuação escolhido. O alinhamento de sequências visa, além de encontrar o escore ótimo, obter um ou mais alinhamentos cujos escores sejam iguais ao escore ótimo. A operação de alinhamento tende a ser mais custosa computacionalmente e, por isso, a comparação de sequências é muitas vezes preferida, mesmo sem produzir o alinhamento. Existem também algoritmos heurísticos capazes de encontrar escores e alinhamentos significativos para os biólogos, embora sem a garantia de obtenção do resultado ótimo. Dentre os algoritmos heurísticos podemos citar o FASTA [13], o BLAST [14] e o MUMmer [44–46].

O objetivo deste capítulo é definir, formalizar e uniformizar termos e símbolos comuns que serão utilizadas ao longo desta tese (Seção 2.1). Além disso, serão apresentados algoritmos exatos de comparação e alinhamento de sequências biológicas (Seção 2.2).

2.1 Definições

Nesta seção, apresentaremos definições comuns que serão utilizadas nos capítulos dessa tese. Inicialmente (Seção 2.1.1), são apresentadas definições de sequências, escores e alinhamentos, comumente usadas na literatura [4][5][6][47]. Ao final (Seção 2.1.2), apresentaremos algumas definições propostas para esta tese. Outras definições mais específicas serão propostas nos Capítulos 7, 8 e 11.

2.1.1 Definições Constantes na Literatura

As definições apresentadas nessa seção foram retiradas de várias fontes [4][5][6][47]. Para uniformizar o uso destas definições, propusemos uma nomenclatura padrão que pudesse ser usada ao longo da tese. A Tabela 2.1 apresenta as definições descritas nesta seção.

Tabela 2.1: Definições

| Definição | Símbolo | Descrição |
|------------------|--|----------------------------|
| Definição 2.1.1 | Σ | Alfabeto |
| Definição 2.1.2 | S | Sequência |
| Definição 2.1.3 | $S[i]$ | Elemento da Sequência |
| Definição 2.1.4 | $ S $ | Comprimento da Sequência |
| Definição 2.1.5 | S' | Complemento |
| Definição 2.1.6 | $rev(S)$ | Reverso |
| Definição 2.1.7 | $rev(S')$ | Complemento Reverso |
| Definição 2.1.8 | A | Alinhamento |
| Definição 2.1.9 | $ A $ | Comprimento do Alinhamento |
| Definição 2.1.10 | $sbt(a, b)$ | Matriz de Substituição |
| Definição 2.1.11 | $\gamma(k)$ | Penalidade de <i>Gaps</i> |
| Definição 2.1.12 | $\gamma(k) = -k \cdot G$ | <i>Linear gap</i> |
| Definição 2.1.13 | $\gamma(k) = -G_{first} - (k - 1) \cdot G_{ext}$ | <i>Affine gap</i> |
| Definição 2.1.14 | s_i | Escore de uma coluna |
| Definição 2.1.15 | $score(A) = \sum s_i$ | Escore de um alinhamento |
| Definição 2.1.16 | $score^{opt}$ | Escore Ótimo |
| Definição 2.1.17 | A^{opt} | Alinhamento Ótimo |

Definição 2.1.1 (Alfabeto) Um alfabeto Σ é o conjunto de caracteres necessários para representar textualmente uma sequência de DNA, de RNA ou de aminoácidos. O alfabeto para as sequências de DNA é $\Sigma = \{A, C, G, T\}$, para sequências de RNA é $\Sigma = \{A, C, G, U\}$ e para sequências de aminoácidos é $\Sigma = \{A, C, D, E, F, G, H, I, K, L, M, N, P, Q, R, S, T, V, W, Y\}$. Adicionalmente, a *International Union of Pure and Applied*

Tabela 2.2: Códigos IUPAC [48]

| Caractere | Descrição |
|-----------|-------------------------|
| A | Adenina |
| C | Citosina |
| G | Guanina |
| T | Timina |
| U | Uracila |
| R | A ou G (Purinas) |
| Y | T, U ou C (Pirimidinas) |
| M | C ou A |
| K | T, U ou G |
| W | T, U ou A |
| S | C ou G |
| B | C, T, U ou G |
| D | A, T, U ou G |
| H | A, T, U ou C |
| V | A, C ou G |
| N | A, C, G, T ou U |

| Caractere | Descrição |
|-----------|-------------------------------|
| A | Alanina |
| B | Ácido aspártico ou Asparagina |
| C | Cisteína |
| D | Ácido aspártico |
| E | Ácido glutâmico |
| F | Fenilalanina |
| G | Glicina |
| H | Histidina |
| I | Isoleucina |
| K | Lisina |
| L | Leucina |
| M | Metionina |
| N | Asparagina |
| O | Pirrolisina |
| P | Prolina |
| Q | Glutamina |
| R | Arginina |
| S | Serina |
| T | Treonina |
| U | Selenocisteína |
| V | Valina |
| W | Triptofano |
| Y | Tirosina |
| Z | Ácido glutâmico ou Glutamina |
| X | qualquer |

Chemistry (IUPAC) [48] complementa este alfabeto com outros caracteres que representam ambiguidades encontradas nas sequências. Os alfabetos completos da IUPAC encontram-se na Tabela 2.2.

Definição 2.1.2 (Sequência) Uma sequência $S = \{c_1, c_2, \dots, c_n\}$ é uma série ordenada de caracteres de um alfabeto Σ .

Definição 2.1.3 (Elementos da Sequência) O i -ésimo elemento de uma sequência $S = \{c_1, c_2, \dots, c_n\}$ é representado como $S[i] = c_i$. Chamamos de i o índice do elemento $S[i]$.

Definição 2.1.4 (Comprimento da Sequência) O número de caracteres na sequência $S = \{c_1, c_2, \dots, c_n\}$ é representado por $|S| = n$.

Definição 2.1.5 (Complemento) As bases nitrogenadas ocorrem aos pares na sequência de DNA. Assim, o complemento da Adenina é a Timina (e vice-versa) e o complemento da Citosina é a Guanina (e vice-versa). Representamos o complemento de uma base c como sendo c' . Analogamente, o complemento S' de uma sequência $S =$

$\{c_1, c_2, \dots, c_n\}$ é definido como $S' = \{c'_1, c'_2, \dots, c'_n\}$

Definição 2.1.6 (Reverso) O reverso de uma sequência $S = \{c_1, c_2, \dots, c_n\}$ é obtido por meio da inversão do sentido de seus caracteres, ou seja, $rev(S) = \{c_n, c_{n-1}, \dots, c_1\}$.

Definição 2.1.7 (Complemento Reverso) O complemento reverso de uma sequência de DNA $S = \{c_1, c_2, \dots, c_n\}$ é obtido por meio da inversão do sentido de seus caracteres e a complementação de cada um deles, ou seja, $rev(S') = \{c'_n, c'_{n-1}, \dots, c'_1\}$.

Definição 2.1.8 (Alinhamento) Um alinhamento é um pareamento entre os caracteres das duas sequências com a possível inserção de espaços (*gaps*). Define-se o alinhamento pela matriz da Equação 2.1, onde cada par de elementos $\binom{a_i}{b_i}$ representa o pareamento entre os caracteres $a_i, b_i \in \Sigma \cup \{-\}$. A ocorrência de um *gap* na posição i é representada por $a_i = -$ ou $b_i = -$. Excluindo os *gaps* '-', os caracteres da sequência $\{a_1, a_2, \dots, a_l\}$ e $\{b_1, b_2, \dots, b_l\}$ formam *substrings* de S_0 e S_1 , respectivamente.

$$A = \begin{matrix} S_0 \\ S_1 \end{matrix} \begin{pmatrix} a_1 & a_2 & a_3 & a_4 & \dots & a_l \\ b_1 & b_2 & b_3 & b_4 & \dots & b_l \end{pmatrix} \text{ onde } a_i, b_i \in \Sigma \cup \{-\} \quad (2.1)$$

O alinhamento exemplificado na Figura 2.1 é representado pela matriz descrita na Equação 2.2.

$$A = \begin{matrix} S_0 \\ S_1 \end{matrix} \begin{pmatrix} A & A & A & T & T & - & - & - & G & T & A & G & C & G & A & - \\ - & - & A & A & T & G & G & C & G & T & - & G & C & T & A & C \end{pmatrix} \quad (2.2)$$

Definição 2.1.9 (Comprimento do Alinhamento) O comprimento do alinhamento é o número de pareamentos entre os caracteres das sequências. Sendo A o alinhamento definido na equação 2.1, o comprimento deste alinhamento é definido como $|A| = l$, onde l é o índice do último pareamento $\binom{a_l}{b_l}$. O alinhamento exemplificado na Figura 2.1 e na Equação 2.2 possui comprimento 16.

Definição 2.1.10 (Matriz de Substituição) Uma matriz de substituição sbt é uma matriz de duas dimensões que indica uma pontuação numérica para cada par de elementos $a, b \in \Sigma$. Quanto maior for o valor numérico $sbt(a, b)$ associado a esse par, mais próximo evolutivamente ele será. No caso de um alfabeto de nucleotídeos, utiliza-se simplificadaamente um valor positivo único ($+ma$) para todos os *matches* (diagonal principal da matriz) e um valor negativo único ($-mi$) para o restante da matriz, conforme exemplificado na Tabela 2.3. Existem também conjunto de matrizes de substituição pré-calculadas para aminoácidos, como por exemplo o conjunto de matrizes BLOSUM [49].

Tabela 2.3: Exemplo de matriz de substituição para DNA.

| | A | C | T | G |
|---|----|----|----|----|
| A | 1 | -3 | -3 | -3 |
| C | -3 | 1 | -3 | -3 |
| T | -3 | -3 | 1 | -3 |
| G | -3 | -3 | -3 | 1 |

Definição 2.1.11 (Penalidade de *Gaps*) Em um alinhamento, as ocorrências de *gaps* são associadas a uma pontuação numérica negativa definida por uma função de penalidade $\gamma(k)$, onde k é o comprimento de uma sequência consecutiva de *gaps*.

Definição 2.1.12 (*Linear gap*) O *linear gap* é uma função de penalidade de *gaps* que atribui o mesmo valor negativo $-G$ para cada *gap* dentro de uma sequência, ou seja, $\gamma(k) = -k \cdot G$.

Definição 2.1.13 (*Affine gap*) O *affine gap* é uma função de penalidade de *gaps* que atribui uma penalidade $-G_{first}$ para o primeiro *gap* de uma sequência de *gaps* e uma penalidade $-G_{ext}$ para os demais *gaps* dessa sequência. Assim, a penalização de uma sequência de k *gaps* consecutivos é dada pela fórmula $\gamma(k) = -G_{first} - (k - 1) \cdot G_{ext}$, sendo G_{first} a penalidade por abrir um *gap* e G_{ext} a penalidade por estendê-lo. O *affine gap* formaliza o raciocínio biológico de que uma remoção de várias bases deve ser considerada como uma única remoção, tornando um *gap* longo mais propício de ocorrer do que vários *gaps* pequenos. O *linear gap* pode ser considerado um caso específico do *affine gap*, onde $G_{ext} = G_{first}$.

Definição 2.1.14 (Escore de uma coluna) Sendo A o alinhamento definido na equação 2.1, o escore s_i da i -ésima coluna $\begin{pmatrix} a_i \\ b_i \end{pmatrix}$ é definido por meio da Equação 2.3, onde gap_i é a pontuação numérica negativa definida pela função de penalidade por *gaps* (Definição 2.1.11) e sbt é a matriz de substituição (Definição 2.1.10).

$$s_i = \begin{cases} gap_i, & \text{se } a_i = - \text{ ou } b_i = - \\ sbt(a_i, b_i) & \text{caso contrário} \end{cases} \quad (2.3)$$

A Figura 2.2 apresenta o escore de cada coluna de um alinhamento, considerando o valor de *match* como +1, de *mismatch* como -1 e de *gaps* como -2 no modelo *linear gap*.

```

A A A T T - - - G T A G C G A -
  : . :           : :   : : . :
- - A A T G G C G T - G C T A C
-2-2+1-1+1-2-2-2+1+1-2+1+1-1+1-2

```

Figura 2.2: Escore de cada coluna de um alinhamento.

Definição 2.1.15 (Escore de um alinhamento) O escore de um alinhamento é a soma $score(A) = \sum s_i$ dos escores de todas as colunas (Definição 2.1.14). No exemplo da Figura 2.2, o escore total do alinhamento é -9 .

Definição 2.1.16 (Escore ótimo) Chamamos de *escore ótimo* o maior escore encontrado no conjunto contendo todos os alinhamentos de um determinado tipo. Por exemplo, em um conjunto com todos os alinhamentos globais entre duas sequências, o maior escore deste conjunto é o escore global ótimo. Analogamente, o mesmo se aplica para os conjuntos de todos os possíveis alinhamentos locais e semi-globais, em que se define, respectivamente, o escore local ótimo e o escore semi-global ótimo. Chamaremos o escore ótimo de $score^{opt}$.

Definição 2.1.17 (Alinhamento ótimo) Chamamos de *alinhamentos ótimos* aqueles cujo escore for o maior possível dentro de um conjunto contendo todos os alinhamentos de um determinado tipo. Embora o escore ótimo seja único, podem existir vários alinhamentos ótimos com o mesmo escore. Por exemplo, em um conjunto com todos os alinhamentos globais entre duas sequências, os alinhamentos globais ótimos são aqueles cujos escores forem iguais ao escore global ótimo. Analogamente, o mesmo se aplica para os conjuntos de alinhamentos locais e semi-globais, em que se define, respectivamente, os alinhamentos locais ótimos e os alinhamentos semi-globais ótimos. Chamaremos o alinhamento ótimo de A^{opt} .

A Figura 2.3 apresenta o alinhamento global ótimo entre duas sequências, cujo escore ótimo é -3 .

```

A A - A T T G T A G C G A -
: : . . . : : : : :
A A T G G C G T - G C T A C

+1+1-2-1-1-1+1+1-2+1+1-1+1-2

```

Figura 2.3: Alinhamento global ótimo entre duas sequências.

2.1.2 Definições Propostas nesta tese

Percebemos que na literatura há várias definições diferentes para alinhamento semi-global [50][51][52]. Por isso, propomos uma definição homogênea para a distinção entre alinhamentos globais, locais e semi-globais. A Tabela 2.4 apresenta as definições descritas nesta seção e que serão utilizadas com uma notação uniforme para os tipos de alinhamentos abordados nesta tese.

Tabela 2.4: Definições

| Definição | Símbolo | Descrição |
|------------------|-----------------------|--|
| Definição 2.1.18 | $coord(A)$ | Coordenadas do Alinhamento |
| Definição 2.1.19 | $start(A)$ e $end(A)$ | Coordenadas de Início e Fim do Alinhamento |
| Definição 2.1.20 | t_{start}/t_{end} | Tipo de Alinhamento |

Definição 2.1.18 (Coordenadas do Alinhamento) As coordenadas $coord(A)$ do alinhamento A (Definição 2.1.8) representam os índices dos elementos pareados das sequências S_0 e S_1 ou dos *gaps*. As coordenadas são formalmente definidas pela matriz da Equação 2.4.

$$coord(A) = \begin{matrix} S_0 \\ S_1 \end{matrix} \begin{pmatrix} a_0 & a_1 & a_2 & a_3 & a_4 & \cdots & a_l \\ b_0 & b_1 & b_2 & b_3 & b_4 & \cdots & b_l \end{pmatrix} \text{ onde } a_i, b_i \in \mathbb{N} \quad (2.4)$$

Para $i > 0$, cada coordenada $\begin{pmatrix} a_i \\ b_i \end{pmatrix}$ respeita as Equações 2.5 e 2.6 indicando: 1) a ocorrência de um *gap* na sequência S_0 (se $a_i = a_{i-1}$); 2) a ocorrência de um *gap* na sequência S_1 (se $b_i = b_{i-1}$); ou o pareamento entre os caracteres $S_0[a_i]$ e $S_1[b_i]$, caso contrário. Adicionalmente, a diferença entre duas coordenadas consecutivas varia entre 0 e 1, ou seja $a_i - a_{i-1} \in \{0, 1\}$ e $b_i - b_{i-1} \in \{0, 1\}$, para $i > 0$. A coordenada $\begin{pmatrix} a_0 \\ b_0 \end{pmatrix}$ é necessária para permitir a indicação de *gaps* no primeiro pareamento $\begin{pmatrix} a_1 \\ b_1 \end{pmatrix}$ do alinhamento.

$$a_{i>0} = \begin{cases} a_i, & \text{em caso de } gap, \text{ ou} \\ a_i + 1 & \text{em caso de } match \text{ ou } mismatch \end{cases} \quad (2.5)$$

$$b_{i>0} = \begin{cases} b_i, & \text{em caso de } gap, \text{ ou} \\ b_i + 1 & \text{em caso de } match \text{ ou } mismatch \end{cases} \quad (2.6)$$

As coordenadas do alinhamento da Equação 2.2 são representadas formalmente pela matriz descrita na Equação 2.7. As coordenadas do alinhamento podem ser representadas visualmente por meio de segmentos de linhas plotados em um plano cartesiano, conforme a Figura 2.4.

$$coord(A) = \begin{matrix} S_0 \\ S_1 \end{matrix} \begin{pmatrix} \mathbf{0} & 1 & 2 & 3 & 4 & 5 & 5 & 5 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & \mathbf{12} \\ \mathbf{0} & 0 & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 8 & 9 & 10 & 11 & 12 & \mathbf{13} \end{pmatrix} \quad (2.7)$$

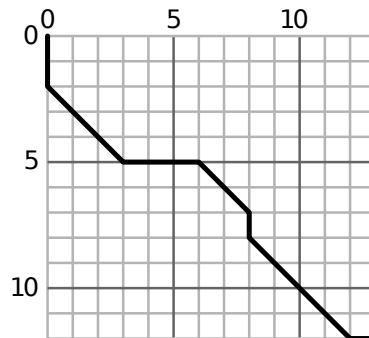


Figura 2.4: Plot de coordenadas de um alinhamento.

Definição 2.1.19 (Coordenadas de Início e Fim do Alinhamento) Sendo $coord(A)$ as coordenadas definidas na Equação 2.4, define-se o início do alinhamento A como sendo a coordenada $start(A) = (a_0, b_0)$ e o fim do alinhamento A como sendo a coordenada $end(A) = (a_l, b_l)$ onde l é o comprimento do alinhamento. O alinhamento exemplificado na Equação 2.2 possui início nos índices $(0, 0)$ e fim nos índices $(12, 13)$.

Definição 2.1.20 (Tipo de Alinhamento) Considerando que o início e o fim do alinhamento A são representados respectivamente por $start(A) = (a_0, b_0)$ e $end(A) = (a_l, b_l)$ (Definição 2.1.19), classificamos os tipos de alinhamento por meio da análise das posições de início e fim do alinhamento, conforme as condições elencadas na Equação 2.8. Dizemos que o alinhamento é do tipo *local* se ele inicia ou termina em qualquer posição. Se todas as condições da Equação 2.8 forem verdadeiras, dizemos que o alinhamento é do tipo *global*. Se ao menos uma das condições for verdadeira, dizemos que o alinhamento é do tipo *semi-global*.

$$\text{condições: } \begin{cases} a_0 = 0 \\ b_0 = 0 \\ a_l = |S_0| \\ b_l = |S_1| \end{cases} \quad (2.8)$$

A Tabela 2.5 apresenta uma classificação com 5 tipos de coordenadas de início e fim. A combinação dos tipos t_{start}/t_{end} de ambas as coordenadas geram 25 diferentes tipos de alinhamento. O tipo $+/+$ representa um alinhamento global, o tipo $*/*$ um alinhamento local e os demais 23 tipos são alinhamentos semi-globais. Dentre esses últimos, um alinhamento que se inicie no primeiro caractere de ambas as sequências e termine em qualquer caractere é classificado como do tipo $+/*$ e um alinhamento que se inicie no primeiro caractere da primeira sequência (S_0) e termine no último caractere da segunda sequência (S_1) é classificado como do tipo $1/2$. A Figura 2.5 ilustra as coordenadas de 25 alinhamentos de tipos distintos.

Tabela 2.5: Classificação dos tipos de coordenada.

| Início do alinhamento | | |
|-----------------------|-------------------------------------|--------------------------------|
| Símbolo | Descrição | Condições |
| 1 | 1º caractere de S_0 | $a_0 = 0$ |
| 2 | 1º caractere de S_1 | $b_0 = 0$ |
| 3 | 1º caractere de S_0 ou S_1 | $a_0 = 0$ ou $b_0 = 0$ |
| + | 1º caractere de S_0 e S_1 | $a_0 = 0$ e $b_0 = 0$ |
| * | Qualquer caractere de S_0 e S_1 | - |
| Fim do alinhamento | | |
| Símbolo | Descrição | Condições |
| 1 | Último caractere de S_0 | $a_l = S_0 $ |
| 2 | Último caractere de S_1 | $b_l = S_1 $ |
| 3 | Último caractere de S_0 ou S_1 | $a_l = S_0 $ ou $b_l = S_1 $ |
| + | Último caractere de S_0 e S_1 | $a_l = S_0 $ e $b_l = S_1 $ |
| * | Qualquer caractere de S_0 e S_1 | - |

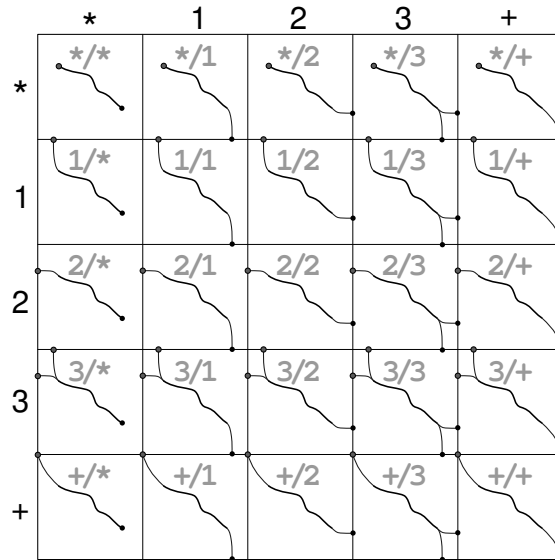


Figura 2.5: Tipos de alinhamento

Considerando todos os alinhamentos possíveis (ótimos ou não) entre duas sequências, podemos definir conjuntos para cada um dos tipos de alinhamento. Por meio da Figura 2.6, pode-se observar que os conjuntos formados por todos os alinhamentos de um determinado tipo podem se sobrepor. Por exemplo, a interseção dos alinhamentos do tipo 1/1 e 2/2 forma o conjunto dos alinhamentos +/+ (global). Por sua vez, a união dos alinhamentos 1/1 e 2/2 formam o conjunto dos alinhamentos 3/3. O conjunto dos alinhamentos do tipo */* (local) contém todos os demais conjuntos.

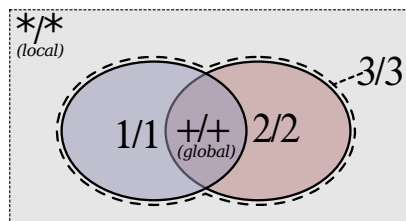


Figura 2.6: Conjuntos formados pelos alinhamentos de tipo 1/1, 2/2, 3/3, +/+ e */*.

2.2 Algoritmos Exatos

Os algoritmos exatos retornam sempre a solução ótima, ou seja, a melhor solução possível. Nessa seção, apresentaremos os algoritmos clássicos para obtenção do alinhamento global ótimo (Seção 2.2.1), do alinhamento local ótimo (Seção 2.2.2), do alinhamento ótimo com *affine gap* (Seção 2.2.3), do alinhamento semi-global ótimo (Seção 2.2.4) e do alinhamento global ótimo com *affine gap* em espaço linear (Seção 2.2.5). Além desses algoritmos, apresentaremos algumas outras variantes exatas, permitindo o *tradeoff* entre o uso de memória e tempo (Seção 2.2.6) ou o consumo de tempo e memória em complexidade $O(kn)$ (Seção 2.2.7 e Seção 2.2.8).

2.2.1 Needleman-Wunsch (NW)

O algoritmo Needleman-Wunsch (NW) [8] obtém o alinhamento global ótimo entre as sequências S_0 e S_1 , permitindo que *gaps* sejam inseridos para melhorar o alinhamento. Note que os alinhamentos globais incluem necessariamente todos os caracteres das sequências (Definição 2.1.20).

O algoritmo NW executa-se em duas fases: 1) cálculo da matriz de programação dinâmica e 2) obtenção do alinhamento ótimo (*traceback*). Cada fase será explicada a seguir.

Fase 1 – Cálculo da matriz de programação dinâmica: Na primeira fase, calcula-se uma matriz H onde o elemento $H_{i,j}$ representa o escore do alinhamento global ótimo entre os prefixos $S_0[1..i]$ e $S_1[1..j]$. Para calcular essa matriz, utiliza-se a técnica de programação dinâmica [53]. Primeiro, inicia-se o elemento $H_{0,0} = 0$, representando o escore entre duas sequências vazias. Em seguida, a primeira linha e a primeira coluna são iniciadas com os valores $H_{i,0} = -i \cdot G$ e $H_{0,j} = -j \cdot G$, onde G é a penalidade linear de *gap*.

A equação de recorrência do NW baseia-se no fato de que o valor $H_{i,j}$ de um alinhamento terminado nos caracteres $S_0[i]$ e $S_1[j]$ é o maior dos escores obtidos nos seguintes cenários: (1) alinhar $S_0[1..i-1]$ e $S_1[1..j-1]$ acrescido dos caracteres $S_0[i]$ e $S_1[j]$; (2) alinhar $S_0[1..i]$ e $S_1[1..j-1]$ e inserir um *gap* alinhado com $S_1[j]$; (3) alinhar $S_0[1..i-1]$ e $S_1[1..j]$ e inserir $S_0[i]$ alinhado com um *gap*. Desta forma, a recorrência é descrita pela Equação 2.9. Esta equação é aplicada para preencher a matriz H , do canto superior esquerdo para o canto inferior direito.

$$H_{i,j} = \max \begin{cases} H_{i-1,j-1} + sbt(S_0[i], S_1[j]) \\ H_{i-1,j} - G \\ H_{i,j-1} - G \end{cases} \quad (2.9)$$

Fase 2 – Obtenção do alinhamento ótimo (*traceback*): A segunda fase tem por objetivo encontrar o alinhamento global ótimo. Para tanto, cada célula $H_{i,j}$ possui um ponteiro indicando quais das células vizinhas $H_{i-1,j}$, $H_{i,j-1}$ ou $H_{i-1,j-1}$ originou o seu valor. Assim, percorre-se a sequência de ponteiros da matriz no sentido reverso, iniciando no canto inferior direito até chegar no canto superior esquerdo. Em alguns casos, o *traceback* poderá percorrer mais de um possível caminho na sequência de ponteiros, o que permite a geração de mais de um alinhamento global ótimo.

A Tabela 2.6 apresenta um exemplo da matriz de programação dinâmica para um alinhamento global ótimo. Os parâmetros utilizados foram: *Match*: +1; *Mismatch*: -1; *Gap*: -2. As células em destaque indicam o percurso reverso (*traceback*) capaz de produzir o alinhamento global ótimo com escore -3, conforme descrito na Figura 2.7.

$$\begin{array}{cccccccc} \text{A} & \text{A} & - & \text{A} & \text{T} & \text{T} & \text{G} & \text{T} & \text{A} & \text{G} & \text{C} & \text{G} & \text{A} & - \\ : & : & & . & . & . & : & : & : & : & . & . & : & \\ \text{A} & \text{A} & \text{T} & \text{G} & \text{G} & \text{C} & \text{G} & \text{T} & - & \text{G} & \text{C} & \text{T} & \text{A} & \text{C} \\ +1+1-2-1-1-1+1+1-2+1+1-1+1-2 & = & -3 \end{array}$$

Figura 2.7: Alinhamento global ótimo entre as sequências $S_0=AAATTGTAGCGA$ e $S_1=AATGGCGTGCTAC$.

Tabela 2.6: Matriz de programação dinâmica gerada pelo alinhamento global ótimo entre as sequências $S_0=AAATTGTAGCGA$ e $S_1=AATGGCGTGCTAC$.

| | | A | A | T | G | G | C | G | T | G | C | T | A | C |
|---|----------|----------|----------|----------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|
| | 0 | -2 | -4 | -6 | -8 | -10 | -12 | -14 | -16 | -18 | -20 | -22 | -24 | -26 |
| A | -2 | 1 | -1 | -3 | -5 | -7 | -9 | -11 | -13 | -15 | -17 | -19 | -21 | -23 |
| A | -4 | -1 | 2 | 0 | -2 | -4 | -6 | -8 | -10 | -12 | -14 | -16 | -18 | -20 |
| A | -6 | -3 | 0 | 1 | -1 | -3 | -5 | -7 | -9 | -11 | -13 | -15 | -15 | -17 |
| T | -8 | -5 | -2 | 1 | 0 | -2 | -4 | -6 | -6 | -8 | -10 | -12 | -14 | -16 |
| T | -10 | -7 | -4 | -1 | 0 | -1 | -3 | -5 | -5 | -7 | -9 | -9 | -11 | -13 |
| G | -12 | -9 | -6 | -3 | 0 | 1 | -1 | -2 | -4 | -4 | -6 | -8 | -10 | -12 |
| T | -14 | -11 | -8 | -5 | -2 | -1 | 0 | -2 | -1 | -3 | -5 | -5 | -7 | -9 |
| A | -16 | -13 | -10 | -7 | -4 | -3 | -2 | -1 | -3 | -2 | -4 | -6 | -4 | -6 |
| G | -18 | -15 | -12 | -9 | -6 | -3 | -4 | -1 | -2 | -2 | -3 | -5 | -6 | -5 |
| C | -20 | -17 | -14 | -11 | -8 | -5 | -2 | -3 | -2 | -3 | -1 | -3 | -5 | -5 |
| G | -22 | -19 | -16 | -13 | -10 | -7 | -4 | -1 | -3 | -1 | -3 | -2 | -4 | -6 |
| A | -24 | -21 | -18 | -15 | -12 | -9 | -6 | -3 | -2 | -3 | -2 | -4 | -1 | -3 |

Complexidade de Tempo e espaço: O algoritmo NW precisa calcular toda a matriz de programação dinâmica (fase 1), que contém $(m+1) \times (n+1)$ elementos, onde $m = |S_0|$ e $n = |S_1|$. O *traceback* é proporcional ao tamanho do alinhamento, que possui um limite superior na ordem de $O(m+n)$. Considerando que o tamanho das duas sequências são próximos, o algoritmo completo possui uma complexidade de $O(n^2)$ tanto em tempo de processamento como em uso de memória. Observe que essa complexidade inviabiliza o uso desse algoritmo para sequências muito grandes. Por exemplo, uma matriz de programação dinâmica entre duas sequências de DNA com 10 MBP (10 milhões de pares de bases) demandaria um uso de memória de 400 TB (Tera bytes), caso cada célula ocupasse 4 bytes na memória.

Se apenas o escore máximo for necessário (sem executar a fase de *traceback*), a matriz de programação dinâmica não precisa ser armazenada completamente, pois cada linha depende apenas da linha anterior (assim como cada coluna depende somente da coluna anterior). Sendo assim, o armazenamento de duas linhas (ou duas colunas) seria suficiente para a obtenção do escore ótimo, tornando a complexidade de espaço do algoritmo linear $O(m+n)$ em termos de espaço. A obtenção do escore sem o alinhamento é útil em caso de comparação entre uma sequência e um banco de sequências, com objetivo de retornar as sequências mais similares. Entretanto, para uma análise mais detalhada, o significado biológico não é tomado apenas pelo escore, mas sim pela representação do alinhamento completo. Então, mesmo na comparação contra um banco genômico, é importante que os alinhamentos das sequências mais similares sejam obtidos.

2.2.2 Smith-Waterman (SW)

Uma situação muito mais comum na Bioinformática é buscar o alinhamento local ótimo entre as sequências S_0 e S_1 , o que pode ser feito através do algoritmo de Smith-Waterman (SW) [9].

O processamento é bastante parecido com o do Needleman-Wunsch, havendo duas diferenças. A primeira diferença ocorre na equação de recorrência, em que se utiliza o valor zero para impedir que números negativos apareçam na matriz de programação

Tabela 2.7: Matriz de programação dinâmica gerada pelo alinhamento local ótimo entre as sequências $S_0=AAATTGTAGCGA$ e $S_1=AATGGCGTGCTAC$.

| | A | A | T | G | G | C | G | T | G | C | T | A | C |
|---|---|---|---|---|---|----------|----------|----------|---|---|---|---|---|
| | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| A | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| A | 0 | 0 | 2 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 |
| A | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 |
| T | 0 | 0 | 0 | 2 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 |
| T | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 |
| G | 0 | 0 | 0 | 0 | 2 | 2 | 0 | 1 | 0 | 2 | 0 | 0 | 0 |
| T | 0 | 0 | 0 | 1 | 0 | 1 | 1 | 0 | 2 | 0 | 1 | 1 | 0 |
| A | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 2 | 0 |
| G | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 1 | 0 | 1 | 0 | 0 | 1 |
| C | 0 | 0 | 0 | 0 | 0 | 0 | 2 | 0 | 0 | 0 | 2 | 0 | 1 |
| G | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 3 | 1 | 1 | 0 | 1 | 0 |
| A | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 2 | 0 | 0 | 2 | 0 |

dinâmica. A ocorrência de um valor zero representa o começo de um novo alinhamento, pois se o alinhamento até um determinado ponto for negativo, então é mais vantajoso começar um novo alinhamento naquela posição. Como consequência, a primeira linha e primeira coluna devem ser preenchidas com zeros. A equação de recorrência de SW está descrita na Equação 2.10.

$$H_{i,j} = \max \begin{cases} H_{i-1,j-1} + sbt(S_0[i], S_1[j]) \\ H_{i-1,j} - G \\ H_{i,j-1} - G \\ 0 \end{cases} \quad (2.10)$$

A segunda mudança em relação ao NW ocorre na forma de se realizar o *traceback*. Em vez de iniciar o *traceback* na posição final $H_{m,n}$, o alinhamento no SW inicia-se na posição $H_{i,j}$ onde ocorrer o maior valor da matriz. O percurso é feito até encontrar uma célula com valor zero. Dessa forma, obtém-se o melhor alinhamento entre duas subsequências. A Tabela 2.7 apresenta um exemplo da matriz de programação dinâmica para o alinhamento local, com score +3. Os parâmetros utilizados foram: *Match*: +1; *Mismatch*: -1; *Gap*: -2. As células em destaque indicam o percurso reverso (*traceback*) capaz de produzir o alinhamento local ótimo da Figura 2.8.

$$\begin{array}{c} \text{G C G} \\ \vdots \\ \text{G C G} \\ \\ +1+1+1 = +3 \end{array}$$

Figura 2.8: Alinhamento local ótimo entre as sequências $S_0=AAATTGTAGCGA$ e $S_1=AATGGCGTGCTAC$.

2.2.3 Gotoh

Uma generalização da Equação 2.9 pode ser feita substituindo a constante G por uma função genérica $\gamma(k)$ que retorna a penalidade de um *gap* com comprimento k (Definição 2.1.11). Obtemos então a Equação 2.11, que pode ser resolvida usando programação dinâmica com a complexidade aumentada para $O(n^3)$ [54].

$$H_{i,j} = \max \begin{cases} H_{i-1,j-1} + sbt(S_0[i], S_1[j]) \\ H_{i-k,j} - \gamma(k), & k = 1, \dots, i-1 \\ H_{i,j-k} - \gamma(k), & k = 1, \dots, j-1 \end{cases} \quad (2.11)$$

Um caso particular ocorre para o modelo de *affine-gap* (Definição 2.1.13), em que temos a fórmula $\gamma(k) = -G_{first} - (k-1) \cdot G_{ext}$. Para este modelo, o algoritmo desenvolvido por Gotoh [10] calcula o alinhamento global ótimo com complexidade de tempo e espaço $O(n^2)$.

Para cada posição (i, j) da matriz de programação dinâmica, mantêm-se três variáveis correspondentes a três situações distintas: (1) $S_0[i]$ alinhado com $S_1[j]$; (2) um *gap* alinhado com $S_1[j]$; (3) $S_0[i]$ alinhado com um *gap*. Definem-se então três matrizes H , E e F para cada uma das situações. A fórmula de recorrência é então modificada conforme as equações 2.12, 2.13 e 2.14. Adicionalmente, pode-se incluir uma condição para limitar o valor mínimo da matriz H em zero, adaptando-o assim para a obtenção do alinhamento local.

$$H_{i,j} = \max \begin{cases} H_{i-1,j-1} + sbt(S_0[i], S_1[j]) \\ E_{i,j} \\ F_{i,j} \\ 0 \end{cases} \quad (2.12)$$

$$E_{i,j} = \max \begin{cases} E_{i,j-1} - G_{ext} \\ H_{i,j-1} - G_{first} \end{cases} \quad (2.13)$$

$$F_{i,j} = \max \begin{cases} F_{i-1,j} - G_{ext} \\ H_{i-1,j} - G_{first} \end{cases} \quad (2.14)$$

2.2.4 Alinhamento Semi-Global Ótimo

Para obter alinhamentos semi-globais conforme descrito na Definição 2.1.20, algumas modificações são necessárias nos algoritmos NW (Seção 2.2.2) ou SW (Seção 2.2.2). A Tabela 2.8 apresenta as modificações necessárias. O tipo de início de alinhamento desejado influencia a forma de inicialização da primeira linha e primeira coluna e a inclusão ou não do limite mínimo 0 na equação de recorrência. Já o tipo de fim do alinhamento somente influencia o critério de busca pelo score ótimo, que pode ser em toda a matriz, na última linha ou coluna ou na última célula. Ressalta-se que os tipos $+/+$ e $*/*$ são equivalentes aos alinhamentos globais e locais, respectivamente, sendo que todos os demais tipos representam alinhamentos semi-globais.

Para o modelo *affine gap*, sempre que uma linha ou coluna for inicializada com zero, as matrizes E e F correspondentes devem ser inicializadas com $-\infty$, impedindo que

Tabela 2.8: Modificações no algoritmo NW para cada tipo de alinhamento.

| Início do alinhamento | | |
|-----------------------|-------------------------------------|---|
| Tipo | Descrição | Modificação |
| 1 | 1º caractere de S_0 | Inicializar primeira linha com 0 |
| 2 | 1º caractere de S_1 | Inicializar primeira coluna com 0 |
| 3 | 1º caractere de S_0 ou S_1 | Inicializar primeira linha e primeira coluna com 0 |
| + | 1º caractere de S_0 e S_1 | Inicializar primeira linha e primeira coluna com penalidade de <i>gaps</i> |
| * | Qualquer caractere de S_0 e S_1 | Inicializar primeira linha e primeira coluna com 0 e proibir valores negativos em toda a matriz |
| Fim do alinhamento | | |
| Tipo | Descrição | Modificação |
| 1 | Último caractere de S_0 | Buscar escore ótimo somente na última linha |
| 2 | Último caractere de S_1 | Buscar escore ótimo somente na última coluna |
| 3 | Último caractere de S_0 ou S_1 | Buscar escore ótimo somente na última coluna e na última linha |
| + | Último caractere de S_0 e S_1 | Buscar escore ótimo somente na última célula (na interseção da última linha e coluna) |
| * | Qualquer caractere de S_0 e S_1 | Buscar escore ótimo em qualquer posição da matriz |

Tabela 2.9: Matriz de programação dinâmica gerada pelo alinhamento semi-global ótimo (tipo 1/1) entre as sequências S_0 =GGATGCT e S_1 =AATGGCGTGCTAC.

| | | | | | | | | | | | | | | |
|---|-----|-----|-----|----------|----------|----------|----------|-----------|----------|----------|----------|----------|----|----|
| | | A | A | T | G | G | C | G | T | G | C | T | A | C |
| | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| G | -2 | -1 | -1 | -1 | 1 | 1 | -1 | 1 | -1 | 1 | -1 | -1 | -1 | -1 |
| G | -4 | -3 | -2 | -2 | 0 | 2 | 0 | 0 | 0 | 0 | 0 | -2 | -2 | -2 |
| A | -6 | -3 | -2 | -3 | -2 | 0 | 1 | -1 | -1 | -1 | -1 | -1 | -1 | -3 |
| T | -8 | -5 | -4 | -1 | -3 | -2 | -1 | 0 | 0 | -2 | -2 | 0 | -2 | -2 |
| G | -10 | -7 | -6 | -3 | 0 | -2 | -3 | 0 | -1 | 1 | -1 | -2 | -1 | -3 |
| C | -12 | -9 | -8 | -5 | -2 | -1 | -1 | -2 | -1 | -1 | 2 | 0 | -2 | 0 |
| T | -14 | -11 | -10 | -7 | -4 | -3 | -2 | -2 | -1 | -2 | 0 | 3 | 1 | -1 |

gaps iniciais sejam considerados como extensões de *gaps* (i.e. sem a penalidade extra de abertura de *gap*).

A Tabela 2.9 apresenta um exemplo da matriz de programação dinâmica para um alinhamento semi-global ótimo (tipo 1/1). Os parâmetros utilizados foram: *Match*: +1; *Mismatch*: -1; *Gap*: -2. As células em destaque indicam o percurso reverso (*traceback*) capaz de produzir o alinhamento semi-global ótimo descrito na Figura 2.9.

2.2.5 Myers e Miller (MM)

A complexidade de memória dos algoritmos vistos anteriormente inviabiliza a obtenção do alinhamento entre sequências longas. Para resolver este problema, algoritmos foram desenvolvidos para obter o alinhamento ótimo utilizando espaço linear, ou seja, na ordem de $O(m + n)$.

Hirschberg [12] desenvolveu um algoritmo para resolver o problema da Maior Subsequência Comum (LCS – *Longest Common Subsequence*) em espaço linear. Este algo-

$$\begin{array}{cccccccc}
& G & G & A & - & T & G & C & T \\
& : & : & . & : & : & : & : & : \\
A & A & T & G & G & C & G & T & G & C & T & A & C \\
& +1 & +1 & -1 & -2 & +1 & +1 & +1 & +1 & & & & = +3
\end{array}$$

Figura 2.9: Alinhamento semi-global ótimo (tipo 1/1) entre as sequências $S_0=GGATGCT$ e $S_1=AATGGCGTGCTAC$.

ritmo foi posteriormente adaptado por Myers e Miller [11] para transformar o algoritmo de Gotoh (Seção 2.2.3) em uma solução em espaço linear. A seguir, o algoritmo de Myers-Miller (MM) será apresentado.

Partindo de um alinhamento global, a ideia é encontrar o ponto médio (*crosspoint*) pelo qual passa um alinhamento ótimo. Para encontrar este ponto, a computação é feita em duas partes. Na primeira parte, o alinhamento é feito por linhas até a linha central $\frac{m}{2}$. Na segunda parte, o processamento é feito sobre as duas sequências invertidas $rev(S_0)$ e $rev(S_1)$ (Definição 2.1.6), até que a mesma linha central $\frac{m}{2}$ seja calculada.

Os valores finais da linha $\frac{m}{2}$ são armazenados nos vetores CC e DD . A diferença entre os dois vetores é que o vetor DD armazena o escore dos alinhamentos que terminam em *gap* e o vetor CC armazena o escore dos alinhamentos que terminam em *match* ou *mismatch*. Analogamente, o processamento das sequências invertidas são armazenadas nos vetores CC' e DD' .

Um alinhamento que cruza a linha $\frac{m}{2}$ e a coluna j terá escore $K_j = \max\{CC_j + CC'_j, DD_j + DD'_j - G_{open}\}$, onde G_{open} é a penalidade por abrir um *gap* (i.e. $G_{open} = G_{first} - G_{ext}$). Observe que, quando soma-se o valor dos dois vetores DD e DD' , considera-se um caso onde existem *gaps* em ambas as direções, logo deve-se remover a penalidade G_{open} de um dos *gaps*.

O escore K_{j^*} é o valor máximo entre todos os valores de K_j , ou seja, $K_{j^*} = \max_{0 \leq j < n} K_j$. Neste caso, conclui-se que a célula $(\frac{m}{2}, j^*)$ é um ponto pelo qual passa um alinhamento ótimo [11].

Em seguida, realiza-se recursivamente o processamento de duas seções da matriz: $(0, 0)$ a $(\frac{m}{2}, j^*)$ e $(\frac{m}{2}, j^*)$ a (m, n) . A cada passo, dividem-se as seções em trechos menores, até que o tamanho de cada seção seja trivial. A Figura 2.10 ilustra dois passos do algoritmo de Myers e Miller.

Haja vista que a cada passo recursivo a área de processamento cai pela metade, pode-se estimar que o próximo passo gastará metade do tempo do passo anterior. Se T_k é o tempo gasto pelo passo k , então $T_k = \frac{T_0}{2^k}$, sendo $0 \leq k < \log_2 m$. Somando todos os tempos

obtemos o tempo total $T = \sum_{k=1}^{\log_2 m - 1} \frac{T_0}{2^k} \leq 2.T_0$, logo, o algoritmo continua na ordem de $O(mn)$ em tempo, mas com o benefício de utilizar apenas $O(m + n)$ de memória.

2.2.6 FastLSA

O Algoritmo FastLSA [55] estende o conceito de Myers e Miller permitindo a divisão da matriz em várias linhas e colunas. Desta forma, é criado um *tradeoff* entre a quantidade de memória utilizada e o tempo necessário para a computação. Ao contrário do algoritmo de Myers-Miller que divide a matriz na linha central, o FastLSA divide a matriz em k

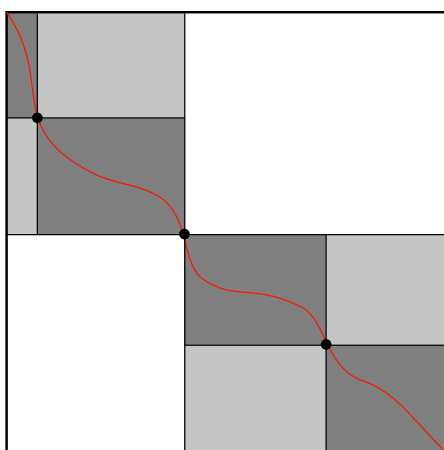


Figura 2.10: Dois níveis de recursão do algoritmo de Myers e Miller.

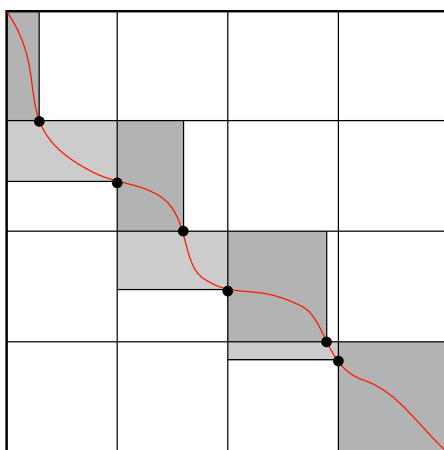


Figura 2.11: Execução do algoritmo FastLSA

partes para cada dimensão, criando um *grid* com $k \times k$ blocos, cada um contendo $\frac{m}{k} \times \frac{n}{k}$ células, onde m e n são os tamanhos das sequências. Quanto maior for o valor k , mais rápido será o *traceback*, com o *tradeoff* de se utilizar uma maior quantidade de memória. A Figura 2.11 apresenta um exemplo de *traceback* com o FastLSA, onde $k = 4$.

2.2.7 Fickett

Fickett [56] desenvolveu uma otimização capaz de reduzir o tempo de execução e memória de um alinhamento para $O(k \cdot n)$, onde k é a largura da faixa de diagonais que contém todo o alinhamento ótimo. Sequências muito similares tendem a possuir poucos *gaps* (i.e. k pequenos), logo o tempo de processamento pode ser consideravelmente reduzido utilizando esta técnica. O algoritmo de Fickett [56] resolve o problema da distância de edição, mas pode ser facilmente adaptado para NW (Seção 2.2.1), SW (Seção 2.2.2) e Gotoh (Seção 2.2.3).

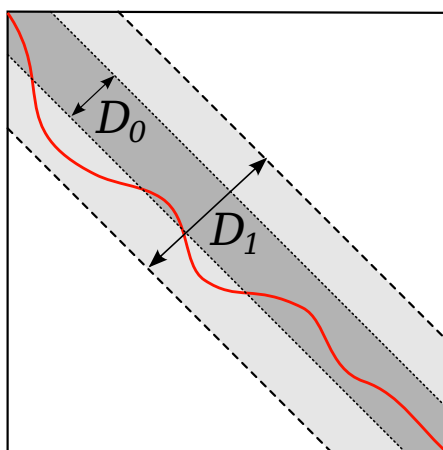


Figura 2.12: Representação do algoritmo de Fickett. O alinhamento ótimo não está inicialmente contido na área delimitada pelo valor D_0 , mas está dentro da área delimitada por D_1 .

A principal ideia do algoritmo é reordenar a computação da matriz de forma a calcular apenas as células d_{ij} da matriz cujo valor seja menor ou igual a um limitador D . Assim, limita-se o cálculo a uma área da matriz cujo alinhamento seja mais significativo.

Existem três formas de se determinar o limitador D :

1. **Heurística:** o limite superior de D pode ser escolhido, de maneira rápida e não-ótima, por meio de uma heurística, como por exemplo o BLAST [14];
2. **Escolha Manual:** o próprio usuário escolhe o valor máximo de D , aproveitando o conhecimento prévio que ele possui das sequências comparadas;
3. **Incremento Automático:** um valor inicial D_0 é fixado pelo algoritmo, que calcula as células tais que $d_{ij} \leq D_0$. Caso não seja possível encontrar o alinhamento ótimo com esta restrição, o algoritmo escolhe um valor $D_1 > D_0$ e calcula uma parte maior da matriz tal que $d_{ij} \leq D_1$. O valor D_1 pode ser escolhido, por exemplo, como o dobro do valor D_0 . O algoritmo continua gerando novos limites $D_2 > D_3 > \dots$ até que seja possível encontrar um alinhamento.

Dado o valor limite D , a matriz é computada da seguinte forma. Calculam-se inicialmente as células $d_{1,1}, \dots, d_{1,L_1}$, sendo L_1 o menor valor tal que $d_{1,L_1} \geq D$. Observe que $d_{1,j} \geq D$ para todo $j > L_1$. Em seguida, calcula-se $d_{2,1}, \dots, d_{2,L_2}$, sendo L_2 o menor valor tal que $d_{2,L_2} \geq D$ e $L_2 > L_1$. Assim, tem-se garantia de que $d_{2,j} > D$ para todo $j > L_2$. Este cálculo é feito linha após linha.

Caso uma linha i comece com K_i valores maiores que D , as K_i colunas iniciais podem ser ignoradas no cálculo das próximas linhas. Sendo assim, pode-se generalizar a computação da seguinte forma: calculados os elementos $d_{i,K_i}, \dots, d_{i,L_i}$ da linha i , a linha $i + 1$ é calculada para os elementos $d_{i+1,K_{i+1}}, \dots, d_{i+1,L_{i+1}}$, sendo: K_{i+1} o menor valor tal que $d_{i+1,K_{i+1}} < D$ e $K_{i+1} > K_i$; e L_{i+1} o menor valor tal que $d_{i,L_i} \geq D$ e $L_{i+1} > L_i$. Os valores K_i e L_i devem ser armazenados em memória para cada linha.

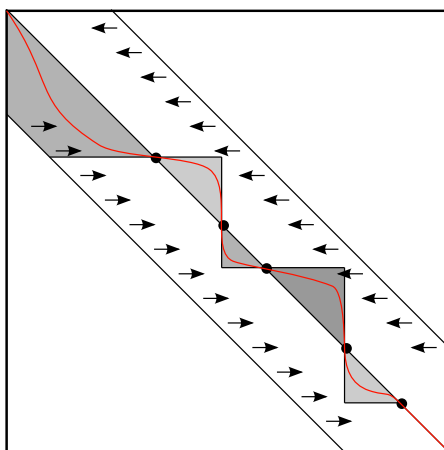


Figura 2.13: Representação do algoritmo de Chao-Pearson-Miller

Se o alinhamento não puder ser encontrado com o limite superior $D = D_0$, escolhe-se um valor $D_1 > D_0$ e continua-se o processamento nas colunas anteriores a K_i e posteriores a L_i . Este procedimento é repetido até que se encontre o alinhamento. A Figura 2.12 ilustra o funcionamento do algoritmo de Fickett.

2.2.8 Chao-Pearson-Miller

Chao e co-autores [57] propuseram um método que agregou as ideias de Fickett e de Myers-Miller. Neste método (Figura 2.13), a matriz também é processada em faixas de diagonais. Entretanto, a diagonal central é utilizada para dividir a matriz em duas metades, cada uma processada em sentidos diferentes. Os resultados da diagonal central são utilizados para encontrar os *crosspoints*, de maneira semelhante ao Myers-Miller, de forma que os *crosspoints* são utilizados para criar subproblemas menores, que são recursivamente processados. Um aspecto deste algoritmo é que em uma diagonal é possível identificar vários *crosspoints*, aumentando o número de subproblemas encontrados em uma única iteração.

Capítulo 3

Arquiteturas de Unidades de Processamento Gráfico (GPUs)

As Unidades de Processamento Gráfico (GPU - *Graphics Processing Unit*) são multiprocessadores com funções altamente otimizadas para acelerar a renderização gráfica [58]. A aceleração obtida pelas GPUs deve-se principalmente à sua capacidade de executar em *hardware* as funções matemáticas mais comuns em computação gráfica, assim como a existência de uma memória dedicada para armazenamento de texturas, polígonos, fontes e representações de objetos. Podemos citar entre as principais funções gráficas de uma GPU as operações de transformação, iluminação, texturização e rasterização.

As GPUs sofreram uma grande evolução nas últimas décadas [59], tornando-se plataformas com grande poder computacional. Esta capacidade de processamento passou a ser aproveitada também em áreas de pesquisa distintas à de computação gráfica, produzindo resultados expressivos em ambientes de baixo custo.

O objetivo deste capítulo é apresentar um histórico da evolução das GPUs (Seção 3.1) e descrever, de maneira mais detalhada, a plataforma CUDA — *Compute Unified Device Architecture* — (Seção 3.2), da NVIDIA, por ser a principal arquitetura escolhida para implementação desta tese de doutorado.

3.1 Evolução das Placas Gráficas

As funções executadas pelas GPUs são usualmente organizadas em *pipeline*. Nas primeiras gerações de GPUs (1970-1980), o *pipeline* era do tipo *fixed function pipeline* (FFP) [58], no qual as rotinas gráficas eram todas embarcadas, não permitindo que programadores criassem rotinas personalizadas.

As gerações seguintes de GPUs (1980-1990) [58] flexibilizaram o *pipeline* permitindo que sequências de instruções executassem em algum dos estágios do *pipeline* das GPUs [60, 61]. Estas sequências de instruções, chamadas de *shaders*, são capazes de manipular as posições dos vértices (*vertex shader*), as primitivas geométricas (*geometric shader*) ou as cores dos *pixels* (*pixel shader* ou *fragment shader*).

Algumas linguagens foram então criadas para permitir a codificação dos *shaders*. Uma das linguagens considerada como precursora [62–64] é a Shade Trees Language [65], datada em 1984, que a despeito de não ter sido usada em GPUs permitiu que objetos fossem graficamente alterados através de árvores de *shaders*. Em 1985, foi proposta a Pixel Stream

Editing Language [66], que permitiu que estruturas de programação fossem utilizadas. Em 1990, a Pixar publicou a RenderMan Shading Language (RSL) [67] para criação de filmes em três dimensões.

Em 1998, foi proposta a linguagem *pfman* [68], similar a RenderMan. Esta linguagem foi a primeira com compilação destinada a processadores gráficos [69], em especial para a plataforma PixelFlow [70].

Os frameworks OpenGL e ActiveX também criaram suas linguagens de *shaders*. Dentre as linguagens de baixo nível podemos citar o ARB - *OpenGL Assembly Language*. Dentre as linguagens de alto nível podemos citar o GLSL (*GL Shading Language* - OpenGL) e o HLSL (*High Level Shading Language* - DirectX) [58].

Com o advento dos *shaders*, as GPUs tornaram-se capazes de executar também algoritmos de propósito geral, ou seja, algoritmos não necessariamente relacionados à computação gráfica. Esta programação de propósito geral é chamada de GPGPU (*General-Purpose computing on Graphics Processing Units*). Algumas linguagens de alto nível foram desenvolvidas especificamente para programação em GPGPU, dentre as quais podemos citar Brook [71], Sh [72, 73] e CUDA [74].

As GPUs também são chamadas de *stream processors*, por causa de sua capacidade de manipular paralelamente vários elementos de dados, conceito chamado de SIMD (*Single Instruction Multiple Data*) na taxonomia de Flynn [75]. O paralelismo é obtido por meio dos vários núcleos interconectados no interior da GPU. As GPUs mais modernas chegam a possuir centenas ou até milhares de núcleos.

Assim como as arquiteturas Cell BE [76], Larrabee [77] e MIC (*Many Integrated Core architecture*) [78], as GPUs modernas encontram-se classificadas como arquiteturas paralelas *manycore* [79], que são aquelas que possuem, em um único *chip*, dezenas, centenas ou até milhares de núcleos [80]. Estas arquiteturas possuem infraestrutura (interconexão, hierarquia de memória, entre outros) desenhada para ser altamente escalável [81]. As plataformas *manycore* diferem-se das CPUs *multicore* tradicionais por utilizarem técnicas de paralelismo massivo para priorizarem a vazão de processamento, o que pode comprometer o tempo de resposta de operações que não exploram paralelismo [79, 82, 83]. Uma arquitetura *manycore* pode possuir núcleos homogêneos ou heterogêneos [79]. Quando heterogêneos, um processador pode conter poucos núcleos de maior tamanho, otimizados para o desempenho de tarefas com uma única *thread*, e vários núcleos pequenos, otimizados para produzir maior vazão de execução com um grande número de *threads* [79]. Além disso, podem existir núcleos de propósito específico para aceleração de tarefas em *hardware*, como acontece nas placas gráficas.

Com o suporte de arquiteturas *manycore*, várias áreas de pesquisa já apresentam resultados bastante expressivos com algoritmos paralelos [20]. Dentre as áreas de pesquisa que utilizam arquiteturas *manycore* podemos citar a Física [84, 85], a Criptografia [86, 87] e a Bioinformática [28, 88–91].

Atualmente existem vários fabricantes que possuem arquiteturas de *hardware manycore* capazes de realizar programação de propósito geral, dentre os quais podemos citar a NVIDIA, a AMD e a Intel.

3.2 CUDA

Compute Unified Device Architecture (CUDA) [74] é a arquitetura de *hardware* e de *software* da NVIDIA para programação em GPGPU através de linguagens de programação como C, C++ e Fortran e de *frameworks* como OpenCL e DirectCompute. Esta arquitetura inclui um conjunto de instruções (ISA), uma plataforma de desenvolvimento (SDK) e o próprio *hardware* em GPU.

A arquitetura G80, lançada em 2006, foi a primeira geração da arquitetura CUDA. Essa arquitetura introduziu algumas inovações em GPGPU, em especial o uso de um processador unificado para executar vários tipos de operações e o conceito de *Single-instruction multiple-thread* (SIMT), onde a programação é baseada em *threads* que são agrupadas de maneira a executar a mesma instrução simultaneamente. Em 2008, a segunda geração da arquitetura CUDA foi lançada com o nome de GT200, a qual aumentou o número de núcleos para até 240 e incluiu a capacidade de realizar operações em ponto flutuante com precisão dupla.

A terceira geração da arquitetura CUDA foi lançada em 2009 com o nome de Fermi (ou GF100) [92, 93]. Nesta geração, melhorias foram realizadas na hierarquia de memória, que passou a ter dois níveis de *cache* (L1 e L2). O número máximo de núcleos foi aumentado para 512, os quais passaram a suportar mais de um programa executado concorrentemente.

Em 2012, a NVIDIA lançou a quarta geração da arquitetura CUDA, chamada de Kepler (ou GK100) [94]. Nesta nova arquitetura, o desempenho de energia foi melhorado em até 3 vezes e a tecnologia GPUDirect permitiu que GPUs trocassem dados por meio de acesso de memória direto (DMA), sem o auxílio da CPU. Além disso, o código executado na GPU (*kernel*) tornou-se capaz de executar outros *kernels* sem o auxílio da CPU [94, 95]. Placas desta arquitetura são capazes de possuir mais que 3000 núcleos.

A NVIDIA iniciou, em 2014, a produção da quinta geração da arquitetura CUDA, chamada de Maxwell [96]. Nesta arquitetura, novas melhorias foram realizadas no desempenho de energia, com aumento da eficiência energética de até 2 vezes. Adicionalmente, a arquitetura Maxwell melhorou a latência de instruções aritméticas e aumentou o tamanho e a eficiência das estruturas de memória compartilhada e cache.

Cada arquitetura CUDA possui um número de versionamento chamado *compute capability*, no formato x.y (versão *major* e *minor* respectivamente), sendo 1.y para as séries G80 e GT200, 2.y para a Fermi, 3.y para a Kepler e 5.y para a Maxwell. Variações no número *y* representam modificações menores na mesma arquitetura, podendo haver novas funcionalidade [74].

A seguir, serão apresentadas as arquiteturas Fermi, Kepler e Maxwell da NVIDIA.

3.2.1 Componentes de *Hardware*

As GPUs das arquiteturas Fermi (Figura 3.1), Kepler (Figura 3.2) e Maxwell (Figura 3.3) são compostas por vários núcleos (*cores*), sendo que cada um possui uma unidade lógica aritmética para operações com inteiros (Int Unit) e outra ALU para operações em ponto flutuante (FP Unit). As unidades de controle dos núcleos executam as instruções em ordem.

Na arquitetura Fermi (Figura 3.1), cada conjunto de 32 núcleos é agrupado em um *Stream Multiprocessor* (SM), que possui uma unidade de controle *MultiThread* com *cache*

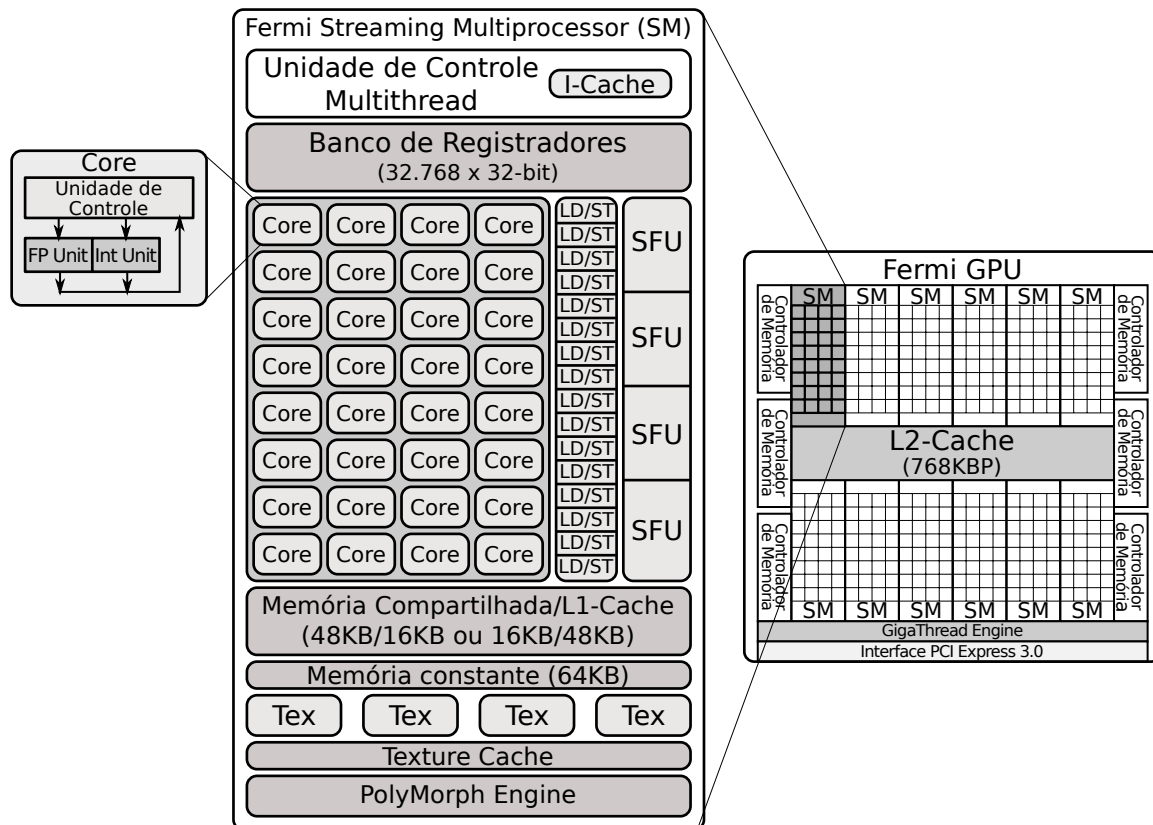


Figura 3.1: Arquitetura CUDA Fermi [93]

de instrução, 32.768 registradores, 4 unidades para processamento de funções especiais (*Special Function Units* - SFUs), 16 unidades para load/store (LD/ST), uma unidade de memória compartilhada, *cache* L1, unidade de memória constante de 64KB (*read-only*), unidades de textura e uma unidade para processamento de tecelagem gráfica (Polymorph Engine). A unidade de *cache* L1 e a memória compartilhada dividem o mesmo banco de 64KB, podendo ser configurado com 48KB de *cache* L1 e 16KB de memória compartilhada (48KB/16KB) ou vice-versa (16KB/48KB). Uma GPU Fermi é formada por um número variado de SMs, uma unidade de *cache* L2 de 768KB, 6 controladores internos de memória, uma unidade para o escalonamento de blocos de *threads* (GigaThread Engine) e a interface de PCI Express.

Na arquitetura Kepler (Figura 3.2), o *Stream Multiprocessor* (SMX) possui os mesmos componentes de um SM da arquitetura Fermi, porém em quantidades maiores: 192 núcleos, 65.536 registradores, 32 SFUs, 32 unidades para load/store (LD/ST) e 16 unidades de textura. O banco para o *cache* L1 e para a memória compartilhada pode ser configurado nas proporções 48KB/16KB, 32KB/32KB e 16KB/48KB. O *cache* L2 foi duplicado para 1536KB.

A arquitetura Maxwell (Figura 3.3), o *Stream Multiprocessor* (SMM) foi particionado em quatro grupos de núcleos, cada grupo com seu próprio banco de registradores e de SFU's [97]. Embora cada SMM possua apenas 128 núcleos, a NVIDIA informa que os 128 núcleos da arquitetura Maxwell possuem 90% do desempenho dos 192 núcleos da

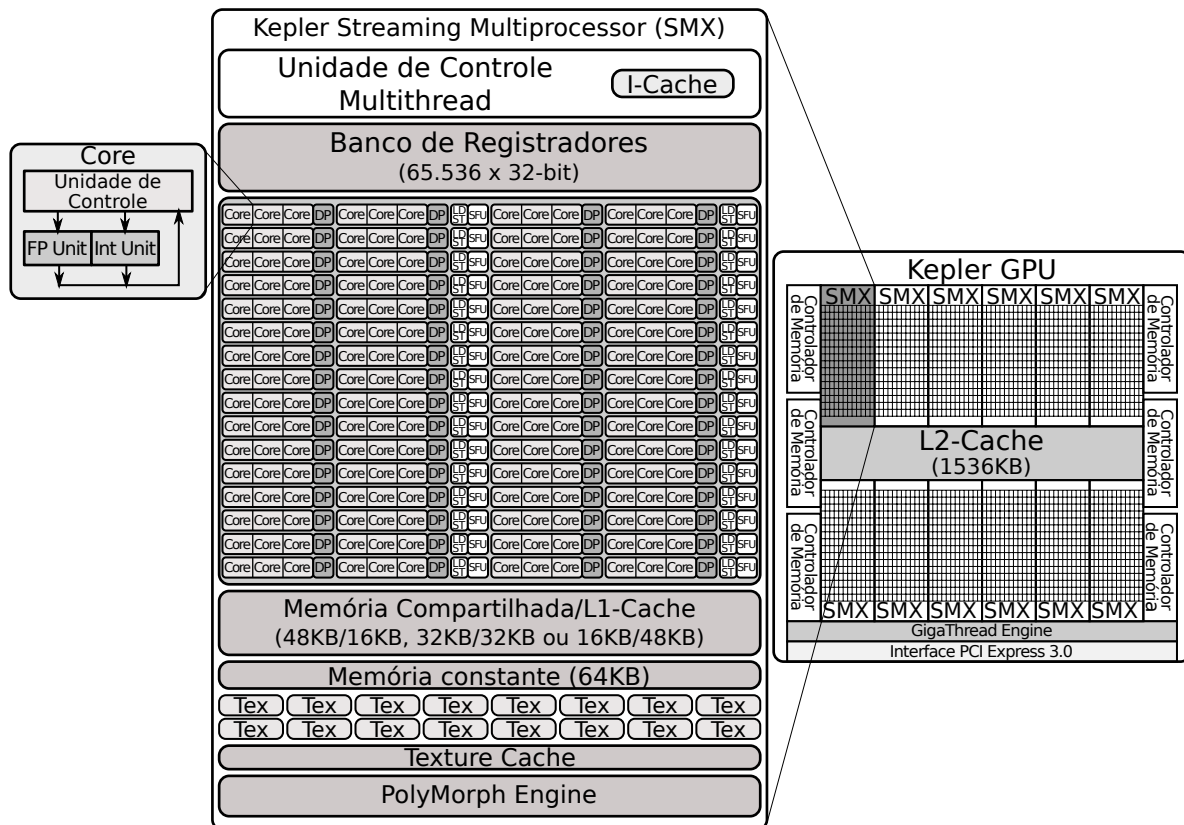


Figura 3.2: Arquitetura CUDA Kepler [94]

arquitetura Kepler, mas ocupando um espaço físico muito menor no processador. Desta forma, a eficiência de energia duplicou [96]. O banco para o *cache* L1 foi unificado com o banco de textura e o *cache* L2 foi aumentado para 2048KB. A memória compartilhada teve seu tamanho aumentado para 64KB, ou até 96 KB em algumas placas. Houve uma enorme redução no número de unidades de ponto flutuante de precisão dupla (DP), sendo que cada SMM possui apenas 4 dessas unidades (na arquitetura Kepler, cada SMX possui 64 DP's). A linha Tesla, destinada à computação de alto desempenho, não possui nenhuma placa com a arquitetura Maxwell, possivelmente por causa da falta de desempenho em operações de ponto flutuante com precisão dupla.

3.2.2 Linhas de Produtos da NVIDIA

A NVIDIA comercializa GPUs em várias linhas de placas, destinadas a diferentes ramos do mercado:

- **GeForce:** A linha GeForce é direcionada para computadores pessoais, em especial para aceleração gráfica de jogos. Por possuir um bom custo-benefício, a linha GeForce é muito utilizada em pesquisas acadêmicas. Essas placas chegam a possuir capacidade computacional na ordem de Teraflops.

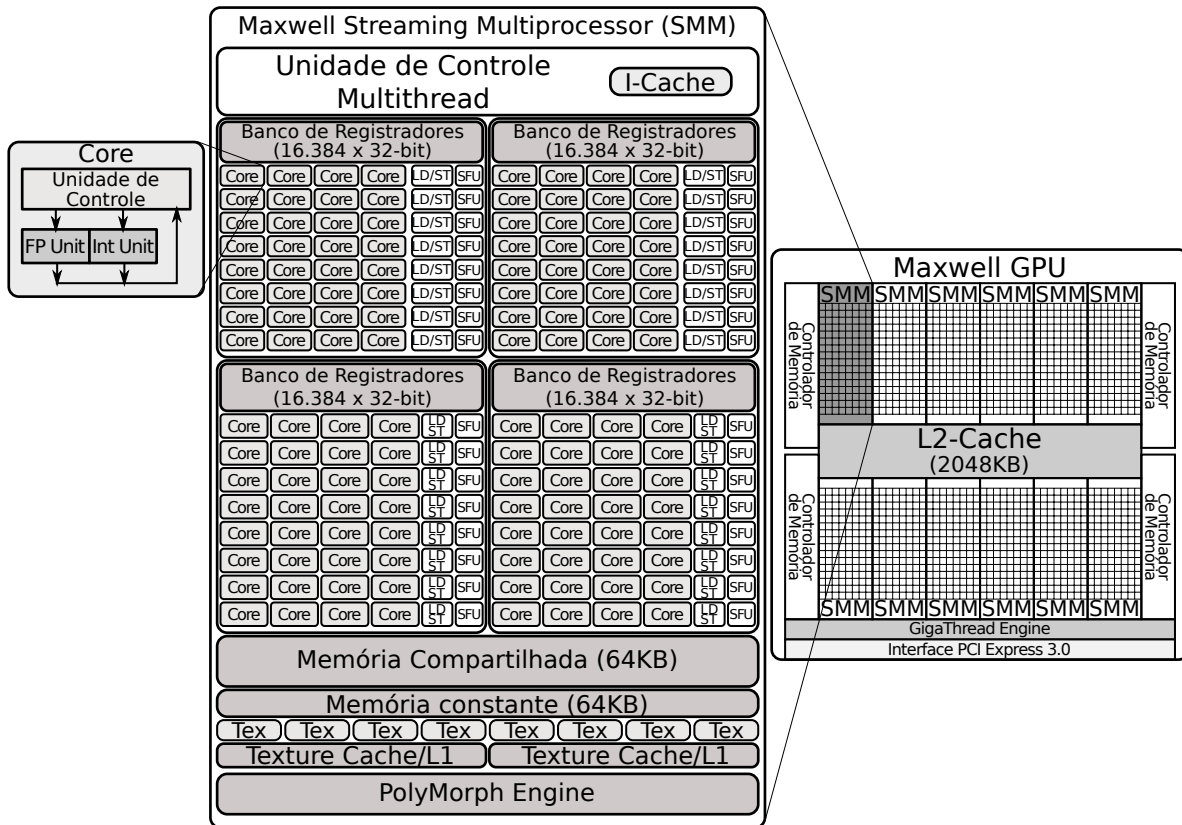


Figura 3.3: Arquitetura CUDA Maxwell [96]

- **Quadro:** A linha Quadro é composta por um conjunto de placas destinadas ao público que trabalha profissionalmente com processamento gráfico, dentre os quais podemos citar os criadores de conteúdo digital e os utilizadores de *softwares* CAD.
- **Tesla:** A linha TESLA contém placas mais caras, especializadas em GPGPU. Embora as placas TESLA não possuam saída de vídeo, elas fornecem recursos de aceleração gráfica assim como as GPUs convencionais.

As tabelas 3.1 e 3.2 apresentam comparativos entre modelos das linhas GeForce e TESLA, respectivamente. O cálculo de GFlops destas tabelas consideram operações de ponto flutuante com precisão simples (32 bit).

3.2.3 Componentes de *Software*

A arquitetura CUDA possui, além do *hardware* de alto desempenho, componentes de *software* que permitem o desenvolvimento de soluções em GPGPU. A figura 3.4 ilustra as camadas de *software* da arquitetura CUDA.

Na camada mais baixa da arquitetura de *software*, existe um módulo de *kernel* do sistema operacional responsável pela inicialização, configuração e comunicação com a GPU. Acima deste módulo do sistema operacional existe o *driver* CUDA em modo usuário, capaz de prover uma interface de programação (API) de baixo nível. Dentro desta API,

Tabela 3.1: Comparação entre algumas placas NVIDIA GeForce [98, 99].

| Modelo | Arquitetura | Clock | Núcleos | GFlops |
|--------------------|-------------|----------|---------|--------|
| GTX Titan Z (dual) | Kepler | 705 MHz | 5760 | 8122 |
| GTX Titan X | Maxwell | 1000 MHz | 3072 | 6144 |
| GTX 980 Ti | Maxwell | 1000 MHz | 2816 | 5632 |
| GTX 690 (dual) | Kepler | 915 MHz | 3072 | 5622 |
| GTX 980 | Maxwell | 1126 MHz | 2048 | 4612 |
| GTX Titan | Kepler | 837 MHz | 2688 | 4500 |
| GTX 780 | Kepler | 863 MHz | 2304 | 3977 |
| GTX 680 | Kepler | 1006 MHz | 1536 | 3090 |
| GTX 590 (dual) | Fermi | 1215 MHz | 1024 | 2488 |
| GTX 670 | Kepler | 915 MHz | 1344 | 2460 |
| GTX 660 Ti | Kepler | 915 MHz | 1344 | 2460 |
| GTX 660 | Kepler | 980 MHz | 960 | 1882 |
| GTX 295 (dual) | GT200 | 1242 MHz | 480 | 1788 |
| GTX 580 | Fermi | 1544 MHz | 512 | 1581 |
| GTX 650 Ti | Kepler | 928 MHz | 768 | 1425 |
| GTX 570 | Fermi | 1464 MHz | 480 | 1405 |
| GTX 560 Ti | Fermi | 1644 MHz | 384 | 1263 |
| GTX 285 | GT200 | 1476 MHz | 240 | 1063 |
| GTX 280 | GT200 | 1296 MHz | 240 | 933 |

Tabela 3.2: Comparação entre modelos da linha TESLA [99].

| Modelo | GPUs | Memória | Arquitetura | Clock | Núcleos | GFlops |
|--------|------|---------|-------------|----------|---------|--------|
| K80 | 2 | 24 GB | Kepler | 875 MHz | 4992 | 8740 |
| K20X | 1 | 6 GB | Kepler | 732 MHz | 2688 | 3950 |
| K20 | 1 | 5 GB | Kepler | 706 MHz | 2496 | 3520 |
| K10 | 2 | 8 GB | Kepler | 745 MHz | 3072 | 4576 |
| S2050 | 4 | 12 GB | Fermi | 1550 MHz | 1792 | 5152 |
| M2090 | 1 | 6 GB | Fermi | 1301 MHz | 512 | 1331 |
| M2070 | 1 | 6 GB | Fermi | 1550 MHz | 448 | 1030 |
| M2050 | 1 | 3 GB | Fermi | 1550 MHz | 448 | 1030 |
| C2075 | 1 | 6 GB | Fermi | 1150 MHz | 448 | 1030 |
| C2070 | 1 | 6 GB | Fermi | 1150 MHz | 448 | 1030 |
| C2050 | 1 | 3 GB | Fermi | 1150 MHz | 448 | 1030 |

existe a definição do conjunto de instruções (ISA) chamada de *Parallel Thread Execution* (PTX). O PTX inclui uma máquina virtual que traduz as instruções PTX em instruções de *hardware*, permitindo que um mesmo código possa ser executado em diversos modelos de GPUs.

Embora as aplicações possam ser desenvolvidas diretamente acima do *driver* CUDA, existem outras camadas superiores que permitem maior facilidade de programação. Dentre estas camadas podemos citar: a plataforma de execução do CUDA, que possui chamadas de mais alto nível e bibliotecas com funções mais específicas, como transformadas de Fourier (FFT) e operações de álgebra linear (CUBLAS); uma camada OpenCL [100], que é um *framework* desenvolvido pelo grupo Khronos [101] para aplicações paralelas em ambientes heterogêneos (GPUs, CPUs, etc); camadas para suporte CUDA em outras

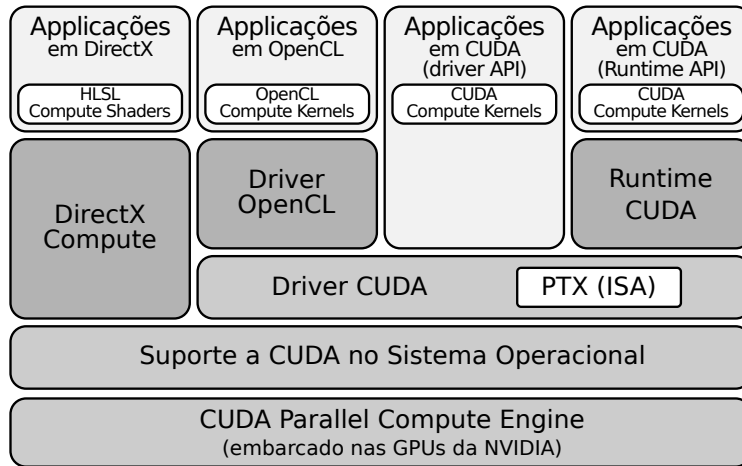


Figura 3.4: Componentes de *software* da arquitetura CUDA [32]

linguagens, tais como Java, Python e .NET. Além disso, OpenGL e DirectX também possuem integração com a arquitetura CUDA.

3.2.4 Programação em CUDA

Um programa desenvolvido para a arquitetura CUDA possui capacidade de executar instruções em unidades de processamento gráfico [32]. O procedimento executado na GPU é chamado de *kernel* e várias instâncias de um *kernel* são executadas em paralelo através de um conjunto de *threads* [74]. Este conjunto de *threads* é organizado em blocos de *threads* e em *grids* de blocos de *threads*. Os blocos e os *grids* podem possuir 1, 2 ou 3 dimensões [32, 74, 102].

Cada *thread* possui um identificador único (`threadIdx`) dentro do seu bloco. Além disso, cada *thread* mantém o seu conjunto de registradores e uma memória local privada.

Cada bloco de *threads* é um conjunto de *threads* que acessa uma mesma memória compartilhada. Cada bloco possui um identificador único dentro do seu *grid*. As *threads* de um bloco podem ser sincronizadas através de uma diretiva barreira (`__syncthreads()`), o que permite acessar a memória compartilhada sem gerar inconsistências.

Um *grid* é uma matriz que executa o mesmo *kernel*. Ao contrário dos blocos, o *grid* só pode ser sincronizado através de uma nova chamada ao *kernel*, não existindo uma diretiva de sincronização específica. Todas as *threads* de um *grid* compartilham a mesma memória global, que deve ser acessada de maneira cuidadosa para não gerar inconsistências.

A Figura 3.5 ilustra cada nível de agrupamento de *threads* e a sua memória relacionada. Além dessas memórias, ainda existem duas memórias somente-leitura: a memória constante, que é uma memória pequena (64 KB¹) e rápida; e a textura, que é uma memória somente-leitura de acesso global que utiliza mecanismos de *cache*. Em especial, a textura permite acelerar o acesso a uma área da memória global utilizando o princípio da localidade espacial, inclusive em matrizes de 2 e 3 dimensões. Não existem mecanismos de coerência de *cache* em texturas, então uma operação de escrita na área de memória

¹Até o momento da escrita desta tese, todas as arquiteturas CUDA (*compute capability* 1.0 a 5.2) possuem memória constante com tamanho de 64 KB.

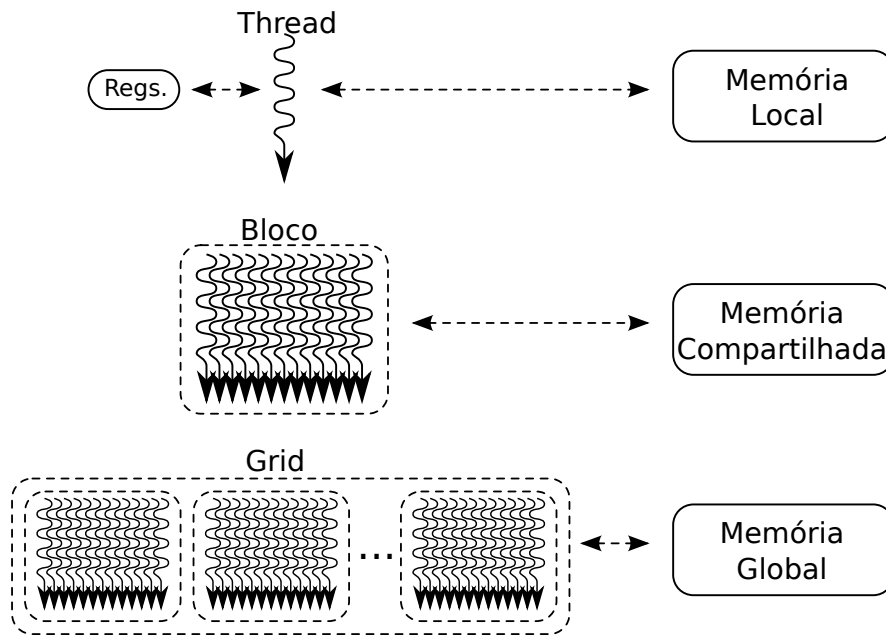


Figura 3.5: Hierarquia de memória da arquitetura CUDA [74].

global mapeada para uma textura pode gerar incoerências no *cache*. O número máximo de elementos em uma textura de uma única dimensão é de 2^{27} .

As *threads* são executadas em grupos chamados *warp*, cada um contendo 32 *threads*. O agrupamento é determinado através do identificador da *thread*, de maneira crescente a partir da *thread* 0, agrupando de 32 em 32. Dentro de um *warp* as *threads* executam paralelamente a mesma instrução por vez (SIMT). Quando um *warp* encontra um *branch* onde as *threads* divergem seus caminhos, o *warp* serializa a execução das instruções. Assim, um grupo de *threads* fica desabilitado até que o outro grupo seja processado. Isso leva a uma perda de desempenho em *branches* divergentes [74, 103]. Para evitar este problema, um cuidado especial deve ser tomado para que, dentro de um mesmo *warp*, todas as *threads* sigam pelo mesmo caminho de execução.

O número de blocos que podem ser executados em paralelo em um mesmo multiprocessador depende do número de registradores que uma *thread* utiliza, assim como a quantidade de memória compartilhada entre as *threads* de um bloco. Os blocos então são enumerados e alocados aos vários multiprocessadores. À medida que um bloco é terminado, um novo bloco é ativado naquele processador. Isso se repete até que todos os blocos tenham sido computados. Observe que a ordem de execução dos blocos não pode ser determinada e, para que não haja inconsistência de dados, eles devem ser capazes de executar independentemente dos demais blocos.

A programação em CUDA é feita utilizando a linguagem C com algumas modificações. A definição de um *kernel* é feita simplesmente com a adição da diretiva `__global__` antes da declaração de uma função. A chamada ao *kernel* deve ser feita informando o número de *threads* T e o número de blocos B através da sintaxe `<<< T, B >>>`. Este par (B, T) é chamado de configuração de execução.

No Programa 3.1, vemos um exemplo de código em CUDA que soma dois vetores A e B , armazenando o resultado no vetor C . O *kernel*, definido com a diretiva `__global__`, é

executado por várias *threads*, sendo que cada *thread* recebe um identificador único acessado através da variável *threadIdx*. Este identificador é utilizado para que cada *thread* acesse um elemento distinto no vetor. A chamada ao *kernel* é feita com N *threads* através da sintaxe `vecAdd<<< 1, N >>>`.

Programa 3.1 Exemplo de código em CUDA [74].

```
// definição do kernel
__global__ void vecAdd(float* A, float* B, float* C)
{
    int i = threadIdx.x;
    C[i] = A[i] + B[i];
}

int main()
{
    // chamada ao kernel
    vecAdd<<<1, N>>>(A, B, C);
}
```

Capítulo 4

Comparação Paralela de Sequências Biológicas

A comparação de sequências biológicas é uma tarefa que consome muitos recursos computacionais. Dependendo do tamanho das sequências ou do número de comparações realizadas, o processamento pode levar vários dias. Esse fato é ainda agravado pois os bancos de sequências públicas crescem exponencialmente. O GenBank, por exemplo, é um banco de sequências que duplica de tamanho a cada 30 meses [104]. Nos algoritmos de comparação de sequências vistos no Capítulo 2, a maior parte do tempo de execução é gasta calculando a matriz de programação dinâmica e esta é a parte dos algoritmos que usualmente é paralelizada.

Este capítulo visa apresentar as estratégias de paralelização (Seção 4.1) e algumas soluções que comparam sequências biológicas de maneira paralela em CPU (Seção 4.2), GPU (Seção 4.3) e outras plataformas (Seção 4.4). Por fim, apresentaremos um quadro comparativo com as soluções abordadas neste capítulo (Seção 4.5). Um artigo estendendo esta revisão bibliográfica foi produzido e submetido ao periódico ACM Computing Surveys [43] e encontra-se em processo de revisão.

4.1 Estratégias de Paralelização

As estratégias de paralelização dos algoritmos de comparação de sequências biológicas são divididas em dois grupos principais: *coarse-grained* e *fine-grained*.

4.1.1 Paralelismo *Coarse-grained*

No paralelismo do tipo *coarse-grained*, cada *thread* recebe uma sequência de busca e um banco de sequências. As *threads* calculam várias matrizes de programação dinâmica (DP) em paralelo, cada uma representando uma comparação entre a sequência de busca e uma das sequências do banco. Visto que cada matriz DP é calculada independentemente, não há necessidade de comunicação entre as *threads*, sendo necessário um mecanismo de balanceamento de carga para distribuição das tarefas entre os vários nós de processamento. Em geral, o banco de sequências é ordenado de acordo com o tamanho das sequências, de forma a uniformizar o tamanho das sequências distribuídas para cada nó.

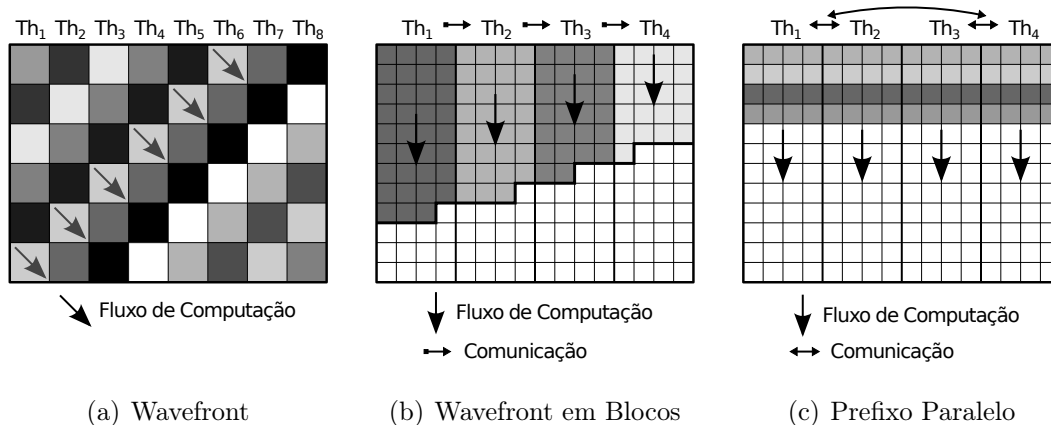


Figura 4.1: Paralelismo *Fine-grained*

4.1.2 Paralelismo *Fine-grained*

Na abordagem *fine-grained*, a comparação entre duas sequências é feita por várias *threads*, sendo que uma única matriz de programação dinâmica é calculada de maneira paralela. Podemos citar três formas de paralelismos *fine-grained*: por *wavefront*, por blocos e por prefixo paralelo.

Processamento em Wavefront

Uma das estratégias tradicionalmente utilizadas para paralelizar o cálculo da matriz de programação dinâmica é o método *wavefront* [105], que processa as células em ondas, uma anti-diagonal por vez.

O padrão de acesso observado nos algoritmos de comparação de sequências biológicas é em geral o seguinte: para calcular a célula $H[i, j]$ é necessário acessar as células $H[i - 1, j]$, $H[i - 1, j - 1]$ e $H[i, j - 1]$. Desta forma, as células em uma mesma anti-diagonal não possuem dependências entre si, permitindo que todas elas sejam processadas em qualquer ordem, ou até mesmo em paralelo.

A Figura 4.1(a) ilustra o método *wavefront*. No começo da computação, apenas a célula superior esquerda pode ser processada, não sendo possível paralelismo neste momento. Em seguida, duas células na diagonal seguinte podem ser processadas em paralelo. O máximo paralelismo é obtido quando a diagonal possui tamanho máximo. O paralelismo decresce nas últimas diagonais, até chegar na última célula no canto inferior direito.

Wavefront baseado em Blocos

A Figura 4.1(b) ilustra uma estratégia semelhante ao *wavefront*. Em vez de cada *thread* processar uma célula da diagonal, a matriz é dividida em diagonais de blocos. Desta forma, cada *thread* processa um dos blocos da diagonal em paralelo. No início do processamento, apenas a primeira *thread* consegue processar os blocos. Assim que as dependências da segunda *thread* forem calculadas, a primeira *thread* envia as células calculadas para a segunda *thread*, permitindo que as duas primeiras *threads* processem blocos em paralelo. Esse mecanismo é propagado para todas as *threads* até que seja possível um paralelismo

máximo com todas as *threads* ativas. Semelhantemente ao *wavefront* original, o paralelismo é perdido no início e no final da computação.

Prefixo Paralelo

A técnica de prefixo paralelo pode ser utilizada para quebrar as dependências entre *threads* vizinhas, permitindo que todas as *threads* iniciem simultaneamente o cálculo da mesma linha (ou coluna) da matriz [23, 106]. Neste método, a matriz é dividida em t partes, onde t é o número de *threads*. A cada iteração, cada *thread* calcula suas células em paralelo utilizando uma equação de recorrência modificada, que permite o cálculo de prefixos máximos locais. No final de cada iteração, $O(\log(t))$ passos de comunicação são realizados para calcular os prefixos máximos de toda a linha. A Figura 4.1(c) ilustra este método.

4.1.3 Métrica de desempenho (CUPS)

A área de computação de alto desempenho vem propondo técnicas específicas para aumentar o desempenho das ferramentas de Bioinformática e de algoritmos de comparação de sequências. A principal métrica para medir o desempenho destas ferramentas é o número de células atualizadas por segundo (CUPS – *Cells updated per second*), calculadas pela Equação 4.1,

$$CUPS = \frac{m \cdot n}{t} \quad (4.1)$$

onde m e n são os tamanhos das sequências e t o tempo total de processamento. Utiliza-se também a métrica derivada em bilhões de células atualizadas por segundo (GCUPS – *Gigacells updated per second*) ou trilhões de células atualizadas por segundo (TCUPS – *Teracells updated per second*).

4.2 Soluções em CPU (SIMD)

4.2.1 SWMMX (Rognes e Seeberg)

O artigo de Rognes e Seeberg [107] apresenta uma implementação do Smith-Waterman para comparação de aminoácidos com *affine gap* (Seção 2.2.3) utilizando o conjunto de instruções vetoriais MMX/SSE da Intel. O padrão de processamento está ilustrado na Figura 4.2(a). O paralelismo é obtido no sentido da sequência S_0 (vertical), e não no sentido do *wavefront* (diagonal). Para isso, empregou-se a otimização SWAT [108], que aproveita o fato de a matriz F do *affine gap* conter normalmente valores próximos de zero. Com isso, a matriz principal H acessa a matriz F apenas quando um dos elementos de H for maior que a penalidade de um *gap* (G_{first}). O cálculo de H é feito em paralelo e, apenas quando necessário, a matriz F é calculada, economizando muito tempo de processamento. Entretanto, se o parâmetro G_{first} for muito baixo, a otimização não é eficaz.

Uma inovação da proposta de Rognes e Seeberg é a precomputação do acesso à matriz de substituição $sbt(S_0[i], x)$ (Seção 2.1.10) em uma matriz $P_{x,i}$, onde $x \in \Sigma$ e $i \in [1..|S_0|]$. Esta técnica é vantajosa pois evita-se o acesso de $sbt(S_0[i], x)$ no *loop* interno do algoritmo e o número de elementos $x \in \Sigma$ é pequeno suficiente para não sobrecarregar a memória.

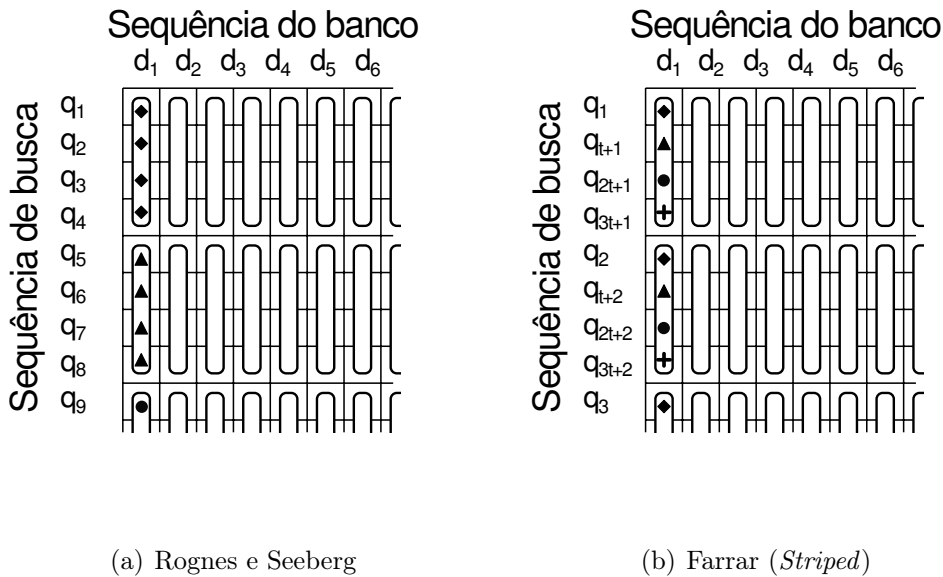


Figura 4.2: Soluções com instruções vetoriais em CPU [107, 110]

A matriz P é chamada de perfil de escores da sequência S_0 (*query-profile*). Após a criação desta matriz, a fase de programação dinâmica é realizada paralelizando o cálculo com instruções SIMD MMX e SSE. Cada instrução MMX/SSE realiza operações sobre 8 elementos da matriz de uma só vez. Os valores são restritos a um intervalo de 8 bits (0 a 255). O MMX/SSE também suporta operações com saturação, diminuindo o número de comparações.

Nos resultados apresentados, a comparação foi realizada entre o banco *Swiss-Prot* e 11 sequências selecionadas (as mesmas usadas pelo artigo do BLAST 2 [109]), com tamanho entre 189 a 567 aminoácidos. A penalidade dos *gaps* foi $G_{first} = -11$ e $G_{ext} = -1$.

O máximo *speedup* obtido em relação às implementações de Smith-Waterman OSEARCH e SSEARCH foram 13x e 6x, respectivamente. Comparando o SWMMX com os algoritmos heurísticos, o Fasta e algumas versões do BLAST foram até 3x mais rápidos do que o SWMMX e o BLAST 2.0.10 foi até 7,5x mais rápido do que o SWMMX.

4.2.2 StripSW (Farrar)

O artigo de Farrar [110] apresenta uma implementação do algoritmo Smith-Waterman com *affine gap* (Seção 2.2.3) utilizando o conjunto de instruções SSE2, aplicando o paralelismo no sentido vertical da matriz. Assim como no SWMMX (Seção 4.2.1), o algoritmo precomputa o *query-profile* da sequência S , armazenando o cálculo de $sbt(S_0[i], x)$ na matriz $P_{x,i}$, onde sbt é a matriz de substituição.

A diferença entre esta implementação e o SWMMX está na dependência de dados durante o processamento das matrizes F e H . O padrão utilizado para o processamento com menor dependência foi chamado de padrão *striped*. Este padrão foi criado para que

a verificação sobre a matriz F seja feita no *loop* externo, e não no *loop* interno como era no SWMMX.

O padrão *striped* está apresentado na Figura 4.2(b). A primeira instrução vetorial processa as células indicadas nas linhas (q_1, q_2, q_3, q_4) ; a segunda instrução, (q_5, q_6, q_7, q_8) ; a i -ésima instrução, $(q_{i \times t + 1}, q_{i \times t + 2}, q_{i \times t + 3}, q_{i \times t + 4})$, sendo $t = 4$. O padrão *striped* ocorre pois as linhas estão intercaladas, ao contrário da solução de Rognes (Figura 4.2(a)).

O teste efetuado foi semelhante ao do SWMMX e ao do BLAST 2, com 11 sequências variando de 143 a 567 aminoácidos. Duas matrizes de substituição foram utilizadas (BLOSUM62 e BLOSUM50) e quatro valores de penalidades de *gap* foram testados $(-10 - k, -10 - 2k, -14 - 2k$ e $-40 - 2k)$.

Percebe-se que quanto maior a penalidade dos *gaps*, menos a matriz F é processada, aumentando a velocidade de execução. Os testes chegaram a ser mais rápidos que o FASTA, mas em média 30% mais lento que o BLAST. Para obter o pico de desempenho de 3 GCUPS, utilizou-se uma penalidade de *gap* bastante elevada: $-40 - 2k$, sendo k o tamanho da sequência de *gaps*.

4.2.3 SWIPE (Rognes)

O artigo de Rognes [27] propôs uma versão otimizada do SW com paralelismo *coarse-grained* e com múltiplas *threads*, utilizando instruções SSE para comparar uma sequência de busca com um banco de sequências. Basicamente, cada operação SIMD computa elementos de comparações diferentes. Cada nó de processamento utiliza um método de auto escalonamento, onde uma tarefa é definida como um conjunto de x sequências do banco. Nos testes, 32 sequências de buscas (24 a 5.478 aminoácidos) foram comparadas com o banco Uniprot 11.0 (4,6 milhões de sequências). Os testes foram executados em dois Intel Xeon com seis núcleos, obtendo um máximo de desempenho de 106 GCUPS com sequências de 375 aminoácidos.

4.2.4 Parallel-LTDP (Maleki)

O artigo de Maleki [111] utiliza conceitos de álgebra linear para transformar problemas de programação dinâmica, incluindo alinhamentos locais e globais com *affine gap*, em conjuntos independentes de multiplicações de matrizes e vetores (MVM). A técnica utiliza os conceitos de *rank convergence* para otimizar a obtenção do alinhamento, em algoritmos do tipo *linear-tropical dynamic programming* (LTDP). O algoritmo proposto (parallel-LTDP) calcula múltiplos estágios de maneira paralela, ignorando as dependência entre os estágios. Posteriormente, a corretude do resultado é garantida por meio de operações adicionais que se baseiam nas propriedades de *rank convergence*. Dependendo do problema de programação dinâmica abordado, as operações adicionais podem gerar um alto custo de reprocessamento. Para o algoritmo de SW, o tempo adicional é bastante pequeno, mas para o NW este tempo é bem maior. Os autores mostraram que para o problema de *Longest Common Subsequence* (LCS) nem sempre consegue convergir para o resultado correto.

Adicionalmente, no algoritmo de SW, os estágios intermediários utilizaram a implementação de Farrar (Seção 4.2.2). Os resultados foram coletados em 8 nós ($2 \times$ Intel Xeon 2,7 GHz com 8 núcleos cada) do cluster Stampede e a comunicação entre os nós foi

feita por MPI. Como sequência de busca, foram utilizadas sequências do tipo *Expressed Sequence Tags* (EST), que possuem no máximo 800 nucleotídeos [112]. A abordagem proposta obteve em torno de 900 GCUPS para o SW, utilizando 128 núcleos.

4.3 Soluções em GPU

4.3.1 DASW (Y. Liu et al.)

No artigo de Liu e co-autores [88], é apresentada uma implementação do algoritmo Smith-Waterman com *affine-gap* duplo em GPU. Nesse caso, além da penalidade já atribuída pelo *affine gap* (Seção 2.2.3), existe uma penalidade extra para os primeiros gaps. Visto que valores de 16-bit são utilizados para armazenar os elementos da matriz de programação dinâmica, o maior escore possível é $2^{16} = 65536$. Além disso, o tamanho da sequência de busca é limitado a 4 MBP. As sequências consultadas são concatenadas colocando um caractere especial entre elas. Durante a computação da matriz, as variáveis são reiniciadas cada vez que um caractere especial for encontrado. Esta otimização reduz o *overhead* de reinicialização do *wavefront*.

Dois modos de operação foram testados: *Alignment Score Mode* (ASM) e *Alignment Traceback Mode* (ATM). No modo ASM, a saída é somente o escore encontrado e no modo ATM o alinhamento completo também é retornado. Nesse último modo, a memória em textura é utilizada para armazenadas a matriz completa, o que limita a matriz em 2^{22} células caso a GPU possua memória de 256 MB. Os testes foram executados em uma GeForce 7800 GTX, comparando uma sequência com 16.384 aminoácidos contra um banco de 983 sequências. Os resultados foram de 0,24 GCUPS e 0,18 GCUPS nos modos ASM e ATM, respectivamente.

4.3.2 GPU-SW (W. Liu et al.)

O artigo de W. Liu e co-autores [113] implementa dois algoritmos em GPU: Smith-Waterman com *affine gap* (Seção 2.2.3) e o ClustalW [114] para alinhamentos múltiplos de sequências (MSA).

O algoritmo de Smith-Waterman implementado retorna apenas os escores ótimos entre uma sequência e um banco genômico, deixando o alinhamento final para a CPU. O percurso da matriz é feito paralelamente com o método *wavefront* (Seção 4.1.2). Para aumentar o paralelismo, várias sequências do banco são alinhadas simultaneamente com a sequência de busca. As diagonais de todas as matrizes são armazenadas lado a lado em uma textura de duas dimensões. Se as diagonais possuírem tamanhos diferentes, a textura terá áreas inúteis (ou supérfluas), desperdiçando tempo de processamento. Para atenuar este problema, as sequências são dispostas em ordem crescente de comprimento. Em seguida, lotes são criados com sequências de tamanhos similares, diminuindo a área supérflua da textura. Na GPU utilizada para os testes, a textura tem um tamanho máximo de 4096×4096 *pixels*, o que limita o tamanho máximo da sequência em 4.096 caracteres e o tamanho de um lote em 4.096 sequências. A linguagem usada para programação da GPU foi a OpenGL Shading Language (GLSL [115, 116]).

No alinhamento múltiplo de sequências (MSA), todas as sequências da entrada são alinhadas duas a duas com o algoritmo de Smith-Waterman e, em seguida, executa-se o algoritmo ClustalW [114] para obter o alinhamento múltiplo.

Outro artigo de W. Liu e co-autores [117] apresenta uma fórmula para prever o tempo de execução da sua implementação de Smith-Waterman em GPU. Os fatores que determinam a fórmula são classificados em *data-constant*, *data-linear* e *computation-dependent*.

Os resultados do Smith-Waterman em GPU mostraram um *speedup* de 16x em relação ao OSEARCH e 10x em relação ao SSEARCH. Já os resultados do alinhamento múltiplo mostraram *speedup* de 7x em relação ao ClustalW.

4.3.3 SWCuda (Manavsky e Valle)

O artigo de Manavsky e Valle [28] apresenta uma implementação do algoritmo Smith-Waterman em GPU, utilizando o ambiente CUDA da NVIDIA (Seção 3.2). O algoritmo utiliza *affine gap* (Seção 2.2.3) para parametrizar as penalidades de *gap*. A abordagem empregada foi distribuir para cada *thread* uma sequência do banco. Desta forma, cada *thread* é responsável por um único alinhamento e cada alinhamento é feito sequencialmente no interior da *thread*. O percurso da matriz é feito coluna a coluna. Dois *buffers* são alternados durante o processamento: um para receber os valores da coluna atual; outro para armazenar os valores da coluna anterior.

O artigo otimiza o acesso à memória utilizando a capacidade de a GPU ler palavras de 128 bits da memória. Assim, 4 elementos (16 bits) de H e de E são lidos em uma única instrução. Após calculados os novos valores, os 4 elementos de cada um dos vetores são escritos no *buffer*, também com uma única instrução. Já o *query-profile* (Seção 4.2.1) é armazenado em textura, sendo que cada elemento possui 8 bits. Desta forma, 4 elementos são lidos em um único inteiro de 32 bits.

O banco de sequências é ordenado pelo tamanho das sequências, para que os blocos de *threads* executem em tempo semelhante. A configuração de execução ótima apresentada pelo artigo foi 64 *threads* em cada um dos 450 blocos, totalizando 28.800 *threads*. O banco de sequências é armazenado na memória global da GPU. Também é realizada a distribuição das sequências do banco para diversas GPUs, considerando o poder computacional de cada uma. O processamento é iniciado com pesos pré-definidos e, após o fim de cada alinhamento, refazem-se os pesos, particionando o banco de acordo com o desempenho anteriormente obtido.

Utilizando uma placa dual GeForce 8800 GTX, a implementação obteve desempenho de até 1,75 GCUPS e 3,42 GCUPS em uma e duas GPUs, respectivamente. O tamanho das sequências de busca variou de 63 a 567 aminoácidos.

4.3.4 LR-SW (L. Ligowski et al.)

O artigo de Ligowski e co-autores [91] propõe um método para comparar uma sequência de busca com um banco de sequências, retornando o escore ótimo obtido por cada alinhamento. Para cada alinhamento, a matriz de programação dinâmica é dividida em 12 linhas. Cada etapa calcula 12 células com apenas 2 acessos à memória global, de forma a acelerar o desempenho do processamento. Cada *thread* processa um par de sequências

distinto. Entretanto, a implementação limita o tamanho da sequência de busca em 1000 caracteres.

Os testes experimentais foram executados na placa dual 9800 GX2. Utilizando o banco *Swiss-Prot*, a comparação executou em até 7,5 GCUPS e 14,5 GCUPS para 1 e 2 GPUs, respectivamente. Comparando os resultados obtidos com o artigo [28] (Seção 4.3.3), foi alegada uma melhoria no desempenho de mais de 3x.

4.3.5 CUDA-SSCA (A. Khajeh-Saeed et al.)

O artigo de A. Khajeh-Saeed e co-autores [118] implementa em GPU cinco *kernels* relacionados ao conjunto de *benchmarks* chamado *Scalable Synthetic Compact Applications* (SSCA) [119]. O *kernel 1* executa o algoritmo de Smith-Waterman para comparar duas sequências e obtém os 200 maiores escores e as suas localizações na matriz de programação dinâmica. O *kernel 2* executa a fase de *traceback* do Smith-Waterman (Seção 2.2.2) para obter o alinhamento completo de cada um dos resultados encontrados no *kernel 1*. O *kernel 3* busca os 100 maiores escores entre as subsequências encontradas no *kernel 2* e a sequência original. O *kernel 4* executa um alinhamento global (Seção 2.2.1) entre todos os resultados obtidos no *kernel 3*. O *kernel 5* executa um alinhamento múltiplo de sequências baseado nos resultados obtidos no *kernel 4*. Em vez de calcular a matriz com o método de *wavefront* (Seção 4.1.2), o paralelismo foi obtido ao decompor a equação de recorrência de tal forma que as *threads* processem paralelamente a mesma linha utilizando prefixo paralelo (Seção 4.1.2). Para utilizar múltiplas GPUs, a matriz de programação dinâmica foi dividida em vários blocos sobrepostos, o que somente foi possível devido ao tamanho restrito das sequências utilizadas nos testes.

Os melhores resultados foram obtidos no Kernel 1 em uma GPU, com speedup de até 45x comparado com a implementação em CPU, resultado em um desempenho de 0,7 GCUPS em uma placa GTX295 e de 2,6 GCUPS com 4 placas GTX295. Os benchmarks utilizaram sequências com até 1024 bases e o alinhamento ótimo foi obtido com sequências com menos de 64 bases.

4.3.6 CudaSW++ (Y. Liu et al.)

O artigo de Y. Liu e co-autores [90] apresenta o CudaSW++ 1.0, uma proposta que executa o Smith-Waterman com *affine gap* em GPUs. Esta proposta divide o problema em várias tarefas, sendo que cada tarefa compara a sequência de busca com uma das sequências do banco. Sendo assim, foram propostos dois métodos para paralelizar o problema: paralelização inter-tarefas e paralelização intra-tarefas.

Na abordagem inter-tarefas, cada tarefa é atribuída e processada por uma única *thread*. Executam-se blocos com $dimBlock = 256 threads$, sendo que a quantidade de blocos executada em paralelo é igual ao número de multiprocessadores da GPU. Já na abordagem intra-tarefas, cada tarefa é atribuída e processada por um bloco de *threads*, sendo que todas as $dimBlock = 256 threads$ deste bloco cooperam para processar a mesma tarefa (*fine-grained*). O paralelismo é obtido por meio da técnica de *wavefront* (Seção 4.1.2), em que as células das diagonais são processadas em paralelo.

A abordagem inter-tarefas é mais rápida que a intra-tarefas, mas a intra-tarefas permite comparar sequências maiores. Sendo assim, a execução da comparação é dividida

em duas fases. A primeira fase utiliza o método inter-tarefas para comparar sequências cujo tamanho é menor que uma constante. Na segunda fase, o método intra-tarefas é executado, comparando as sequências restantes.

Para comparação da maioria das sequências, faz-se necessário o uso da memória global para salvar os resultados intermediários de uma *thread*. Para reduzir o tempo de acesso à memória global, as *threads* acessam a memória em um padrão ordenado (*coalesced*).

O CudaSW++ 1.0 foi testado na GPU GTX 280 e na GPU GTX 295 (dual). Ambas as placas foram conectadas a um AMD Opteron 248 com 2,2 GHz rodando o sistema operacional Linux. Foram utilizadas para a análise 25 sequências de busca com tamanhos entre 144 e 5.478 aminoácidos. O banco de dados escolhido foi o *Swiss-Prot release 56.6*. A placa GTX 280 conseguiu um desempenho entre 9,04 e 9,66 GCUPS e a placa GTX 295 conseguiu de 10,66 a 16,09 GCUPS. Utilizando a GTX 280, o Cudasw++ foi 6,27 vezes mais rápido que o NCBI-BLAST e, utilizando a GTX 295, o desempenho foi 9,55 vezes mais rápido (empregou-se a matriz BLOSUM50).

O artigo de Doug Hains e co-autores [120] propôs o CudaSW++1.0i, estendendo o CudaSW++1.0 por meio de otimizações no *kernel* relacionado ao processamento intra-tarefas. As melhorias efetuadas no processamento intra-tarefas foram a diminuição do número de acessos à memória global, estudo dos parâmetros utilizados na execução, a inserção da otimização *query profile* e de *loop unrolling*. Utilizando uma sequência de busca com 576 aminoácidos e o banco de proteínas *Swiss-Prot*, o ganho da otimização apresentado foi de até 39,3% na placa Tesla C2050 e de 67,0% na Tesla C1060. Desta forma, alterou-se a constante que seleciona o tamanho das sequências que serão calculadas pelo método intra-tarefas de 3.072 para 1.500. Em seguida, testou-se a implementação com sequências com tamanho variando de 144 a 5478 e utilizando vários outros bancos de proteínas. O GCUPS máximo obtido foi de 29,29 GCUPS com a Tesla C2050.

Na versão 2.0 do CudaSW++ [29], duas implementações do Smith-Waterman com *af-fine gap* foram propostas. A primeira é uma otimização da versão inter-tarefas que, nesta nova versão, utiliza duas otimizações: *sequential query profile* e *packed data format*. A segunda implementação é uma variante do padrão *striped* proposto por Farrar (Seção 4.2.2). O CudaSW++ 2.0 foi testado com sequências de busca com até 5.478 aminoácidos. O CudaSW++2.0 obteve, com a otimização da versão inter-tarefas, um desempenho de 16,9 GCUPS na GTX 280 e de 28,8 GCUPS na GTX 295 (dual-GPU). Já com a otimização de Farrar, o desempenho foi de 17,8 GCUPS na GTX 280 e de 29,7 GCUPS na GTX 295, resultados obtidos com a penalidade de *gap* igual a $-40 - 3k$. O ganho de desempenho comparado com o CudaSW++1.0 foi de até 1,77x.

A versão 3.0 do CudaSW++ [121] aumentou o desempenho da comparação por meio de balanceamento de carga entre CPU e GPU. Para o código em CPU, foram utilizadas instruções de 8-bit baseadas em SSE (SIMD) para acelerar o processamento, de maneira similar ao do algoritmo de Rognes (Seção 4.2.1). Para o código em GPU, foram utilizadas instruções SIMD de vídeo existentes no conjunto de instruções PTX (Seção 3.2.3). Essas instruções de vídeo, disponíveis a partir da arquitetura Kepler, operam simultaneamente sobre $\frac{1}{4}$ de palavras, sendo cada $\frac{1}{4}$ de palavra possui 8-bit e representa um alinhamento independente. Visto que as operações de 8-bit podem gerar *overflow*, um passo extra precisa ser executado para as comparações cujos escores estão próximos do valor máximo.

O CudaSW++ 3.0 limita o tamanho máximo da sequência de busca em 3.072 para a GPU, deixando as sequências maiores para serem processadas em CPU. Ainda assim, o

artigo informa que esse valor pode ser ajustado em tempo de compilação para até 65.536, visto que a arquitetura Kepler permite até 512 KB de memória local.

Os resultados experimentais utilizaram 20 sequências de busca com tamanhos entre 144 e 5.478. O banco de proteínas utilizado foi o *Swiss-Prot* com 538.585 sequências, sendo que a maior sequência possui tamanho de 35.213 aminoácidos. Também foi utilizado um banco simulado de proteínas com 200.000 sequências de 3.000 aminoácidos. Os testes foram executados em uma CPU quad-core Intel i7 e individualmente com as GPUs GTX680 e GTX690 (dual). Com o banco *Swiss-Prot*, o CudaSW++ 3.0 obteve um pico de 119,0 GCUPS na GTX680 e de 185,6 GCUPS na GTX690, resultando em um *speed* de até 3,2x comparando os resultados do CudaSW++ 2.0 nas mesmas placas. Utilizando o banco simulado, cujas sequências possuem todas o mesmo tamanho, o desempenho médio foi de 118,0 GCUPS na GTX 680 e de 196,2 GCUPS na GTX 690, resultando em um *speed* de até 2,1x comparando os resultados do CudaSW++ 2.0 nas mesmas placas.

Os testes foram novamente executados somente em GPU, desabilitando o processamento em CPU e o passo extra para reprocessamento das sequências com *overflow*. Visto que o tamanho máximo das sequências em GPU foi limitado, removeu-se as sequências com mais de 3.072 aminoácidos do banco de proteína, resultando em um banco 1,59% menor. O CudaSW++ 3.0 obteve até 83,3 GCUPS utilizando a GTX680, resultando em um *speed* de até 1,6x comparando os resultados do CudaSW++ 2.0. Concluiu-se que, em média, a proporção do poder de computação entre CPU e GPU foi de aproximadamente 1:2.

4.3.7 CUDAlign 1.0 (Sandes et al.)

O CUDAlign 1.0 [122] é a ferramenta proposta na dissertação de mestrado [33] do autor desta tese, de forma a comparar sequências longas de DNA com o algoritmo Smith-Waterman utilizando *affine gap* (Seção 2.2.3) e memória linear. Ao contrário da maioria das implementações, o CUDAlign 1.0 compara sequências longas de DNA (maiores que 10 MBP). Notamos que os outros métodos, de forma geral, possuem foco no problema de comparar uma sequência pequena (menores que 100 KBP) de aminoácidos com inúmeras sequências em um banco de proteínas.

No CUDAlign 1.0, as células da matriz de programação dinâmica são agrupadas em blocos, formando um *grid* G com B colunas de blocos, que são processados em *wavefront*. Cada bloco possui T *threads*, sendo que cada *thread* é responsável por processar α linhas. As *threads* processam as diagonais internas ao bloco também em *wavefront*. Quando a primeira *thread* (T_0) alcança a última coluna do bloco, todas as outras *threads* param de executar, deixando algumas células pendentes. Estas células são delegadas para o próximo bloco, permitindo que a primeira diagonal interna do próximo bloco inicie seu processamento já com paralelismo máximo, ou seja, com todas as T *threads* processando simultaneamente.

Sequências reais de até 47 MBP foram utilizadas em [122] para os resultados experimentais. O maior par de sequências comparado foi o cromossomo 21 do homem com o cromossomo 22 do chimpanzé, o qual levou 21 horas em uma GTX 280, resultando em 20,4 GCUPS de desempenho. O Capítulo 5 descreverá o CUDAlign 1.0 com maiores detalhes.

4.3.8 FGCS (Ino et al.)

A solução de Ino e co-autores [123] utiliza um grid composto por múltiplas GPUs ociosas para executar o algoritmo de Gotoh (Seção 2.2.3). Quando a proteção de tela é ativada, várias tarefas independentes são executadas. Os autores propuseram uma arquitetura mestre-escravo utilizando uma política de auto-escalonamento. O trabalho [124] estende o método original de forma a permitir o uso da GPU em pequenos intervalos de ociosidade, mesmo antes da proteção de tela ser ativada. Os testes utilizaram pequenas sequências de busca de até 511 bases, comparadas em 8 GPUs distribuídas. O desempenho máximo obtido foi de 64 GCUPS.

4.3.9 SW# (Korpar e Sikic)

O artigo de Korpar e Sikic [125] implementa o algoritmo de Myers-Miller (Seção 2.2.5) com as estratégias de paralelização e otimização de *Block Pruning* propostas na presente tese (Capítulo 8) e em [35]. A ferramenta SW# é capaz de, logo no primeiro estágio, utilizar até duas GPUs para acelerar a computação da matriz. A solução é capaz de obter o alinhamento completo utilizando a estratégia de dividir para conquistar de Myers-Miller. Sequências de até 47 MBP foram alinhadas nos experimentos, obtendo um desempenho de 65,2 GCUPS em uma GPU dual GTX690.

4.3.10 GSWABE (Liu e Schmidt)

O artigo de Liu e Schmidt [50] propõe uma estratégia chamada GSWABE que retorna alinhamentos ótimos com *affine gap* para pequenas sequências de DNA. Considerando a nomenclatura proposta na Definição 2.1.20, esta implementação é capaz de produzir três tipos de alinhamentos semi-globais (1/1, 2/2 e 3/3), além do alinhamento local (*/*) e global (+/+). No GSWABE, a matriz de programação dinâmica é dividida em pequenos blocos de 4×4 e cada *thread* executa uma comparação diferente. Visto que as sequências comparadas são menores que 500 bases, os alinhamentos são retornados utilizando espaço quadrático. Um desempenho máximo de 58,3 GCUPS foi obtido em uma Tesla K40.

4.4 Soluções em outras plataformas

4.4.1 CBESW - Cell BE (A. Wirawan)

No artigo de A. Wirawan [126], o Smith-Waterman com *affine gap* foi implementado em um processador Cell BE do Playstation 3. A otimização de Farrar (Seção 4.2.2) foi implementada, precomputando o cálculo de $sbt(S_0[i], x)$ num vetor $P_{x,i}$, onde sbt é a matriz de substituição, e realizando os cálculos de *gap* somente quando necessários. Desta forma, o paralelismo é obtido no sentido das colunas, e não na diagonal. Além disso, a execução paralela ocorre dentro dos *Synergistic Processor Elements* (SPEs) do Cell BE. Cada coluna é dividida em segmentos e cada segmento é processado paralelamente. O Cell BE não possui operações aritméticas saturadas e a operação de subtração saturada foi implementada em software.

Cada uma das k SPEs alinha uma sequência de busca contra $|D|/k$ sequências do banco D . Apenas os b maiores escores são retornados ao processo mestre, que executa

no *Power Processor Element* (PPE). O processo mestre, por sua vez, é responsável por selecionar os b maiores escores de todos os SPEs.

Ao final do artigo, uma comparação com outras implementações é realizada. A comparação é feita utilizando 18 sequências avaliadas pelas outras implementações, cujo tamanho da sequência de busca varia de 63 a 852 aminoácidos. O CBESW atingiu pico de 3,64 GCUPS na comparação da sequência de tamanho 852. Comparando com as implementações SSEARCH e StripSW (Seção 4.2.2) executadas em um Intel Core 2 Duo 2,4 GHz, foram obtidos *speedups* de 30,1x e 1,64x respectivamente. Comparando com o SWCuda (Seção 4.3.3), executado em uma GeForce 8800GTX, foi obtido um *speedup* de 3x.

4.4.2 PP-Cell - Cell BE (A. Sarje)

No artigo [127], uma abordagem paralela foi proposta utilizando técnicas de redução de memória (Seção 2.2.5) e de prefixo paralelo (Seção 4.1.2) para obter os alinhamentos locais e globais ótimos. A proposta utiliza mecanismo de *double-buffer* e otimizações na comunicação e na computação para acelerar o desempenho do algoritmo no Cell BE. Os autores implementaram dois tipos diferentes de algoritmos (*spliced* e *syntenic*). O melhor desempenho obtido em um IBM S20 Cell BE com 16 SPEs foi de 0,8 GCUPS com a sequência com 2048 caracteres, utilizando o alinhamento do tipo *syntenic*.

4.4.3 NoC-SW - Asic (Sarkar)

No artigo de Sarkar [30], uma solução em ASIC (*Application Specific Integrated Circuit*) foi proposta com vários elemento de processamento (PEs) integrados por meio de uma rede de comunicação embarcada (Network-on-Chip - NoC). Esta solução é capaz de produzir alinhamentos locais, semi-globais e globais de sequências de DNA ou proteínas, utilizando o modelo de *affine-gap*. Os autores desenharam um array sistólico capaz de utilizar o paralelismo em *wavefront* ou em prefixo paralelo (Seção 4.1.2). Cada PE foi implementado utilizando RTL (Register-Transfer Level) e sintetizado com o padrão de 90nm. Nos resultados experimentais, sequências de até 1024 nucleotídeos foram alinhados utilizando as técnicas de wavefront e prefixo paralelo. O pico de desempenho foi obtido quando sequências de tamanho 1KBP \times 1KBP foram comparadas com 16 PEs utilizando a técnica de prefixo paralelo. Neste caso, o desempenho de 243,8 GCUPS foi obtido.

4.4.4 SW-Rivyera - FPGA (L. Wienbrandt)

Em [128], a plataforma multi-FPGA Rivyera S3-5000 é utilizada para acelerar a execução de SW com *gap* linear. Esta plataforma contém 128 FPGAs (*Field Programmable Gate Array*) conectadas por um barramento de comunicação de alto desempenho. Os autores propuseram um projeto de array sistólico [129] para comparar sequências pequenas de DNA ou proteínas. Quatro comparações distintas são executadas simultaneamente em cada FPGA, atingindo um pico de 3.040 GCUPS de desempenho.

4.4.5 XSW (Wang et al.)

O artigo [130] propõe uma estratégia para executar o algoritmo de Gotoh (Seção 2.2.3) na arquitetura Intel Phi, produzindo o escore ótimo como saída. Um subconjunto de sequências de um banco é distribuído para cada thread, que por sua vez processa as sequências utilizando instruções vetoriais baseadas o algoritmo de Rognes (Seção 4.2.3). Os resultados foram coletados em um Intel Xeon Phi 7110 (61 núcleos) com 244 threads e sequências com até 1.000 aminoácidos foram utilizadas como sequência de busca, sendo capaz de obter um pico de 70 GCUPS.

4.4.6 SWAPHI-LS (Liu et al.)

O artigo [131] propõe uma estratégia para executar o algoritmo de Gotoh (Seção 2.2.3) para sequências longas em mais de um Intel Phi. Como saída, apenas o escore ótimo é fornecido. A matriz de programação dinâmica é dividida em blocos que são processados em *wavefront*. Cada bloco é dividido em pequenas áreas que são processadas vetorialmente por cada *thread*. Um grupo de 4 Intel Phis foi utilizado, conectados por uma infraestrutura em MPI. Um desempenho máximo de 111,4 GCUPS foi obtido com sequências de até 50 MBP.

4.5 Comparação dos artigos

A Tabela 4.1 apresenta um resumo comparativo entre os artigos discutidos neste capítulo. A coluna “Saída” indica se somente o escore (“escore”) de similaridade é retornado ou se o alinhamento completo (“alinh.”) também é informado. Podemos ver que a maioria dos artigos apresenta apenas o escore ótimo, excetuando os artigos Parallel-LTDP (Seção 4.2.4), DASW (Seção 4.3.1), CUDA-SSCA#1 (Seção 4.3.5), SW# (Seção 4.3.9), GSWABE (Seção 4.3.10), PP-Cell (Seção 4.4.2) e NoC-SW (Seção 4.4.3).

A coluna “Tam. Max.” informa qual o maior tamanho de sequência comparada, sendo que em caso de comparações de DNA consideramos a menor das duas sequências e nas comparações de proteínas consideramos o maior tamanho das sequências que são comparadas com o banco. Apenas três das propostas listadas foram capazes de comparar sequências maiores que 1 MBP: o CUDAlign 1.0 (Seção 4.3.7), o SW# (Seção 4.3.9) e o SWAPHI-LS (Seção 4.4.6). Conforme pode ser visto, o maior tamanho de sequência foi de 50 MBP (Seção 4.4.6). Os resultados apresentados nos demais artigos ficaram restritos a sequências com no máximo 35.213 caracteres. Limitar o tamanho da sequência de busca permite que as estruturas de dados sejam armazenadas nas memórias mais rápidas das GPUs, acelerando o processamento.

A coluna “GCUPS” registra o desempenho médio medido em Bilhões de Células Atualizadas por segundo. O desempenho obtidos nos artigos que usaram GPUs variaram até 196,2 GCUPS, sendo que o poder de processamento das placas gráficas utilizadas por cada uma das soluções é distinto. O maior desempenho entre as soluções apresentadas foi em FPGA (Seção 4.4.4), obtendo 3,04 TCUPS e, em CPU, o melhor desempenho foi 900 GCUPS (Seção 4.2.4). A coluna “Plataforma” indica qual arquitetura foi utilizada na solução. O CUDASW++3.0 utiliza uma abordagem híbrida (GPU+CPU), conseguindo o maior desempenho dentre os artigos em GPU (196,2 GCUPS). Algumas soluções utili-

Tabela 4.1: Artigos de comparação paralela com Smith-Waterman.

| Seção | Artigo | Ano | Saída | Tam. Máx. | GCUPS | Plataforma | |
|--------|---------------|------|--------|-------------------|----------------|-----------------|--------|
| 4.2.1 | SWMMX | 2000 | escore | 567 | 0,2 | Pentium III | CPU |
| 4.2.2 | StripSW | 2007 | escore | 567 | 3,0 | Intel Xeon | |
| 4.2.3 | SWIPE | 2011 | escore | 5.478 | 106,0 | 2 × Intel Xeon | |
| 4.2.4 | Parallel-LTDP | 2014 | alinh. | 800 | 900,0 | 16 × Intel Xeon | |
| 4.3.1 | DASW | 2006 | alinh. | 16.384 | 0,2 | 7800 GTX | GPU |
| 4.3.2 | GPU-SW | 2007 | escore | 4.095 | 0,7 | 7900GTX | |
| 4.3.3 | SW-CUDA | 2008 | escore | 567 | 3,6 | 8800GTX | |
| 4.3.4 | LR-SW | 2009 | escore | 1.000 | 14,5 | 9800GX2 | |
| 4.3.5 | CUDA-SSCA#1 | 2010 | alinh. | 1.024 | 1,0 | GTX295 | |
| 4.3.6 | CUDASW++1.0 | 2009 | escore | 5.478 | 16,1 | GTX295 | |
| 4.3.6 | CUDASW++1.0i | 2011 | escore | 5.478 | 29,3 | C2050 | |
| 4.3.6 | CUDASW++2.0 | 2010 | escore | 5.478 | 29,7 | GTX295 | |
| 4.3.7 | CUDAalign 1.0 | 2010 | escore | 32.799.110 | 20,3 | GTX285 | |
| 4.3.8 | FGCS | 2012 | escore | 511 | 64,0 | 6×GTX e 2×FX | |
| 4.3.6 | CUDASW++3.0 | 2013 | escore | 35.213 | 196,2 | GTX690+i7 | |
| 4.3.9 | SW# | 2013 | alinh. | 32.799.110 | 65,2 | GTX690 | |
| 4.3.10 | GSWABE | 2014 | alinh. | 500 | 58,3 | K40 | |
| 4.4.1 | CBESW | 2008 | escore | 852 | 3,7 | Cell BE | Outros |
| 4.4.2 | PP-Cell | 2009 | alinh. | 2.048 | 0,8 | Cell BE | |
| 4.4.3 | NoC-SW | 2010 | alinh. | 1.024 | 243,8 | ASIC 90nm | |
| 4.4.4 | SW-Rivyera | 2013 | escore | 100 | 3.040,0 | 128 × FPGA | |
| 4.4.5 | XSW | 2014 | escore | 1.000 | 70,0 | Xeon Phi | |
| 4.4.6 | SWAPHI-LS | 2014 | escore | 50.000.000 | 114,4 | 4 × Xeon Phi | |

zam mais de um nó de processamento, tais como o SWAPHI-LS (Seção 4.4.6), o FGCS (Seção 4.3.8) e a solução de FPGA (Seção 4.4.4).

Capítulo 5

Visão Geral do CUDAlign 1.0

O CUDAlign 1.0 (Seção 4.3.7) foi a ferramenta proposta na dissertação de mestrado do autor desta tese [33][122] com o objetivo de obter o escore da comparação de sequências longas de DNA utilizando o algoritmo Smith-Waterman com *affine gap* (Seção 2.2.3) e com uso linear de memória. A plataforma escolhida para a sua implementação foi a arquitetura CUDA da NVIDIA (Seção 3.2).

Visto que a presente tese utiliza algumas ideias do CUDAlign 1.0, este capítulo fornece uma visão geral dessa ferramenta. Para simplificar as explicações que se seguem, as matrizes H , E e F (utilizadas no cálculo do *affine gap* - Seção 2.2.3) são consideradas como componentes de uma única matriz de programação dinâmica. Apenas será feita menção aos componentes H , E e F quando forem relevantes para a explicação.

Este capítulo está estruturado da seguinte maneira. Na Seção 5.1 são apresentadas as técnicas de paralelismo (externo e interno) utilizadas. A Seção 5.2 apresenta as principais estruturas de dados do CUDAlign 1.0. A Seção 5.3 descreve as otimizações propostas. Finalmente, a Seção 5.4 apresenta fórmulas de previsão de desempenho.

5.1 Técnicas de Paralelismo

O CUDAlign 1.0 foi projetado para obter paralelismo no cálculo da equação de recorrência do Smith-Waterman com *affine-gap* (Seção 2.2.3) utilizando a técnica de *wavefront* (Seção 4.1.2) em GPU. No CUDAlign 1.0, o conceito de *wavefront* foi aplicado em blocos de células, de forma a criar anti-diagonais de blocos que são processados paralelamente. Com isso, a técnica de *wavefront* foi aplicada em dois níveis: o paralelismo externo e o paralelismo interno.

5.1.1 Paralelismo externo

O primeiro nível de paralelismo ocorre entre blocos de células. O agrupamento das células é feito dividindo-se a matriz de programação dinâmica em blocos com R linhas e C colunas ($R \times C$ células), o que resulta em um grid G com $\frac{m}{R} \times \frac{n}{C}$ blocos, onde m e n são os tamanhos das sequências S_0 e S_1 , respectivamente.

Os valores de R e C são escolhidos de acordo com o número B de blocos concorrentes e o número T de *threads* por bloco. Os valores B e T definem o número de blocos e de *threads* que serão executados para a chamada do *kernel* e compõem a configuração

| | | | | | |
|----|------------------|------------------|------------------|----|------------------|
| | 0 | 12 | 24 | 36 | |
| 0 | G _{0,0} | G _{0,1} | G _{0,2} | | |
| 6 | G _{1,0} | G _{1,1} | G _{1,2} | | |
| 12 | G _{2,0} | G _{2,1} | G _{2,2} | | ← D ₄ |
| 18 | G _{3,0} | G _{3,1} | G _{3,2} | | |
| 24 | G _{4,0} | G _{4,1} | G _{4,2} | | |
| 30 | G _{5,0} | G _{5,1} | G _{5,2} | | |
| 36 | | | | | |

Figura 5.1: Divisão da matriz em blocos. A diagonal externa D_4 está com blocos em destaque, observando-se que o número de colunas de blocos é $B = 3$ [122].

de execução em CUDA (Seção 3.2.4), representada por $\lll B, T \ggg$. Estes valores são escolhidos de acordo com as especificações da GPU e com os resultados empíricos obtidos em cada placa.

Dados os valores B e T , o número de linhas R e de colunas C em cada bloco são calculados da seguinte forma: $C = \frac{n}{B}$ e $R = \alpha.T$, sendo α uma constante inteira representando o número de linhas que cada *thread* é responsável por processar. Por exemplo, suponha que as sequências S_0 e S_1 possuam tamanho igual a 36 bases ($m = 36$ e $n = 36$). Além disso, suponha que sejam utilizados 3 blocos ($B = 3$), 3 *threads* por bloco ($T = 3$) e que cada *thread* calcule 2 linhas ($\alpha = 2$). Nesse caso, o grid G terá 6×3 blocos e cada bloco terá 6×12 células (Figura 5.1).

Feito este primeiro agrupamento, os blocos são novamente agrupados em anti-diagonais. A anti-diagonal D_k ($k = 0, 1, 2, \dots$) contém os blocos definidos pela equação $D_k = \{G_{ij} | i + j = k\}$. Essas anti-diagonais são chamadas de *diagonais externas*. O número de diagonais externas é $|D| = B + \frac{m}{\alpha T} - 1$ e o número de blocos em uma diagonal externa varia de 1 a B . A Figura 5.1 destaca os blocos da diagonal externa D_4 .

A técnica de *wavefront* é então aplicada em cada diagonal externa. Para isso, a CPU invoca uma execução do *kernel* para processar todos os blocos da diagonal externa. Cada execução do *kernel* possui B blocos e T *threads* por bloco. Quando a GPU completa a execução de todos os blocos de uma diagonal externa, a CPU sincroniza o processamento e reinvoca o *kernel* para a próxima diagonal externa, repetindo este procedimento sucessivamente até o fim do cálculo da matriz.

5.1.2 Paralelismo interno

Um outro nível de paralelismo ocorre no interior de um bloco. Para cada bloco, T *threads* cooperam para processar as $R \times C$ células deste bloco. O paralelismo é realizado de maneira similar ao do paralelismo externo. As células internas ao bloco são agrupadas em *diagonais internas*, de forma que a diagonal interna d_k ($k = 0, 1, 2, \dots$) é definida pela expressão $d_k = \{(i, j) | \lfloor \frac{i}{\alpha} \rfloor + j = k\}$, considerando que os valores i e j são relativos à célula superior esquerda deste bloco. Cada bloco possui T *threads* e a *thread* T_k processa

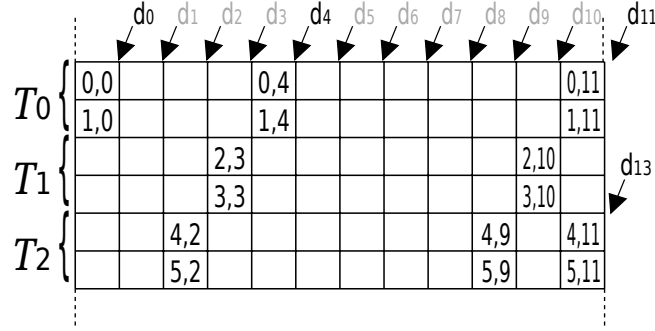


Figura 5.2: Bloco retangular com 3 *threads*. Cada *thread* é responsável por 2 linhas do bloco. As diagonais internas d_0 , d_4 , d_{11} e d_{13} estão indicadas na figura [122].

as linhas αk a $\alpha k + \alpha - 1$ deste bloco. O sentido do processamento de uma *thread* é da esquerda para a direita.

Para permitir a correta execução do *wavefront*, todas as *threads* de um bloco devem calcular em paralelo a mesma diagonal interna. Para isso, a *thread* T_i deve processar as células da coluna j ao mesmo tempo que a *thread* T_{i+1} processa as células da coluna $j - 1$. As *threads* são sincronizadas ao fim de cada diagonal interna por meio da diretiva `__syncthreads()`, que é a diretiva de barreira definida pela arquitetura CUDA (Seção 3.2).

A Figura 5.2 ilustra a execução de um bloco com 3 *threads*, sendo cada *thread* responsável por $\alpha = 2$ linhas. Note que as células da diagonal interna d_4 podem ser calculadas por cada *thread* de maneira paralela.

5.2 Estruturas em memória

Visto que o CUDAlign 1.0 processa sequências longas de DNA, as estruturas de dados em memória precisaram ser projetadas de maneira coerente com os objetivos dos vários tipos de memória disponíveis na arquitetura CUDA. Nesta seção descreveremos as estruturas de dados do CUDAlign 1.0 e suas principais características. Na matriz de programação dinâmica (DP), cada célula possui três componentes: H , F e E (Seção 2.2.3), sendo que cada componente possui 32 *bits* (4 *bytes*).

As estruturas de dados do CUDAlign 1.0 armazenadas em memória da GPU (Seção 3.2.4) são as seguintes:

- **Sequências:** As sequências são armazenadas em textura (memória somente-leitura). Cada nucleotídeo ocupa um *byte* em memória.
- **Memória Compartilhada:** Os valores da última linha de cada *thread* são transferidos através de memória compartilhada, salvo no caso da última *thread*, que envia os valores através da memória global para a primeira *thread* do próximo bloco. Os componentes que precisam ser transferidos entre as *threads* são apenas os componentes H e F .
- **Barramento Horizontal:** Os valores da última linha de cada bloco são armazenados nesta área da memória global para que o bloco imediatamente inferior carregue

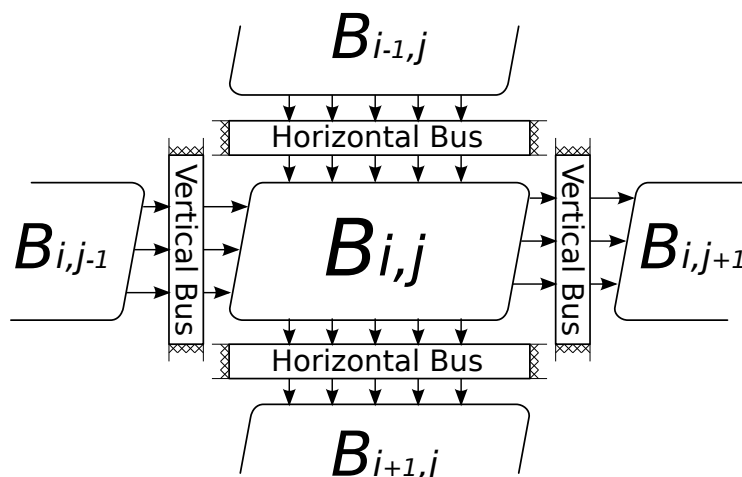


Figura 5.3: Barramentos de comunicação do CUDAAlign 1.0 [122].

Tabela 5.1: Tamanho e tipo de memória utilizada para as estruturas do CUDAAlign 1.0.

| Estrutura | Tamanho | Tipo |
|-----------------------|---------------------------------------|--------------------------|
| Sequências | $m + n + 2 \cdot B \cdot T$ bytes | Textura |
| Memória Compartilhada | $16 \cdot T$ bytes | Memória Compartilhada |
| Barramento Horizontal | $8n$ bytes | Memória Global + Textura |
| Barramento Vertical | $(8\alpha + 8) \cdot B \cdot T$ bytes | Memória Global |

esses dados e continue o seu processamento. As operações de escrita são feitas diretamente na memória global, mas as leituras são feitas através de textura. A Figura 5.3 ilustra o barramento horizontal transferindo dados entre os blocos $B_{i-1,j}$ e $B_{i,j}$ e entre os blocos $B_{i,j}$ e $B_{i+1,j}$.

- **Barramento Vertical:** Os valores das últimas células calculadas por cada *thread* são armazenados nesta área da memória global para que o bloco imediatamente à direita carregue esses dados e continue o seu processamento. A Figura 5.3 ilustra o barramento vertical transferindo dados entre os blocos $B_{i,j-1}$ e $B_{i,j}$ e entre os blocos $B_{i,j}$ e $B_{i,j+1}$.

A Tabela 5.1 apresenta um resumo das estruturas em memória. O total de memória global utilizada pelo CUDAAlign 1.0 é de $9n + m + (8\alpha + 10) \cdot B \cdot T$, o que permite concluir que o algoritmo utiliza memória na ordem de $O(m + n)$. Visto que o CUDAAlign 1.0 permite a comparação de sequências muito grandes, o tamanho do barramento horizontal torna-se a maior restrição do CUDAAlign 1.0. Podemos então estimar que a quantidade de memória global utilizada pelo CUDAAlign 1.0 é de aproximadamente $9n + m$ bytes. Com essa estimativa, uma GPU com 1GB de memória seria capaz de comparar duas sequências de aproximadamente 100 milhões de bases. Entretanto, devido ao uso não dedicado das GPUs, assim como a existência de outras estruturas utilizadas durante o processamento, o tamanho máximo das sequências pode ser menor.

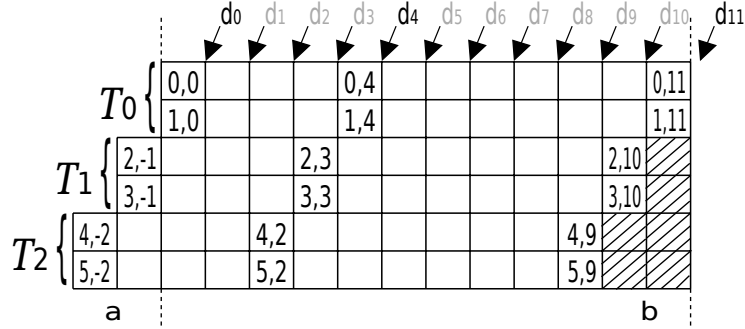


Figura 5.4: Bloco não retangular com 3 threads. O processamento continua enquanto o paralelismo da diagonal interna for máximo (até a diagonal d_{11}). Células hachuradas indicam as células pendentes, que serão processadas por outro bloco na próxima diagonal externa. Células com coluna negativa indicam células delegadas por um bloco da diagonal externa anterior [122].

5.3 Otimizações

O desempenho de um programa na arquitetura CUDA é determinado pelo paralelismo permitido pela plataforma. Entretanto, detalhes de implementação podem influenciar bastante na capacidade de explorar o paralelismo do hardware. Esta seção irá detalhar várias técnicas utilizadas para melhorar o desempenho do CUDAlign 1.0.

5.3.1 Delegação de Células

Conforme visto na seção 5.1, a técnica de *wavefront* é aplicada tanto entre os blocos (paralelismo externo) como entre as threads internas ao bloco (paralelismo interno). Dentro de cada bloco, o processamento é feito pelas diagonais internas, sendo que o grau de paralelismo de uma diagonal é proporcional ao número de células que ela possui. Observa-se que no início e no final de um bloco retangular as diagonais possuem um número menor de células, o que torna os extremos do bloco regiões com baixo nível de paralelismo. Na Figura 5.2, pode-se observar que as diagonais d_0 e d_{13} possuem apenas duas células, as diagonais d_1 e d_{12} possuem quatro células e todas as diagonais de d_2 até d_{11} possuem seis células, o que permite, nesse último caso, o máximo de paralelismo com 3 threads.

Para manter o paralelismo no máximo durante o maior tempo possível, o CUDAlign 1.0 propôs uma otimização chamada de *delegação de células*. Em vez de processar todas as células de um bloco, o CUDAlign 1.0 computa as células até a última diagonal interna cujo paralelismo seja máximo. Para isso, um bloco com $R \times C$ células terá C diagonais processadas, deixando as últimas células pendentes de processamento. Para que todas as células da matriz sejam computadas, o bloco sucessor é sempre responsável por calcular as células pendentes do bloco anterior.

A Figura 5.4 ilustra como essa técnica afeta o processamento de um bloco, de forma que encerra-se o bloco na diagonal d_{11} deixando seis células pendentes (células hachuradas). A Figura 5.5 ilustra a delegação de células entre os blocos. Anteriormente os blocos eram descritos como retângulos, mas por causa da delegação de células eles passam a ser representados por paralelogramos não retangulares. As áreas inclinadas desses paralelogramos indicam justamente as células delegadas entre os sucessivos blocos.

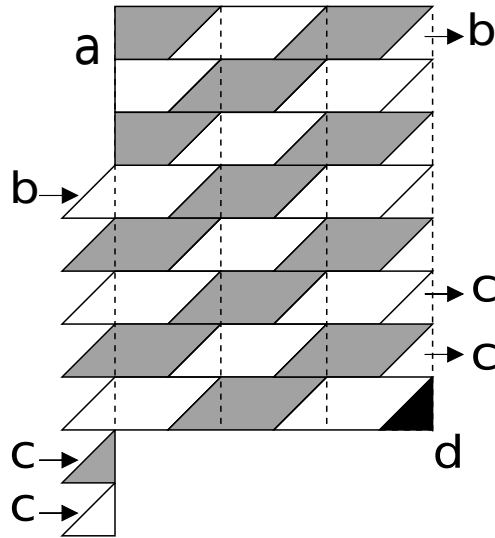


Figura 5.5: Delegação de células entre os blocos. A região (a) indica os blocos iniciais da matriz. O bloco (b) à direita da figura delega células para o bloco (b) à esquerda da matriz. Os blocos à direita indicados por (c) delegam células para blocos que preenchem o lado esquerdo (c) da matriz. Um bloco extra precisa ser criado para processar as células pendentes do bloco (d), por ser este bloco o último da matriz [122].

5.3.2 Divisão de Fases

Um detalhe que deve ser observado no processo de delegação de células é que os blocos da mesma diagonal externa não podem possuir dependência de dados entre si. Isso se deve ao fato de o escalonador CUDA executar os blocos em qualquer ordem, em paralelo ou em série, podendo até mesmo distribuí-los em diferentes unidades de processamento gráfico (Seção 3.2.4). Entretanto, por haver uma área do bloco que foi delegada para a diagonal externa seguinte, esta área cria uma dependência entre dois blocos da próxima diagonal.

A Figura 5.6 apresenta esta dependência durante a execução dos blocos 1, 2 e 3. O bloco 1 está em uma diagonal e os blocos 2 e 3 estão na diagonal seguinte. Por estar em diagonais distintas, o bloco 1 sempre será executado em uma chamada anterior aos blocos 2 e 3. A barra preta abaixo do bloco 1 indica as células que foram completamente calculadas no final da execução do bloco 1, sendo que o bloco 2 poderá ler esses valores corretamente. Em seguida, a próxima diagonal externa será executada, mas suponha que o escalonador escolha processar completamente o bloco 2 antes de iniciar o bloco 3. Neste caso, o bloco 2 lerá todos os valores dos blocos superiores, inclusive os valores indefinidos, que ainda não foram calculados pelo bloco 3. Na Figura 5.6 os valores indefinidos estão representados como uma barra cinza.

Para eliminar esta dependência entre blocos da mesma diagonal externa, deve-se calcular todas as células pendentes antes delas serem lidas e os blocos devem ser sincronizados para garantir a correta leitura dos valores das células. Para sincronizar a execução de todos os blocos, não existe uma primitiva que faça isso na GPU, o que torna necessário à CPU fazer isso explicitamente. No CUDAlign 1.0, esse procedimento é feito dividindo-se a execução do bloco em duas fases: a fase curta e a fase longa. Cada uma das fases é

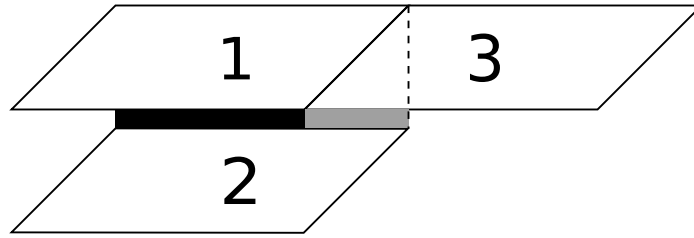


Figura 5.6: Problema de dependência de dados entre blocos. Caso o bloco 2 execute antes do bloco 3, o bloco 2 receberá valores indeterminados da área cinza.

implementada em um *kernel* distinto.

- **Fase Curta:** O objetivo dessa fase é terminar de processar todas as células pendentes. Para isso, processam-se as $T - 1$ primeiras diagonais internas de todos os blocos e, em seguida, o controle é retornado à CPU para forçar o sincronismo entre os blocos. Com isso, a área representada por uma barra cinza na Figura 5.6 estará corretamente calculada pelo bloco 3, permitindo o término do bloco 2.
- **Fase Longa:** Essa fase termina de processar as $\frac{n}{B} - (T - 1)$ diagonais internas restantes desse bloco. Essa fase foi chamada de *longa* pois usualmente o número de colunas C de cada bloco é muito maior que o número de *threads* T que esse bloco possui. Sendo assim, o tempo de execução da fase longa será muito maior do que o da fase curta.

A Figura 5.7 ilustra a divisão de fases, sendo que as áreas com final .1 indicam as fases curtas e áreas com final .2 indicam as fases longas.

A única condição que precisa ser garantida na divisão de fases é que o tamanho da fase curta ($T - 1$) seja menor ou igual que o tamanho da fase longa ($\frac{n}{B} - (T - 1)$), caso contrário os vários blocos da fase curta terão dependência de dados entre si, novamente gerando possíveis inconsistências. Para que isso não ocorra, é necessário que a Inequação 5.1 seja satisfeita [33]. Por simplicidade, esta condição será arredondada para apenas $n \geq 2BT$.

$$n \geq 2B(T - 1) \tag{5.1}$$

A condição $n \geq 2BT$ é chamada de **condição do tamanho mínimo** e, caso este critério não seja satisfeito para alguma sequência de tamanho pequeno, os parâmetros B e T podem ser reduzidos para satisfazê-lo. Além disso, se B for igual a 1 a condição do tamanho mínimo não se aplica, pois neste cenário os blocos são executados de maneira serial. Sendo assim, o CUDAlign 1.0 é capaz de alinhar sequências de qualquer tamanho, bastando alguns ajustes dos parâmetros B e T , caso necessário.

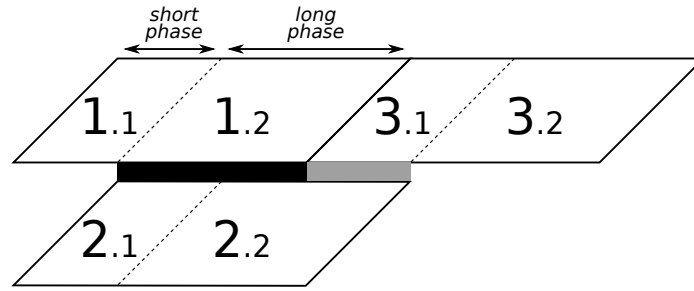


Figura 5.7: Execução do bloco dividida em duas fases. A fase curta possui sufixo .1 e a fase longa possui sufixo .2 [122].

5.4 Previsão de desempenho

A dissertação de mestrado [33] apresentou um método para prever o tempo de execução do CUDAalign 1.0 para duas sequências de tamanhos m e n . O método baseia-se na Equação 5.2, cujas constantes c_1 , c_2 , c_3 e c_4 são específicas para cada ambiente de execução. As constantes são obtidas por meio de regressão linear sobre os tempos de execução mensurados por *benchmarks*. Este modelo estatístico foi avaliado em situações reais e a diferença entre o tempo de execução real e o tempo previsto foi menor que 1% nos casos testados [33].

$$t(m, n) = c_1 + c_2m + c_3n + c_4mn \quad (5.2)$$

Para obtermos uma previsão do desempenho em CUPS (Seção 4.1.3), basta dividirmos o número de células da matriz ($m \times n$) pelo tempo de execução $t(m, n)$, conforme descrito na Equação 5.3.

$$CUPS(m, n) = \frac{mn}{t(m, n)} = \frac{mn}{c_1 + c_2m + c_3n + c_4mn} \quad (5.3)$$

Quando os tamanhos m e n das sequências aumentam, o desempenho previsto tende a aumentar até um limite máximo $CUPS_{max}$. Este limite é calculado pela Equação 5.4.

$$CUPS_{max} = \lim_{m, n \rightarrow (\infty, \infty)} \frac{mn}{c_1 + c_2m + c_3n + c_4mn} = \frac{1}{c_4} \quad (5.4)$$

Parte II
Contribuições

Capítulo 6

CUDAlign 2.0: Obtenção de alinhamentos longos em uma GPU

O CUDAlign 2.0 [34] é a primeira contribuição desta tese. A principal ideia do algoritmo é encontrar algumas coordenadas de um alinhamento ótimo e incrementar iterativamente o número dessas coordenadas até que seja possível obter o alinhamento completo utilizando uma quantidade restrita de memória. Ele foi desenvolvido com estratégias baseadas nos algoritmos de Myers e Miller (Seção 2.2.5) e FastLSA (Seção 2.2.6), com três novas otimizações, de forma a obter um alinhamento local ótimo de sequências longas de DNA em memória linear e tempo reduzido: o *matching* baseado em objetivo, a execução ortogonal e a divisão balanceada. Estas otimizações reduzem o número de vezes que a matriz é reprocessada para encontrar o alinhamento.

O objetivo deste capítulo é detalhar o projeto do CUDAlign 2.0. Cada um dos seus estágios e otimizações estão apresentados na Seção 6.1 e os resultados experimentais estão descritos na Seção 6.2. Por fim, uma discussão sobre o CUDAlign 2.0 será apresentada na Seção 6.3.

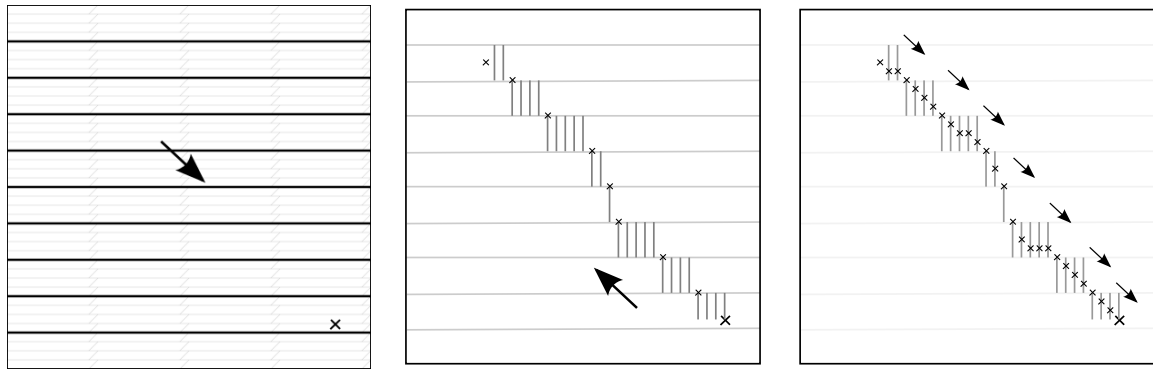
6.1 Projeto do CUDAlign 2.0

O CUDAlign 2.0 foi dividido em seis estágios (Figura 6.1), sendo que alguns deles podem ser ignorados dependendo do grau de similaridade entre as sequências. Em seguida cada estágio será detalhado.

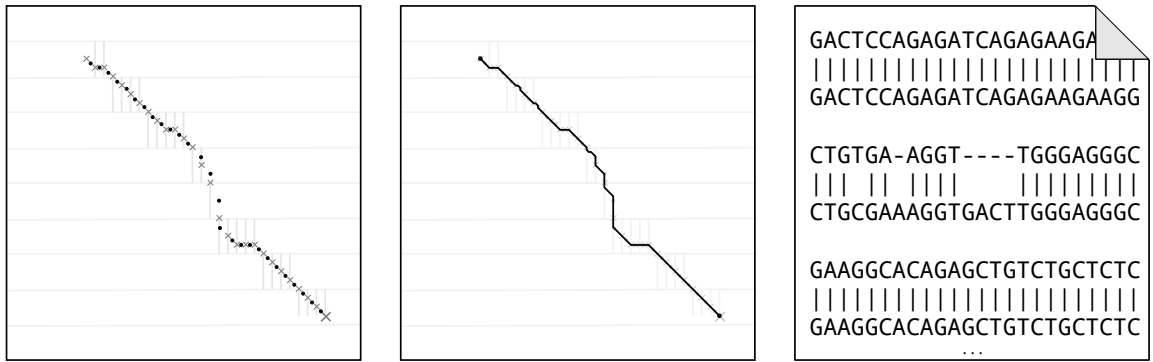
6.1.1 Estágio 1 - Obtenção do Escore Ótimo

O objetivo do Estágio 1 é encontrar o escore ótimo e a coordenada final do alinhamento. O Estágio 1 calcula o escore ótimo utilizando o modelo de *affine-gap* (Seção 2.2.3). O Estágio 1 executa o algoritmo básico do CUDAlign 1.0 (Capítulo 5) com uma modificação: algumas linhas da matriz (linhas especiais) são salvas em disco. As linhas especiais são utilizadas para executar o procedimento de *matching* no estágio 2 e para efetuar *check-points*, permitindo que a execução seja posteriormente restaurada em caso de interrupção do algoritmo.

As linhas especiais são obtidas do barramento horizontal (Seção 5.2) a cada intervalo de x blocos. Visto que o barramento horizontal armazena apenas a última linha de



(a) O Estágio 1 encontra o escore ótimo e sua posição. Linhas especiais são salvas em disco.
 (b) O Estágio 2 encontra as coordenadas nas quais o alinhamento ótimo intercepta as linhas especiais. Colunas especiais são salvas em disco.
 (c) O Estágio 3 encontra as coordenadas que interceptam o alinhamento ótimo pelas colunas especiais salvas no Estágio 2.



(d) O Estágio 4 executa o algoritmo de Myers e Miller em cada partição formada por coordenadas sucessivas.
 (e) O Estágio 5 obtém o alinhamento completo concatenando o alinhamento de cada partição.
 (f) O Estágio 6 permite a visualização textual e gráfica do alinhamento ótimo obtido.

Figura 6.1: Execução do CUDAlign 2.0 dividida em seis estágios.

cada bloco, apenas as linhas cujos índices são múltiplos da altura de um bloco (αT) são candidatas a serem salvas como uma linha especial.

A área reservada para armazenar as linhas especiais no disco é chamada de *special rows area* (SRA) e o seu tamanho é limitado por uma constante $|SRA|$ definida em tempo de execução. Cada célula de uma linha especial contém dois valores de 4 bytes, representando os componentes H e F da matriz de programação dinâmica. Sendo assim, uma linha especial completa possui $8n$ bytes e o número máximo de linhas especiais que podem ser salvas em disco é, no máximo, $\frac{|SRA|}{8n}$. O número mínimo de blocos entre cada linha especial é calculado pela Equação (6.1) e deve ser garantido que o SRA seja capaz de armazenar, no mínimo, 1 linha especial completa.

$$\left\lceil \frac{8mn}{\alpha \cdot T \cdot |SRA|} \right\rceil \quad (6.1)$$

A Figura 6.1(a) apresenta as saídas do Estágio 1. Neste exemplo, o intervalo entre as linhas especiais é de 4 blocos, então a cada 4 fileiras de blocos existe uma linha mais

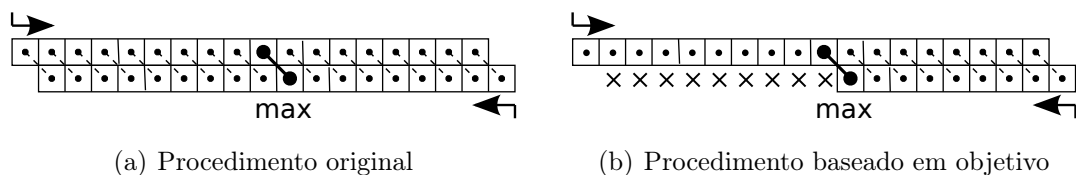


Figura 6.2: Procedimentos de *Matching*. O procedimento original de *matching* do algoritmo Myers-Miller (a) compara todos os pares de células para encontrar onde ocorre o escore máximo (*max*). No procedimento de *matching* baseado em objetivo (b), o maior escore já é conhecido, então o procedimento encerra assim que o escore máximo (*max*) for encontrado. O ‘x’ indica as células que não foram processadas.

escura representando as linhas especiais. A posição do escore ótimo está representada por um “x”.

6.1.2 Estágio 2 - *Traceback* Parcial

Encontrados os resultados do Estágio 1, o Estágio 2 calcula a equação de recorrência do alinhamento semi-global do tipo *+/** (Definição 2.1.20) na direção reversa, iniciando da coordenada final do alinhamento. O objetivo do Estágio 2 é encontrar as coordenadas de um alinhamento ótimo que cruzam as linhas especiais salvas no Estágio 1. Além disso, o Estágio 2 deve encontrar a coordenada inicial deste alinhamento ótimo. As saídas do Estágio 2 estão ilustradas na Figura 6.1(b).

Duas otimizações foram propostas para esse estágio: 1) procedimento de *matching* baseado em objetivo e 2) execução ortogonal.

***Matching* baseado em objetivo:** O algoritmo Myers-Miller original (Seção 2.2.5) executa um procedimento de *matching* para encontrar a célula da linha central através da qual o alinhamento ótimo passa. Nesse procedimento, todas as células das linhas centrais são analisadas, de forma a encontrar o par de células cujo escore somado seja o maior possível. No estágio 2, diferentemente do algoritmo Myers-Miller original, o escore máximo já é conhecido e este é chamado de escore-alvo. Assim que o escore-alvo for encontrado, o procedimento de *matching* pode encerrar a sua execução e iniciar uma nova iteração em busca da próxima coordenada (relativa à próxima linha especial). Inicialmente, o escore-alvo é o próprio escore ótimo obtido no Estágio 1, mas à medida que novas coordenadas são encontradas, o escore-alvo é atualizado com o escore da última coordenada obtida da linha especial. A Figura 6.2 ilustra a diferença entre o procedimento de *matching* do Myers-Miller original e o procedimento de *matching* baseado em objetivo.

Execução ortogonal: Para haver ganho de desempenho ao utilizar o procedimento de *matching* baseado em objetivo, as *threads* do Estágio 2 não podem executar na mesma direção horizontal do Estágio 1. Em vez disso, a execução das *threads* deve ser feita verticalmente, em direção ortogonal ao Estágio 1. Desta forma, diminui-se a área processada até que o procedimento de *matching* ocorra. A Figura 6.3 ilustra a redução da área processada e a Figura 6.4(a) apresenta, em detalhe, uma região onde a coordenada do alinhamento foi encontrada durante o procedimento de *matching*.

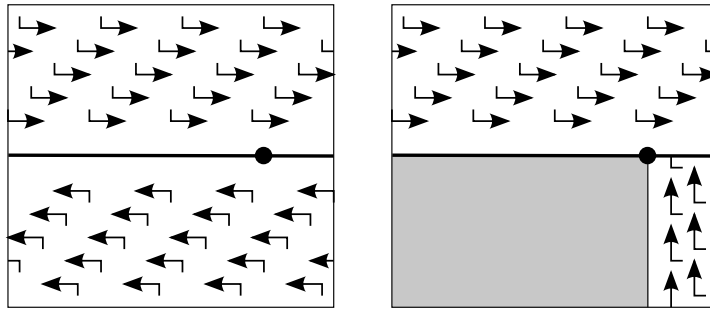
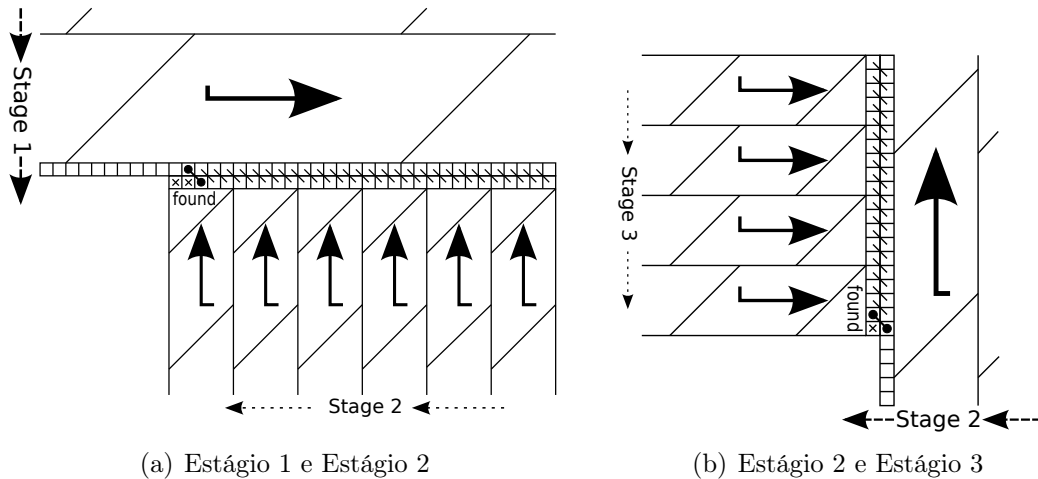


Figura 6.3: Execução ortogonal. A execução original do algoritmo Myers-Miller (à esquerda) processa as duas metades da matriz na mesma direção horizontal. Utilizando a execução ortogonal (à direita), o processamento da parte inferior é feito na vertical, em direção ortogonal ao da parte superior. A área em cinza não precisa ser processada, pois o escore-alvo foi encontrado no círculo preto.



(a) Estágio 1 e Estágio 2

(b) Estágio 2 e Estágio 3

Figura 6.4: Detalhe no procedimento de *matching* ortogonal.

Em termos gerais, a execução do Estágio 2 é bastante similar ao do Estágio 1. As principais diferenças são:

- Equação de recorrência: A equação de recorrência utilizada no Estágio 2 é baseada no algoritmo de alinhamento semi-global (tipo $+/*$ conforme a Definição 2.1.20), ao contrário do Estágio 1, que utiliza alinhamento local (tipo $*/*$).
- Área processada: Ao contrário do Estágio 1, que processa toda a matriz de programação dinâmica, o Estágio 2 processa áreas restritas da matriz. Cada área inicia-se na última coordenada encontrada, na direção reversa das sequências, delimitada à linha especial mais próxima. O percurso de *traceback* é realizado até que o início do alinhamento seja encontrado.
- Busca do escore máximo: A busca pelo escore máximo só precisa ser realizada caso o escore-alvo possa ocorrer antes da próxima linha especial. Isso geralmente ocorre somente quando o alinhamento está bastante próximo da coordenada inicial.

- Requisito do tamanho mínimo: O valor n utilizado para o requisito do tamanho mínimo $n \geq 2BT$ (Inequação 5.1) aplica-se à distância entre linhas especiais. Visto que n é muito menor que o tamanho da sequência, o requisito é mais restritivo e, por isso, o número de blocos B é normalmente reduzido.

6.1.3 Estágio 3 - Divisão de Partições

O objetivo do Estágio 3 é aumentar o número de coordenadas conhecidas que cruzam o alinhamento ótimo. O Estágio 3 é praticamente idêntico ao Estágio 2. A diferença entre ambos é que, no Estágio 3, as coordenadas sucessivas formam partições com começo e fim bem definidos, ao contrário do Estágio 2, que conhece a coordenada final e a coordenada inicial precisa ser encontrada.

Visto que existem partições bem definidas no Estágio 3, executa-se o mesmo algoritmo básico em GPU para encontrar as coordenadas por onde o alinhamento ótimo intercepta as linhas especiais salvas no Estágio 2. Note que não existe dependência entre as partições e, por isso, a ordem de execução das partições é irrelevante. Sendo assim, as partições podem ser processadas em paralelo ou até em GPUs distintas.

A Figura 6.1(c) apresenta as saídas do Estágio 3. Cada partição é dividida em várias coordenadas, uma para cada intersecção entre o alinhamento ótimo e as linhas especiais salvas no Estágio 2. A Figura 6.4(b) apresenta, em detalhe, uma região onde a coordenada do alinhamento foi encontrada durante o procedimento de *matching*.

6.1.4 Estágio 4 - Myers-Miller otimizado

O Estágio 4 executa em CPU o algoritmo de Myers-Miller (Seção 2.2.5) em cada partição formada por coordenadas sucessivas encontradas ao final do Estágio 3. O objetivo do Estágio 4 é aumentar, iterativamente, o número de coordenadas do alinhamento ótimo conhecidas até que o tamanho de cada partição formada entre as coordenadas seja menor que uma constante chamada de *tamanho máximo de partição*. O tamanho máximo de partição foi escolhido empiricamente como sendo 16 bases, de forma que o estágio 5 obtivesse o alinhamento destas pequenas partições em poucos segundos.

Cada iteração do Estágio 4 pode aumentar até $2\times$ o número de coordenadas conhecidas. Por isso, várias iterações podem ser necessárias até que os tamanhos das partições sejam inferiores ao tamanho máximo de partição. Assim como no Estágio 3, a ordem de processamento das partições é irrelevante, então as partições são processadas por várias *threads* em paralelo.

Conforme visto na Seção 2.2.5, o algoritmo Myers-Miller divide a matriz em sua linha central e processa as duas metades da matriz em sua totalidade e em sentido contrário. Por fim, as últimas linhas de cada uma das metades da matriz são comparadas de forma a encontrar a coordenada através do qual o alinhamento ótimo passa. No Estágio 4, o algoritmo Myers-Miller foi otimizado com uma proposta de *divisão balanceada* e a execução ortogonal foi utilizada para acelerar a obtenção dos *crosspoints*.

A otimização de *Divisão Balanceada* define que a matriz poderá ser dividida tanto em sua linha central como em sua coluna central. A ideia desta otimização é que a maior dimensão de cada partição seja dividida a cada iteração, prevenindo que partições com lados muito desproporcionais sejam mantidas por muitas iterações. A Figura 6.5 apresenta

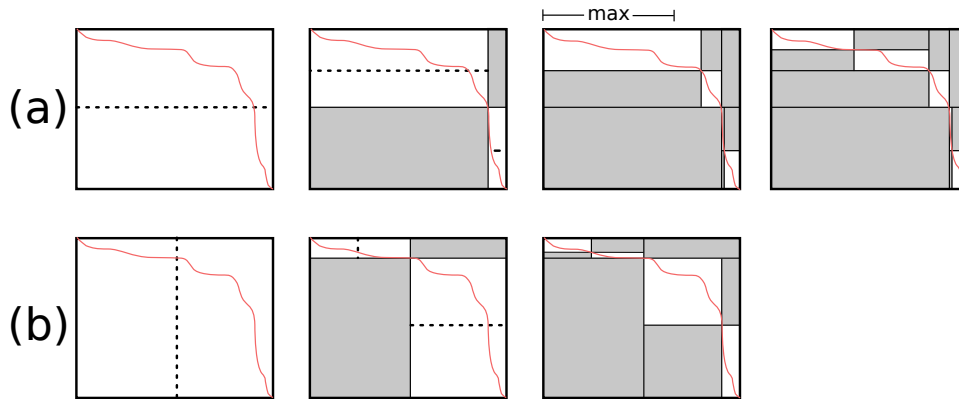


Figura 6.5: No algoritmo original de Myers-Miller (a), a divisão é sempre feita na linha central, então 3 iterações foram necessárias para obter partições com tamanho menor que “*max*”. Com a divisão balanceada (b), a divisão é feita na dimensão mais larga, então 2 iterações foram suficientes.

um exemplo utilizando a divisão balanceada e não balanceada. Após a segunda iteração, a divisão não balanceada cria uma partição com tamanho maior que o tamanho máximo (*max*), exigindo uma iteração extra. Já na divisão balanceada, isso não ocorre.

A *execução ortogonal* aproveita o fato de que os escores das coordenadas inicial e final da partição são conhecidas, logo o escore ótimo da partição é a diferença entre esses escores. Aproveitando essa observação, o algoritmo Myers-Miller (Seção 2.2.5) foi adaptado da seguinte forma. Uma das metades da matriz continua sendo processada em sua totalidade, mas a outra metade é processada somente até encontrar a coordenada do alinhamento ótimo. Para isso, a segunda metade é processada em direção contrária ao da primeira metade. Por exemplo, se a primeira metade processar a matriz linha após linha, a segunda metade processará a matriz coluna após coluna. Em média, a coordenada é encontrada no centro da matriz, o que exige apenas que 50% da segunda metade seja processada. Considerando ambas as metades da matriz, apenas 75% da partição precisará ser calculada em média, resultando em um ganho de 25% de eficiência no estágio 4.

A Figura 6.1(d) apresenta uma iteração do Estágio 4, resultando em novas coordenadas que formam partições ainda menores.

6.1.5 Estágio 5 - Obtenção do alinhamento completo

O Estágio 5 alinha em CPU cada partição obtida no Estágio 4, utilizando o algoritmo Needleman-Wunsch (Seção 2.2.1). O alinhamento de cada partição é concatenado, resultando no alinhamento ótimo completo, conforme podemos ver na Figura 6.1(e).

Após a execução do Estágio 4, é garantido que os tamanhos das partições serão limitados ao tamanho máximo de partição. Este tamanho deve ser pequeno suficiente para permitir que o Estágio 5 seja rápido. Visto que esse tamanho é constante, a complexidade do uso de memória em cada partição é constante e o alinhamento completo pode ser obtido em memória linear.

Para reduzir o tamanho da saída do Estágio 5, um arquivo binário foi criado contendo as seguintes informações: as coordenadas inicial (i_0, j_0) e final (i_1, j_1) (Definição 2.1.19); o escore ótimo (Definição 2.1.16); duas listas GAP_1 e GAP_2 , cada uma contendo tuplas $(i_{gap}, j_{gap}, length)$ onde i_{gap} e j_{gap} são a posição de cada abertura de *gap* e *length* o número de *gaps* sucessivos. A lista GAP_1 representa os *gaps* na sequência S_1 e a lista GAP_2 representa os *gaps* na sequência S_2 .

O arquivo binário permite representar um alinhamento sem armazenar as bases da sequência. Desta forma, o volume de dados necessário para a transferência e o armazenamento dos alinhamentos é reduzido consideravelmente em comparação com a representação textual do alinhamento. Entretanto, para reconstruir o alinhamento é necessário que as sequências originais estejam disponíveis. O alinhamento completo pode ser visualizado por meio do Estágio 6.

6.1.6 Estágio 6 - Visualização

O Estágio 6 é utilizado opcionalmente para visualização da representação binária do alinhamento. Dadas as sequências de entrada S_0 e S_1 , as coordenadas inicial (i_0, j_0) e final (i_1, j_1) e as listas de *gaps* GAP_1 e GAP_2 , a reconstrução do alinhamento é feita por meio da união de todos os *gaps*. Iniciando com a coordenada $(i, j) = (i_0, j_0)$, o *gap* mais próximo é obtido das listas GAP_1 e GAP_2 e a próxima coordenada (i, j) será o final do *gap* mais próximo escolhido. Este procedimento é feito até que se encontre a posição final (i_1, j_1) .

Esse procedimento permite que as representações textual (Figura 6.1(f)) ou gráfica (Figuras 2.4 e 6.7) sejam obtidas. Visto que a representação binária é muito menor que a textual, o Estágio 6 somente é necessário quando uma análise detalhada do alinhamento for necessária. Adicionalmente, foi desenvolvido um utilitário que interpreta os arquivos binários produzidos no estágio 5 e apresenta os alinhamentos em uma interface gráfica interativa (Figura 6.6).

6.2 Resultados Experimentais

O CUDAlign 2.0 foi testado em uma placa NVIDIA GeForce GTX 285 (Tabela 3.1), com 1GB de memória e 240 núcleos. Os parâmetros do Smith-Waterman foram setados em *match*: +1; *mismatch* -3; *first gap*: -5; *extension gap*: -2. Os parâmetros de execução foram: $\alpha = 4$; o número de blocos e de *threads* para o Estágio 1 foram $B_1 = 240$ e $T_1 = 2^6$; o número de blocos e de *threads* para os Estágios 2 e 3 foram $B_2 = B_3 = 60$ e $T_2 = T_3 = 2^7$.

Os testes executados para o CUDAlign 2.0 utilizaram sequências reais obtidas do site NCBI (www.ncbi.nlm.nih.gov). A Tabela 6.1 lista os nomes e os tamanhos das sequências, que variaram de 162 KBP a 47 MBP. As sequências foram as mesmas utilizadas no CUDAlign 1.0 [122].

Para todos os pares de sequências apresentados na Tabela 6.1, a Tabela 6.2 lista o melhor escore, o comprimento do alinhamento, o número de *gaps* encontrados e o número de células processadas no Estágio 1. Nota-se que 1,54 peta células são calculadas na comparação $32799K \times 46944K$.

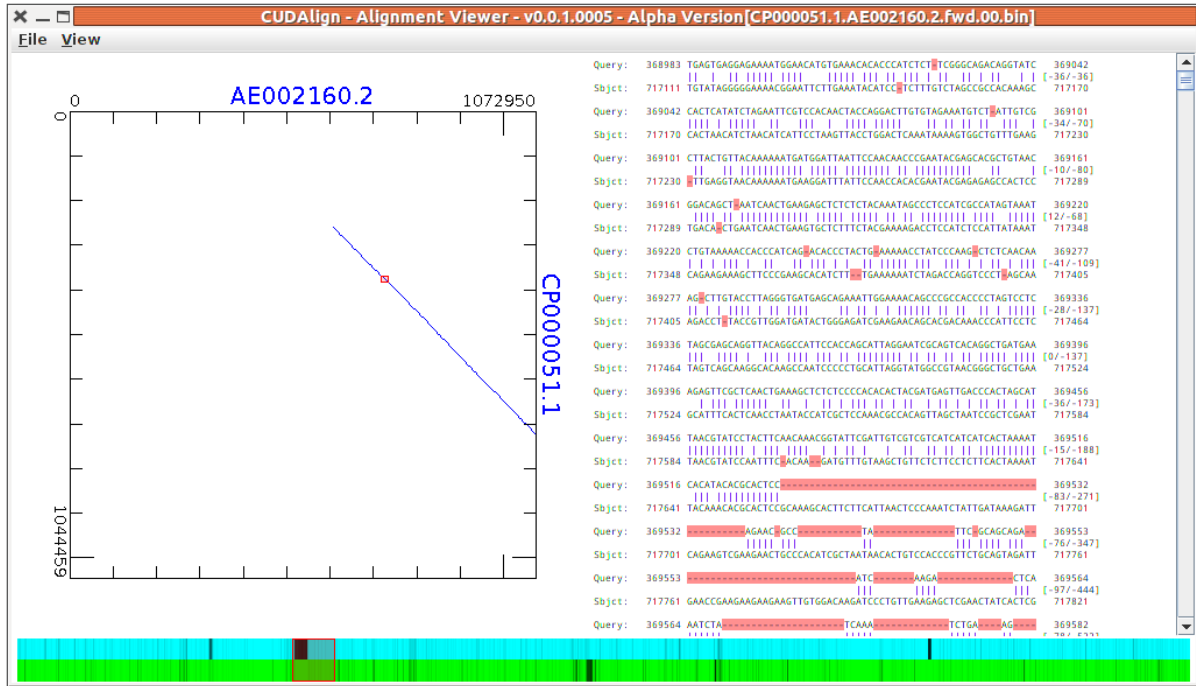


Figura 6.6: Visualizador de alinhamentos do CUDAlign

A Tabela 6.3 apresenta o tempo de execução e o GCUPS do Estágio 1, com e sem salvar as linhas especiais em disco. Note que, para sequências maiores, o *overhead* causado para salvar as linhas especiais é aproximadamente 1% do tempo de execução, o que consideramos muito baixo. Nos testes, o tamanho do *Special Rows Area* (SRA) foi escolhido empiricamente.

A Tabela 6.4 mostra o tempo de execução para todos os estágios. Quando o comprimento do alinhamento é pequeno, os tempos de execução dos estágios 2 a 6 são desprezíveis, como pode ser visto para as comparações $162K \times 172K$, $543K \times 536K$ e $7146K \times 5227K$. Em todos os casos analisados, o tempo de execução do estágio 1 é bem maior do que o tempo de execução dos demais estágios. Na comparação $32799K \times 46944K$, o estágio 1 demora mais de 18 horas e os demais estágios somam 25 minutos.

Os detalhes do alinhamento ótimo obtido na comparação $32799K \times 46944K$ estão apresentados na Figura 6.7, que é a saída gráfica provida pelo Estágio 6. A representação textual do início deste mesmo alinhamento está na Figura 6.8.

6.3 Conclusão do Capítulo

Na Seção 6.2, mostramos que o CUDAlign 2.0 é capaz de recuperar alinhamentos longos entre cromossomos completos. Isso foi possível devido às otimizações propostas nessa tese (*matching* baseado em objetivo, execução ortogonal e divisão balanceada), aplicadas no processamento dos estágios 2 a 4 do CUDAlign 2.0.

No entanto, notamos que o número de células calculadas da matriz de programação dinâmica é bastante elevado para sequências longas (na ordem de peta células) e que

Tabela 6.1: Detalhes das sequências reais. Os tamanhos das sequências variaram de 162 KBP a 47 MBP.

| Comparação | Tamanho | Número de Acesso | Nome |
|---------------|---------------|------------------|---|
| 162K×172K | 162,114 BP | NC_000898.1 | <i>Human herpesvirus 6B</i> |
| | 171,823 BP | NC_007605.1 | <i>Human herpesvirus 4</i> |
| 543K×536K | 542,868 BP | NC_003064.2 | <i>Agrobacterium tumefaciens</i> |
| | 536,165 BP | NC_000914.1 | <i>Rhizobium sp.</i> |
| 1044K×1073K | 1,044,459 BP | CP000051.1 | <i>Chlamydia trachomatis</i> |
| | 1,072,950 BP | AE002160.2 | <i>Chlamydia muridarum</i> |
| 3147K×3283K | 3,147,090 BP | BA000035.2 | <i>Corynebacterium efficiens</i> |
| | 3,282,708 BP | BX927147.1 | <i>Corynebacterium glutamicum</i> |
| 5227K×5229K | 5,227,293 BP | AE016879.1 | <i>Bacillus anthracis</i> str. Ames |
| | 5,228,663 BP | AE017225.1 | <i>Bacillus anthracis</i> str. Sterne |
| 7146K×5227K | 7,145,576 BP | NC_005027.1 | <i>Rhodopirellula baltica</i> SH 1 |
| | 5,227,293 BP | NC_003997.3 | <i>Bacillus anthracis</i> str. Ames |
| 23012K×24544K | 23,011,544 BP | NT_033779.4 | <i>Drosophila melanog.</i> chromosome 2L |
| | 24,543,557 BP | NT_037436.3 | <i>Drosophila melanog.</i> chromosome 3L |
| 32799K×46944K | 32,799,110 BP | BA000046.3 | <i>Pan troglodytes</i> DNA, chromosome 22 |
| | 46,944,323 BP | NC_000021.7 | <i>Homo sapiens</i> chromosome 21 |

Tabela 6.2: Resultados dos alinhamentos ótimos

| Comparação | Células | Escore | Tamanho | Gaps |
|---------------|----------|----------|----------|---------|
| 162K×172K | 2,79E+10 | 18 | 18 | 0 |
| 543K×536K | 2,91E+11 | 48 | 92 | 0 |
| 1044K×1073K | 1,12E+12 | 88353 | 471858 | 14021 |
| 3147K×3283K | 1,03E+13 | 4226 | 14554 | 891 |
| 5227K×5229K | 2,73E+13 | 5220960 | 5229192 | 2430 |
| 7146K×5227K | 3,74E+13 | 172 | 565 | 18 |
| 23012K×24544K | 5,65E+14 | 9063 | 9107 | 6 |
| 32799K×46944K | 1,54E+15 | 27206434 | 33583457 | 1371283 |

muitas das células não contribuem para a obtenção do resultado ótimo. Essa observação seria ainda mais perceptível para sequências na ordem de 100 MBP. Isso nos motivou a desenvolver o algoritmo de descarte de células (*Block Pruning*), que será abordado no Capítulo 7 (CUDAlign 2.1) e analisado teoricamente no Capítulo 8.

Tabela 6.3: Tempo de execução (em segundos) do Estágio 1 do CUDAlign 2.0 com e sem salvar linhas especiais em disco.

| Comparação | Sem SRA | | Com SRA | | |
|---------------|---------|-------|---------|-------|-------|
| | Tempo | GCUPS | SRA | Tempo | GCUPS |
| 162K×172K | 1,4 | 19,77 | 5M | 1,5 | 18,68 |
| 543K×536K | 12,9 | 22,55 | 50M | 13,6 | 21,42 |
| 1044K×1073K | 48,3 | 23,21 | 250M | 51,6 | 21,71 |
| 3147K×3283K | 436 | 23,71 | 1G | 448 | 23,04 |
| 5227K×5229K | 1147 | 23,82 | 3G | 1185 | 23,07 |
| 7146K×5227K | 1568 | 23,82 | 3G | 1604 | 23,28 |
| 23012K×24544K | 23620 | 23,91 | 10G | 23750 | 23,78 |
| 32799K×46944K | 64507 | 23,87 | 50G | 65153 | 23,63 |

Tabela 6.4: Tempo de execução (em segundos) de cada estágio do CUDAlign 2.0 na placa GTX 285, variando o tamanho da comparação. O tempo total inclui todos os estágios e o tempo de leitura das sequências.

| Comparação | Estágio | | | | | Total |
|---------------|---------|------|------|------|------|-------|
| | 1 | 2 | 3 | 4 | 5+6 | |
| 162K×172K | 1.5 | <0.1 | <0.1 | <0.1 | <0.1 | 1.8 |
| 543K×536K | 13.6 | <0.1 | <0.1 | <0.1 | <0.1 | 13.9 |
| 1044K×1073K | 51.6 | 3.1 | 1.0 | 5.4 | 0.1 | 61.6 |
| 3147K×3283K | 448 | 0.1 | <0.1 | 0.3 | <0.1 | 449 |
| 5227K×5229K | 1185 | 65.9 | 20.3 | 47.6 | 1.9 | 1321 |
| 7146K×5227K | 1604 | <0.1 | <0.1 | <0.1 | <0.1 | 1605 |
| 23012K×24544K | 23750 | 0.3 | <0.1 | 0.7 | <0.1 | 23755 |
| 32799K×46944K | 65153 | 805 | 236 | 376 | 9 | 66579 |

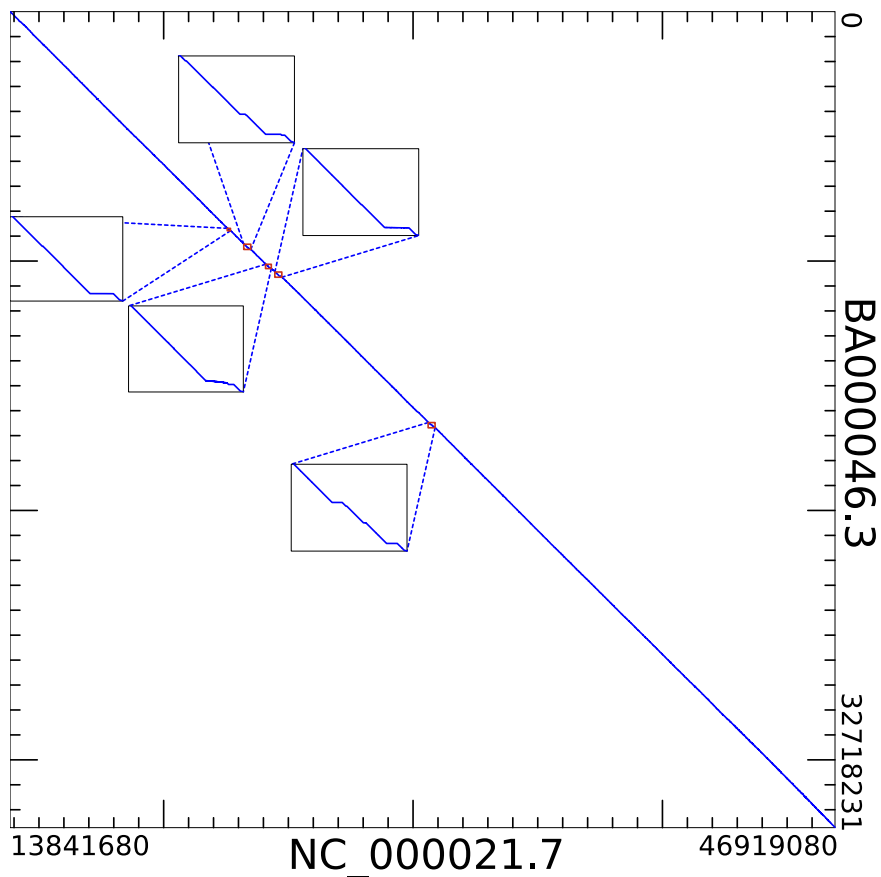


Figura 6.7: Representação gráfica do alinhamento entre cromossomos 21 do homem e do chimpanzé. Cinco seções relevantes foram destacadas.

```

Query:      1 GAGCTCTATTGTTCCATTTTCATTGGCTTCTTCAGGTGCTCAGAAAAAGAAAAGACCCTG      61
            ||||| ||||||||||||||||||||||||||||||||||||||||| ||| ||||| ||||| [44/44]
Sbjct: 13841681 GAGCTGTATTGTTCCATTTTCATTGGCTTCTTCAGGTGCTCAGTAAAGGAAAATACCCTG 13841741

Query:      61 TTTTTCATGGAGAAAACATTGGGCAAATTGATTCAATCAAGAACTTTCTTAGTATATAAG      121
            ||||||||||||||||||||||||||||||||||||||||| ||||||||||||||||| [56/100]
Sbjct: 13841741 TTTTTCATGGAGAAAACATTGGGCAAATTGATTCAATCAAGAAATTTCTTAGTATATAAG 13841801

Query:      121 ATGAAAGTGGTGAACACATGAACTCTGAACCTGTTTCTTACCTAAATCCATTCTTTTGT      181
            || ||||||||| ||||||||||||||||||||||||||||||||||||||||| | [48/148]
Sbjct: 13841801 ATTAAAGTGGTAGAACACATGAACTCTGAACCTGTTTCTTACCTAAATCCATTCTTTTTT 13841861

            [...]

```

Figura 6.8: Representação textual de um trecho do alinhamento entre o cromossomo 21 do homem e do chimpanzé.

Capítulo 7

CUDAlign 2.1: Obtenção de alinhamentos em uma GPU com descarte de blocos

No CUDAlign 2.1 [35], a otimização *Block Pruning* (BP) é proposta para acelerar o processamento da matriz de programação dinâmica quando deseja-se obter um único alinhamento local ótimo. Dependendo da similaridade entre as sequências, essa otimização permite que blocos de células da matriz de programação dinâmica sejam descartados. Com isso, elimina-se o cálculo de blocos de células que comprovadamente não contribuem para o alinhamento ótimo. Estes blocos possuem um escore tão baixo que torna-se matematicamente impossível que um alinhamento que passe por ele seja melhor do que o melhor alinhamento encontrado até então.

Na Seção 7.1, a otimização *Block Pruning* é proposta e a Seção 7.2 apresenta os resultados experimentais. Ao final do capítulo, a Seção 7.3 apresenta uma conclusão sobre o capítulo.

7.1 Otimização Block Pruning

O *Block Pruning* (BP) utiliza idéia similar à proposta em [132], mas com duas diferenças: (a) o *Block Pruning* é aplicado em conjunto de blocos, e não em células individuais, o que diminui o *overhead* causado pelos cálculos adicionais; e (b) no CUDAlign 2.1, o *Block Pruning* é aplicado no estágio 1, onde o escore ótimo ainda não é conhecido, acelerando o estágio mais longo do processamento.

7.1.1 Definições

Considere que as sequências comparadas são S_0 e S_1 , cujos tamanhos são respectivamente m e n , e que a função $p(i, j)$ representa a pontuação obtida entre as bases $S_0[i]$ e $S_1[j]$, podendo ser $p(i, j) = ma$ em caso de *match* e $p(i, j) = mi$ em caso de *mismatch*.

Para uma célula (i, j) da matriz de programação dinâmica, a distância-i (Δ_i) é a distância entre a linha i e a última linha da matriz ($\Delta_i = m - i$) e a distância-j (Δ_j) é a distância entre a coluna j e a última coluna da matriz ($\Delta_j = n - j$). Chamamos uma

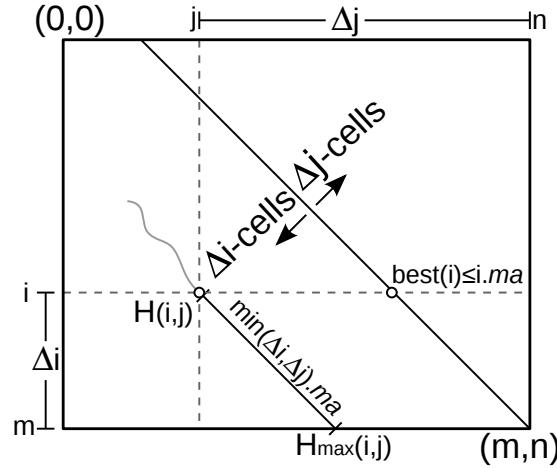


Figura 7.1: Definições geométricas do *Block Pruning*.

célula de Δ_i -cell uma célula (i, j) que está mais próxima da última linha do que da última coluna ($\Delta_i < \Delta_j$) e chamamos uma célula de Δ_j -cell no caso inverso ($\Delta_j < \Delta_i$).

Suponha que uma célula (i, j) da matriz possua escore $H(i, j)$. O escore máximo que um alinhamento local pode possuir passando por essa célula é de $H_{max}(i, j) = H(i, j) + \min(\Delta_i, \Delta_j).ma$, onde ma é a pontuação de um *match*. Essa situação ilustra o caso onde ocorre um *match* perfeito entre as substrings $S_0[i..(i + \min(\Delta_i, \Delta_j))] = S_1[j..(j + \min(\Delta_i, \Delta_j))]$.

O máximo escore possível que pode ser encontrado até a linha i ocorre em um caso de *match* perfeito entre as *substrings* $S_0[1..i] = S_1[1..i]$, e o máximo escore possível nesta linha i é representada por $best(i) = i.ma$. Uma célula é considerada *prunable* se a célula não pode originar um alinhamento com escore maior que o escore máximo encontrado no momento, ou seja, a célula (i, j) é *prunable* se $H_{max}(i, j) \leq best(i)$. Com isso, todas as células *prunable* podem ser ignoradas sem impacto no cálculo do escore ótimo final. A Figura 7.1 apresenta algumas dessas definições em representação geométrica.

7.1.2 Procedimento de *Pruning*

Para reduzir o número de células processadas, devemos identificar uma célula *prunable* antes mesmo de calcularmos o seu valor. Isso pode ser feito observando que se as células $(i - 1, j)$, $(i, j - 1)$ e $(i - 1, j - 1)$ são *prunable*, então a célula (i, j) também será.

Considerando que a matriz é processada linha após linha, o procedimento de *Pruning* funciona da seguinte maneira.

Chamamos de janela não-*prunable* o intervalo $[k_s..k_e]$ de colunas que devem ser processadas em uma determinada linha. Inicialmente, definimos $[k_s..k_e] = [0..n]$. Para cada linha i , o algoritmo calcula todas as células $(i, j \in [k_s..k_e])$ e, então, ele atualiza a janela não-*prunable* para os valores $[k'_s..k'_e]$, onde os valores k'_s e k'_e são, respectivamente, a primeira e a última célula não-*prunable* desta linha.

Assumindo que as células $(i - 1, j \in [0..k_s])$ são *prunable*, então todas as células $(i, j \in [0..k_s])$ também serão e, por indução, todas as células $(i' \geq i, j \in [0..k_s])$ também serão.

Se todas as células $(i-1, j \in [k_e..n])$ forem *prunable* e, para algum $x \in [k_e..n]$, a célula (i, x) for *prunable*, logo todas as células $(i, j \in [x+1..m])$ também serão *prunable*. Se a célula $(i, k_e + 1)$ for *prunable*, então o novo valor k'_e será a última célula não-*prunable* no intervalo $(i, j \in [k_s..k_e])$, onde $k'_e \leq k_e$. Caso contrário, devemos calcular as células restantes da linha i até que encontremos a primeira célula *prunable* $(i, x \in [k_e + 2..n])$, de forma que a novo valor de k'_e será $x - 1$.

As células $(i, j \in [0..k'_e])$ são chamadas de células *prunable* à esquerda e as células no intervalo $(i, j \in [k'_e + 1..n])$ são chamadas de células *prunable* à direita.

A otimização *Block Pruning* é o procedimento descrito nesta seção aplicado a blocos de células. O procedimento aplicado a blocos é usualmente mais rápido, pois a verificação da condição de *pruning* é feita uma única vez para todo o bloco, considerando o pior caso onde a célula com maior escore encontra-se no canto superior direito do bloco. Entretanto, se o bloco for muito grande, o procedimento de *pruning* será tão pouco granular que será difícil encontrar blocos *prunable*.

Os *kernels* do CUDAlign 2.1 verificam se o bloco a ser calculado está dentro da janela não-*prunable*. Se o bloco estiver fora da janela, então o bloco não será calculado e as áreas associadas aos barramentos vertical e horizontal (Seção 5.2) serão zeradas para prevenir qualquer inconsistência de dados. Após cada diagonal externa ser calculada, a janela não-*prunable* é atualizada considerando o escore máximo obtido por cada bloco e o valor máximo que cada um pode obter. A atualização da janela não-*prunable* é feita em CPU.

7.1.3 Teoremas

Os teoremas seguintes assumirão que o cálculo da matriz de programação dinâmica é feito linha após linha e que cada bloco possui uma única célula.

Teorema 7.1.1 Uma célula (i, j) do tipo Δ_i -cell somente pode ser *prunable* se $i \geq m/2$.

Demonstração. Uma célula Δ_i -cell é uma célula (i, j) tal que $\Delta_i < \Delta_j$ e o maior escore possível em uma linha i é $best(i) = i.ma$. Então, temos as inequações (7.1):

$$\begin{aligned}
H_{max}(i, j) &\leq best(i) \leq i.ma \\
H(i, j) + \min(\Delta_i, \Delta_j).ma &\leq i.ma \\
H(i, j) + \Delta_i.ma &\leq i.ma \\
H(i, j) + (m - i).ma &\leq i.ma \\
H(i, j) + m.ma &< 2.i.ma \\
\frac{H(i, j)}{2.ma} + \frac{m}{2} &\leq i \\
\frac{m}{2} &\leq i, \text{ visto que } H(i, j) \geq 0
\end{aligned} \tag{7.1}$$

□

Teorema 7.1.2 Uma célula (i, j) do tipo Δ_j -cell somente pode ser *prunable* se $i \geq n - j$.

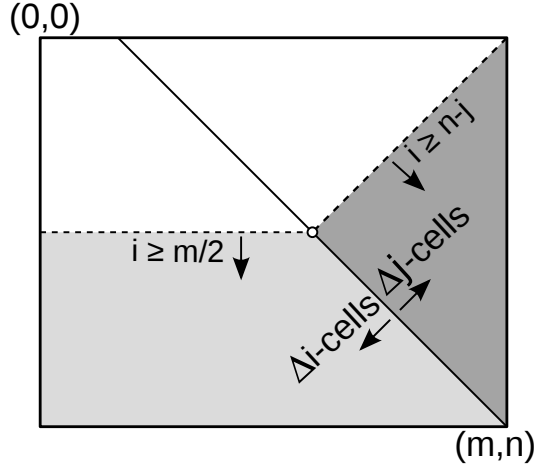


Figura 7.2: Áreas prunable (em cinza).

Demonstração. Uma célula Δ_j -cell é uma célula (i, j) tal que $\Delta_j < \Delta_i$ e o maior escore possível em uma linha i é $best(i) = i.ma$. Então, temos as inequações (7.2):

$$\begin{aligned}
 H_{max}(i, j) &\leq best(i) \leq i.ma \\
 H(i, j) + \min(\Delta_i, \Delta_j).ma &\leq i.ma \\
 H(i, j) + \Delta_j.ma &\leq i.ma \\
 H(i, j) + (n - j).ma &\leq i.ma \\
 \frac{H(i, j)}{ma} + (n - j) &\leq i \\
 (n - j) &\leq i, \text{ visto que } H(i, j) \geq 0
 \end{aligned} \tag{7.2}$$

□

Corolário 7.1.3 Uma célula (i, j) somente pode ser *prunable* se $i \geq n - j$ e $i \geq m/2$.

Demonstração. Esta é a união dos Teoremas 7.1.1 e 7.1.2

□

A Figura 7.2 apresenta as áreas onde podemos encontrar células Δ_i -cell *prunable* (cinza claro) e células Δ_j -cell *prunable* (cinza escuro). Além disso, a área onde não podemos encontrar nenhuma célula *prunable* está apresentada em branco (Corolário 7.1.3). Analisando geometricamente esta área, podemos concluir que ela contém aproximadamente $\frac{mn}{2} - \frac{m^2}{8}$ células se $m \leq 2n$, ou $\frac{n^2}{2}$ quando $m > 2n$. Dividindo este número pelo total de células (mn), temos um limite superior para o método de *pruning* definido pela equação 7.3. Logo, por exemplo, se existem duas sequências de tamanho igual ($m = n$), temos um limite superior de *pruning* restrito em 62.5%. No pior caso, todas as células serão calculadas e o limite inferior é de 0%. Desta forma, o Estágio 1 continua executando em complexidade $O(mn)$ de tempo.

$$\frac{1}{2} + \frac{1}{8} \frac{m}{n}, \quad \text{se } m \leq 2n$$

$$1 - \frac{1}{2} \frac{n}{m}, \quad \text{se } m > 2n$$
(7.3)

7.2 Resultados Experimentais

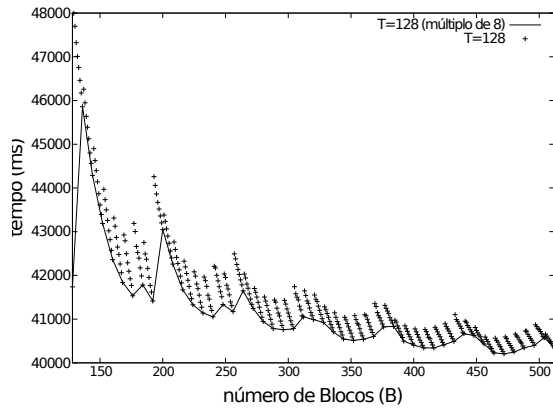
O CUDAlign 2.1 foi testado em uma placa NVIDIA GeForce GTX 560 Ti (Tabela 3.1), com 1GB de memória e 384 núcleos. Os parâmetros do Smith-Waterman foram definidos em *match*: +1; *mismatch* -3; *first gap*: -5; *extension gap*: -2.

Os testes executados para o CUDAlign 2.1 utilizaram sequências reais obtidas do site NCBI (www.ncbi.nlm.nih.gov). As mesmas sequências do CUDAlign 2.0 (Tabela 6.1) foram utilizadas, acrescida da comparação 59374K×23953K entre o cromossomo Y do *Homo sapiens* (NC_000024.9) e o cromossomo Y do *Pan troglodytes* (NC_006492.2). Com isso, o tamanho das sequências utilizados para os testes variaram de 162 KBP a 59 MBP.

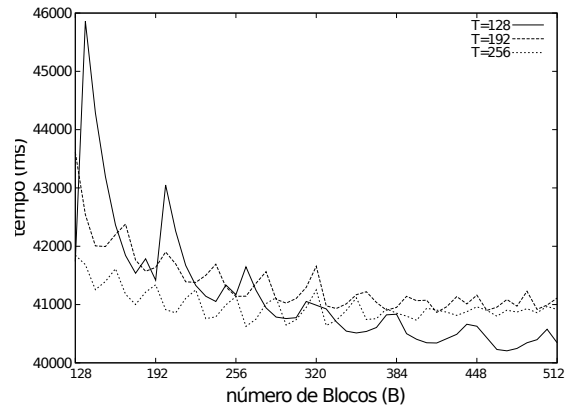
Para encontrar os melhores parâmetros de blocos (B) e *threads* (T), executamos o CUDAlign 2.1 com várias configurações diferentes. A Figura 7.3(a) apresenta o tempo de execução do estágio 1 (sem *Block Pruning*) da comparação 1044K×1073K, variando o valor de B entre 128 a 512, usando $T = 128$ *threads*. Podemos ver que os melhores tempos ocorrem com valores de B que são múltiplos de 8. Isso é explicado pelo fato de que a GTX 560 Ti possui 8 multiprocessadores, produzindo melhores resultados quando há um número uniforme de blocos alocados por multiprocessador. Considerando apenas os valores de B que são múltiplos de 8, a Figura 7.3(b) apresenta as mesmas execuções para $T = 128$, $T = 192$ e $T = 256$. O melhor resultado foi obtido quando $T = 128$ e $B = 472$. Visto que o tempo de execução da potência de 2 mais próxima ($B = 512$) foi apenas 0,35% mais lento, foi preferido um valor de B um pouco maior, que tende a produzir uma melhor eficácia de *pruning*. Para os estágios 2 e 3, preferiu-se utilizar valores de B menores. Sendo assim, os parâmetros de execução escolhidos foram: $\alpha = 4$; o número de blocos e de *threads* para o Estágio 1 foram $B_1 = 512$ e $T_1 = 128$; o número de blocos e de *threads* para os Estágios 2 e 3 foram $B_2 = B_3 = 32$ e $T_2 = T_3 = 128$.

A Tabela 7.1 apresenta o tempo de execução e o GCUPS para o Estágio 1, com e sem a otimização de *Block Pruning* (sem salvar linhas especiais em disco). O GCUPS _{P} representa a performance considerando a fórmula $\frac{mn-p}{t \times 10^9}$, onde p é o número de células que não processadas (*pruned*) por causa da otimização. O número de blocos *pruned* aumenta quanto maior for a similaridade, conforme podemos ver na coluna “%Pruned”. Note que no caso da comparação 5227K × 5229K, o ganho de desempenho chegou a 51,2%, com um total de 53,7% de blocos *pruned*.

A Tabela 7.2 mostra o tempo de execução de todas as comparações em todos os estágios, executados com todas as otimizações propostas. O tamanho do SRA (Seção 6.1.1) foi escolhido empiricamente para cada comparação. Comparando as tabelas 7.2 e 7.1, podemos constatar um *overhead* de 3 segundos por Gigabyte no SRA, gerando *overhead* total de aproximadamente 2%.



(a) $T = 128$



(b) $T = 128, T = 192$ e $T = 256$

Figura 7.3: Tempo de execução da comparação de $1044K \times 1073K$ variando o número de *threads* e blocos

A Figura 7.4 apresenta um gráfico em escala logarítmica com o tempo de execução do estágio 1 em relação ao número de células da matriz de programação dinâmica. Neste teste, o *Block Pruning* foi desabilitado. Os resultados mostram a escalabilidade do CUDAlign 2.1, onde o GCUPS é praticamente constante para sequências maiores que 1 MBP. Pela Tabela 7.2, é possível ver que as execuções estabilizam em torno de 28 GCUPS na placa GTX 560 Ti.

A Figura 7.5 ilustra os gráficos dos alinhamentos mais similares, provenientes do estágio de visualização (Estágio 6). A área cinza representa os blocos que foram descartados pela otimização *Block Pruning*. Note que quanto mais similares as sequências forem, maior o ganho de desempenho da otimização.

7.3 Conclusão do Capítulo

Ao analisar os resultados do CUDAlign 2.1, é evidente que a otimização *Block Pruning* permite acelerar bastante o cálculo da matriz de programação dinâmica quando as sequências possuem alto grau de similaridade. Neste contexto, decidimos explorar melhor essa otimização, investigando em detalhe os fatores que contribuem para o aumento da taxa de *pruning*. O detalhamento dessa investigação encontra-se no Capítulo 8, onde a otimização *Block Pruning* é estendida, sendo abordados vários aspectos teóricos sobre a sua eficácia.

Tabela 7.1: Tempo de execução (em segundos) do Estágio 1 do CUDAlign 2.1 sem salvar linhas especiais em disco. Os tempos foram apresentados com e sem a otimização de *block pruning*. O GCUPS original considera o número de células de toda a matriz. O GCUPS *pruned* (GCUPS_P) considera apenas as células que não foram descartadas.

| Comparação | Original | | Block Pruning | | | | % Pruned |
|---------------|----------|-------|---------------|--------------------|-------|-------|----------|
| | Tempo(s) | GCUPS | Tempo(s) | GCUPS _P | GCUPS | Ganho | |
| 543K×536K | 10,8 | 26,97 | 10,8 | 26,98 | 27,00 | 0,1% | <0,1% |
| 1044K×1073K | 40,3 | 27,81 | 36,2 | 27,54 | 30,95 | 10,1% | 11,0% |
| 3147K×3283K | 363,6 | 28,41 | 363,2 | 28,40 | 28,44 | 0,1% | 0,1% |
| 5227K×5229K | 962,4 | 28,40 | 469,5 | 26,95 | 58,21 | 51,2% | 53,7% |
| 7146K×5227K | 1309 | 28,53 | 1309 | 28,53 | 28,54 | <0,1% | <0,1% |
| 23012K×24544K | 19701 | 28,67 | 19694 | 28,67 | 28,68 | <0,1% | <0,1% |
| 59374K×23953K | 49634 | 28,65 | 46869 | 28,51 | 30,34 | 5,6% | 6,0% |
| 32799K×46944K | 53869 | 28,58 | 29133 | 27,44 | 52,85 | 45,9% | 48,1% |

Tabela 7.2: Tempo de execução (em segundos) de cada estágio do CUDAlign 2.1 na placa GTX 560 Ti, variando o tamanho da comparação. O tempo total inclui todos os estágios e o tempo de leitura das sequências.

| Comparação | SRA | Tempos de cada Estágio | | | | | Total | |
|---------------|------|------------------------|------|------|------|------|-------|-------|
| | | 1 | 2 | 3 | 4 | 5+6 | Tempo | GCUPS |
| 162K×172K | 5M | 1,2 | <0,1 | <0,1 | <0,1 | <0,1 | 2,1 | 13,26 |
| 543K×536K | 50M | 11,0 | <0,1 | <0,1 | <0,1 | <0,1 | 11,8 | 24,67 |
| 1044K×1073K | 250M | 37,1 | 1,9 | 0,6 | 2,8 | 0,2 | 43,4 | 25,82 |
| 3147K×3283K | 1G | 366 | <0,1 | <0,1 | 0,2 | <0,1 | 367 | 28,15 |
| 5227K×5229K | 3G | 480 | 38,6 | 7,3 | 26,1 | 4,7 | 558 | 48,98 |
| 7146K×5227K | 3G | 1320 | <0,1 | <0,1 | <0,1 | <0,1 | 1321 | 28,28 |
| 23012K×24544K | 10G | 19755 | 0,2 | <0,1 | 0,4 | <0,1 | 19757 | 28,59 |
| 59374K×23953K | 50G | 47039 | 52,9 | 5,5 | 11,0 | 0,9 | 47123 | 30,18 |
| 32799K×46944K | 50G | 29333 | 592 | 175 | 251 | 7,5 | 30369 | 50,70 |

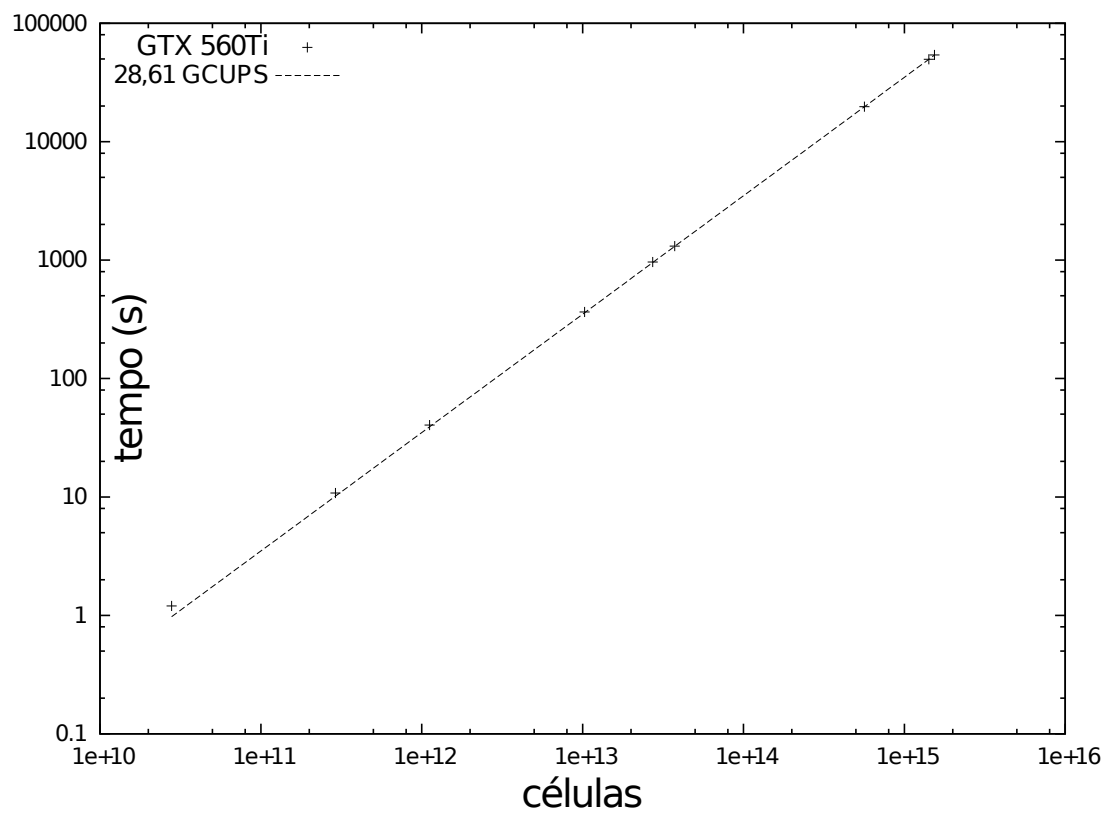
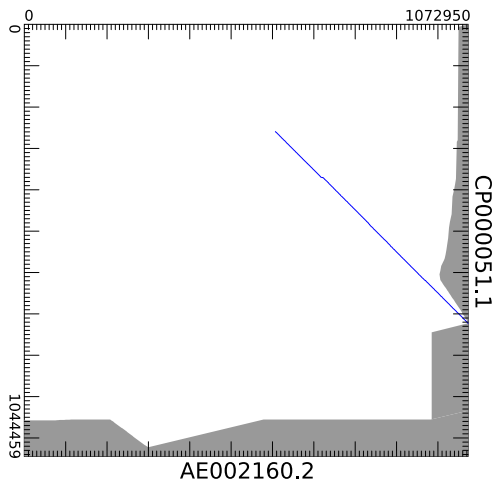
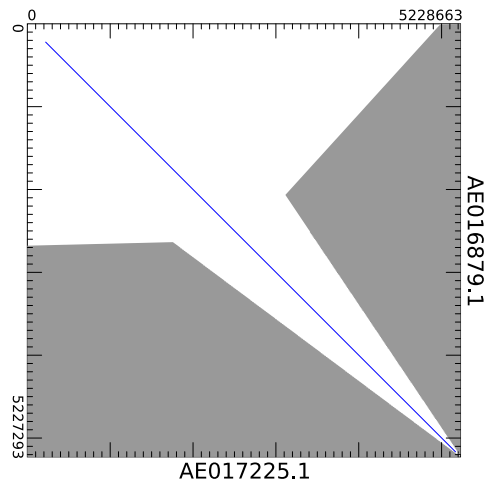


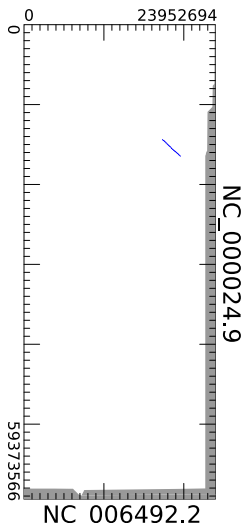
Figura 7.4: Tempo de execução \times tamanho da matriz (em células)



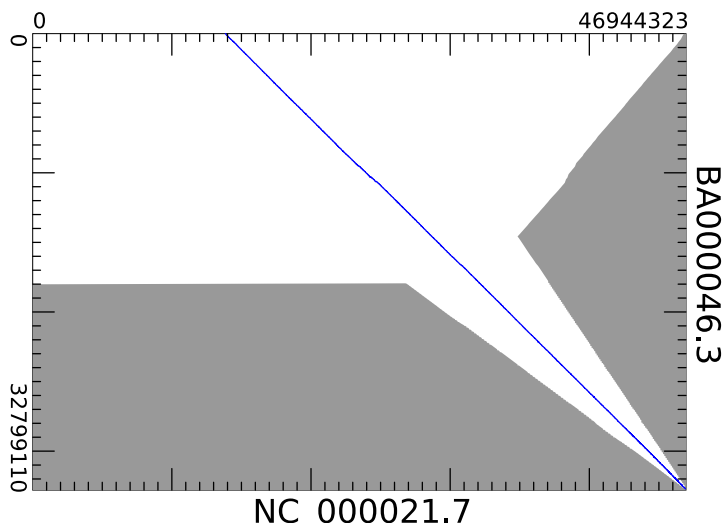
(a) $1044K \times 1073K$. 11,0% descartado.



(b) $5227K \times 5229K$. 53,7% descartado.



(c) $59374K \times 23953K$.
6,0% descartado.



(d) $32799K \times 46944K$. 48,1% descartado.

Figura 7.5: Gráficos de alguns alinhamentos, com a área *pruned* em cinza.

Capítulo 8

Formalização e Generalização do Descarte de Blocos (*Pruning*)

A matriz de programação dinâmica usada para a comparação de sequências biológicas é normalmente calculada com uma equação de recorrência que envolve operações de máximo sobre valores previamente calculados de três células adjacentes ($H_{i-1,j-1}$, $H_{i-1,j}$ e $H_{i,j-1}$). Podemos observar que a diferença entre os valores destas células adjacentes é restrita a um limite superior e a um limite inferior. Por meio dessa propriedade, conhecido o valor de uma célula qualquer, é possível determinar os valores máximo e mínimo de quaisquer outras células da matriz, adjacentes ou não. Quanto mais distante estiver uma célula com valor conhecido, maior será o intervalo de valores que uma célula poderá possuir. Se as duas células estiverem conectadas por um caminho de *traceback*, então o intervalo de valores será menor.

Com isso, é possível ignorar algumas células da matriz antes mesmo de calculá-las, conforme explicado no Capítulo 7. Este procedimento é capaz de reduzir o número de células processadas da matriz sem prejudicar a obtenção do alinhamento ótimo. Além disso, o *pruning* também pode se aplicar a blocos de células, diminuindo assim o custo computacional deste método. Este capítulo dedica-se a formalizar o *Block Pruning*, estendendo-o para diversas formas de processamento da matriz de programação dinâmica. Além disso, será mostrado que a eficácia do *pruning* depende: 1) da forma de processamento da matriz; 2) dos parâmetros de *match*, *mismatch* e *gap* da equação de recorrência; e 3) das características das sequências.

Neste capítulo, inicialmente serão apresentados alguns trabalhos relacionados que realizam descarte de células (Seção 8.1). Em seguida, serão definidos conceitos que serão utilizados ao longo do capítulo (Seção 8.2). A seguir, discorreremos sobre a diferença de valores entre células da matriz (Seção 8.3), com ênfase nos limites superior e inferior. O método de *pruning* proposto nesta tese, aplicado tanto para células como para blocos, será detalhado na Seção 8.4. Na Seção 8.5, será apresentado um arcabouço teórico para avaliação da eficácia do método de *pruning* em alguns cenários. A Seção 8.6 apresenta, por meio de simulações, uma previsão da área da matriz que será descartada durante o *pruning*. Finalmente, a Seção 8.7 conclui o capítulo.

8.1 Trabalhos relacionados

Na literatura, alguns trabalhos foram propostos para reduzir a área de processamento da matriz de programação dinâmica. Duas linhas de trabalho foram propostas, sendo a primeira relacionada a algoritmos em grafos e a segunda relacionada a algoritmos baseados em propriedades da matriz de programação dinâmica.

A primeira linha de trabalhos desenvolveu-se considerando a matriz de programação dinâmica como um grafo, onde cada célula é representada por nós e as dependências entre as células vizinhas são representadas como arestas. Desta forma, o problema de encontrar o alinhamento ótimo é transformado em um problema de caminho mínimo, descartando (*pruning*) nós que não contribuam para o resultado ótimo. Utilizando o algoritmo de *A-star* [133] com o auxílio de uma função heurística para estimar a distância de um nó de origem ao destino, é possível encontrar, de maneira mais eficiente, o melhor caminho que liga o início do alinhamento ao final do alinhamento, reduzindo assim o número de nós processados na matriz. Entretanto, o algoritmo *A-star* exige um consumo muito grande de memória, pois o algoritmo deve manter uma fila de prioridades contendo nós que estão sendo expandidos. Deste modo, o algoritmo *A-star* somente seria viável em sequências muito pequenas. O algoritmo *IDA-star* [134] amenizou o problema de memória, mas o seu funcionamento não é escalável no problema de comparação de sequências devido ao número exponencial de caminhos no *traceback*, inviabilizando o processamento de matrizes maiores que 10×10 [135]. Em um trabalho subsequente, o *DCFA-star* [136] aplicou técnicas de dividir para conquistar semelhantes ao de Hirschberg [12] para alinhar sequências de até 6.000 caracteres. O *DCFA* diminuiu o uso de memória, porém o tempo de processamento aumentou. Entretanto, o algoritmo chamado *Partial Expansion A-star* [137] foi capaz de reduzir ainda mais o consumo de memória, embora o tamanho médio das sequências testadas tenha sido menor que 500 caracteres.

A segunda linha de trabalhos aplica otimizações específicas para o processamento da matriz de programação dinâmica. Pesquisadores que adotaram esta linha observaram que os problemas em grafo, por mais otimizados que fossem, necessitam de muita memória para armazenar os nós. Visto que o procedimento de descarte é feito para cada nó expandido, o custo de *pruning* torna-se ainda maior. Fickett (Seção 2.2.7) propõe uma solução que processa a matriz descartando células que estejam fora de uma faixa de processamento. A largura desta faixa é inicialmente pequena, mas seu tamanho é incrementado até que o alinhamento ótimo seja encontrado. O *LBD-Align* [138] processa a matriz de programação dinâmica por diagonais, efetuando o procedimento de *pruning* somente nas extremidades de cada diagonal, de maneira adaptativa durante o cálculo da matriz. Desta forma, cria-se uma janela que encolhe ou expande de acordo com os valores das células, eliminando regiões que não contribuem para o score ótimo. O *DCLBD-Align* [138] aplica o método de dividir para conquistar para obter o alinhamento completo em memória linear. Ao contrário do algoritmo de Myers-Miller (Seção 2.2.5), o *DCLBD-Align* divide a matriz nas diagonais centrais em vez de na linha central. O trabalho em [132] reduz o número de células processadas no *traceback* excluindo células que obtiveram valores não positivos. Entretanto, o primeiro estágio do algoritmo processa a matriz em sua totalidade.

Até onde sabemos, existia somente um trabalho que aplica os conceitos de *pruning* sobre blocos de células da matriz de programação dinâmica. Esse trabalho, SW#[125], implementou o *Block Pruning* proposto no *CUDAAlign 2.1* [35] (Capítulo 6), que é uma

das contribuições desta tese. Além disso, os algoritmos de *pruning* existentes na literatura utilizam padrões de processamento pré-determinados, não existindo trabalho que estenda estes conceitos para mecanismos que processem a matriz em uma ordem genérica. Por fim, nenhum trabalho encontrado avalia matematicamente a eficácia destes métodos considerando as características das sequências, parâmetros de comparação e a forma de processamento da matriz.

8.2 Definições

Nesta seção, apresentaremos algumas definições comuns que serão utilizadas ao longo do capítulo. Por simplicidade, consideraremos inicialmente a equação de Needleman-Wunsch (Equação 2.9). Posteriormente, apresentaremos as diferenças das propriedades em relação às Equações de Smith-Waterman (Equação 2.10) e Gotoh (Equação 2.12). A Tabela 8.1 apresenta os nomes e os símbolos para cada uma das definições feitas em seguida.

Tabela 8.1: Definições

| Definição | Símbolo | Descrição |
|------------------|-------------------------------------|------------------------------------|
| Definição 8.2.1 | $H_{i-1,j-1}, H_{i-1,j}, H_{i,j-1}$ | Células Adjacentes |
| Definição 8.2.2 | \mathcal{G} | Grafo de Traceback |
| Definição 8.2.3 | $(diag), (up), (left)$ | Casos de dependência de dados |
| Definição 8.2.4 | $H_{i,j}^*$ | Célula de Referência |
| Definição 8.2.5 | Δ_i e Δ_j | Distâncias Vertical e Horizontal |
| Definição 8.2.6 | $H_{i+\Delta_i,j+\Delta_j}$ | Célula Qualquer |
| Definição 8.2.7 | $\bar{H}_{i+\Delta_i,j+\Delta_j}$ | Célula Conectada |
| Definição 8.2.8 | $\delta_{i,j}$ | Diferença entre Células |
| Definição 8.2.9 | $\bar{\delta}_{i,j}$ | Diferença entre Células Conectadas |
| Definição 8.2.10 | $H_{i,j}^{max}$ | Escore Derivado Máximo |
| Definição 8.2.11 | $H_{i,j}^{min}$ | Escore Derivado Mínimo |
| Definição 8.2.12 | $best_{i,j}$ | Melhor Escore Provisório |
| Definição 8.2.13 | $\phi_{i,j}$ | Iteração |

Definição 8.2.1 (Células Adjacentes) Uma célula $H_{i,j}$ possui 3 células adjacentes: $H_{i-1,j-1}$, $H_{i-1,j}$ e $H_{i,j-1}$, cada uma representando uma dependência na equação de recorrência.

Definição 8.2.2 (Grafo de *traceback*) Conforme visto na equação de Needleman-Wunsch (Equação 2.9), cada célula possui seu valor calculado por uma operação de máximo sobre valores previamente calculados de células adjacentes. O *grafo de traceback* representa as dependências que geraram o valor máximo em cada uma das células, indicando a direção que o *traceback* deverá percorrer. Definimos formalmente este grafo como $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, onde os nós \mathcal{V} representam cada uma das células da matriz de programação dinâmica e as arestas \mathcal{E} indicam quais células adjacentes produziram o valor máximo em cada célula, ou seja, $(a, b) \in \mathcal{E}$ se e somente se a célula b produz o valor máximo em a

considerando a equação de recorrência. A Figura 8.1 apresenta o grafo de *traceback* entre as sequências TTACACTT e TGCACACAGG. É possível observar que todas as células possuem 1, 2 ou 3 arestas de saída, exceto no caso da célula $H_{0,0}$, que não possui nenhuma aresta. As arestas em destaque representam os alinhamentos globais ótimos.

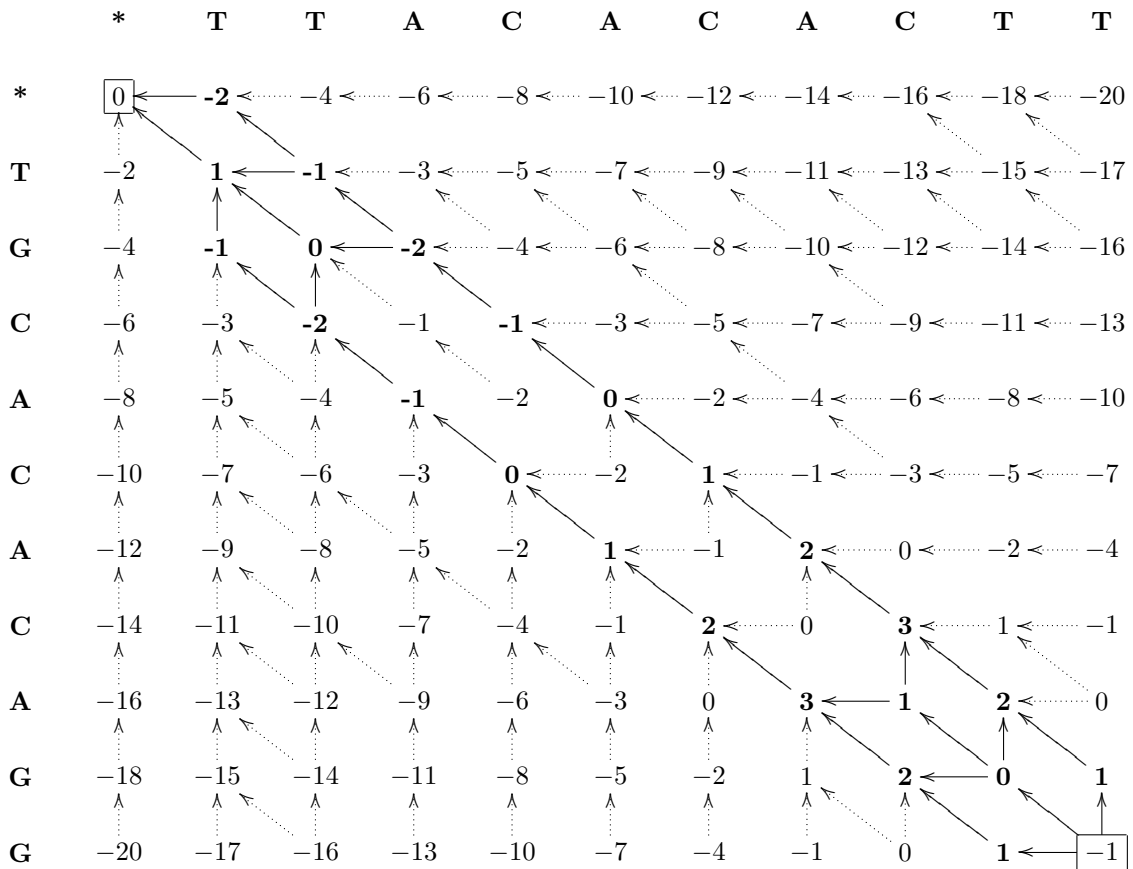


Figura 8.1: Grafo de *Traceback*.

Definição 8.2.3 (Casos de dependência (*diag*), (*up*) e (*left*)) Conforme visto na equação de Needleman-Wunsch (Equação 2.9), o valor da célula $H_{i,j}$ é determinado pelo valor máximo entre três casos de dependências, que chamaremos de (*diag*), (*up*) e (*left*), dependendo respectivamente de suas células adjacentes $H_{i-1,j-1}$, $H_{i-1,j}$ e $H_{i,j-1}$. Um ou mais casos simultâneos podem se aplicar a uma mesma célula, indicando as direções em que o *traceback* pode percorrer. A Figura 8.2 ilustra as dependências da equação, onde cada um dos três casos está representado como arestas do grafo de *traceback*.

Definição 8.2.4 (Célula de referência $H_{i,j}^*$) Chama-se de célula de referência uma célula com valor já conhecido, usada como ponto de referência durante alguma análise da matriz. Utiliza-se o símbolo \star para indicar a célula de referência.

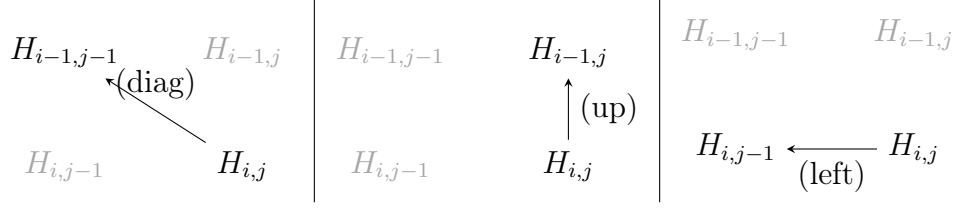


Figura 8.2: Casos de dependência de $H_{i,j}$.

Definição 8.2.5 (Distâncias vertical e horizontal Δ_i e Δ_j) As distâncias vertical e horizontal entre uma célula qualquer e a célula de referência serão denominadas Δ_i e Δ_j , respectivamente.

Definição 8.2.6 (Célula qualquer $H_{i+\Delta_i,j+\Delta_j}$) Determinada uma célula de referência $H_{i,j}^*$ com valor conhecido, chama-se de célula qualquer $H_{i+\Delta_i,j+\Delta_j}$ uma célula, com valor conhecido ou não, onde Δ_i e Δ_j são as distâncias vertical e horizontal para a célula de referência.

Definição 8.2.7 (Célula conectada $\bar{H}_{i+\Delta_i,j+\Delta_j}$) Determinada uma célula de referência $H_{i,j}^*$ com valor conhecido, diz-se que a célula $\bar{H}_{i+\Delta_i,j+\Delta_j}$ está conectada à célula de referência se existir um caminho de arestas entre elas no grafo de *traceback*. Desta forma, poderemos efetuar o *traceback* a partir da célula mais inferior até a outra célula, percorrendo um caminho de vértices conectados pelas arestas. A Figura 8.1 destaca em um retângulo as células $H_{0,0}$ e $H_{|S_0|,|S_1|}$, que estão conectadas por caminhos de *traceback*.

Definição 8.2.8 (Diferença entre células $\delta_{i+\Delta_i,j+\Delta_j}$) Considerando uma célula de referência $H_{i,j}^*$ de valor conhecido, $\delta_{i+\Delta_i,j+\Delta_j}$ é a diferença entre os valores das células $H_{i+\Delta_i,j+\Delta_j}$ e $H_{i,j}^*$, definida conforme a Equação 8.1.

$$\delta_{i+\Delta_i,j+\Delta_j} = H_{i+\Delta_i,j+\Delta_j} - H_{i,j}^* \quad (8.1)$$

Definição 8.2.9 (Diferença entre Células Conectadas $\bar{\delta}_{i+\Delta_i,j+\Delta_j}$) Se as células $H_{i,j}^*$ e $H_{i+\Delta_i,j+\Delta_j}$ forem conectadas pelo grafo de *traceback*, chama-se a diferença de valor entre elas de $\bar{\delta}_{i+\Delta_i,j+\Delta_j}$.

Definição 8.2.10 (Escore Derivado Máximo $H_{i,j}^{max}$) Dado que o valor da célula $H_{i,j}$ é conhecido, o *Escore Derivado Máximo* ($H_{i,j}^{max}$) é o maior escore que o alinhamento ótimo poderá possuir caso ele atravesse a célula $H_{i,j}$.

Definição 8.2.11 (Escore Derivado Mínimo $H_{i,j}^{min}$) Analogamente, o *Escore Derivado Mínimo* ($H_{i,j}^{min}$) é o menor escore que um alinhamento ótimo poderá produzir caso ele atravesse a célula $H_{i,j}$.

Definição 8.2.12 (Melhor Escore Provisório $best_{i,j}$) O melhor escore provisório $best_{i,j}$ é definido como o maior escore derivado mínimo $H_{i',j'}^{min}$ de todas as células (i', j') processadas antes do cálculo de $H_{i,j}$. Ao final do processamento da matriz, o valor $best_{m,n}$ da última célula é o escore ótimo real.

Definição 8.2.13 (Iteração $\phi_{i,j}$) Considerando que as células da matriz H são processadas por iterações, chamaremos de $\phi_{i,j}$ a iteração em que a célula $H_{i,j}$ foi processada. Desta forma, se $\phi_{i',j'} < \phi_{i,j}$ se e somente se $H_{i',j'}$ foi processada em uma iteração anterior a $H_{i,j}$.

8.3 Diferença de valores entre células da matriz

Nesta seção, apresentaremos fórmulas que definem limites superiores e inferiores entre os valores de células da matriz. No restante do capítulo, essas propriedades serão utilizadas para definir e analisar o procedimento de redução da área de processamento da matriz.

8.3.1 Limites das diferenças entre células conectadas ($\bar{\delta}$)

Considerando células adjacentes diretamente conectadas por arestas, define-se a diferença de valor $\bar{\delta}$ entre as células utilizando os próprios valores da equação de recorrência de NW: $\bar{\delta}_{i-1,j-1} = +mi$ ou $-ma$, $\bar{\delta}_{i-1,j} = +G$ e $\bar{\delta}_{i,j-1} = +G$ (Figura 8.3).

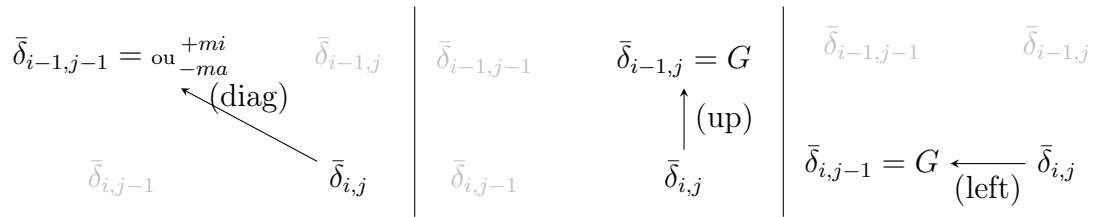


Figura 8.3: Diferença $\bar{\delta}$ entre células adjacentes conectadas por arestas.

Pode-se encontrar a diferença máxima entre duas células conectadas não adjacentes considerando todos os caminhos possíveis entre elas. Para existir um caminho entre duas células, a célula superior deve estar à esquerda da célula inferior, pois as equações de recorrência em estudo permitem que o *traceback* seja realizado somente para a esquerda, para cima ou para a diagonal superior esquerda. A Tabela 8.2 apresenta os valores máximos de $\bar{\delta}$ para a vizinhança de uma célula de referência $H_{i,j}^*$, indicada na tabela pelo símbolo \star .

A Inequação 8.2 generaliza a diferença máxima de $\bar{\delta}_{i+\Delta_i,j+\Delta_j}$ entre os valores de uma célula de referência $H_{i,j}^*$ (Definição 8.2.4) e uma célula qualquer $H_{i+\Delta_i,j+\Delta_j}$ (Definição 8.2.6), onde Δ_i e Δ_j são as distâncias vertical e horizontal para a célula de referência $H_{i,j}^*$.

$$\bar{\delta}_{i+\Delta_i,j+\Delta_j} \leq \begin{cases} -max(\Delta_i, \Delta_j)mi + |\Delta_i - \Delta_j|G: & \text{se } \Delta_i \leq 0 \text{ e } \Delta_j \leq 0 \\ min(\Delta_i, \Delta_j)ma - |\Delta_i - \Delta_j|G: & \text{se } \Delta_i \geq 0 \text{ e } \Delta_j \geq 0 \end{cases} \quad (8.2)$$

Tabela 8.2: Diferença máxima $\bar{\delta}$ entre células conectadas.

| | | $\leftarrow \Delta_j \rightarrow$ | | | | | | |
|------------|-----|-----------------------------------|-----------|-----------|-----------|-----------|-----------|-----------|
| | | -3 | -2 | -1 | 0 | +1 | +2 | +3 |
| Δ_i | ... | | | | | | | |
| | -3 | $+3mi$ | $+2mi+1G$ | $+1mi+2G$ | $+3G$ | --- | --- | --- |
| | -2 | $+2mi+1G$ | $+2mi$ | $+1mi+1G$ | $+2G$ | --- | --- | --- |
| | -1 | $+1mi+2G$ | $+1mi+1G$ | $+1mi$ | $+1G$ | --- | --- | --- |
| | 0 | $+3G$ | $+2G$ | $+1G$ | $0\star$ | $-1G$ | $-2G$ | $-3G$ |
| | +1 | --- | --- | --- | $-1G$ | $+1ma$ | $+1ma-1G$ | $+1ma-2G$ |
| | +2 | --- | --- | --- | $-2G$ | $+1ma-1G$ | $+2ma$ | $+2ma-1G$ |
| +3 | --- | --- | --- | $-3G$ | $+1ma-2G$ | $+2ma-1G$ | $+3ma$ | |

Por simetria, o valor mínimo de $\bar{\delta}_{i+\Delta_i, j+\Delta_j}$ é igual ao valor mínimo de $-\bar{\delta}_{i-\Delta_i, j-\Delta_j}$. Com isso, definimos o valor mínimo de $\bar{\delta}_{i+\Delta_i, j+\Delta_j}$ conforme a Inequação 8.3.

$$\bar{\delta}_{i+\Delta_i, j+\Delta_j} \geq \begin{cases} -\min(\Delta_i, \Delta_j)mi - |\Delta_i - \Delta_j|G: & \text{se } \Delta_i \geq 0 \text{ e } \Delta_j \geq 0 \\ \max(\Delta_i, \Delta_j)ma + |\Delta_i - \Delta_j|G: & \text{se } \Delta_i \leq 0 \text{ e } \Delta_j \leq 0 \end{cases} \quad (8.3)$$

Como exemplo, suponha que $ma = 1$, $mi = 3$ e $G = 5$. Podemos afirmar que uma célula qualquer $H_{i+\Delta_i, j+\Delta_j}$, conectada a uma célula de referência $H_{i,j}^*$, deslocada $\Delta_i = 100$ linhas abaixo e $\Delta_j = 70$ colunas à direita de $H_{i,j}^*$, terá uma diferença relativa (Definição 8.2.8) limitada em $-360 \leq \bar{\delta}_{i+\Delta_i, j+\Delta_j} \leq -80$. Considerando que o valor da célula de referência é $H_{i,j}^* = 1000$, teremos os limites $640 \leq \bar{H}_{i+\Delta_i, j+\Delta_j} \leq 920$.

8.3.2 Limites das diferenças entre células quaisquer (δ)

Nesta subseção consideraremos células da matriz que estejam conectadas ou não por um caminho de *traceback*. Inicialmente, definiremos os limites das diferenças entre células adjacentes. Em seguida definiremos os limites para células quaisquer da matriz.

Limite superior de células adjacentes: Pela equação de recorrência de NW (Equação 2.9), as inequações 8.4, 8.5 e 8.6 aplicam-se a qualquer um dos casos (diag), (up) e (left).

$$H_{i,j} \geq H_{i-1, j-1} - mi \quad (8.4)$$

$$H_{i,j} \geq H_{i-1, j} - G \quad (8.5)$$

$$H_{i,j} \geq H_{i, j-1} - G \quad (8.6)$$

Considerando Equação 8.1 (Definição 8.2.8), os limites superiores das diferenças das células vizinhas são (Figura 8.4): $\delta_{i-1, j-1} \leq +mi$, $\delta_{i-1, j} \leq +G$ e $\delta_{i, j-1} \leq +G$.

| | |
|------------------------------|--------------------------|
| $\delta_{i-1, j-1} \leq +mi$ | $\delta_{i-1, j} \leq G$ |
| $\delta_{i, j-1} \leq G$ | $\delta_{i, j} = 0$ |

Figura 8.4: Limites superiores das células adjacentes

Limite inferior de células adjacentes: Para determinar os limites inferiores, provaremos o Teorema 8.3.1. Lembramos que, por definição, G e ma são positivos. Também assumiremos que $mi < 2G$, caso contrário todas as ocorrência de *mismatches* seriam substituídas por dois *gaps*.

Teorema 8.3.1 $H_{i,j} \leq H_{i-1,j} + ma + G$, $H_{i,j} \leq H_{i,j-1} + ma + G$ e $H_{i,j} \leq H_{i-1,j-1} + ma$.

Demonstração. O teorema será provado por indução nas linhas e colunas da matriz H .

Base de indução: Conforme visto na Seção 2.2.1, a primeira linha ($i = 0$) e a primeira coluna ($j = 0$) da matriz de programação dinâmica são iniciadas com os valores $H_{i,0} = -i \cdot G$ e $H_{0,j} = -j \cdot G$ respectivamente. Além disso, por meio das Equações 8.2 e 8.3, sabe-se que os valores das células da segunda linha ($i = 1$) e da segunda coluna ($j = 1$) são restritos aos limites definidos das Equações 8.7 e 8.8, respectivamente.

$$(1 - j)G - mi \leq H_{1,j} \leq (1 - j)G + ma \quad (8.7)$$

$$(1 - i)G - mi \leq H_{i,1} \leq (1 - i)G + ma \quad (8.8)$$

Os possíveis valores das duas primeiras linhas e colunas estão ilustrados na Figura 8.5.

| | j=0 | j=1 | j=2 | j=3 | ... | jG | ... | nG |
|-----|-----|--------------------------|------------------|------------------|-----|--------------------------|-----|--------------------------|
| i=0 | 0 | -1G | -2G | -3G | ... | -jG | ... | -nG |
| i=1 | -1G | +ma -mi | -1G+ma -1G-mi | -2G+ma -2G-mi | ... | -(j-1)G+ma -(j-1)G-mi | ... | -(n-1)G+ma -(n-1)G-mi |
| i=2 | -2G | -1G+ma -1G-mi | | | | | | |
| i=3 | -3G | -2G+ma -2G-mi | | | | | | |
| | ... | ... | | | | | | |
| | -iG | -(i-1)G+ma -(i-1)G-mi | | | | | | |
| | ... | ... | | | | | | |
| i=m | -mG | -(m-1)G+ma -(m-1)G-mi | | | | | | |

Figura 8.5: Possíveis valores nas duas primeiras linhas e colunas.

Considerando as características apresentadas das duas primeiras linhas da matriz, a Figura 8.6 apresenta, para qualquer $1 \leq j \leq n$, o limite superior de $H_{1,j}$, o limite inferior de $H_{1,j-1}$ e os valores das células $H_{0,j-1}$ e $H_{0,j}$.

| | |
|--------------------------------|------------------------------|
| $H_{0,j-1} = (1 - j)G$ | $H_{0,j} = -jG$ |
| $H_{1,j-1} \geq (2 - j)G - mi$ | $H_{1,j} \leq (1 - j)G + ma$ |

Figura 8.6: Limites das células adjacentes a $H_{1,j}$

Com os valores apresentados na Figura 8.6, o limite superior da célula $H_{1,j}$ pode ser expresso conforme as Equações 8.9, 8.10 e 8.11, que validam a base da indução para as células da segunda linha ($i = 1$). Em relação à Equação 8.11, ressalta-se que $mi < 2G$.

$$H_{1,j} \leq H_{0,j-1} + ma \quad (8.9)$$

$$H_{1,j} \leq H_{0,j} + ma + G \quad (8.10)$$

$$H_{1,j} \leq H_{1,j-1} + ma - G + mi \leq H_{1,j-1} + ma + G \quad (8.11)$$

Por simetria, chega-se a resultados análogos para as células da segunda coluna ($j = 1$).

Hipótese de Indução: Dada uma célula $H_{i,j}$, consideramos que o teorema é verdadeiro para suas células adjacentes $H_{i-1,j}$ (Equação 8.12), $H_{i,j-1}$ (Equação 8.13) e $H_{i-1,j-1}$ (Equação 8.14).

$$H_{i-1,j} \leq H_{i-1,j-1} + ma + G \quad (8.12)$$

$$H_{i,j-1} \leq H_{i-1,j-1} + ma + G \quad (8.13)$$

$$H_{i-1,j-1} \leq H_{i-2,j-2} + ma \quad (8.14)$$

Passo de Indução: A equação de recorrência do NW (Equação 2.9) limita-se ao valor máximo de todos os seus termos. Então, sabendo que o valor da função $sbt(S_0[i], S_1[j])$ é $+ma$ ou $-mi$ e que, pela hipótese de indução, os valores máximos de $H_{i-1,j}$ e $H_{i,j-1}$ estão respectivamente definidos nas Equações 8.12 e 8.13, podemos concluir que $H_{i,j} \leq H_{i-1,j-1} + ma$ (Equação 8.15).

$$H_{i,j} = \max \begin{cases} H_{i-1,j-1} + sbt(S_0[i], S_1[j]) \\ H_{i-1,j} - G \\ H_{i,j-1} - G \end{cases}$$

$$H_{i,j} \leq \max \begin{cases} H_{i-1,j-1} + ma \\ (H_{i-1,j-1} + ma + G) - G \\ (H_{i-1,j-1} + ma + G) - G \end{cases}$$

$$H_{i,j} \leq H_{i-1,j-1} + ma \quad (8.15)$$

Pela equação de recorrência de NW, temos as Inequações 8.16 e 8.17.

$$H_{i-1,j-1} \leq H_{i,j-1} + G \quad (8.16)$$

$$H_{i-1,j-1} \leq H_{i-1,j} + G \quad (8.17)$$

Aplicando as Inequações 8.16 e 8.17 na Equação 8.15, temos as Equações 8.18 e 8.19 que provam que o passo da indução está correto.

$$H_{i,j} \leq H_{i-1,j-1} + ma \leq H_{i-1,j} + G + ma \quad (8.18)$$

$$H_{i,j} \leq H_{i-1,j-1} + ma \leq H_{i,j-1} + G + ma \quad (8.19)$$

□

Pelo Teorema 8.3.1 e Definição 8.2.8 temos o Corolário 8.3.2.

Corolário 8.3.2 Os limites inferiores das diferenças das células vizinhas são (Figura 8.7): $\delta_{i-1,j-1} \geq -ma$, $\delta_{i-1,j} \geq -ma - G$ e $\delta_{i,j-1} \geq -ma - G$.

$$\begin{array}{|c|c|} \hline \delta_{i-1,j-1} \geq -ma & \delta_{i-1,j} \geq -ma - G \\ \hline \delta_{i,j-1} \geq -ma - G & \delta_{i,j} = 0 \\ \hline \end{array}$$

Figura 8.7: Limites inferiores das células adjacentes

Limites das diferenças entre células não adjacentes: Utilizando os limites definidos pelas Figuras 8.4 e 8.7, podemos estendê-los para encontrar a diferença máxima entre duas células não adjacentes da matriz. A Tabela 8.3 apresenta os limites máximos entre as diferenças entre células (adjacentes ou não), sendo que o símbolo \star indica a célula de referência $H_{i,j}^*$.

Tabela 8.3: Diferença máxima $\delta_{i+\Delta_i,j+\Delta_j}$ entre células.

| | | $\leftarrow \Delta_j \rightarrow$ | | | | | | |
|--|-----------|-----------------------------------|-----------|-----------|-----------|-----------|-----------|-----------|
| | | -3 | -2 | -1 | 0 | +1 | +2 | +3 |
| \uparrow Δ_i \downarrow | ... | | | | | | | |
| | -3 | $+3mi$ | $+2mi+1G$ | $+1mi+2G$ | $+3G$ | $+1ma+4G$ | $+2ma+5G$ | $+3ma+6G$ |
| | -2 | $+2mi+1G$ | $+2mi$ | $+1mi+1G$ | $+2G$ | $+1ma+3G$ | $+2ma+4G$ | $+3ma+5G$ |
| | -1 | $+1mi+2G$ | $+1mi+1G$ | $+1mi$ | $+1G$ | $+1ma+2G$ | $+2ma+3G$ | $+3ma+4G$ |
| | 0 | $+3G$ | $+2G$ | $+1G$ | $0\star$ | $+1ma+1G$ | $+2ma+2G$ | $+3ma+3G$ |
| | +1 | $+1ma+4G$ | $+1ma+3G$ | $+1ma+2G$ | $+1ma+1G$ | $+1ma$ | $+2ma+1G$ | $+3ma+2G$ |
| | +2 | $+2ma+5G$ | $+2ma+4G$ | $+2ma+3G$ | $+2ma+2G$ | $+2ma+1G$ | $+2ma$ | $+3ma+1G$ |
| +3 | $+3ma+6G$ | $+3ma+5G$ | $+3ma+4G$ | $+3ma+3G$ | $+3ma+2G$ | $+2ma+2G$ | $+3ma$ | |

A Inequação 8.20 generaliza a diferença máxima de $\delta_{i+\Delta_i,j+\Delta_j}$ entre os valores de uma célula de referência $H_{i,j}^*$ (Definição 8.2.4) e uma célula qualquer $H_{i+\Delta_i,j+\Delta_j}$ (Definição 8.2.6), onde Δ_i e Δ_j são as distâncias vertical e horizontal para a célula de referência $H_{i,j}^*$.

$$\delta_{i+\Delta_i,j+\Delta_j} \leq \begin{cases} -\max(\Delta_i, \Delta_j)mi + |\Delta_i - \Delta_j|G: & \text{se } \Delta_i < 0 \text{ e } \Delta_j < 0 \\ \max(\Delta_i, \Delta_j)ma + |\Delta_i - \Delta_j|G: & \text{se } \Delta_i \geq 0 \text{ ou } \Delta_j \geq 0 \end{cases} \quad (8.20)$$

Por simetria, o valor mínimo de $\delta_{i+\Delta_i,j+\Delta_j}$ é igual ao valor mínimo de $-\delta_{i-\Delta_i,j-\Delta_j}$. Com isso, definimos o valor mínimo de $\delta_{i+\Delta_i,j+\Delta_j}$ conforme a Inequação 8.21.

$$\delta_{i+\Delta_i,j+\Delta_j} \geq \begin{cases} -\min(\Delta_i, \Delta_j)mi - |\Delta_i - \Delta_j|G: & \text{se } \Delta_i > 0 \text{ e } \Delta_j > 0 \\ \min(\Delta_i, \Delta_j)ma - |\Delta_i - \Delta_j|G: & \text{se } \Delta_i \leq 0 \text{ ou } \Delta_j \leq 0 \end{cases} \quad (8.21)$$

Como exemplo, suponha que $ma = 1$, $mi = 3$ e $G = 5$. Podemos afirmar que uma célula qualquer $H_{i+\Delta_i,j+\Delta_j}$ deslocada $\Delta_i = 100$ linhas abaixo e $\Delta_j = 70$ colunas à direita de uma célula de referência $H_{i,j}^*$ terá uma diferença relativa (Definição 8.2.8) limitada em

$-360 \leq \delta_{i+\Delta_i, j+\Delta_j} \leq 250$. Considerando que o valor da célula de referência é $H_{i,j}^* = 1000$, teremos os limites $640 \leq H_{i+\Delta_i, j+\Delta_j} \leq 1250$.

8.3.3 Outras equações de recorrência

Nas seções anteriores, analisamos as diferenças máxima e mínima entre células da matriz calculada pelo algoritmo de Needleman-Wunsch (NW), que produz o alinhamento global ótimo. Para outras equações de recorrência como o Smith-Waterman (Seção 2.2.2) e o Gotoh (Seção 2.2.3), a análise das diferenças é bastante similar.

Smith-Waterman (SW): A equação de SW difere do NW pois as células calculadas no SW são limitadas a valores não negativos ($H_{i,j} \geq 0$). Com isso, a diferença entre uma célula de referência $H_{i,j}^*$ e uma célula qualquer $H_{i+\Delta_i, j+\Delta_j}$ será sempre menor que o próprio valor da célula de referência $H_{i,j}^*$ (Inequação 8.22).

$$\begin{aligned} \delta_{i+\Delta_i, j+\Delta_j} &= \underbrace{H_{i+\Delta_i, j+\Delta_j} - H_{i,j}^*}_{\geq 0} \\ \delta_{i+\Delta_i, j+\Delta_j} &\geq -H_{i,j}^* \end{aligned} \quad (8.22)$$

Ressalta-se que os limites inferiores das diferenças $\delta_{i+\Delta_i, j+\Delta_j}$ aplicadas para o NW (Equações 8.3 e 8.21) também se aplicam ao SW, sendo que a Inequação 8.22 torna-se um limite inferior mais estrito quando $H_{i,j}^*$ for menor que o valor definido pela Inequação 8.21.

Gotoh (affine-gap): Para adaptar as diferenças máxima e mínima para equações com *affine-gap*, podemos substituir a variável G das equações apresentadas na Seção 8.3.1 pela penalidade do primeiro gap $\gamma(1) = G_{first}$, sabendo que o primeiro gap de uma sequência de gaps possui penalidade maior que os demais (i.e. $G_{first} > G_{ext}$). Desta forma, as equações que apresentam as diferenças máximas e mínimas do NW (Equações 8.2, 8.3, 8.20 e 8.21) também se aplicam ao Gotoh, embora tornando-se limites não estritos.

8.3.4 Escore derivado de uma célula ($H_{i,j}^{max}$ e $H_{i,j}^{min}$)

Para encontrar os limites superior e inferior dos escores derivados de uma célula (Definições 8.2.10 e 8.2.11), devemos considerar o tipo de alinhamento escolhido. Nesta seção, serão definidos os limites para alinhamentos globais e locais. Neste contexto, chamaremos de $m = |S_0|$ e $n = |S_1|$ os tamanhos das sequências, Δ_i a distância vertical entre a célula $H_{i,j}$ e a última linha da matriz ($\Delta_i = m - i$) e Δ_j a distância horizontal entre esta célula e a última coluna da matriz ($\Delta_j = n - j$).

Alinhamento Global: No caso do alinhamento global, o alinhamento ótimo termina, por definição, na última célula da matriz $H_{m,n}$. Considerando a Equação 8.2, o valor de $H_{i,j}^{max}$ pode ser definido conforme a Equação 8.23.

$$\begin{aligned} H_{m,n} &= H_{i,j} + \bar{\delta}(m, n) \\ H_{m,n} &\leq \underbrace{H_{i,j} + \min(m - i, n - j) \cdot ma - |(m - i) - (n - j)| \cdot G}_{=H_{i,j}^{max}} \end{aligned} \quad (8.23)$$

Analogamente, considerando a Inequação 8.3, o valor de $H_{i,j}^{min}$ pode ser definido conforme a Equação 8.24.

$$\begin{aligned} H_{m,n} &= H_{i,j} + \bar{\delta}(m, n) \\ H_{m,n} &\geq \underbrace{H_{i,j} - \min(m-i, n-j) \cdot mi - |(m-i) - (n-j)| \cdot G}_{=H_{i,j}^{min}} \end{aligned} \quad (8.24)$$

Alinhamento Local: No caso do alinhamento local, o alinhamento ótimo pode terminar em qualquer célula da matriz. Devemos então considerar o valor máximo possível de todas as células que possam estar conectadas a $H_{i,j}$. Considerando a Inequação 8.2, o valor de $H_{i,j}^{max}$ pode ser definido conforme a Equação 8.25.

$$\begin{aligned} H_{i,j}^{max} &= H_{i,j} + \max_{(i',j') \leq (i,j) \leq (m,n)} \bar{\delta}(i', j') \\ H_{i,j}^{max} &= H_{i,j} + \min(m-i, n-j) \cdot ma \end{aligned} \quad (8.25)$$

Visto que, no alinhamento local, o alinhamento pode encerrar-se na própria célula $H_{i,j}$, então o valor mínimo $H_{i,j}^{min}$ de um alinhamento que passe por essa célula é o próprio valor $H_{i,j}$, conforme descrito na Equação 8.26.

$$H_{i,j}^{min} = H_{i,j} \quad (8.26)$$

A Figura 8.8 ilustra geometricamente os valores de $H_{i,j}^{max}$ e $H_{i,j}^{min}$ para alinhamentos globais e locais. As linhas que saem da célula $H_{i,j}$ representam os melhores (Figuras 8.8(a) e 8.8(c)) e os piores (Figuras 8.8(b) e 8.8(d)) cenários de alinhamentos, tais como os casos onde ocorrem *matches* ou *mismatches* entre todas os caracteres das sequências.

8.4 Método de *Pruning*

O método de *pruning* proposto nesta tese consiste em descartar células da matriz de programação dinâmica que jamais poderão contribuir para o alinhamento ótimo. Em outras palavras, uma célula pode ser descartada se seu escore derivado máximo $H_{i,j}^{max}$ (Seção 8.3.4) for menor que um limite inferior *bound* do alinhamento ótimo. A Inequação 8.27 será chamada de **condição de *pruning***.

$$H_{i,j}^{max} \leq bound \quad (8.27)$$

Ao avaliar a condição de *pruning* durante o cálculo da matriz, podemos reduzir o número de células processadas, acelerando assim o tempo de processamento. Nesta seção formalizaremos algumas definições adicionais para o método de *pruning* e apresentaremos fórmulas capazes de estimar a eficácia do método em determinadas condições.

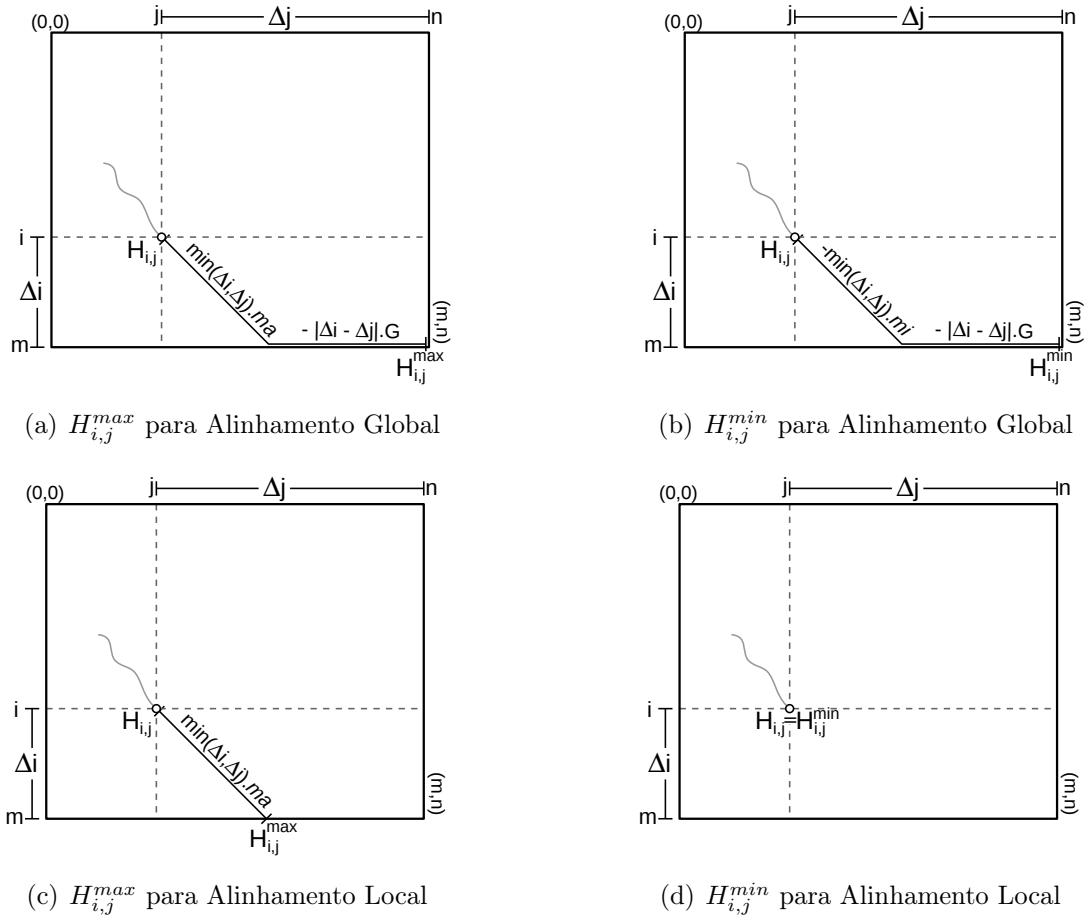


Figura 8.8: Representações geométricas de $H_{i,j}^{max}$ e $H_{i,j}^{min}$.

8.4.1 Definições

Definição 8.4.1 (Células *Prunable*) Se a condição expressa na Inequação 8.27 for atendida para uma célula $H_{i,j}$, então $H_{i,j}$ é dita uma “célula *prunable*”, pois seu cálculo poderia ter sido ignorado visto que nenhum alinhamento ótimo jamais passará por ela.

Definição 8.4.2 (Células *Pruned*) Chama-se de “célula *pruned*” uma célula que foi identificada como *prunable* antes mesmo de ser calculado o seu valor $H_{i,j}$. Ao ignorarmos o cálculo destas células, o número de células processadas da matriz é reduzido. A verificação de uma célula *pruned* é feita observando os valores de suas dependências. Caso as 3 células adjacentes ($H_{i-1,j-1}$, $H_{i-1,j}$ e $H_{i,j-1}$) sejam todas *prunable*, a célula $H_{i,j}$ será obrigatoriamente *prunable* (i.e. célula *pruned*), sem a necessidade de calcularmos o valor de $H_{i,j}$. Um caso especial ocorre quando as células *prunable* estão na primeira coluna ou primeira linha. Se a célula $H_{y,0}$ for *prunable*, todas as células restantes $H_{i>y,0}$ também serão *prunable* por indução. Analogamente, se a célula $H_{0,x}$ for *prunable*, todas as células restantes $H_{0,j>x}$ também serão *prunable* por indução. A Figura 8.9 apresenta um exemplo onde as células cinza claro representam as células *prunable* e as células cinza escuro representam as células que foram consideradas *prunable* por indução e, conseqüentemente,

podem ser ignoradas no processamento da matriz. As células com um círculo no centro foram processadas e as demais (células *pruned*) foram ignoradas.

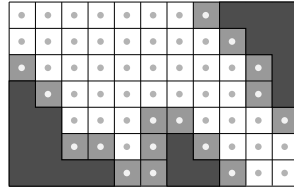


Figura 8.9: Visualização das induções na condição de *pruning*.

Definição 8.4.3 (Limite inferior *bound*) Por meio da Equação 8.27, serão ignorados os alinhamentos cujo escore sejam menores que o limite inferior *bound*.

Definição 8.4.4 (Limite inferior inicial $bound_0$) O limite inferior *bound* possui um valor inicial $bound_0$, que pode ser definido das seguintes maneiras:

- **Irrestrito:** Define-se $bound_0 = -\infty$ de modo a não restringir, inicialmente, nenhum alinhamento.
- **Manual:** O usuário define $bound_0$ de modo a restringir a obtenção de alinhamentos somente com escores maiores que este limite.
- **Heurística:** Um algoritmo heurístico encontra um alinhamento não necessariamente ótimo e o escore deste alinhamento é utilizado como limite inferior.
- **Oráculo:** Um conceito teórico onde o escore ótimo é conhecido antes da execução do algoritmo.

Definição 8.4.5 (Atualização do limite inferior) O limite inferior *bound* pode ser atualizado com novos valores $bound_1, bound_2, \dots$, nas seguintes situações:

- **por célula:** O valor de *bound* é atualizado com o maior escore derivado mínimo $H_{i,j}^{min}$ (Equações 8.24 ou 8.26) imediatamente após o cálculo de cada célula.
- **por bloco:** A atualização é feita semelhantemente à atualização por célula. Entretanto, a atualização considera apenas o escore derivado mínimo $H_{i,j}^{min}$ (Equações 8.24 ou 8.26) da célula com o maior escore deste bloco. Comparando com a atualização por célula, o cálculo de $H_{i,j}^{min}$ não precisa ser feito para todas as células, reduzindo o *overhead* deste cálculo.
- **por tempo:** Atualiza-se o valor de *bound* a cada intervalo de tempo. A atualização considera o escore derivado mínimo $H_{i,j}^{min}$ (Equações 8.24 ou 8.26) da célula com o maior escore encontrado até o momento. Visto que a atualização é feita periodicamente, o cálculo de $H_{i,j}^{min}$ pode ser feito através de alguma heurística em vez de utilizar as Equações 8.24 e 8.26. Embora a heurística possa consumir mais tempo do que o cálculo das equações, o valor

de *bound* pode convergir mais rapidamente para o escore ótimo, melhorando a eficácia do método de *pruning*.

- **por execução:** Esta forma de atualização se aplica quando $bound_0$ é definido manualmente, pois a primeira execução pode não encontrar nenhum alinhamento. Neste caso, a matriz de programação dinâmica é calculada em várias execuções até que um alinhamento com escore maior que *bound* seja encontrado. Para cada nova execução, atualiza-se o valor de *bound* com valores sucessivamente menores ($bound_0 > bound_1 > bound_2 > \dots$).

8.4.2 Algoritmos de *Pruning*

Nesta seção serão apresentados quatro algoritmos de *pruning* (Algoritmos 2 a 5). Por simplicidade, apenas os casos para alinhamento local serão tratados, sendo que os ajustes necessários para abordar alinhamentos globais estão citados na Seção 8.3.3. Os quatro algoritmos de *pruning* propostos são os seguintes:

- (Algoritmo 2) **Específico por linha:** Aplica-se somente ao processamento por linha ou por coluna. A complexidade de memória é constante $O(1)$.
- (Algoritmo 3) **Específico por diagonal:** Aplica-se somente ao processamento por diagonal. A complexidade de memória é constante $O(1)$.
- (Algoritmo 4) **Genérico quadrático:** Aplica-se a qualquer forma de processamento da matriz. A complexidade de memória é quadrática $O(m \cdot n)$.
- (Algoritmo 5) **Genérico linear:** Aplica-se a qualquer forma de processamento da matriz. A complexidade de memória é linear $O(m + n)$.

As formas de processamento da matriz são ilustradas na Figura 8.10.

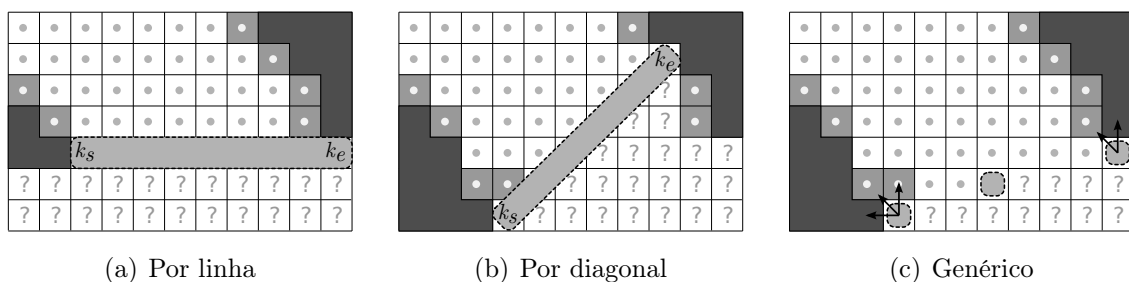


Figura 8.10: Algoritmos de *Pruning*

Inicialmente, as características comuns aos 4 algoritmos serão apresentadas. Em seguida, as características individuais serão abordadas separadamente.

Características comuns : O Algoritmo 1 apresenta um esqueleto que será utilizado de maneira comum em todos os demais algoritmos de *pruning* (Algoritmos 2 a 5). Neste esqueleto, estão apresentados os códigos comuns e algumas áreas que serão substituídas por códigos específicos a cada um dos algoritmos de *pruning*.

O esqueleto é dividido em 4 partes: 1) inicialização; 2) função ISPRUNABLE; 3) função ISPRUNED; e 4) procedimento PRUNINGUPDATE.

Algorithm 1 Esqueleto dos Algoritmos de *Pruning*

```
1:  $bound \leftarrow -\infty$ 
2: ... Inicializações Específicas ...

3: function ISPRUNABLE( $i, j, H_{i,j}, bound$ )
4:    $H_{i,j}^{max} \leftarrow H_{i,j} + \min(m - i, n - j) \cdot ma$ 
5:   return  $H_{i,j}^{max} \leq bound$ 
6: end function

7: function ISPRUNED( $i, j$ )
8:   ... Código Específico ...
9: end function

10: procedure PRUNINGUPDATE( $i, j, H_{i,j}$ )
11:   ... Código Específico ...
12: end procedure
```

A inicialização é feita nas primeiras linhas dos algoritmos, onde estão declarados os requisitos de memória e os valores iniciais de suas variáveis. Assume-se, sem perda de generalidade, que o limite inferior (Definição 8.4.5) é sempre inicializado com $bound_0 = -\infty$. Nos Algoritmos 2 a 5 pode-se observar as inicializações das variáveis específicas de cada método, onde os dois métodos genéricos possuem complexidade de memória em $O(m \cdot n)$ (quadrático) e $O(m + n)$ (linear) e os algoritmos específicos utilizam memória em $O(1)$ (constante).

A função ISPRUNABLE (linhas 3-6), responsável por identificar se uma célula é *prunable* ou não (Definição 8.4.1), será utilizada para todos os algoritmos de *pruning*. Essa função é consequência direta das Equações 8.25 e 8.27.

A função ISPRUNED é chamada antes do cálculo de cada célula e é responsável por identificar se uma célula poderá ser ignorada (*pruned*) sem comprometer o resultado ótimo (Definição 8.4.2). Cada algoritmo de *pruning* terá uma forma diferente de identificar células *pruned*.

O procedimento PRUNINGUPDATE é chamado após o cálculo de cada célula. Este procedimento é responsável por atualizar as variáveis que auxiliam o *pruning* e por atualizar o valor do limite inferior $bound$ (Definição 8.4.5). Cada algoritmo de *pruning* terá uma forma diferente de atualizar as suas variáveis.

Algoritmo Específico por Linha: Considerando que a matriz é processada por linhas (ou analogamente por coluna), o procedimento de *Pruning* funciona da seguinte maneira. Chamamos de janela não *prunable* o intervalo $[k_s..k_e]$ de colunas que devem ser processadas em uma determinada linha. Inicialmente, definimos $(k_s, k_e) = (0, n)$. Para cada linha i , o algoritmo calcula todas as células ($i, j \in [k_s..k_e]$) e, então, ele atualiza a janela não *prunable* para os valores $[k'_s..k'_e]$, onde os valores k'_s e k'_e são, respectivamente, a primeira e a última célula não-*prunable* desta linha.

Assumindo que as células ($i - 1, j \in [0..k_s]$) são *prunable*, então todas as células ($i, j \in [0..k_s]$) também serão e, por indução, todas as células ($i' \geq i, j \in [0..k_s]$) também serão. Se todas as células ($i - 1, j \in [k_e..n]$) são *prunable* e a célula ($i, x \in [k_e..n]$) é *prunable*, logo todas as células ($i, j \in [x + 1..m]$) também serão *prunable*. Se a célula ($i, k_e + 1$) for *prunable*, então o novo valor k'_e será a última célula não-*prunable* no intervalo ($i, j \in [k_s..k_e]$), onde $k'_e \leq k_e$. Caso contrário, devemos calcular as células restantes da

linha i até que encontremos a primeira célula *prunable* ($i, x \in [k_e + 2..n]$), de forma que o novo valor de k'_e será $x - 1$. As células ($i, j \in [0..k'_s]$) são chamadas de células *prunable* à esquerda e as células no intervalo ($i, j \in [k'_e + 1..n]$) são chamadas de células *prunable* à direita.

No Algoritmo 2, a função ISPRUNED (linhas 20-22) simplesmente retorna um booleano indicando se a coluna j está fora da janela $[k_s..k_e]$. O procedimento PRUNINGUPDATE (linhas 2-19) apresenta a atualização da janela de *pruning* após o cálculo de todas as células $H_{i,[0..n]}$ da linha i . A linha 5 indica a atualização do valor k_s e as linhas 10 e 15 as atualizações do valor k_e .

Algorithm 2 Algoritmo Específico por Linha

```

1:  $(k_s, k_e) \leftarrow (0, n)$ 
2: procedure PRUNINGUPDATE( $i, H_{i,[0..n]}$ )
3:    $bound \leftarrow \max(bound, H_{i,[0..n]})$ 
4:   while  $k_s < n$  and IsPrunable( $i, k_s, H_{i,k_s}, bound$ ) do
5:      $k_s \leftarrow k_s + 1$  ▷ aumento da área pruning à esquerda
6:   end while
7:   if  $k_e < n$  and  $\neg$ IsPrunable( $i, k_e, H_{i,k_e}, bound$ ) then
8:      $k_e \leftarrow k_e + 1$ 
9:     while  $k_s < B$  and  $\neg$ IsPrunable( $i, k_e, H_{i,k_e}, bound$ ) do
10:       $k_e \leftarrow k_e + 1$  ▷ redução da área pruning à direita
11:    end while
12:   else
13:      $k_e \leftarrow k_e - 1$ ;
14:     while  $k_e \geq k_s$  and IsPrunable( $i, k_e, H_{i,k_e}, bound$ ) do
15:        $k_e \leftarrow k_e - 1$  ▷ aumento da área pruning à direita
16:     end while
17:      $k_e \leftarrow k_e + 1$ ;
18:   end if
19: end procedure
20: function ISPRUNED( $j$ )
21:   return  $j < k_s$  or  $j > k_e$ 
22: end function

```

Algoritmo Específico por Diagonal: O Algoritmo 3 é similar ao Algoritmo 2, com a principal diferença em que a janela $[k_s..k_e]$ indica quais os elementos da diagonal atual deverá ser processada. A função ISPRUNED (linhas 17-19) é idêntica a do Algoritmo 2. O procedimento PRUNINGUPDATE (linhas 2-19) difere em dois aspectos. O primeiro é que o vetor $H_{d-k,k}$ (onde $k \in [0..n]$) possui as células da diagonal d da matriz. O segundo aspecto é visto na redução da área *pruning* à direita (linha 8), onde a redução somente pode ser feita em intervalos de 1 elemento durante cada atualização. O algoritmo de *pruning* por diagonal foi implementado no CUDAlign 2.1 (Capítulo 7);

Algoritmo Genérico Quadrático: O Algoritmo Genérico Quadrático (Algoritmo 4) é capaz de manter informações de *pruning* para todas as células, permitindo assim que a matriz seja processada em qualquer ordem, inclusive seguindo um fluxo genérico de processamento (*dataflow*). Este algoritmo mantém uma matriz k , de complexidade $O(m \cdot n)$ em memória, indicando a condição de *pruning* de cada célula. O valor de $k_{i,j}$ é **true**

Algorithm 3 Algoritmo Específico por Diagonal

```

1:  $(k_s, k_e) \leftarrow (0, n)$ 
2: procedure PRUNINGUPDATE( $d, H_{d-k,k}$ ) ▷  $k \in [0..n]$ 
3:    $bound \leftarrow \max(bound, H_{d-k,k})$ 
4:   while  $k_s < n$  and  $\text{IsPrunable}(d - k_s, k_s, H_{d-k_s, k_s}, bound)$  do
5:      $k_s \leftarrow k_s + 1$  ▷ aumento da área pruning à esquerda
6:   end while
7:   if  $k_e < n$  and  $\neg \text{IsPrunable}(d - k_e, k_e, H_{d-k_e, k_e}, bound)$  then
8:      $k_e \leftarrow k_e + 1$ ; ▷ redução da área pruning à direita
9:   else
10:     $k_e \leftarrow k_e - 1$ ;
11:    while  $k_e \geq k_s$  and  $\text{IsPrunable}(d - k_e, k_e, H_{d-k_e, k_e}, bound)$  do
12:       $k_e \leftarrow k_e - 1$  ▷ aumento da área pruning à direita
13:    end while
14:     $k_e \leftarrow k_e + 1$ ;
15:  end if
16: end procedure

17: function ISPRUNED( $j$ )
18:   return  $j < k_e$  or  $j > k_e$ 
19: end function

```

se e somente se a célula $H_{i,j}$ for identificada como *prunable*. A inicialização da matriz k é feita conforme a Equação 8.28, de forma que as primeiras linhas e colunas possam ser descartadas por indução e que a primeira célula da matriz $H_{1,1}$ seja sempre calculada. A inicialização da matriz está apresentada na primeira linha do Algoritmo 4.

$$k[0..m][0..n] = \begin{pmatrix} \text{false} & \text{true} & \text{true} & \cdots & \text{true} \\ \text{true} & \text{false} & \text{false} & \cdots & \text{false} \\ \text{true} & \text{false} & \text{false} & \cdots & \text{false} \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ \text{true} & \text{false} & \text{false} & \cdots & \text{false} \end{pmatrix} \quad (8.28)$$

A função ISPRUNED (linhas 2-9) é sempre chamada antes do cálculo de cada célula e verifica a condição de *pruning* das células vizinhas ($H_{i-1,j}$, $H_{i-1,j-1}$ e $H_{i,j-1}$) para poder inferir se a célula $H_{i,j}$ pode ser descartada (*pruned*). Caso seja identificada uma célula *pruned*, a célula $k_{i,j}$ é marcada como **true**.

Caso a célula não seja identificada como *pruned*, a computação do seu valor é feita normalmente e, em seguida, a função PRUNINGUPDATE (lines 10-15) é chamada para atualizar o estado da célula $k_{i,j}$ caso ela seja identificada como *prunable* pela função ISPRUNABLE. Além disso, o limite inferior *bound* é atualizado caso o valor recém-calculado da célula $H_{i,j}$ seja maior que ele (linha 11).

Algoritmo Genérico Linear: O Algoritmo Genérico Linear (Algoritmo 5) é similar em funcionalidade com o Algoritmo Genérico Quadrático (Algoritmo 4), com a diferença de utilizar uma estrutura muito mais simples em memória. Em vez de utilizar uma matriz quadrática k , o algoritmo mantém dois vetores lineares, um para cada dimensão k_h (horizontal) e k_v (vertical), os quais indicam qual foi a última célula *prunable* em

Algorithm 4 Algoritmo Genérico Quadrático

```
1:  $k[0..m][0..n] \leftarrow$  Equation 8.28
2: function ISPRUNED( $i, j$ )
3:   if  $k[i-1][j] = \text{true}$  and  $k[i][j-1] = \text{true}$  and  $k[i-1][j-1] = \text{true}$  then
4:      $k[i][j] \leftarrow \text{true}$ 
5:     return true;
6:   else
7:     return false;
8:   end if
9: end function
10: procedure PRUNINGUPDATE( $i, j, H_{i,j}$ )
11:    $bound \leftarrow \max(bound, H_{i,j})$ 
12:   if IsPrunable( $i, j, H_{i,j}, bound$ ) then
13:      $k[i][j] \leftarrow \text{true}$ 
14:   end if
15: end procedure
```

uma determinada coluna ou linha, respectivamente. Desta forma, o algoritmo reduz a complexidade de memória para $O(m+n)$.

Para identificar se uma célula $H_{i,j}$ pode ser descartada antes do seu cálculo, a função ISPRUNED (linhas 3-11) precisa identificar se as células $H_{i-1,j}$, $H_{i,j-1}$ e $H_{i-1,j-1}$ são todas *prunable* (linha 4). Neste momento, se $k_h[j] = i-1$ então a última célula *prunable* da coluna j foi a célula $H_{i-1,j}$. Analogamente, se $k_w[i] = j-1$ então a última célula *prunable* da linha i foi a célula $H_{i,j-1}$.

Entretanto, note que a indução também precisa levar em conta a célula da diagonal $H_{i-1,j-1}$. Para isso, adiciona-se um fator de ajuste $-ma - G$ (linha 14) relacionado com a diferença de valores entre células adjacentes (Figura 8.7). Com isso, os vetores k_h e k_w são ajustados para que a condição $k_h[j] = i-1$ ocorra somente se $H_{i-1,j}$ e $H_{i-1,j-1}$ forem *prunable*. Analogamente, $k_w[i] = j-1$ deve ocorrer somente se $H_{i,j-1}$ e $H_{i-1,j-1}$ forem *prunable*. Com isso, o teste da linha 4 também considera a célula $H_{i-1,j-1}$ na indução.

Assim como no Algoritmo 4, a função PRUNINGUPDATE (linhas 12-18) é chamada para atualizar o estado dos elementos $k_w[i]$ e $k_h[j]$ caso a célula $H_{i,j}$ seja identificada como *prunable* pela função ISPRUNABLE. Além disso, o limite inferior $bound$ é atualizado caso o valor recém calculado da célula $H_{i,j}$ seja maior que ele.

8.4.3 Block Pruning

Um bloco é definido um conjunto de células contíguas. Para que não haja dependência cíclica entre blocos, os blocos devem respeitar a seguinte regra: se as células $H_{i,j}$ e $H_{i-1,j-1}$ estiverem em um mesmo bloco, então as células $H_{i-1,j}$ e $H_{i,j-1}$ também estarão. A Figura 8.11 apresenta 3 formatos de blocos que respeitam essa propriedade.

Um *grid* é uma matriz $B_h \times B_w$ de blocos ordenados em h linhas e w colunas. Para cada bloco $B_{x,y}$, definimos como (i', j') e (i'', j'') as coordenadas mínimas e máximas em cada dimensão do bloco, conforme ilustra a Figura 8.11.

A dependência entre os blocos de um grid depende diretamente da forma dos blocos. Na Figura 8.12(a) podemos ver que blocos retangulares possuem dependências análogas às das células. Entretanto, blocos em paralelogramo (Figura 8.12(b)) geram uma de-

Algorithm 5 Algoritmo Genérico Linear

```

1:  $k_v[0..m] \leftarrow \{-\infty, -1, -1, \dots, -1\}$ 
2:  $k_h[0..n] \leftarrow \{-\infty, -1, -1, \dots, -1\}$ 

3: function ISPRUNED( $i, j$ )
4:   if  $k_v[i] = j - 1$  and  $k_h[j] = i - 1$  then
5:      $k_v[i] \leftarrow b_j$ 
6:      $k_h[j] \leftarrow b_i$ 
7:     return true;
8:   else
9:     return false;
10:  end if
11: end function

12: procedure PRUNINGUPDATE( $i, j, H_{i,j}$ )
13:   $bound \leftarrow \max(bound, H_{i,j})$ 
14:  if IsPrunable( $i, j, H_{i,j} - G - ma, bound$ ) then
15:     $k_v[i] \leftarrow b_j$ 
16:     $k_h[j] \leftarrow b_i$ 
17:  end if
18: end procedure

```

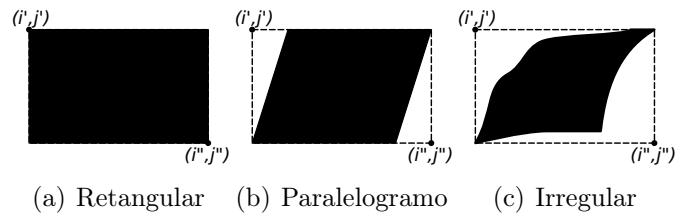


Figura 8.11: Diferentes formatos de blocos e suas coordenadas máximas mínimas.

pendência diferenciada, onde o bloco $B_{x,y}$ depende dos blocos $B_{x-1,y}$, $B_{x,y-1}$ e $B_{x+1,y-1}$. Chamaremos esta dependência a $B_{x+1,y-1}$ de *dependência diferenciada do grid*. Todas as dependências diferenciadas devem ser levadas em consideração durante o processamento da matriz, caso contrário o resultado poderá ficar inconsistente.

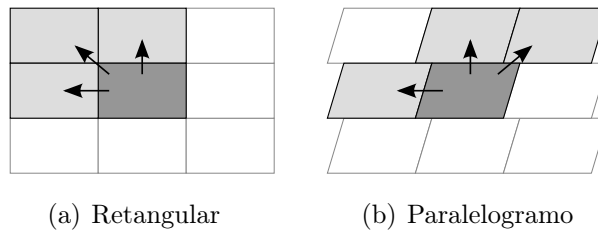


Figura 8.12: Dependências entre blocos de um grid

Estritamente falando, o escore derivado máximo de qualquer alinhamento local que passe através de um bloco $B_{x,y}$, pode ser definido conforme a Equação 8.29, considerando a Equação 8.25.

$$\begin{aligned}
H_{B_{x,y}}^{max} &= \max_{(i,j) \in B_{x,y}} [H_{i,j}^{max}] \\
H_{B_{x,y}}^{max} &= \max_{(i,j) \in B_{x,y}} [H_{i,j} + \min(m - i, n - j) \cdot ma]
\end{aligned} \tag{8.29}$$

Entretanto, para evitar o cálculo de $H_{i,j}^{max}$ para todas as células, a Equação 8.30 apresenta uma fórmula menos estrita para o escore derivado máximo deste bloco, onde $H_{B_{x,y}}$ é o maior escore do bloco $B_{x,y}$.

$$H_{B_{x,y}}^{max} = \underbrace{\max_{(i,j) \in B_{x,y}} [H_{i,j}]}_{H_{B_{x,y}}} + \max_{(i,j) \in B_{x,y}} [\min(m - i, n - j) \cdot ma] \tag{8.30}$$

Embora o valor de $H_{B_{x,y}}$ seja naturalmente obtido ao se calcular todas as células de um bloco, ainda assim o cálculo de $\min(m - i, n - j) \cdot ma$ na Equação 8.30 necessitaria de ser feito para todas as células. Para evitar esse cálculo, substitui-se este termo considerando a coordenada mais distante das bordas da matriz, que é a coordenada mínima (i', j') deste bloco. A Equação 8.31 apresenta a forma menos estrita onde, conhecido o escore máximo $H_{B_{x,y}}$, é possível determinar escore derivado máximo $H_{B_{x,y}}^{max}$ em tempo constante.

$$H_{B_{x,y}}^{max} = H_{B_{x,y}} + \min(m - i', n - j') \cdot ma \tag{8.31}$$

O Algoritmo 6 apresenta a função ISPRUNABLE adaptada para blocos, considerando a Equação 8.31.

Algorithm 6 Condição de *Pruning* para Blocos

```

1: function ISPRUNABLE( $x, y, H_{B_{x,y}}, bound$ )
2:    $(i', j') := \text{GETMINCOORD}(B_{x,y})$ 
3:    $H_{B_{x,y}}^{max} = H_{B_{x,y}} + \min(m - i', n - j') \cdot ma$ 
4:   return  $H_{B_{x,y}}^{max} \leq bound$ 
5: end function

```

8.5 Método de Avaliação Teórica

Nesta seção será definida uma metodologia de avaliação teórica do método de *pruning*, com o objetivo de definir a eficácia do método. Restringiremos a análise desta seção a apenas alinhamentos locais. A metodologia proposta leva em consideração dois aspectos: (1) os valores de cada célula $H_{i,j}$ da matriz de programação dinâmica; (2) a forma de processamento da matriz.

Em relação aos valores das células, serão considerados alguns casos especiais para definir $H_{i,j}$ por meio de fórmulas matemáticas que utilizam conceitos probabilísticos. As células $H_{i,j}$ dependem dos parâmetros de *match*, *mismatch* e *gap*, além das características das sequências. Embora a matriz de programação dinâmica seja formada por coordenadas e valores inteiros, a metodologia apresentada nesta seção considera coordenadas e valores no conjunto dos números Reais, assemelhando-se à matriz original. Com esta abordagem, a análise e a simulação do método de *pruning* podem ser feitas matematicamente, sem a

necessidade de calcularmos as equações de recorrência, que são muito mais custosas em termos de processamento e memória.

Apresentaremos a seguir 3 fórmulas de $H_{i,j}$ relativas a casos especiais onde o alinhamento ótimo encontra-se na diagonal principal.

- **Perfect Match com letras iguais e repetidas (PM_r):** Neste cenário, o alfabeto das sequências possui um único caractere $\Sigma = \{c_1\}$ e as duas sequências são formadas por uma repetição do mesmo caractere (ex.: $S_0 = c_1c_1c_1\dots c_1$ e $S_1 = c_1c_1c_1\dots c_1$). Neste cenário, definimos os valores das células conforme a Equação 8.32.

$$H_{i,j} = \min(i, j) \cdot ma \quad (8.32)$$

- **Perfect Match com letras distintas ($PM^{1.0}$):** Supondo que o alfabeto das sequências possui um número infinito de caracteres $\Sigma = \{c_1, c_2, c_3, \dots\}$ e que as duas sequências são formadas por letras únicas (ex.: $S_0 = c_1c_2c_3\dots c_m$ e $S_1 = c_1c_2c_3\dots c_n$), definimos os valores das células conforme a Equação 8.33. O caso de *Perfect Match* com letras distintas é uma generalização do caso de *Perfect Match* com letras iguais e repetidas (PM_r) quando $G = 0$.

$$H_{i,j} = \max(0, \min(i, j) \cdot ma - |i - j|G) \quad (8.33)$$

- **Semi-Perfect Match (PM^p):** Definimos um *Semi-Perfect Match* como um caso similar ao *Perfect Match* com letras distintas, mas alterando alguns caracteres da sequência S_0 por um caractere não existente na outra sequência, de forma a gerar um *mismatch* naquela posição. Considerando que a probabilidade de alteração é $\psi \in [0, 1]$ e que a alteração é uniformemente distribuída ao longo da sequência, definimos os valores das células de maneira aproximada conforme a Equação 8.34, onde $p = \psi - \frac{mi}{ma}(1 - \psi)$ e $p \in [0, 1]$. O caso de *Semi-Perfect Match* é uma generalização do caso de *Perfect Match* com letras distintas considerando que $p = 1$. Analogamente, o caso de *Semi-Perfect Match* é uma generalização do caso de *Perfect Match* com letras iguais e repetidas quando $p = 1$ e $G = 0$.

$$\begin{aligned} H_{i,j} &= \max(0, \min(i, j) \cdot \underbrace{(ma \cdot \psi - mi(1 - \psi))}_{ma \cdot p} - |i - j|G) \\ &= \max(0, \min(i, j) \cdot ma \cdot p - |i - j|G) \end{aligned} \quad (8.34)$$

O segundo aspecto considerado na metodologia proposta nesta tese é a forma de processamento da matriz. Na Figura 8.13 apresentamos os processamentos: linha por linha (Figura 8.13(a)), coluna por coluna (Figura 8.13(b)), por antidiagonais (Figura 8.13(c)), com inclinação (Figura 8.13(d)), em forma de quadrado (Figura 8.13(e)), em forma de quadrado invertido (Figura 8.13(f)) e de maneira genérica (Figura 8.13(g)). O processamento por inclinação (considerando um ângulo θ) generaliza os processamentos por linha ($\theta = 0^\circ$), por coluna ($\theta = 90^\circ$) e por diagonal ($\theta = 45^\circ$). O processamento genérico não será abordado na avaliação teórica visto que a ordem de execução pode variar a cada execução.

A forma de processamento da matriz reflete diretamente no melhor escore provisório $best_{i,j}$ (Definição 8.2.12) encontrado ao se calcular a célula $H_{i,j}$. Considerando que o valor

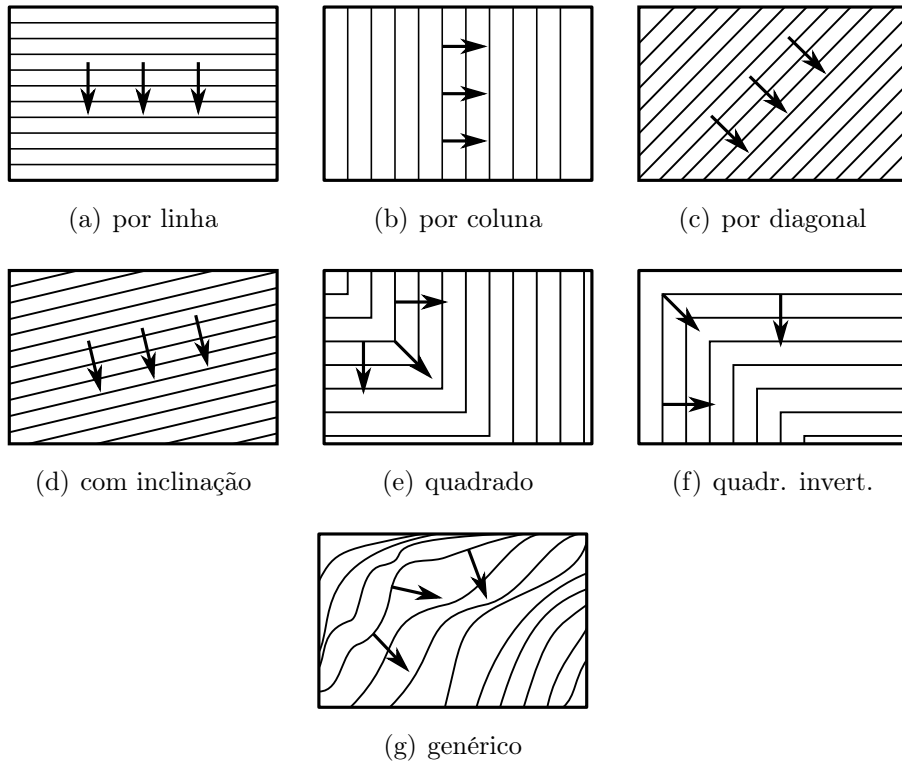


Figura 8.13: Formas de processamento.

$H_{i,j}$ é conhecido, o valor de $best_{i,j}$ pode ser definido por meio de fórmulas. A análise de $best_{i,j}$ torna-se mais simplificada para o alinhamento local, pois $H_{i',j'}^{min} = H_{i',j'}$ para todas as células da matriz (Equação 8.26).

Visto que nos cenários PM_r , $PM^{1.0}$ e PM^p o alinhamento ótimo está na diagonal principal da matriz, podemos definir o valor de $best_{i,j}$ de acordo com a iteração $\phi_{i,j}$ (Definição 8.2.13), conforme a Equação 8.35.

$$best_{i,j} = H_{\phi_{i,j},\phi_{i,j}} = \phi_{i,j} \cdot ma.p \quad (8.35)$$

A Figura 8.14 apresenta geometricamente a posição de $best_{i,j}$ para as diferentes formas de processamento, aplicando-se a qualquer um dos cenários PM_r , $PM^{1.0}$ e PM^p .

O valor de $\phi_{i,j}$ é então definido para cada forma de processamento conforme as Equações 8.36 a 8.41. Ressaltamos que $\phi_{i,j}$ considera valores no conjunto dos reais por questões de simplificação.

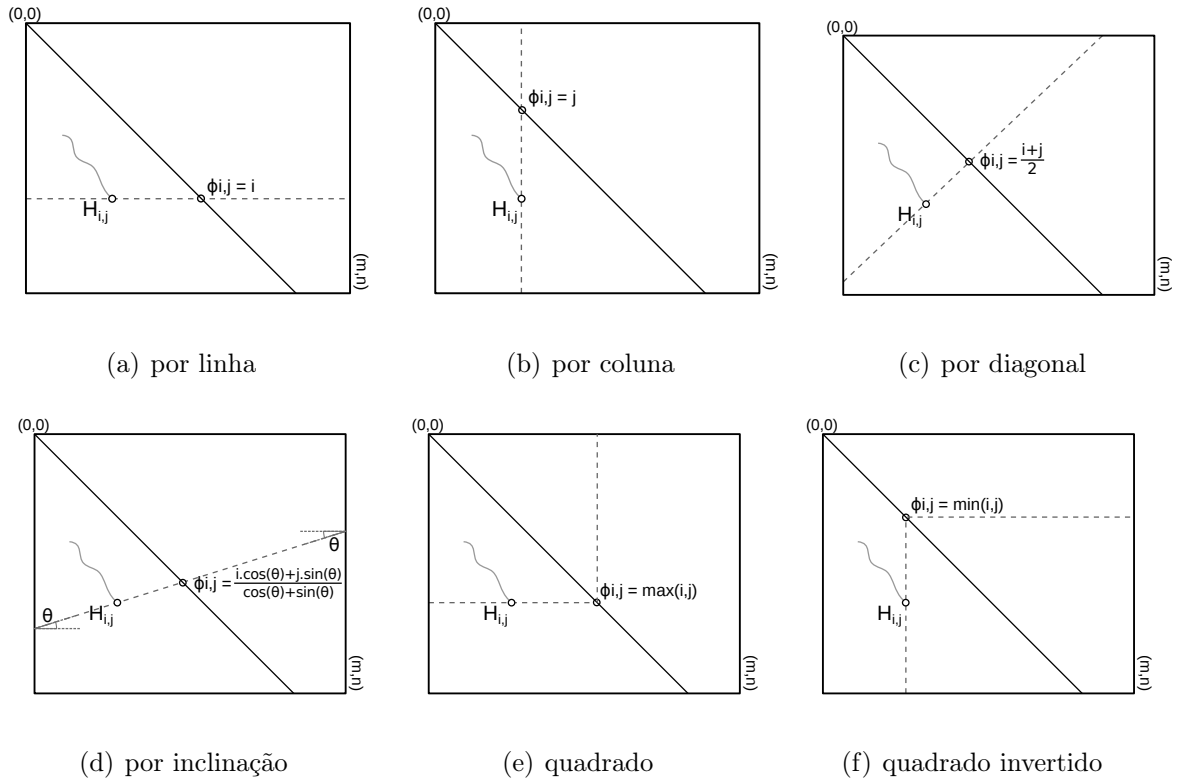


Figura 8.14: Maior escore provisório ($best_{i,j}$) em diversas formas de processamento.

$$\text{por linha:} \quad \phi_{i,j} = i \quad (8.36)$$

$$\text{por coluna:} \quad \phi_{i,j} = j \quad (8.37)$$

$$\text{por diagonal:} \quad \phi_{i,j} = \frac{i+j}{2} \quad (8.38)$$

$$\text{por inclinação } \theta: \quad \phi_{i,j} = \frac{i \cdot \cos(\theta) + j \cdot \sin(\theta)}{\cos(\theta) + \sin(\theta)} \quad (8.39)$$

$$\text{quadrado:} \quad \phi_{i,j} = \max(i, j) \quad (8.40)$$

$$\text{quadrado invertido:} \quad \phi_{i,j} = \min(i, j) \quad (8.41)$$

8.5.1 Eficácia do método de *Pruning*

A Eficácia de *Pruning* é a razão entre o número de células *prunable* e o número de células da matriz, conforme a Equação 8.42.

$$\text{eficácia} = \frac{n^{\circ} \text{ células } \textit{prunable}}{|S_0| \times |S_1|} \quad (8.42)$$

Podemos dizer também que a eficácia de *pruning* é geometricamente definida pela razão entre a área com células *prunable* (área *prunable*) e a área total da matriz. Podemos

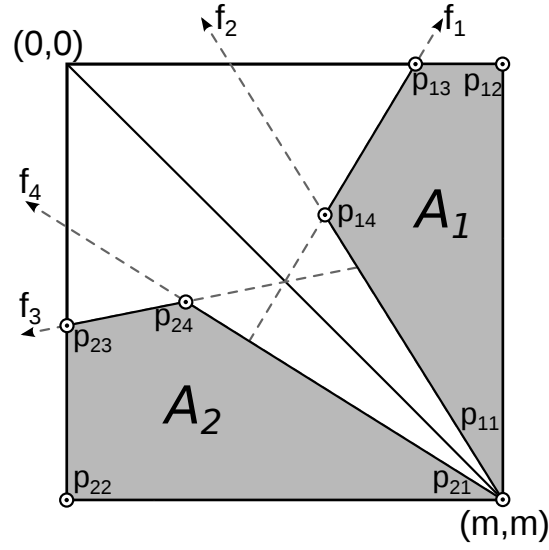


Figura 8.15: Representações geométricas da área de *Block Pruning*.

identificar a área *prunable* verificando em quais áreas da matriz a condição $H_{i,j}^{max} < best_{i,j}$ é verdadeira.

No restante desta seção, definiremos o tamanho da área de *pruning* considerando os cenários PM_r , $PM^{1.0}$ e PM^p (onde o valor $H_{i,j}$ é definido conforme as Equações 8.32, 8.33 e 8.34, respectivamente) e os diferentes modos de processamento (onde o valor $\phi_{i,j}$ é definido conforme as Equações 8.36 a 8.41). Analisaremos apenas os casos de alinhamento local e assumiremos que as sequências possuem tamanhos iguais ($|S_1| = |S_2| = m$). Os valores de $H_{i,j}^{max}$ e $H_{i,j}^{min}$ serão definidos, respectivamente, pelas Equações 8.25 e 8.26.

A Figura 8.15 apresenta as definições geométricas das áreas de *pruning*. Podemos visualizar duas áreas de *pruning* distintas, unidas ao ponto inferior direito (m, m) do quadrado. Chamaremos de A_1 e A_2 as áreas que estão, respectivamente, à direita e à esquerda da diagonal principal. As áreas possuem no máximo 4 lados, limitados pelas retas f_1, f_2, f_3, f_4 e pelas bordas da matriz. A definição das funções f_1, f_2, f_3 e f_4 baseia-se na condição de *pruning*, com algumas premissas que serão definidas nas Equações 8.46. Os vértices das áreas são definidos como $A_1 = (p_{11}, p_{12}, p_{13}, p_{14})$ e $A_2 = (p_{21}, p_{22}, p_{23}, p_{24})$, cujas coordenadas estão declaradas nas Equações 8.43. Os vértices p_{14} e p_{24} são formados pelas intersecções das retas $f_1 \cap f_2$ e $f_3 \cap f_4$, respectivamente.

$$\begin{aligned}
 p_{11} &= (0, f_1(0)) & p_{21} &= (f_3(0), 0) \\
 p_{12} &= (0, m) & p_{22} &= (m, 0) \\
 p_{13} &= (m, m) & p_{23} &= (m, m) \\
 p_{14} &= (i', f_1(i')) = (i', f_2(i')) & p_{24} &= (f_3(j'), j') = (f_4(i'), j')
 \end{aligned} \tag{8.43}$$

Definidos os pontos das Equações 8.43, podemos encontrar o tamanho das áreas A_1 e A_2 utilizando a Equação 8.44, que calcula, em valores absolutos, a área do polígono

formado por quatro pontos quaisquer.

$$A = \frac{1}{2} \cdot \text{abs} \left[\begin{array}{cc} x_1 & y_1 \\ x_2 & y_2 \\ x_3 & y_3 \\ x_4 & y_4 \end{array} \right] = \text{abs} \left[\frac{(x_1y_2 + x_2y_3 + x_3y_4 + x_4y_1) - (y_1x_2 + y_2x_3 + y_3x_4 + y_4x_1)}{2} \right] \quad (8.44)$$

Para encontrar as definições das funções f_1 , f_2 , f_3 e f_4 , consideraremos a condição de *pruning* $H_{i,j}^{max} \leq best_{i,j}$ assumindo os valores $H_{i,j}$ (Equação 8.34), $\phi_{i,j}$ (Equações 8.36 a 8.41), $best_{i,j} = \phi_{i,j}.ma.p$ (Equação 8.35) e $H_{i,j}^{max}$ (Equação 8.25), o que resulta na Inequação 8.45:

$$\begin{aligned} H_{i,j}^{max} &\leq best_{i,j} \\ H_{i,j} + \min(m-i, m-j).ma &\leq \phi_{i,j}.ma.p \\ \max(0, \min(i, j)ma.p - |i-j|G) + \min(m-i, m-j).ma &\leq \phi_{i,j}.ma.p \end{aligned} \quad (8.45)$$

Para formalizar as funções f_1 , f_2 , f_3 e f_4 , serão utilizadas diferentes premissas para a seleção dos operandos das funções max e min dentro da Inequação 8.45. As premissas utilizadas em cada função são definidas em (8.46):

$$\begin{aligned} f1: & j > i \text{ e } H_{i,j} = 0 \\ f2: & j > i \text{ e } H_{i,j} \neq 0 \\ f3: & j \leq i \text{ e } H_{i,j} = 0 \\ f4: & j \leq i \text{ e } H_{i,j} \neq 0 \end{aligned} \quad (8.46)$$

Ao transformar a Inequação 8.45 em equações baseadas nas 4 premissas acima, encontramos as Equações 8.47 a 8.50.

caso f_1 : $j > i$ e $H_{i,j} = 0$

$$\begin{aligned} H_{i,j} + \min(m-i, m-j).ma &= \phi_{i,j}.ma.p \\ 0 + (m-j).ma &= \phi_{i,j}.ma.p \\ j &= m - \phi_{i,j}.p \end{aligned} \quad (8.47)$$

caso f_2 : $j > i$ e $H_{i,j} \neq 0$

$$\begin{aligned} H_{i,j} + \min(m-i, m-j).ma &= \phi_{i,j}.ma.p \\ \min(i, j)ma.p - |i-j|G + \min(m-i, m-j).ma &= \phi_{i,j}.ma.p \\ i.ma.p - (j-i)G + (m-j).ma &= \phi_{i,j}.ma.p \\ j(G+ma) &= i(ma.p+G) + m.ma - \phi_{i,j}.ma.p \\ j &= \frac{i(ma.p+G) + m.ma - \phi_{i,j}.ma.p}{G+ma} \end{aligned} \quad (8.48)$$

caso f_3 : $j \leq i$ e $H_{i,j} = 0$

$$\begin{aligned} H_{i,j} + \min(m-i, m-j).ma &= \phi_{i,j}.ma.p \\ 0 + (m-i).ma &= \phi_{i,j}.ma.p \\ i &= m - \phi_{i,j}.p \end{aligned} \quad (8.49)$$

caso f_4 : $j \leq i$ e $H_{i,j} \neq 0$

$$\begin{aligned} H_{i,j} + \min(m-i, m-j).ma &= \phi_{i,j}.ma.p \\ \min(i, j)ma.p - |i-j|G + \min(m-i, m-j).ma &= \phi_{i,j}.ma.p \\ j.ma.p - (i-j)G + (m-i).ma &= \phi_{i,j}.ma.p \\ i(G+ma) &= j(ma.p+G) + m.ma - \phi_{i,j}.ma.p \\ i &= \frac{j(ma.p+G) + m.ma - \phi_{i,j}.ma.p}{G+ma} \end{aligned} \quad (8.50)$$

Para todos os casos acima, devemos resolver as equações considerando os valores de $\phi_{i,j}$. Por exemplo, digamos que $\phi_{i,j} = i$ (processamento por linhas). As funções f_1 , f_2 , f_3 e f_4 são definidas como se segue:

caso f_1 : (Equação 8.47)

$$\begin{aligned} j &= m - \phi_{i,j}.p \\ j &= m - i.p \\ f_1(i) &= m - i.p \end{aligned}$$

caso f_2 : (Equação 8.48)

$$\begin{aligned} j &= \frac{i(ma.p+G) + m.ma - \phi_{i,j}.ma.p}{G+ma} \\ j &= \frac{i(ma.p+G) + m.ma - i.ma.p}{G+ma} \\ f_2(i) &= \frac{i.G + m.ma}{G+ma} \end{aligned}$$

caso f_3 : (Equação 8.49)

$$\begin{aligned} i &= m - \phi_{i,j}.p \\ i &= m - i.p \\ f_3(j) &= \frac{m}{p+1} \end{aligned}$$

caso f_4 : (Equação 8.50)

$$\begin{aligned} i &= \frac{j(ma.p+G) + m.ma - \phi_{i,j}.ma.p}{G+ma} \\ i &= \frac{j(ma.p+G) + m.ma - i.ma.p}{G+ma} \\ f_4(j) &= \frac{j(ma.p+G) + m.ma}{G+ma.(p+1)} \end{aligned}$$

```

/* definicoes de phi */
phi(i,j):=i; /* row-by-row */
phi(i,j):=j; /* col-by-col */
phi(i,j):=(i+j)/2; /* diagonal */
phi(i,j):=(i*cos(theta)+j*sin(theta))/(cos(theta)+sin(theta)); /* angular */
phi(i,j):=max(i,j); /* square */
phi(i,j):=min(i,j); /* anti-square */

/* resolucao das funcoes f1, f2, f3 e f4 */
f1(i):=ev(rhs(solve(j=-phi(i,j)*ma*p/ma+m, j)[1]));
f2(i):=ev(rhs(solve(j=(-phi(i,j)*ma*p+m*ma+i*(G+ma*p))/(G+ma), j)[1]));
f3(j):=ev(rhs(solve(i=-phi(i,j)*ma*p/ma+m, i)[1]));
f4(j):=ev(rhs(solve(i=(-phi(i,j)*ma*p+m*ma+j*(G+ma*p))/(G+ma), i)[1]));

/* intersecoes s0:f1-f2 e s1:f3-f4 */
ii:ev(rhs(solve(f1(i)=f2(i), i)[1]));
jj:ev(rhs(solve(f3(j)=f4(j), j)[1]));

/* calculo das areas */
area1:(-f1(0)*ii - f1(ii)*m + ii*m + m*m)/2;
area2:(-f3(0)*jj - f3(jj)*m + jj*m + m*m)/2;
area:area1+area2;

/* calculo da eficiencia */
factor(area/m/m);

```

Figura 8.16: Código para resolução das equações de área - Ferramenta Maxima

Supondo que $p = 1$, $ma = 1$ e $G = 3$, podemos encontrar as seguintes coordenadas das áreas de *pruning*:

$$\begin{array}{ll}
p_{11} = (0, m) & p_{21} = (m\frac{1}{2}, 0) \\
p_{12} = (0, m) & p_{22} = (m, 0) \\
p_{13} = (m, m) & p_{23} = (m, m) \\
p_{14} = (m\frac{3}{7}, m\frac{4}{7}) & p_{24} = (m\frac{1}{2}, m\frac{3}{8})
\end{array} \tag{8.51}$$

Utilizando a Equação 8.44, pode-se encontrar o tamanho das áreas $A_1 = \frac{3}{14}m^2$ e $A_2 = \frac{11}{32}m^2$. Sendo assim, a eficácia de *pruning* neste cenário é de $\frac{A_1+A_2}{m^2} = 55.8\%$.

Para auxiliar a resolução das equações com parametrização variável, utilizamos a ferramenta computacional *Maxima* [139] para resolução de equações algébricas. O código fonte na Figura 8.16, escrito para a ferramenta *Maxima*, permite encontrar todas as equações de eficácia de *pruning* para cada uma das 6 formas de processamento apresentadas ($\phi_{i,j}$ - Equações 8.36 a 8.41), considerando os cenários PM_r , $PM^{1.0}$ e PM^p . A Tabela 8.4 apresenta as 18 equações, utilizando as variáveis ma , mi , G , p e θ . Por meio dessas equações, podemos concluir que, nos cenários apresentados, os processamentos por linha e coluna apresentam exatamente as mesmas eficácias. Além disso, a eficácia do processamento por linha ou coluna está na mediana entre a eficácia do processamento por quadrado e por quadrado invertido.

Tabela 8.4: Eficácia de *Pruning*

| Wave | $H_{i,j}$ | Eficácia (%) |
|---|------------|---|
| | PM_r | $\frac{1}{4} = 25.0\%$ |
| | $PM^{1.0}$ | $\frac{1}{2} \cdot \frac{G}{2G+ma} + \frac{1}{4} \cdot \frac{3G+2ma}{2G+2ma}$ |
| | PM^p | $\frac{1}{2} \cdot \frac{pG}{G+pG+map} + \frac{1}{2} \cdot \frac{p}{p+1} \cdot \frac{2G+pG+map+map^2}{1G+pG+map+map^2}$ |
| | PM_r | $\frac{1}{4} = 25.0\%$ |
| | $PM^{1.0}$ | $\frac{1}{4} \cdot \frac{3G+2ma}{2G+2ma} + \frac{1}{2} \cdot \frac{G}{2G+ma}$ |
| | PM^p | $\frac{1}{2} \cdot \frac{p}{p+1} \cdot \frac{2G+pG+map+map^2}{1G+pG+map+map^2} + \frac{1}{2} \cdot \frac{pG}{G+pG+map}$ |
| | PM_r | $\frac{1}{3} = 33.3\%$ |
| | $PM^{1.0}$ | $\frac{1}{3} \cdot \frac{8G+3ma}{4G+3ma}$ |
| | PM^p | $\frac{p}{p+2} \cdot \frac{6G+2pG+2map+map^2}{2G+2pG+2map+map^2}$ |
| | PM_r | $\frac{2c_0-1}{5c_0-1}$ |
| | $PM^{1.0}$ | $\frac{1}{2} \cdot \frac{c_0}{c_2+c_0} \cdot \frac{(c_1+c_2)G+(c_2+c_3)ma}{(c_0+c_0)G+(c_0+c_2)ma} + \frac{1}{2} \cdot \frac{c_0}{c_5+c_0} \cdot \frac{(c_4+c_5)G+(c_5+c_6)ma}{(c_0+c_0)G+(c_0+c_5)ma}$ |
| | PM^p | $\frac{1}{2} \cdot \frac{c_0 p}{c_2 p+c_0} \cdot \frac{c_1 G+c_2 pG+c_2 ma p+c_3 ma p^2}{c_0 G+c_0 pG+c_0 ma p+c_2 ma p^2} + \frac{1}{2} \cdot \frac{c_0 p}{c_5 p+c_0} \cdot \frac{c_4 G+c_5 pG+c_5 ma p+c_6 ma p^2}{c_0 G+c_0 pG+c_0 ma p+c_5 ma p^2}$ |
| onde: $s = \text{sen}\theta, \quad c = \text{cos}\theta, \quad \alpha = (s+c), \quad c_0 = \alpha^2,$ $c_1 = \alpha(\alpha+s), \quad c_2 = \alpha s, \quad c_3 = s^2,$ $c_4 = \alpha(\alpha+c), \quad c_5 = \alpha c, \quad c_6 = c^2$ | | |
| | PM_r | $\frac{1}{2} = 50\%$ |
| | $PM^{1.0}$ | $\frac{1}{2} \cdot \frac{3G+2ma}{2G+2ma}$ |
| | PM^p | $\frac{p}{p+1} \cdot \frac{2G+pG+map+map^2}{1G+pG+map+map^2}$ |
| | PM_r | 0 |
| | $PM^{1.0}$ | $\frac{G}{2G+ma}$ |
| | PM^p | $\frac{pG}{G+pG+map}$ |

8.5.2 Análise das fórmulas de eficácia de *pruning*

Por meio das equações listadas na Tabela 8.4, é possível analisar o impacto de cada uma das variáveis na eficácia do método de *Pruning*. Os gráficos das Figuras 8.17 a 8.20 apresentam a variação da eficácia variando alguns parâmetros. Para esses gráficos, foram utilizados os seguintes valores padrões para as pontuações do SW: $G = 3, ma = 1, mi = 3$.

Na Figura 8.17, variamos o ângulo θ de 0° a 90° nos cenários $PM^{1.0}$, $PM^{0.6}$, PM_r e $PM^{0.2}$, com todos os demais parâmetros fixos. Podemos observar que o pico dos gráficos ocorre aos 45° em todos os cenários plotados, indicando que o *pruning* é capaz de descartar mais células quando a matriz é processada em um *wavefront* de 45° . As eficácias a 0° e 90° são sempre iguais e apresentam os valores mínimos dos gráficos. Também podemos ver que a variação da eficácia é relativamente pequena nos cenários de PM^p . Por exemplo, com $PM^{1.0}$, a eficácia variou de 55,80% (0° e 90°) a 60,00% (45°), um intervalo de 4,20%. Com $PM^{0.6}$, a eficácia variou de 45,18% (0° e 90°) a 47,89% (45°), um intervalo de 2,71%. Já com $PM^{0.2}$, a eficácia variou de 22,74% (0° e 90°) a 23,37% (45°), um intervalo de

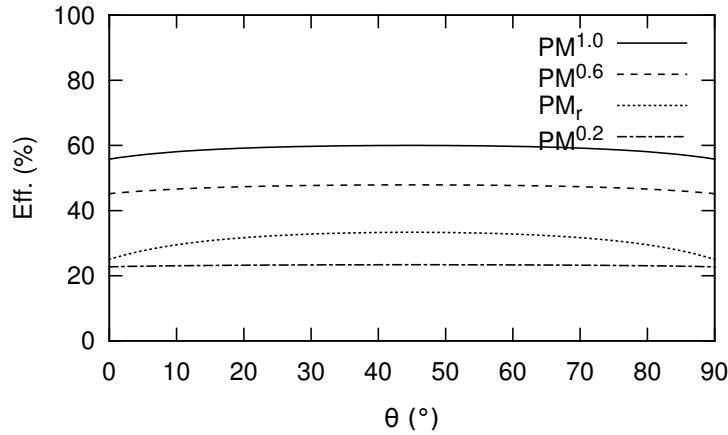


Figura 8.17: Eficácia de *Pruning* vs. ângulo de processamento

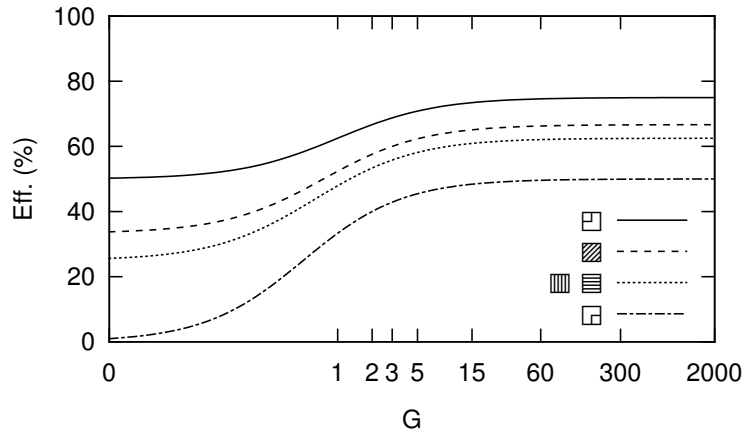


Figura 8.18: Eficácia de *Pruning* vs. penalidade de *gap*

0.63%. A eficácia do cenário PM_r apresentado na Figura 8.17 demonstra uma variação um pouco maior, entre 25% (0° e 90°) a 33,33% (45°), um intervalo de 8,33%. Podemos concluir que o cenário PM_r é o que sofre maior influência do ângulo θ de processamento. Além disso, quanto menor o valor de p no cenário PM^p , menor será a influência de θ .

A Figura 8.18 ilustra a eficácia de *pruning* no cenário $PM^{1.0}$ variando a penalidade de *gap* G entre 0 até 2000. Apresentamos as 5 formas de processamento, sendo que o processamento por linha e coluna possuem a mesma eficácia. Pelos gráficos, também podemos ver o processamento por linha ou coluna estão no centro do intervalo entre as eficácias dos processamentos por quadrado e quadrado invertido. O caso PM_r equivale ao cenário onde $G = 0$. Podemos ver que a eficácia aumenta rapidamente entre $G = 1$ e $G = 5$, estabilizando em seguida a valores mais constantes. Pelos gráficos, podemos ver que a eficácia é sempre crescente quando aumentamos o valor de G . Com G tendendo a infinito, a eficácia é de 75%, 66,67%, 62,50% e 50% para as formas de processamento por quadrado, por diagonal ($\theta = 45^\circ$), por linha/coluna e por quadrado invertido, respectivamente.

A Figura 8.19 apresenta a eficácia de *pruning* no cenário PM^p quando alteramos a similaridade ψ entre as sequências. O valor de p depende de ψ , conforme descrito na Equação 8.34 ($p = \psi - \frac{mi}{ma}(1 - \psi)$). Podemos ver que, quando a similaridade entre

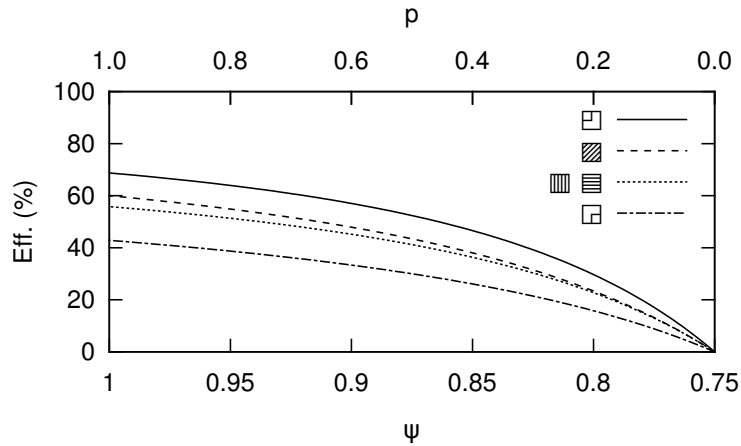
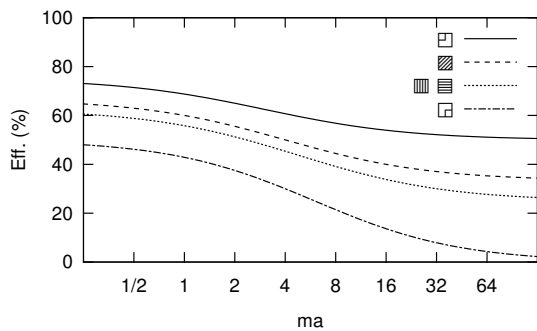


Figura 8.19: Eficácia de *Pruning* vs. o valor de ψ (ou p)

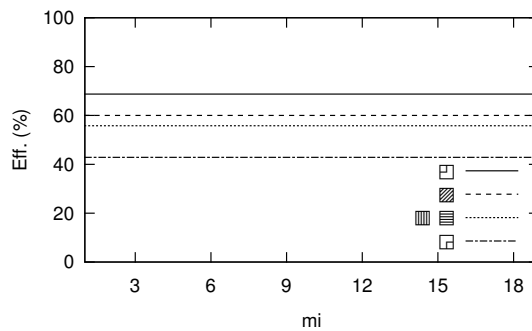
as sequência atinge um valor tal que $p = 0$, a eficácia é zero para qualquer forma de processamento. Pelos gráficos, podemos ver que a eficácia é sempre decrescente quando diminuimos a similaridade ψ da sequência, até chegar no zero da função, que ocorre quando $\psi = \frac{mi}{ma+mi}$.

As Figuras 8.20(a) e 8.20(b) descrevem a eficácia de *pruning* no cenário de *perfect match* $PM^{1.0}$ quando variamos, respectivamente, os valores de *match* (ma) e *mismatch* (mi). Neste cenário de *perfect match*, aumentar o valor de ma é equivalente ao efeito de diminuir, proporcionalmente, a penalidade de gap G . Quando ma tende ao infinito, o valor converge para o cenário PM_r , onde $G = 0$. Sendo assim, a Figura 8.20(a) é similar à Figura 8.18, embora em sentidos opostos. Conclui-se então que, quanto aumentamos a pontuação de *match*, a eficácia tende a diminuir no caso de *perfect match*. Já com a Figura 8.20(b), podemos verificar que a pontuação de *mismatch* mi não afeta a eficácia de *pruning* no cenário de *perfect match*.

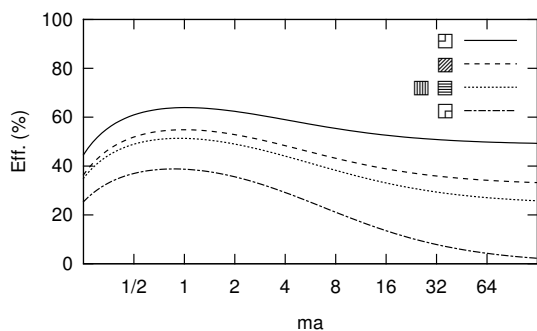
As Figuras 8.20(e) e 8.20(f) descrevem a eficácia de *pruning* no cenário PM^p com similaridade $\psi = 0.9$ e quando variamos, respectivamente, os valores de *match* (ma) e *mismatch* (mi). Devemos considerar que $p = \psi - \frac{mi}{ma}(1 - \psi)$ (Equação 8.34), sendo assim o valor de p também é alterado quando variamos ma ou mi . Visto que estamos tratando um caso de *semi-perfect match* com $\psi = 0.9$, valores muito pequenos de ma inviabilizam a existência de alinhamentos, pois o *mismatch* (mi) seria proporcionalmente muito maior que o *match* (ma). Por isso, a Figura 8.20(e) ilustra que a eficácia de *pruning* é zero para valores pequenos. Após o zero da função, a eficácia cresce até o pico do gráfico, para em seguida começar a decrescer, convergindo no infinito para o cenário de PM_r . O pico deste gráfico dependerá da forma de processamento da matriz e da proporção entre os demais parâmetros do SW. Já com a Figura 8.20(f), podemos verificar que a pontuação de *mismatch* mi afeta na eficácia de *pruning* no cenário de *semi-perfect match*, visto que se o *mismatch* (mi) for muito maior que o *match* (ma), sequências não muito similares terão score próximos a zero. As Figuras 8.20(c) e 8.20(d) descrevem a eficácia de *pruning* no cenário intermediário com similaridade $\psi = 0.95$.



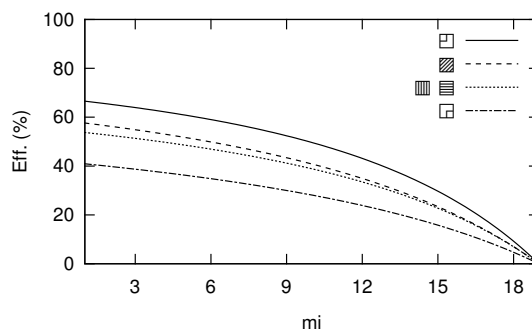
(a) *match* ($\psi = 1.0$)



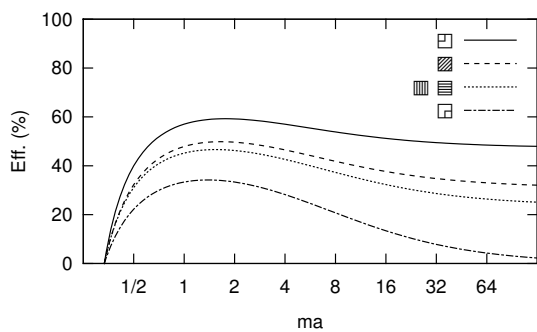
(b) *mismatch* ($\psi = 1.0$)



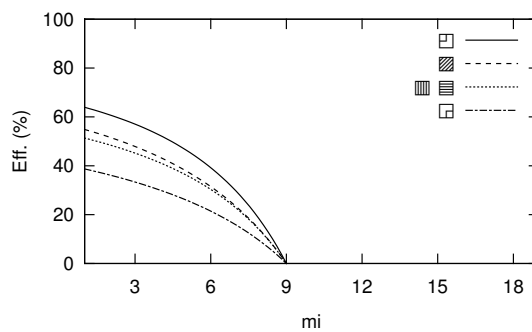
(c) *match* ($\psi = 0.95$)



(d) *mismatch* ($\psi = 0.95$)



(e) *match* ($\psi = 0.9$)



(f) *mismatch* ($\psi = 0.9$)

Figura 8.20: Eficácia de *Pruning* vs. pontuação de *match* ou *mismatch*

8.6 Simulação do *Pruning*

Conforme visto nas Seções 8.5.1 e 8.5.2, a eficácia de *pruning* é afetada por diversas características e parâmetros do processamento da matriz, entre eles a forma de processamento, os parâmetros da equação de recorrência e a similaridade das sequências.

Nesta seção apresentaremos simulações feitas com um *software* de plotagem de gráfico (*Gnuplot*) para visualizarmos as áreas de *pruning*. O script do *Gnuplot* encontra-se no Anexo II. Em cada simulação, apresentaremos a eficácia de *pruning*, medida pela razão entre a quantidade de pixels pretos (células *prunable*) e o número de *pixels* da figura original (sem bordas). O objetivo desta seção é validar, por meio das simulações, as fórmulas obtidas na Tabela 8.4 e os gráficos da Seção 8.5.2. Adicionalmente, apresentaremos geometricamente os efeitos dos diversos parâmetros nas áreas de *pruning*. Assim como na Seção 8.5.2, utilizaremos como padrão os valores $G = 3$, $ma = 1$, $mi = 3$.

8.6.1 Formas de processamento da matriz

Na primeira simulação (Figura 8.21), visualizamos a diferença do *pruning* para o perfect match com letras iguais e repetidas PM_r (Equação 8.32), o perfect match com letras distintas $PM^{1.0}$ (Equação 8.33) e semi-perfect match PM^p (Equação 8.34, com $p=0.6$) considerando as diferentes formas de processamento da matriz. Conforme pode ser visto, podemos observar uma maior área de *pruning* na matriz $PM^{1.0}$ e com o processamento em quadrado. Por outro lado, as menores áreas são observadas na matriz PM_r , visto que ocorrem *matches* em todas as posições da matriz e os valores das células da matriz são mais elevados que nos outros cenários. Podemos observar também que o processamento em quadrado invertido prejudica a eficácia do *pruning*, visto que esta forma de processamento atrasa a obtenção dos melhores escores da matriz. Em compensação, o processamento por quadrados adianta a obtenção dos melhores escores, melhorando a eficácia do método.

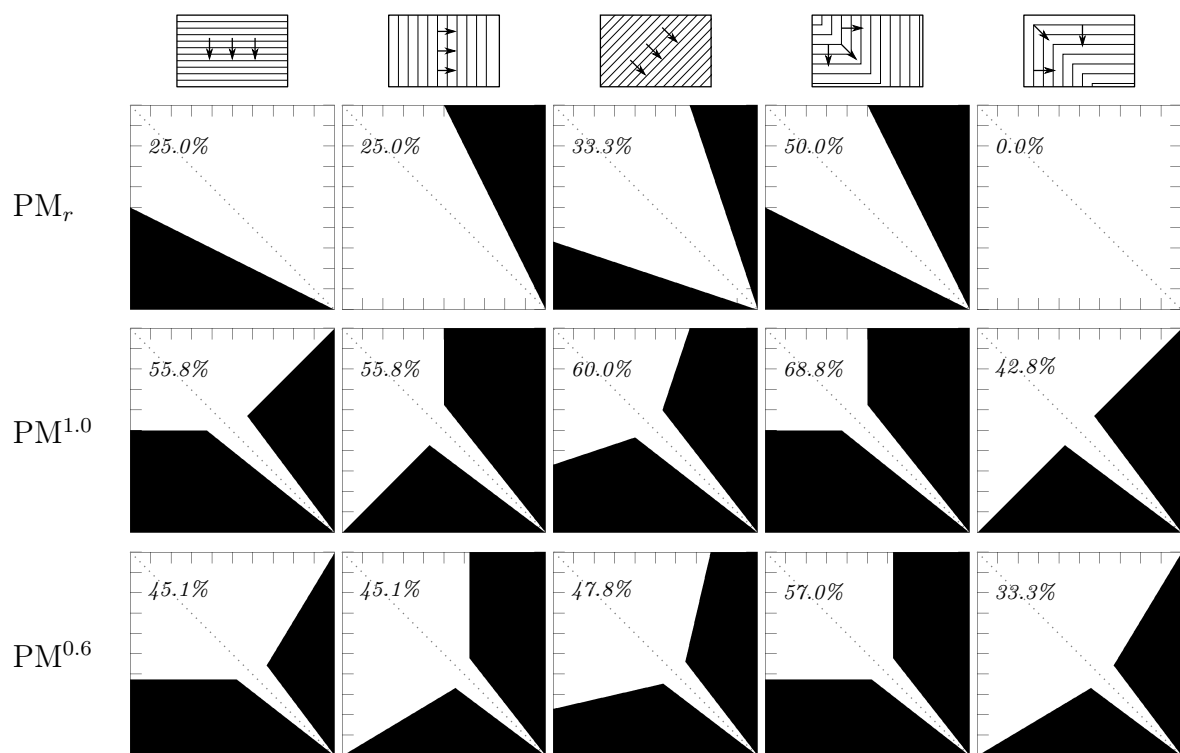


Figura 8.21: Eficácia de *pruning* para os cenários PM_r , $PM^{1.0}$ e PM^p em diferentes formas de processamento

8.6.2 Ângulos de processamento da Matriz

Na segunda simulação (Figura 8.22), consideramos vários ângulos θ para o processamento por inclinação. Nos cenários PM_r , $PM^{1.0}$ e PM^p , o *pruning* é mais eficaz quando $\theta = 45^\circ$, sendo que a eficácia é simétrica com ângulos menores e maiores que 45° . Por exemplo, a eficácia com o ângulo $\theta = 15^\circ$ é igual ao ângulo $\theta = 75^\circ$. O ângulo $\theta = 0^\circ$ é equivalente ao processamento por linhas e o ângulo $\theta = 90^\circ$ é equivalente ao processamento por colunas.

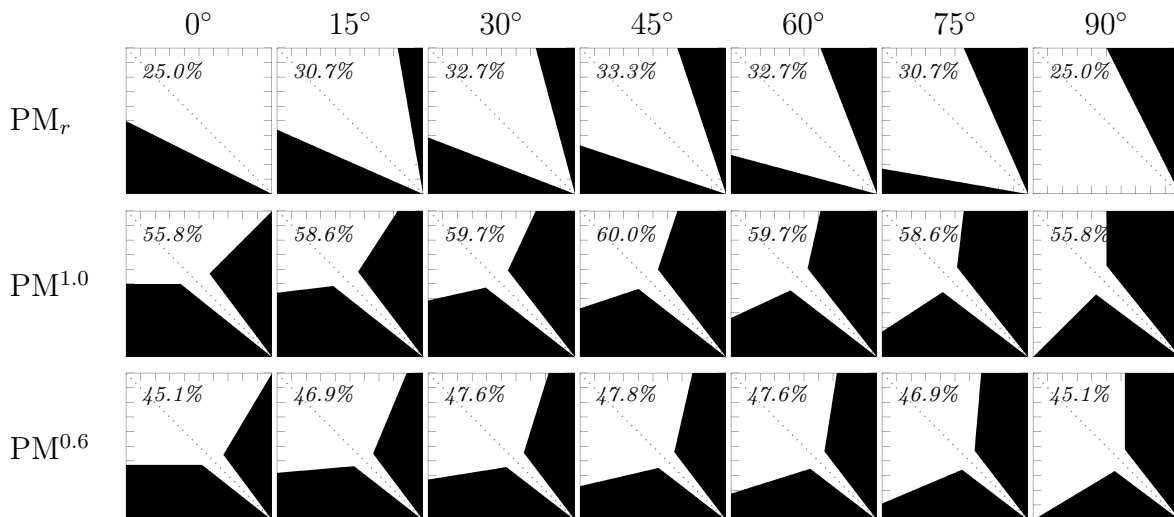


Figura 8.22: Variação do ângulo de processamento

8.6.3 Similaridade entre as sequências

Na terceira simulação (Figura 8.23), consideramos valores diferentes valores de p para o *Semi-Perfect Match* (PM^p). Conforme esperado, o *pruning* diminui sua eficácia quando diminuimos o valor de p . É importante notar que, conforme esperado, o processamento em quadrados consegue os melhores resultados em todos os cenários apresentados, pois nos cenários estudados o alinhamento ótimo está na diagonal principal, coincidindo com a posição do vértice de cada iteração de processamento. Também podemos ver que o *pruning* torna-se ineficaz para sequências com $p = 0$.

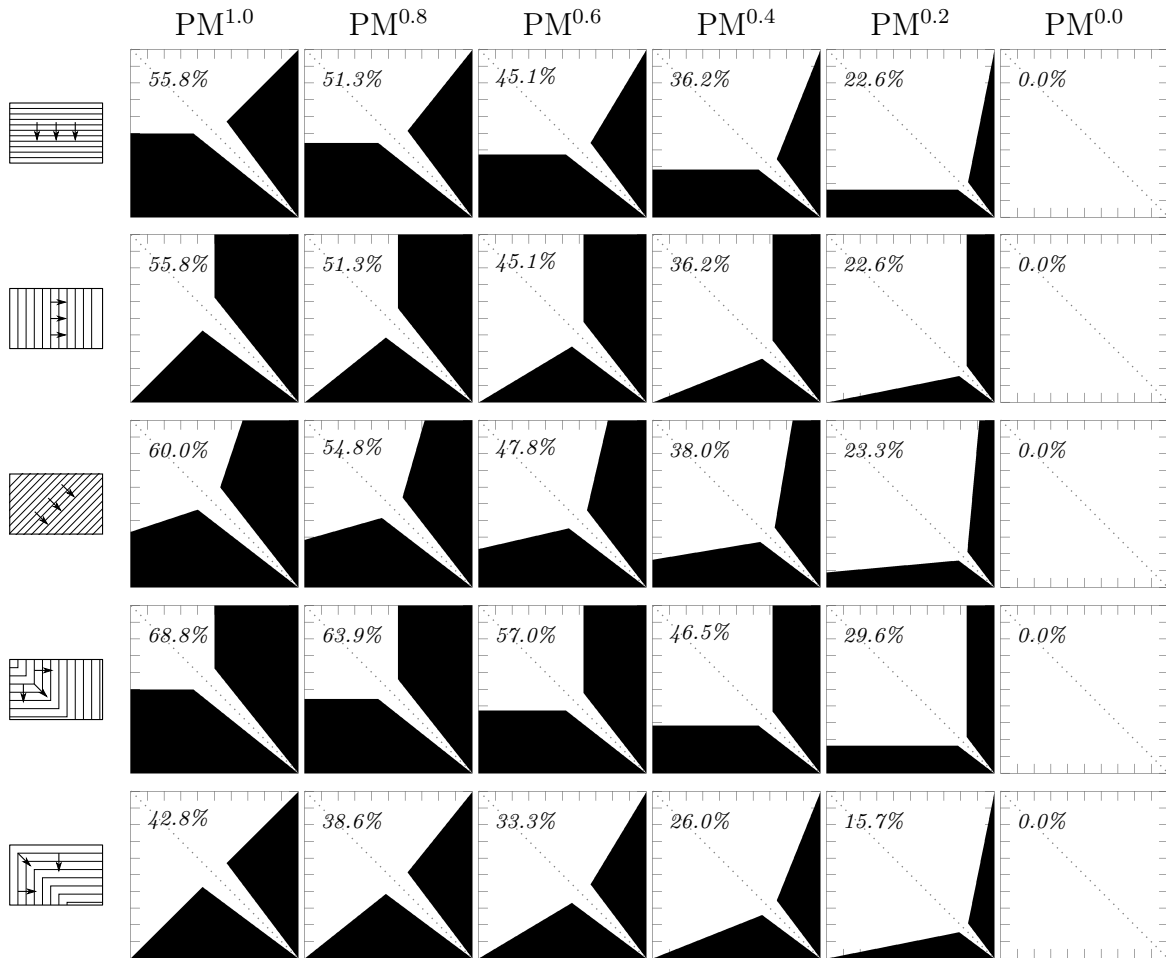


Figura 8.23: Variação da taxa p de crescimento do escore ótimo

8.6.4 Penalidade de Gaps

Na quarta simulação (Figura 8.24), consideramos valores diferentes penalidades de gap G . O *Block Pruning* aumenta a sua eficácia quando aumentamos o valor de G . Podemos observar que o ângulo de abertura formado pelas retas f_2 e f_4 (Figura 8.15) é reduzido quando aumentamos o valor de G , o que representa um maior decaimento dos valores da matriz causados por penalidades de *gap*. O valor de $G = 0$ é análogo o caso de perfect match com letras únicas e repetidas, onde podemos perceber a menor eficácia de *pruning*.

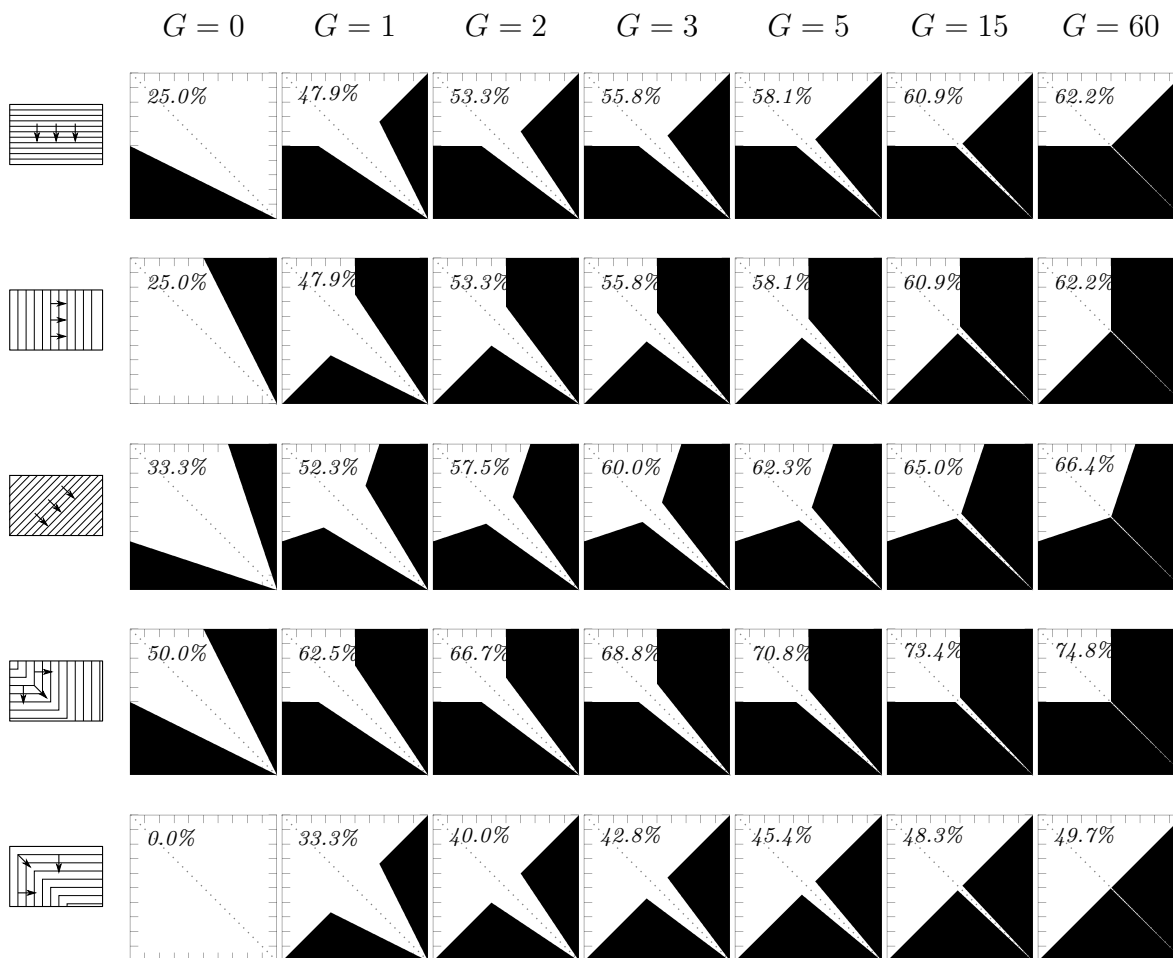


Figura 8.24: Variação da penalidade de *gap*

8.6.5 Sequências de tamanhos diferentes

Finalmente, na quinta simulação (Figura 8.25), apresentamos sequências de tamanhos diferentes, com proporção $\frac{n}{m}$ variando de 1 a 3. Em geral, o *pruning* aumenta a sua eficácia quando aumentamos a proporção $\frac{n}{m}$. O único caso em que verificou-se um decréscimo da eficácia foi quando a matriz é mais alongada no sentido horizontal e processamos linha por linha. Embora não apresentado na Figura 8.25, o mesmo estudo pode ser feito alongando a matriz na vertical ($\frac{n}{m} < 1$). Simetricamente, o único caso em que podemos ver a eficácia diminuída ocorre quando a matriz é mais alongada no sentido vertical e processamos coluna por coluna.

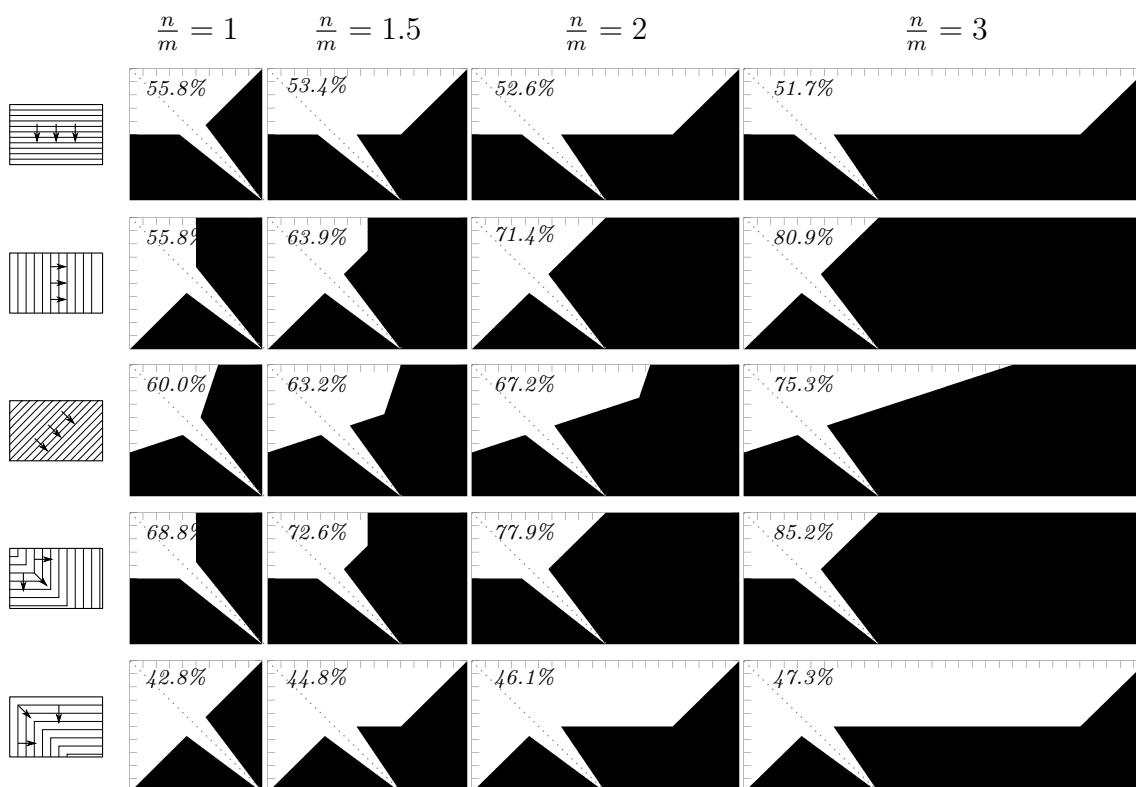


Figura 8.25: Variação da proporção do tamanho das sequências

8.7 Conclusão do Capítulo

O *Block Pruning* (BP) mostrou-se uma otimização de grande importância para acelerar o tempo de execução de comparações longas com alto grau de similaridade. Embora ela tenha sido inicialmente desenvolvida para o CUDAlign 2.1 (Capítulo 7), o presente capítulo apresentou uma generalização do BP para diferentes formas de processamento. Além disso, foi apresentada uma formalização do BP que permitiu estimar a eficácia de *pruning* em alguns cenários. Essas fórmulas possibilitaram identificar que a eficácia desta otimização é influenciada pela forma de processamento da matriz, o ângulo do *wavefront* (θ), a similaridade da sequência (ψ), a proporção das sequências ($\frac{m}{n}$) e os parâmetros do SW para *match* (ma), *mismatch* (mi) e *gaps* (G). Por exemplo, a modificação do ângulo

de processamento do *wavefront* pode alterar a eficácia do método de 25% para 33%. Com sequências diferentes, o descarte pode chegar a mais que 85% da matriz em alguns cenários. A ordem de processamento dos blocos faz com que a área descartada da matriz possa variar de 0% a 68% com duas sequências idênticas. Em relação aos parâmetros do SW, o aumento da penalidade de *gap* (G) melhora a eficácia de *pruning*, mas o aumento do *match* (ma) ou do *mismatch* (mi) diminui o número de células descartadas. Quando as sequências não são totalmente idênticas ($\psi < 1$), valores muito baixo de *match* (ma) também não são efetivos.

Por meio das fórmulas obtidas e de simulações, vimos que o processamento por ondas em forma de quadrado apresentou os melhores resultados para sequências onde o alinhamento ótimo encontra-se na diagonal principal. Em contrapartida, o processamento por quadrado invertido apresentou o pior desempenho. O processamento por linhas e colunas descreveram um resultado na mediana entre os processamentos por quadrado e quadrado invertido. Embora o CUDAlign utilize processamento por antidiagonais, no Capítulo 12 propomos uma implementação em CPU que utiliza o *wavefront* em forma de quadrados, apresentando a vantagem prática deste método.

Capítulo 9

CUDAlign 3.0: Comparação de sequências em Múltiplas GPUs

Apesar de o CUDAlign 2.1 ter conseguido acelerar a obtenção de resultados utilizando somente uma GPU, a comparação de sequências maiores que 33 MBP ainda demora mais que 8 horas para se completar (Seção 7.2). Sendo assim, ficou claro que comparações entre cromossomos muito longos (> 100 MBP) deveriam ser feitas com múltiplas GPUs. O CUDAlign 3.0 [36], versão do CUDAlign proposta neste capítulo, é capaz de executar uma comparação entre sequências muito longas utilizando diversas GPUs.

Nessa versão, foi proposta uma nova arquitetura capaz de distribuir o processamento de uma única comparação em várias GPUs simultaneamente. Desta forma, o poder computacional das GPUs é somado com o intuito de acelerar o tempo de execução do estágio 1, que é a etapa mais demorada do CUDAlign.

O restante deste capítulo está organizado da seguinte forma. Na Seção 9.1, a arquitetura da solução com múltiplas GPUs é apresentada. A Seção 9.2 apresenta o projeto dos *buffers* de comunicação. Na Seção 9.3, apresentamos fórmulas para previsão de desempenho em ambientes com múltiplas GPUs. Os resultados experimentais são apresentados e discutidos na Seção 9.4. Finalmente, o capítulo é concluído na Seção 9.5.

9.1 Arquitetura Multi-GPU

Na arquitetura desenvolvida para o CUDAlign 3.0, cada GPU é responsável por calcular um intervalo de colunas da matriz de programação dinâmica. O tamanho de cada intervalo de colunas é escolhido de forma a igualar o número de linhas processadas por segundo em cada GPU, criando então uma distribuição balanceada de carga entre todas as GPUs. Caso as GPUs possuam o mesmo poder de processamento, as colunas são distribuídas uniformemente, mas, em caso de GPUs heterogêneas, as colunas são distribuídas de acordo com a proporção entre suas capacidades de processamento. As GPUs vizinhas transferem continuamente células de suas colunas divisórias e, caso essa transferência não seja bem projetada, o tempo de execução de todas as demais GPUs será impactado. A Figura 9.1 ilustra uma divisão uniforme da matriz de programação dinâmica entre 4 GPUs.

Durante a execução do CUDAlign 3.0, cada GPU é associada a um processo e cada processo possui três *threads*: uma *thread* gerente e duas *threads* de comunicação. A *thread* gerente é responsável por gerenciar a execução da GPU e por transferir as células da matriz

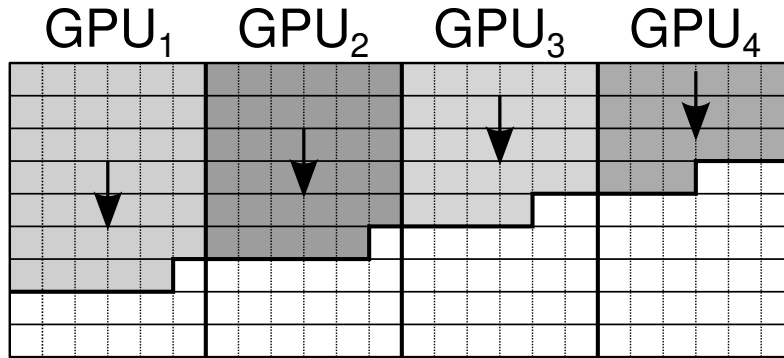


Figura 9.1: Divisão de colunas entre múltiplas GPUs.

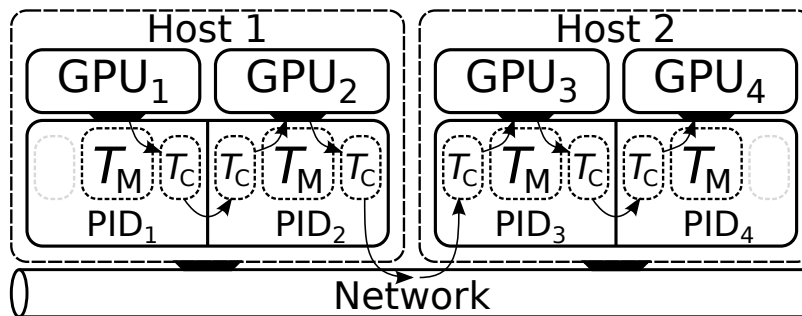


Figura 9.2: Encadeamento de *threads* em ambiente Multi-GPU.

entre a GPU e as *threads* de comunicação. Por sua vez, as *threads* de comunicação são responsáveis por transferir as células de maneira assíncrona entre os processos, através de *sockets*. A Figura 9.2 ilustra a comunicação entre 4 GPUs, onde cada *host* possui duas GPUs e cada GPU possui um processo associado, com uma *thread* gerente (T_M) e duas *threads* de comunicação (T_C), com exceção do primeiro e último processos, que possuem somente uma *thread* de comunicação.

9.2 Buffers de Comunicação

Cada *thread* de comunicação é associada a um *buffer* circular, que pode ser utilizado como *buffer* de saída ou de entrada. Os *buffers* desvinculam a comunicação do processamento, o que permite esconder a latência de comunicação entre as GPUs. A Figura 9.3 ilustra um encadeamento de *buffers* entre 4 GPUs. Os *buffers* I_2, I_3 e I_4 são os *buffers* de entrada e os *buffers* O_1, O_2 e O_3 são os *buffers* de saída. Cada par de *buffers* de entrada e saída ($O_{j-1} \rightarrow I_j$) de GPUs consecutivas estão continuamente transferindo dados. Estes *buffers* são responsáveis por esconder a latência da comunicação inter-processos de tal forma que o *overhead* e pequenas variações na qualidade do serviço de rede sejam pouco perceptíveis para o desempenho do *wavefront*. Além disso, a quantidade de dados aguardando em cada *buffer* é uma indicação da qualidade do balanceamento do processamento entre as GPUs. Por exemplo, o par de *buffers* ($O_1 \rightarrow I_2$) na Figura 9.3 possui mais dados aguardando no *buffer* de entrada I_2 que no *buffer* de saída O_1 , indicando que a GPU₁ está produzindo dados mais rápido que a GPU₂ pode processar. Já no par ($O_3 \rightarrow I_4$), os *buffers* estão

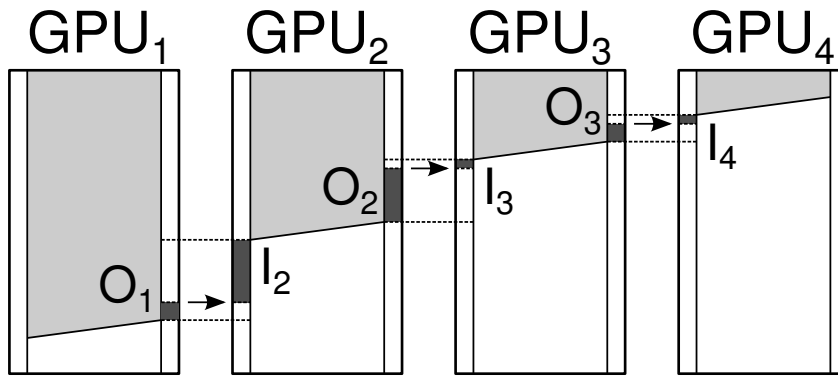


Figura 9.3: *Buffers* de comunicação Multi-GPU.

praticamente vazios, indicando que a GPU₃ e a GPU₄ estão processando praticamente na mesma velocidade.

Note que quando o *wavefront* entre as múltiplas GPUs está balanceado, o uso de todos os *buffers* é praticamente constante. Nesta situação, a quantidade de dados produzidos por uma GPU em um determinado período de tempo é completamente consumida pela próxima GPU, que por sua vez nunca se bloqueia aguardando dados da GPU anterior. Quando o *wavefront* está desbalanceado, os *buffers* de saída podem aumentar até que fiquem cheios, ou os *buffers* de entrada podem diminuir até que se tornem vazios, reduzindo a velocidade de computação devido a bloqueios na *thread* gerente.

9.3 Método de Previsão de desempenho

Em *clusters* com um grande número de GPUs, precisamos especificar o tempo máximo de execução das tarefas submetidas em suas filas. Para execuções que levam horas, subestimar o tempo de execução pode terminar a aplicação prematuramente e superestimar o tempo pode diminuir a prioridade da tarefa na fila de execução. Deste modo, um método para previsão de desempenho é bastante importante para o uso eficiente dos recursos. Adicionalmente, a previsão de *speedup* auxilia a definição do número apropriado de GPUs para obter uma eficiência de paralelismo satisfatória.

Considerando que as sequências S_0 e S_1 possuem tamanhos $|S_0| = m$ and $|S_1| = n$, o algoritmo de SW possui na complexidade $O(mn)$ de tempo (Seção 2.2.2). Embora a maior parte do tempo seja consumida com o cálculo da matriz de programação dinâmica, existem várias outras atividades que consomem tempo, tais como a inicialização das estruturas de dados, leitura das sequências, escrita de resultados, e todas essas operações precisam ser levadas em conta. Deste modo, o tempo de execução de uma comparação em uma única GPU pode ser prevista conforme a Equação 9.1 (Seção 5.4), onde c_1 , c_2 , c_3 e c_4 são constantes relacionadas com cada nó de processamento. A constante c_1 corresponde a procedimentos de inicialização. As constantes c_2 e c_3 correspondem ao tempo gasto com operações de complexidade $O(m)$ ou $O(n)$, tais como operações sobre vetores. A constante c_4 corresponde principalmente ao cálculo da matriz. As constantes c_1 , c_2 , c_3 e

c_4 podem ser obtidas estatisticamente por métodos de regressão sobre os tempos obtidos por comparações menores com tamanhos variados.

$$t(m, n) = c_1 + c_2m + c_3n + c_4mn \quad (9.1)$$

Com a fórmula de previsão de tempo para uma GPU (Equação 9.1), podemos derivar o número de células processadas por segundo (Equação 9.2). O limite de CUPS(m, n) quando m e n tendem ao infinito é igual a $\frac{1}{c_4}$, conforme visto na Equação 5.4

$$CUPS(m, n) = \frac{mn}{t(m, n)} = \frac{mn}{c_1 + c_2m + c_3n + c_4mn} \quad (9.2)$$

Para prever o tempo de execução para uma comparação em múltiplas GPUs, considera-se o tempo de execução da última GPU e o tempo que ela aguarda até que o *wavefront* tenha iniciado em todas as GPUs anteriores. Assim, a fórmula de previsão de desempenho é dada pela Equação 9.3.

$$T_p(m, n) = t(m, \frac{n}{p}) + (p - 1) \times t(\beta, \frac{n}{p})/2 \quad (9.3)$$

onde p é o número de GPUs, $t(m, n/p)$ é o tempo de execução total previsto para a última GPU, β o número de linhas necessárias para que o *wavefront* atinja seu paralelismo máximo e $t(\beta, n/p)/2$ é uma estimativa do tempo necessário para o início do processamento pela última GPU. O valor de β é definido como $\alpha \times B \times T$. Ressalta-se que essa análise considera que o ambiente com múltiplas GPUs é dedicado e homogêneo, de forma que as colunas são distribuídas de maneira uniforme ($\frac{n}{p}$) entre todas as p GPUs.

9.4 Resultados Experimentais

O CUDAlign 3.0 foi testado em três ambientes com múltiplas GPUs descritos a seguir:

- **Minotauro:** é um *cluster* de GPUs cujo desempenho máximo é equivalente a 185,78 TFlops. Ele é o sistema com maior eficiência de energia entre os sistemas baseados na arquitetura Fermi da NVIDIA, chegando a ocupar a 6^a colocação no www.green500.org considerando o tipo de processador. O *cluster* pertence ao *Barcelona Supercomputing Center* (BSC), Espanha, e contém 128 hosts do modelo Bull B505. Cada host possui dois processadores Intel Xeon hexa-core E5649 e duas GPUs Tesla M2090 da NVIDIA. Cada uma das GPUs possui 512 núcleos (Tabela 3.2). Os nós são interconectados por uma rede Infiniband de 40 Gbit/s.
- **Panoramix:** é uma máquina com GPUs heterogêneas. Este host encontra-se na *Universitat Politecnica de Catalunya* (UPC), Espanha, no *Department of Computer Architecture* (DAC). O Panoramix possui um processador quad-core Intel i7-930 conectado a três GPUs: 1 Tesla K20c e 2 Tesla C2050. A GPU Tesla K20c (arquitetura Kepler) contém 2496 núcleos e as GPUs Tesla C2050 (Arquitetura Fermi) contém 448 núcleos (Tabela 3.2).
- **Laico:** é o Laboratório de sistemas Integrados e Concorrentes, hospedado na Universidade de Brasília (UnB), no Departamento de Ciência da Computação. Para os

Tabela 9.1: Sequências utilizadas nos testes.

| Chr. | Homem | | Chimpanzé | | Escore |
|-------|--------------|---------|-------------|---------|----------|
| | Accession | Tamanho | Accession | Tamanho | |
| chr1 | NC_000001.10 | 249M | NC_006468.3 | 228M | 84608525 |
| chr19 | NC_000019.9 | 59M | NC_006486.3 | 64M | 17297608 |
| chr20 | NC_000020.10 | 63M | NC_006487.3 | 62M | 40050427 |
| chr21 | NC_000021.8 | 48M | NC_006488.2 | 46M | 36006054 |
| chr22 | NC_000022.10 | 51M | NC_006489.3 | 50M | 31510791 |

| Comp. | Sequência 1 | | Sequência 2 | | Escore |
|-------|-------------|---------|-------------|---------|----------|
| | Accession | Tamanho | Accession | Tamanho | |
| 5M | AE016879.1 | 5M | AE017225.1 | 5M | 5220960 |
| 10M | NC_017186.1 | 10M | NC_014318.1 | 10M | 10235188 |
| 23M | NT_033779.4 | 23M | NT_037436.3 | 25M | 9063 |

testes, foram selecionados 3 hosts, cada qual com uma GPU. As GPUs utilizadas foram uma GTX 580 (Fermi), com 512 núcleos, e duas GTX 680 (Kepler), com 1536 núcleos cada (Tabela 3.1). Os nós são interconectados por uma rede Ethernet de 1 Gbit/s.

Os resultados experimentais foram obtidos comparando sequências reais provenientes do National Center for Biotechnology Information (NCBI), disponíveis em www.ncbi.nlm.nih.gov. Nos experimentos, foram utilizados alguns dos cromossomos do chimpanzé e do homem. Também foram escolhidas algumas sequências menores para permitir uma melhor análise de escalabilidade. As sequências escolhidas estão descritas na Tabela 9.1, com comprimento variando de 5 MBP até 249 MBP. As sequências de 5 MBP a 23 MBP também foram utilizadas nos testes do CUDAlign 2.0 (Tabela 6.1) e do 2.1 (Seção 7.2). Utilizamos *buffers* de 8 MB, espaço suficiente para armazenar 1 milhão de células.

9.4.1 Tempos de Execução e GCUPS

No Minotauro, as sequências 5M, 10M e 23M, chr19, chr20, chr21 e chr22 foram comparadas com 1, 2, 4, 8 e 16 GPUs e os resultados estão apresentados na Tabela 9.2. As comparações que utilizaram 1 ou 2 GPUs foram realizadas em uma única máquina, visto que cada máquina possui duas GPUs. Já as execuções com mais de duas GPUs utilizaram mais de uma máquina. O tempo de execução variou de 1h18m (chr21 com 16 GPUs) até 33h20m (chr20 com 1 GPU). O desempenho obtido em GCUPS foi aproximadamente 32, 64, 128, 250 e 480 GCUPS utilizando 1, 2, 4, 8 e 16 GPUs do modelo Tesla M2090. A diferença entre o máximo e o mínimo GCUPS utilizando a mesma quantidade de GPUs foi bastante baixa para os cromossomos: 0,49%, 0,34%, 0,63%, 0,86% e 2,29% para, respectivamente, 1, 2, 4, 8 e 16 GPUs.

No Panoramic (Tabela 9.3), o CUDAlign 3.0 obteve desempenho em torno de 100 GCUPS, utilizando uma distribuição de colunas igual a 46,10% para a GPU K20c (Kepler) e 26,95% para cada uma das duas GPUs C2050 (Fermi). A variação do desempenho em GCUPS foi menos que 0,76% entre todas as sequências.

Tabela 9.2: Tempo de Execução e GCUPS no *cluster* Minotauro.

| Comp. | Tamanho | 1×M2090 | | 2×M2090 | | 4×M2090 | |
|-------|---------|---------|-------|---------|-------|---------|--------|
| | | Tempo | GCUPS | Tempo | GCUPS | Tempo | GCUPS |
| chr20 | 63M×62M | 119980s | 32,43 | 60098s | 64,74 | 30360s | 128,15 |
| chr19 | 59M×64M | 115933s | 32,46 | 58189s | 64,67 | 29318s | 128,36 |
| chr22 | 51M×50M | 78536s | 32,49 | 39432s | 64,71 | 19949s | 127,91 |
| chr21 | 48M×46M | 68665s | 32,59 | 34681s | 64,52 | 17541s | 127,56 |
| 23M | 23M×25M | 17419s | 32,42 | 8805s | 64,15 | 4504s | 125,41 |
| 10M | 10M×10M | 3241s | 32,33 | 1667s | 62,85 | 877s | 119,52 |
| 5M | 5M×5M | 851s | 32,11 | 449s | 60,94 | 248s | 110,26 |

| Comp. | Tamanho | 8×M2090 | | 16×M2090 | |
|-------|---------|---------|--------|----------|--------|
| | | Tempo | GCUPS | Tempo | GCUPS |
| chr20 | 63M×62M | 15452s | 251,78 | 7969s | 488,21 |
| chr19 | 59M×64M | 14954s | 251,66 | 7744s | 485,97 |
| chr22 | 51M×50M | 10172s | 250,85 | 5307s | 480,83 |
| chr21 | 48M×46M | 8963s | 249,63 | 4688s | 477,27 |
| 23M | 23M×25M | 2355s | 239,86 | 1277s | 442,31 |
| 10M | 10M×10M | 485s | 216,11 | 288s | 363,90 |
| 5M | 5M×5M | 149s | 182,85 | 100s | 273,54 |

Tabela 9.3: Tempo de Execução e GCUPS no Panoramax (1×K20c + 2×C2050) e no Laico (1×GTX580 + 2×GTX680).

| Comp. | Tamanho | Panoramix | | Laico | |
|-------|---------|-----------|--------|--------|--------|
| | | Tempo | GCUPS | Tempo | GCUPS |
| chr20 | 63M×62M | 38537s | 100,96 | 27728s | 140,31 |
| chr19 | 59M×64M | 37252s | 101,02 | 26957s | 139,60 |
| chr22 | 51M×50M | 25416s | 101,38 | 18180s | 140,36 |
| chr21 | 48M×46M | 22238s | 100,62 | 16024s | 139,63 |

No Laico (Tabela 9.3), o desempenho obtido foi aproximadamente 140 GCUPS, com uma distribuição de colunas igual a 30,71% para a GTX 580 e 36,64% para cada uma das duas GTX 680. A variação de desempenho entre todas as comparações de sequências também foi muito baixa, ficando abaixo de 0,55%.

9.4.2 Comparação do Cromossomo 1

Os cromossomos 1 do homem e do chimpanzé foram comparados no Laico (3 GPUs) e no Minotauro (16, 32 e 64 GPUS). Os resultados estão apresentados na Tabela 9.4. O tempo de execução desta comparação no Laico foi de 5 dias e 13 horas (118,75 GCUPS). No Minotauro, o tempo de execução foi de 1 dia e 10 horas (459,92 GCUPS) com 16 GPUs, 17 horas e 30 minutos (903,48 GCUPS) com 32 GPUs e 9 horas e 9 minutos (1726,47 GCUPS) com 64 GPUs.

Tabela 9.4: Comparação entre os cromossomos 1 do homem e do chimpanzé no Laico e no Minotauro.

| GPUs | Tempo | GCUPS |
|---------------------|---------|---------|
| Laico - 3 GPUs | 479276s | 118,75 |
| Minotauro - 16 GPUs | 123742s | 459,92 |
| Minotauro - 32 GPUs | 62992s | 903,48 |
| Minotauro - 64 GPUs | 32965s | 1726,47 |

Tabela 9.5: Constantes para previsão de desempenho.

| GPU | c_1 | c_2 | c_3 | c_4 |
|--------|-------|-----------|-----------|-------------|
| GTX680 | 2,41 | 3,610e-07 | 6,663e-07 | 2,04015e-11 |
| K20c | 6,42 | 7,216e-07 | 7,161e-07 | 2,12281e-11 |
| GTX580 | 2,97 | 8,521e-07 | 7,255e-07 | 2,28598e-11 |
| M2090 | 2,99 | 8,727e-07 | 9,227e-07 | 3,06697e-11 |
| C2050 | 6,92 | 8,942e-07 | 7,308e-07 | 3,62182e-11 |

9.4.3 *Overhead* de comunicação

Para medir o *overhead* de comunicação, comparamos regiões do cromossomo 21 formando comparações de tamanho $16M \times 4M$, $16M \times 2M$ e $16M \times 1M$. Comparando o tempo de execução destas comparações com e sem o código que executa a transferência de dados, verificamos que esta transferência introduz um *overhead* de no máximo 0.3% do tempo total.

Embora a atividade de comunicação seja sobreposta com a de computação, é importante saber se a vazão da rede é suficiente para transferir os dados sem causar gargalos. Caso contrário, os *buffers* irão encher até bloquear a *thread* de execução. Durante a comparação dos cromossomos 21, a taxa medida de transferência de células por GPU foi de 10.370 células por segundo no Minotauro (16 GPUs). Considerando que cada célula transferida possui 8 bytes (64 bits), esta execução consumiu 664 kbit/s no Minotauro, um valor desprezível quando comparado com sua rede Infiniband de 40 Gbit/s.

9.4.4 Previsão de Desempenho

A fórmula de previsão de desempenho para 1 GPU (Equação 5.2) apresentada na Seção 5.4 foi avaliada para os 5 modelos de GPU utilizados nos experimentos. Para executar o método de previsão, dividimos o cromossomo 21 em 5 sequências menores com os seguintes tamanhos: 1M, 3M, 5M, 7M e 9M. Então, geramos 5×5 diferentes combinações dessas sequências, resultando em 25 comparações com matrizes de tamanhos variando de 10^{12} a 10^{14} células. Os tempos de execução destas comparações foram obtidos em cada uma das GPUs e assim obtivemos as constantes c_1 , c_2 , c_3 e c_4 listadas na Tabela 9.5. A Figura 9.4(a) utiliza pontos para mostrar os tempos de execução reais de cada uma das 25 comparações na placa M2090 e linhas pontilhadas para apresentar os tempos previstos. Para o Minotauro, o erro da regressão comparando as 25 comparações realizadas foi de, no máximo, 0,7%.

Comparamos então os tempos previstos utilizando a Equação 9.1 e os tempos reais de execução com 1 GPU no Minotauro (M2090). A diferença entre o tempo previsto e

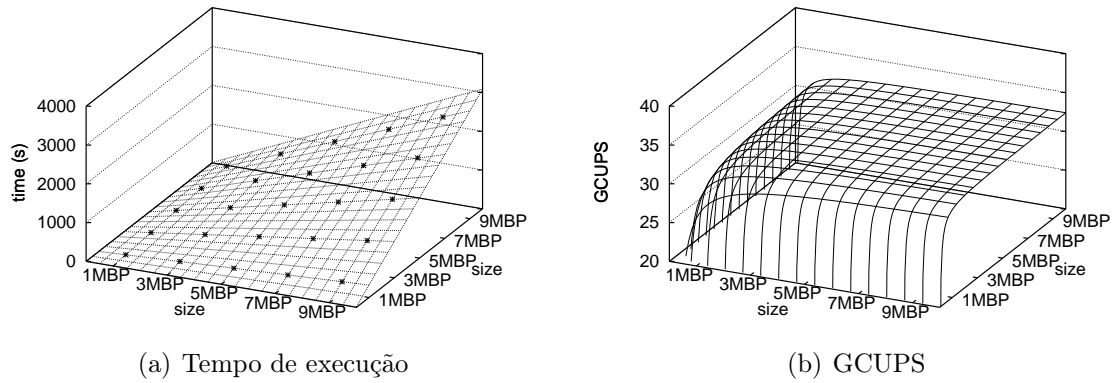


Figura 9.4: Previsões de tempo e GCUPS no Minotauro (1×M2090)

o real foi menor que 0,45%, mostrando que a previsão foi bastante acurada para estas comparações.

A Figura 9.4(b) apresenta o CUPS previsto para um intervalo de tamanhos, calculado pela Equação 9.2. A curva observada com sequências menores que 1M indica que a inicialização e outros *overheads* do algoritmo possuem maior impacto para comparações com sequências pequenas.

O limite da métrica CUPS, definida por $CUPS_{max} = 1/c_4$ (Seção 5.4), foi estimada para 32,6 GCUPS na placa M2090. Observe que na Tabela 9.2 o GCUPS para 1×M2090 é próximo do valor $CUPS_{max}$ de 32,6 GCUPS.

Dada a equação de previsão do tempo de execução para múltiplas GPUs M2090 (Equação 9.3), a Figura 9.5 apresenta o *speedup* previsto no Minotauro para comparações utilizando 1 a 2048 GPUs (2^9 a 2^{20} núcleos CUDA). Note que a curva de *speedup* é mais próxima do linear quando as sequências são maiores. A Tabela 9.6 apresenta o *speedup* previsto para cada comparação considerando: a) 16 GPUs; b) o maior número de GPUs que podemos utilizar mantendo a eficiência de paralelismo maior que 90%; e c) o maior *speedup* previsto.

Tabela 9.6: *Speedup* previsto no Minotauro.

| Cmp. | 16 GPUs | 90% efic. | | <i>Speedup</i> Máx. | |
|-------|----------------|-----------|----------------|---------------------|----------------|
| | <i>Speedup</i> | GPUs | <i>Speedup</i> | GPUs | <i>Speedup</i> |
| chr1 | 15,84 | 138 | 124,28 | 1044 | 385,70 |
| chr20 | 15,34 | 36 | 32,42 | 273 | 100,31 |
| chr19 | 15,31 | 34 | 30,69 | 268 | 97,83 |
| chr22 | 15,18 | 29 | 26,15 | 221 | 81,32 |
| chr21 | 15,12 | 27 | 24,37 | 207 | 76,17 |
| 23M | 14,14 | 13 | 11,80 | 104 | 37,96 |
| 10M | 11,64 | 6 | 5,45 | 45 | 16,48 |
| 5M | 8,10 | 3 | 2,78 | 23 | 8,46 |

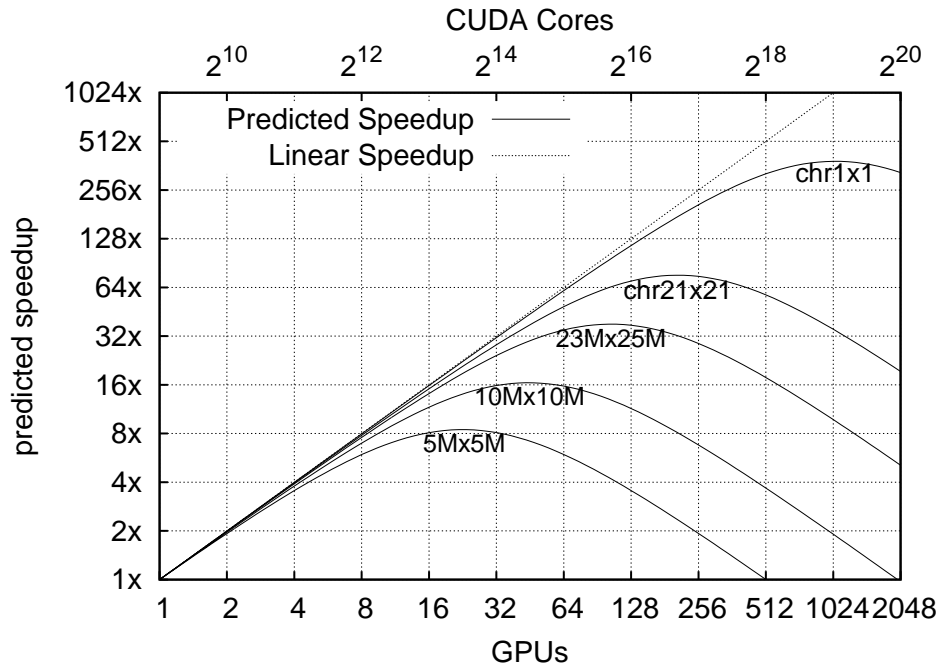


Figura 9.5: *Speedup* previsto no Minotauro.

9.4.5 *Speedups* obtidos

Considerando os resultados obtidos no *cluster* Minotauro, a Figura 9.6 apresenta o *speedup* das comparações até 16 GPUs (8192 núcleos CUDA). Nota-se que o *speedup* aumenta quando aumentamos o tamanho das sequências, sendo que as comparações dos cromossomos apresentaram um *speedup* próximo do linear. Com 16 GPUs, os *speedups* foram 8,5x (5M), 11,3x (10M), 13,6x (23M), 14,7x (chr21), 14,8x (chr22), 15,0x (chr19) e 15,1x (chr20). As diferenças entre os *speedups* previstos e os reais são menores que 5%, como pode ser visto na Tabela 9.7.

Tabela 9.7: Erros da previsão de *speedup* no Minotauro

| Cmp. | Erro absoluto % | | | |
|-------|-----------------|-------|-------|--------|
| | 2×GPU | 4×GPU | 8×GPU | 16×GPU |
| chr20 | 0,09% | 0,40% | 1,06% | 1,86% |
| chr19 | 0,10% | 0,30% | 1,13% | 2,24% |
| chr22 | 0,08% | 0,59% | 1,17% | 2,55% |
| chr21 | 0,66% | 1,09% | 1,79% | 3,22% |
| 23M | 0,34% | 1,10% | 2,39% | 3,65% |
| 10M | 1,04% | 2,39% | 4,36% | 3,42% |
| 5M | 1,60% | 3,41% | 3,80% | 4,78% |

A Figura 9.7 mostra o desempenho relativo (GCUPS) de todas as comparações de cromossomos, exceto o cromossomo 1. Abaixo do eixo x , pode-se ver o número de células calculadas em cada comparação. Podemos notar que o GCUPS é praticamente idêntico considerando uma carga de trabalho variando de 2.24×10^{15} a 3.49×10^{15} células.

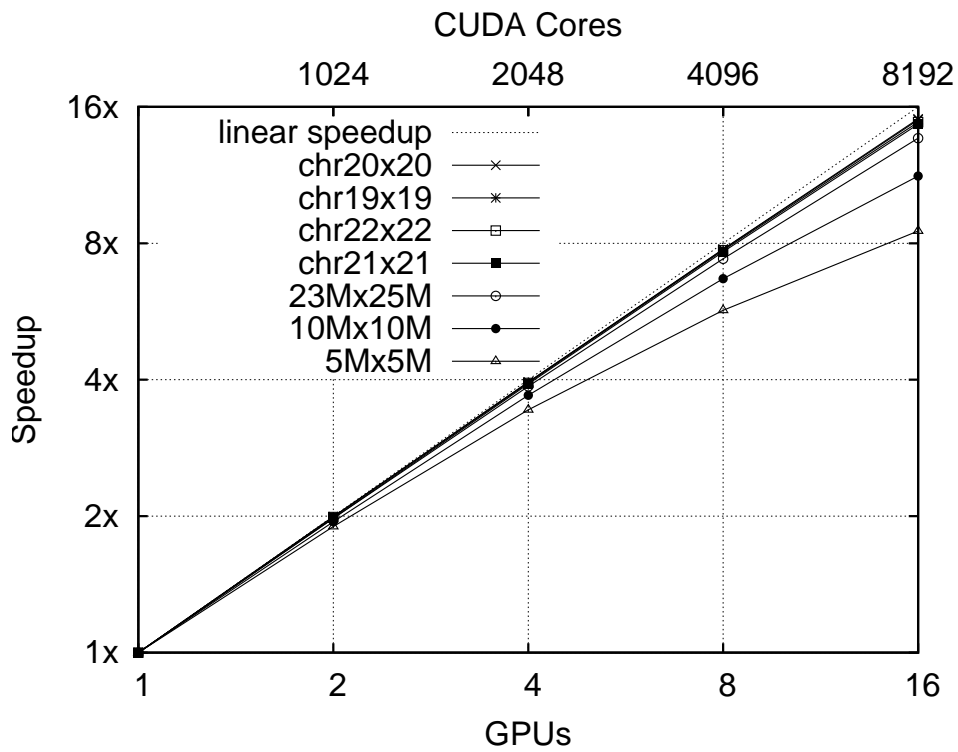


Figura 9.6: *Speedups* obtidos no Minotauro.

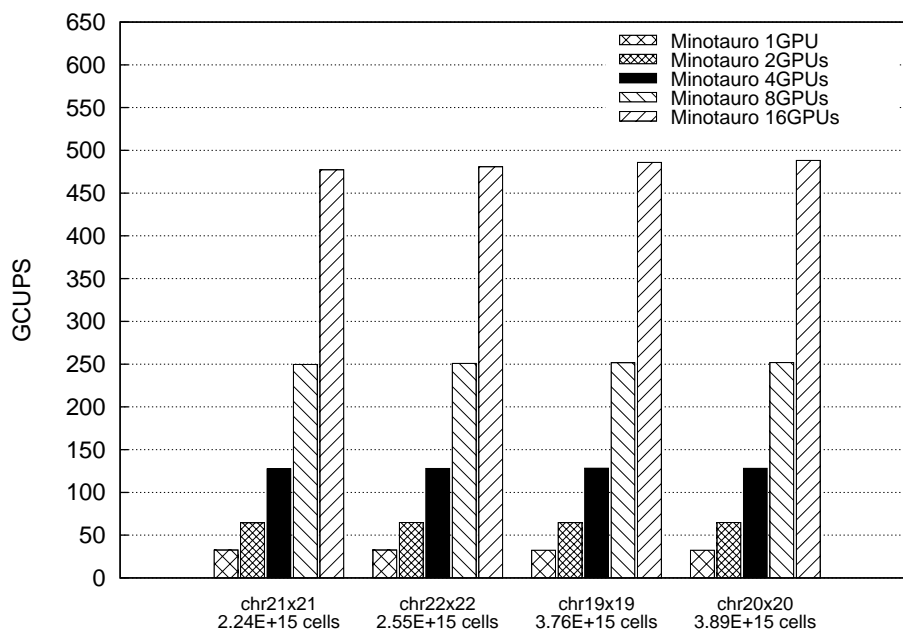


Figura 9.7: Desempenho obtido quando variamos os tamanhos das seqüências

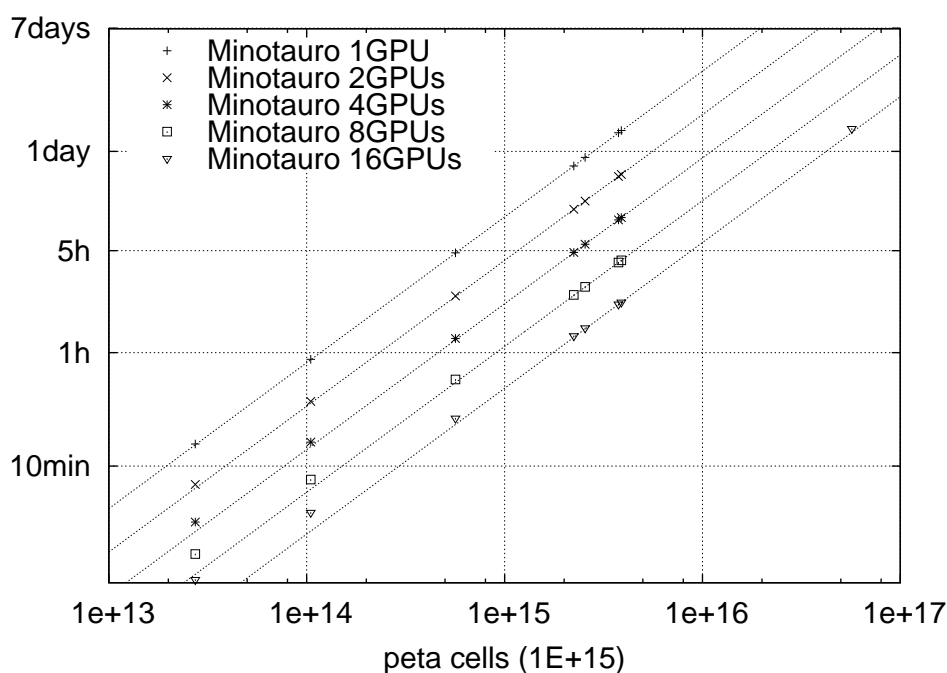


Figura 9.8: Tempos de execução em escala logarítmica.

A Figura 9.8 apresenta, em escala logarítmica, o tempo de execução de todas as comparações listadas na Tabela 9.1 do Minotauro (até 16 GPUs). As linhas tracejadas representam o máximo GCUPS obtido para 1, 2, 4, 8 e 16 GPUs. De cima para baixo, estas linhas correspondem, respectivamente, a 32,59, 64,74, 128,36, 251,78 and 488,21 GCUPS. Na Figura 9.8, podemos notar que as execuções mantêm suas altas taxas de GCUPS desde seqüências menores (5M) até seqüências muito grandes (249M). A comparação dos cromossomos 1 do homem e do chimpanzé possui um *overhead* adicional devido a remoção da textura durante a execução, visto que este tipo de memória possui um limite de 134 milhões (2^{27}) de elementos. Mesmo assim, o GCUPS máximo obtido nessa comparação (Tabela 9.4) foi bastante expressivo comparado com os trabalhos relacionados (Tabela 4.1).

9.5 Conclusão do Capítulo

O CUDAlign 3.0 mostrou-se uma ferramenta bastante eficiente para comparar seqüências longas de DNA em *clusters* com múltiplas GPUs. Por meio do CUDAlign 3.0, fomos capazes de comparar seqüências de até 249 milhões de pares de base (MBP), obtendo um pico de desempenho de 1726 GCUPS com 64 GPUs homogêneas. A estratégia de paralelismo proposta também mostrou-se eficiente em ambientes com GPUs heterogêneas. O método de previsão de desempenho proposto foi bastante acurado para estimar o tempo de execução e o *speedup*, permitindo uma alocação racional dos recursos disponibilizados. Entretanto, o CUDAlign 3.0 só apresenta como resultado o escore ótimo de similaridade entre as seqüências, sem apresentar o alinhamento completo. O Capítulo 10 apresentará

novos métodos e otimizações que permitem obter, de maneira eficiente, o alinhamento completo com múltiplas GPUs.

Capítulo 10

CUDAlign 4.0: Obtenção de alinhamentos em múltiplas GPUs

No presente capítulo, propomos e avaliamos o CUDAlign 4.0, uma ferramenta capaz de recuperar alinhamentos ótimos longos com múltiplas GPUs. O grande desafio do CUDAlign 4.0 foi paralelizar de maneira eficiente a etapa de recuperação do alinhamento ótimo (estágios 2 a 5 do CUDAlign), que é inerentemente sequencial. Para tanto, propomos aqui a estratégia *Incremental Speculative Traceback* (IST) que usa o tempo onde uma GPU estaria ociosa para especular valores de *crosspoints* e recuperar partes do alinhamento com base nesses valores. Os resultados obtidos com o IST mostram que seu *overhead* é muito pequeno e que, na comparação dos cromossomos 4 do homem e do chimpanzé, o IST é 21 vezes mais rápido do que a estratégia sem especulação.

A organização deste capítulo é a seguinte. Na Seção 10.1 apresentamos as mudanças feitas no estágio 1 do CUDAlign 3.0. Duas estratégias de recuperação do alinhamento ótimo para múltiplas GPUs são propostas na Seção 10.2. A Seção 10.3 apresenta e discute os resultados experimentais. Finalmente, a Seção 10.4 conclui o capítulo.

10.1 Estratégia Multi-GPU para o Estágio 1

Para recuperar o alinhamento (*traceback*) em múltiplas GPUs, duas modificações foram introduzidas no estágio 1 do CUDAlign 3.0 (Capítulo 9). Primeiramente, o limite de 2^{27} elementos na textura da GPU impede o seu uso para sequências maiores que 134 MBP. No CUDAlign 3.0, a textura foi removida para sequências maiores que este limite. Compilando esta versão sem textura em uma arquitetura com *compute capability* 2.0 (`-arch=sm_20`), observou-se uma perda de desempenho de 7,0% na GTX 580 (Fermi) e de 15,5% na GTX 680 (Kepler). Para permitir o alinhamento de sequências de tamanho arbitrário sem a remoção da textura, o CUDAlign 4.0 implementa um conceito de subparticionamento, em que a matriz de programação dinâmica é subdividida em quadrantes que caibam dentro do limite de 2^{27} elementos na textura. Cada subpartição é calculada individualmente, uma após a outra, de forma que a primeira linha/coluna de uma subpartição é obtida da última linha/coluna das subpartições vizinhas. O cálculo de cada subpartição utiliza os mesmos mecanismos de paralelismo já apresentados nas versões anteriores do CUDAlign (Capítulos 5, 6, 7 e 9).

A segunda modificação inserida no estágio 1 é a escrita de linhas especiais no sistema de arquivo. De maneira semelhante ao CUDAlign 2.0 e 2.1, estas linhas são utilizadas nos estágios subsequentes de forma a encontrar os pontos onde o alinhamento ótimo passa através delas. As linhas especiais são salvas em um diretório exclusivo para cada GPU. Adicionalmente, foi implementado um mecanismo que também permite salvar as linhas em memória RAM, evitando assim o *overhead* causado pelas operações de entrada/saída em disco.

10.2 *Traceback* em Múltiplas GPUs

Para a obtenção do alinhamento completo utilizando múltiplas GPUs, foram desenvolvidas duas estratégias de *traceback*. A primeira estratégia, utilizada como *baseline*, foi chamada de *Pipelined Traceback* (PT), onde cada GPU executa o *traceback* em um *pipeline* de 3 estágios (estágios 2 a 4 do CUDAlign). A segunda estratégia, chamada de *Incremental Speculative Traceback* (IST), utiliza as GPUs ociosas dentro do *pipeline* para especular a localização do alinhamento ótimo, com o objetivo de acelerar a execução do *traceback* (estágios 2 a 5).

O algoritmo de *traceback* para uma única GPU (Algoritmo 7) será utilizado como base das explicações dos métodos Multi-GPU. No Algoritmo 7, a linha 1 executa o estágio 1 e obtém o escore ótimo e sua coordenada. O estágio 2 (linha 2) é executado a partir do escore ótimo e uma lista de *crosspoints* é obtida. Os estágios 3 (linha 3) e 4 (linha 4) recebem a lista de *crosspoints* do estágio anterior e criam listas com maior número de *crosspoints*. O estágio 5 (linha 5) recebe os *crosspoints* do estágio 4 e gera um arquivo binário contendo o alinhamento ótimo. Por fim, o estágio 6 (linha 6) transforma o arquivo binário em um arquivo com a representação textual do alinhamento.

Algorithm 7 *Traceback* em uma única GPU

| | |
|--|--------------------------|
| 1: $best \leftarrow \text{STAGE1}()$ | ▷ Fase 1 (GPU) |
| 2: $list_2 \leftarrow \text{STAGE2}(best)$ | ▷ <i>Traceback</i> (GPU) |
| 3: $list_3 \leftarrow \text{STAGE3}(list_2)$ | ▷ <i>Traceback</i> (GPU) |
| 4: $list_4 \leftarrow \text{STAGE4}(list_3)$ | ▷ <i>Traceback</i> (CPU) |
| 5: $bin \leftarrow \text{STAGE5}(list_4)$ | ▷ <i>Traceback</i> (CPU) |
| 6: $txt \leftarrow \text{STAGE6}(bin)$ | ▷ Conversão (CPU) |

As estratégias PT e IST serão explicadas nas subseções 10.2.1 e 10.2.2, respectivamente.

10.2.1 *Pipelined Traceback* (PT) - Estratégia de *baseline*

A estratégia multi-GPU utilizada como *baseline* (*Pipelined Traceback* - PT) executa o estágio 1 modificado (Seção 10.1) e cria um *pipeline* para calcular os estágios 2 a 4 do CUDAlign para diferentes partições em paralelo. Para fins de simplicidade desta seção, sempre que dissermos que a GPU_{*i*} está processando um dos estágios 4, 5 ou 6, isto significa que o processo associado à GPU_{*i*} está executando o determinado estágio em CPU. Adicionalmente, sem perda de generalidade, admitimos que a coordenada que marca o final do alinhamento ótimo encontra-se na última GPU (GPU_{*p*}), onde *p* é o número de GPUs.

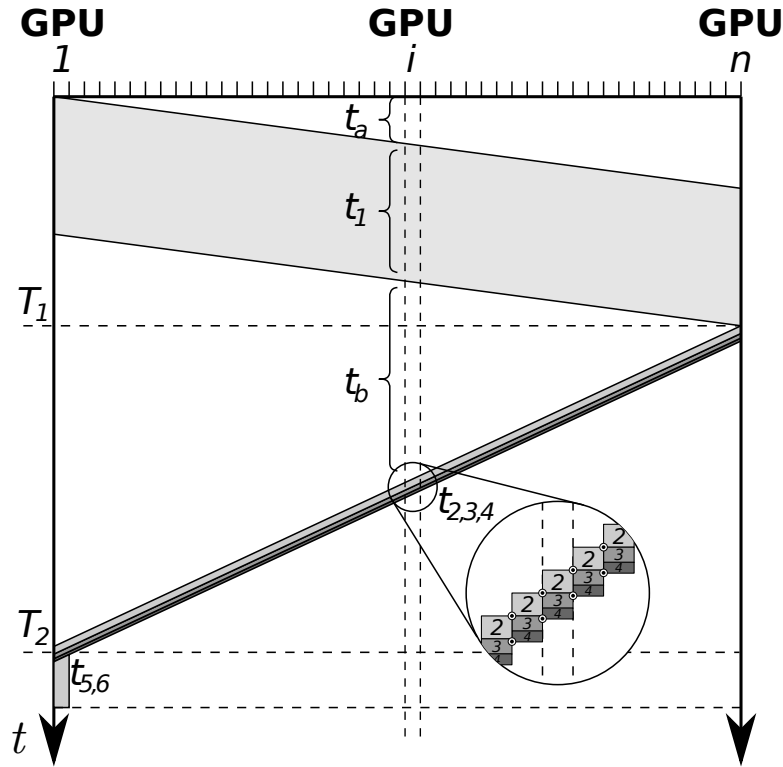


Figura 10.1: Linhas de tempo para o *Pipelined Traceback* (PT)

Após o estágio 1 ser completado, a última GPU inicia a execução do estágio 2 a partir da última partição. Sempre que um *crosspoint* de uma coluna intermediária for encontrado, sua coordenada é enviada para a GPU à esquerda (GPU_{i-1}), a qual será responsável por calcular a próxima partição vizinha. A GPU à esquerda deve então iniciar o cálculo do estágio 2 para esta nova partição. Uma cadeia de dependência é formada entre as partições (Figura 6.1(b)) e, conseqüentemente, entre as GPUs. Esta dependência cria um caminho crítico de execução que resulta em um tempo ocioso entre o fim do estágio 1 e o início do estágio 2, exceto para a última GPU. Entretanto, após as dependências do estágio 2 serem resolvidas, a execução dos estágios 2, 3 e 4 podem ser calculadas em *pipeline*. Esta estratégia é utilizada para todas as GPUs. Os estágios 5 e 6 exigem menos tempo computacional e, por isso, eles são executados apenas pela primeira GPU, utilizando como entrada todos os *crosspoints* encontrados nos estágios anteriores pelas demais GPUs.

A Figura 10.1 ilustra a linha de tempo da execução do *Pipelined Traceback*. O eixo y refere-se ao tempo e o eixo x representa cada uma das GPUs utilizada, sendo que cada ponto do gráfico mostra qual estágio estava sendo executado em cada GPU ao longo do tempo. A área branca no topo do gráfico (t_a) ilustra o início do preenchimento do *wavefront*. As áreas cinzas representam o tempo computacional utilizado para os estágios 1 a 6 ($t_{1,2,3,4,5,6}$). A área branca (t_b) entre os estágios 1 e 2 é o tempo ocioso apresentado após cada GPU terminar o estágio 1 para a sua partição, enquanto aguarda o *crosspoint* resultante do estágio 2 da GPU à sua direita. Considerando todas as GPUs, o tempo de término do estágio 1 é representado por T_1 e do estágio 2 é representado por T_2 .

Análise de Complexidade Considerando que p GPUs são utilizadas no estágio 2 e que existem mais GPUs que linhas especiais, podemos estimar grosseiramente que cada GPU irá calcular uma área de $\frac{m}{p} \times \frac{n}{p}$ células, resultando em uma complexidade de tempo de $O(\frac{mn}{p^2})$ por GPU. Visto que a execução das p GPUs possui uma cadeia de dependências, sua execução é serializada e o tempo de execução total é $O(\frac{mn}{p})$ — a mesma complexidade de tempo do estágio 1. Embora a estratégia de *pipeline* paralelize a execução dos estágios 2, 3 e 4, a cadeia de execução do estágio 2 faz parte de um mesmo caminho crítico de execução (Figura 10.1). Este problema será resolvido com o uso da estratégia *Incremental Speculative Traceback* (IST) proposta na Seção 10.2.2.

Pseudocódigo O pseudocódigo para a estratégia *Pipeline Traceback* (PT) está apresentado no Algoritmo 8, o qual modifica o *traceback* original do CUDAlign 2.1 (Algoritmo 7) para executar em múltiplas GPUs. Utilizamos a variável n para representar a n -ésima GPU e $Recv_i$ e $Send_j$ são as chamadas para recebimento e envio de mensagens da i -ésima e j -ésima GPUs, respectivamente. Para simplificar o pseudocódigo, ignoramos alguns casos de borda (primeira e última GPU), onde $Recv_i$ e $Send_j$ não são executados quando i e j estão fora dos limites. Utilizando a distribuição de colunas discutida no CUDAlign 3.0, o estágio 1 (linha 1) é executado para as colunas atribuídas à n -ésima GPU e a variável $best'_n$ recebe o máximo escore desta GPU e a posição do escore nestas colunas. Então, o melhor escore acumulado $best_n$ das GPUs 1 a n é passado através das GPUs (linhas 2 a 4). Na última GPU, a variável $crosspoint$ irá conter o escore ótimo de todas as GPUs e a sua coordenada. Em seguida, o estágio 2 irá iniciar a execução (linha 10) a partir da posição com o melhor escore (se ele estiver dentro do intervalo de colunas da última GPU). Além disso, a coordenada superior-esquerda armazenada em $list_2[0]$ (linha 11), encontrada na primeira coluna do intervalo, é enviada para a GPU à esquerda (linha 12), a qual a recebe na variável $crosspoint$ (linha 8). Enquanto a GPU que acabou de receber a coordenada calcula o estágio 2, a GPU que enviou a coordenada executa em *pipeline* os estágios 3 e 4 (linhas 13 e 14).

Após cada GPU terminar o cálculo do estágio 4, cada GPU irá receber as coordenadas calculadas pela GPU à direita (linha 15), concatenar as próprias coordenadas a esta lista, e enviar a lista resultante para a GPU à esquerda (linha 20). Este processo continua até a primeira GPU (ou a que possui o início do alinhamento). A primeira GPU calcula o alinhamento completo de todas as micro partições por meio dos estágios 5 e 6 (linhas 17 e 18), utilizando as listas de coordenadas ($list_4 + list'_4$) de todas as GPUs.

10.2.2 *Incremental Speculative Traceback* (IST)

Ao analisar várias execuções do *traceback* com o *Pipelined Traceback* (Seção 10.2.1), é possível observar que os *crosspoints* encontrados nas colunas intermediárias tendem a coincidir com as posições onde ocorrem os escores máximos dessas colunas (máximo local). A estratégia *Incremental Speculative Traceback* (IST) utiliza esta ideia para estimar onde o alinhamento ótimo irá cruzar as colunas intermediárias, permitindo que o estágio 2 seja executado em paralelo para múltiplas partições, mesmo antes de serem conhecidos os resultados ótimos do estágio 1 nas últimas GPUs.

Em paralelo ao cálculo especulativo, o estágio 2 também inicia sua execução na última GPU, utilizando o escore ótimo real (máximo global) conforme descrito na Seção 10.2.1.

Algorithm 8 *Pipelined Traceback* (nth GPU)

```
1:  $best'_n \leftarrow \text{STAGE1}()$ 
2:  $\text{RCV}_{n-1}(best_{n-1})$ 
3:  $best_n \leftarrow \max(best_{n-1}, best'_n)$ 
4:  $\text{SEND}_{n+1}(best_n)$ 
5: if  $n$  is last GPU then
6:    $crosspoint \leftarrow best_n$ 
7: else
8:    $\text{RCV}_{n+1}(crosspoint)$ 
9: end if
10:  $list_2 \leftarrow \text{STAGE2}(crosspoint)$ 
11:  $left\_crosspoint \leftarrow list_2[0]$ 
12:  $\text{SEND}_{n-1}(left\_crosspoint)$ 
13:  $list_3 \leftarrow \text{STAGE3}(list_2)$ 
14:  $list_4 \leftarrow \text{STAGE4}(list_3)$ 
15:  $\text{RCV}_{n+1}(list'_4)$ 
16: if  $n$  is first GPU then
17:    $bin \leftarrow \text{STAGE5}(list_4 + list'_4)$ 
18:    $txt \leftarrow \text{STAGE6}(bin)$ 
19: else
20:    $\text{SEND}_{n-1}(list_4 + list'_4)$ 
21: end if
```

Com o IST, entretanto, a GPU que recebe o *crosspoint* correto (o ponto que realmente pertence ao alinhamento ótimo) irá verificar se o alinhamento a partir deste ponto já foi calculado corretamente durante a fase de especulação. Caso tenha sido, esta GPU não irá recomputar a partição e irá redirecionar o próximo *crosspoint* para a GPU à esquerda. Caso contrário, esta GPU irá recalculer a partição com o *crosspoint* correto.

Também é possível observar que em regiões com muitos *gaps* e *mismatches* os *crosspoints* corretos são mais difíceis de prever. Desta forma, em vez de somente especular sobre os melhores *crosspoints* das colunas intermediárias, o IST utiliza os resultados parciais especulados pelas GPUs vizinhas. Desta forma, cada GPU irá redirecionar o *crosspoint* encontrado durante a fase de especulação para as outras GPUs, as quais irão utilizar como novas especulações (exceto se o *crosspoint* recebido já tiver sido calculado anteriormente). Esta estratégia executa uma especulação incremental, aumentando a probabilidade de sucesso a medida que novos valores são especulados. É importante ressaltar que os resultados de cada especulação, tais como os escores e as linhas especiais, precisam ser armazenados temporariamente em uma área de memória que chamamos de *cache*. O número de especulações é limitado por uma constante *max_spec_tries*, de forma a limitar o uso de memória deste *cache*. Por meio do *cache*, os resultados dos valores anteriormente especulados podem ser rapidamente obtidos, sem a necessidade de um novo cálculo da partição.

A Figura 10.2 ilustra o *Incremental Speculative Traceback*. A execução do estágio 1 é idêntica ao do *Pipelined Traceback* (Seção 10.2.1), mas a especulação do estágio 2 é executada durante o intervalo de tempo que, na estratégia PT, as GPUs estavam ociosas. A especulação incremental é ilustrada como várias faixas de recomputação, no sentido diagonal para o canto inferior esquerdo. A faixa de recomputação que se inicia na última GPU refere-se ao caminho crítico de execução do estágio 2, o qual se apresenta em faixas diagonais nos momentos de recálculo devido às especulações que falharam (i.e. que pre-

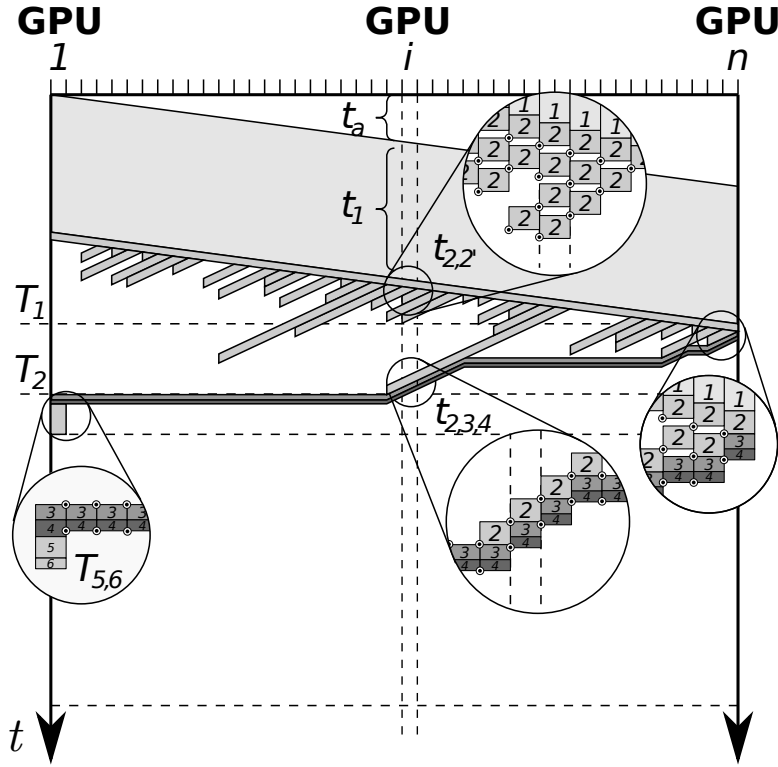


Figura 10.2: Linhas de tempo para o *Incremental Speculative Traceback* (IST)

viram *crosspoints* diferentes dos corretos). Note que os estágios 3 a 6 seguem a execução do estágio 2 para os *crosspoints* corretos, que fazem parte do alinhamento ótimo.

Análise de Complexidade A análise de complexidade do estágio 2 com IST, considerando p GPUs, pode ser formalizada com base no tempo gasto nas faixas de recomputação. Inicialmente, observa-se que as especulações do estágio 2 iniciam-se imediatamente após o término do estágio 1, sendo que este término ocorre em momentos diferentes em cada GPU, devido à propagação do *wavefront*. Seja T_1^i o tempo de execução do estágio 1 na GPU i , definido pela Equação 10.1, onde t_0 é o deslocamento de tempo de início entre GPUs sucessivas, causado pela propagação do *wavefront*.

$$T_1^i = T_1^1 + (i - 1) \cdot t_0 \quad (10.1)$$

Seja l_i o comprimento da faixa de recomputação que se inicia na GPU i e termina na GPU $i-l_i$. Além disso, seja t_2 o tempo médio em que o estágio 2 é executado para uma partição. Desta forma, a faixa de recomputação que se inicia na GPU i ocorre no intervalo de tempo iniciando em T_1^i e terminando em $T_1^i + l_i \cdot t_2$.

Seja T_2 o tempo adicional executado pelo estágio 2 em todas as p GPUs após a última GPU terminar a execução do estágio 1. Este tempo adicional depende do comprimento l_i e do tempo inicial de cada faixa de recomputação, observado que uma faixa longa que se iniciou nas primeiras GPUs pode terminar antes de uma faixa curta que se iniciou nas

últimas GPUs. Com isso, T_2 é definido pela Equação 10.2.

$$\begin{aligned}
T_2 &= \max_{1 \leq i \leq p} (T_1^i + l_i \cdot t_2) - T_1^p \\
&= \max_{1 \leq i \leq p} (T_1^1 + (i-1) \cdot t_0 + l_i \cdot t_2) - T_1^1 + (p-1) \cdot t_0 \\
&= \max_{1 \leq i \leq p} ((i-p) \cdot t_0 + l_i \cdot t_2)
\end{aligned} \tag{10.2}$$

Considerando que t_2 é $O(\frac{mn}{p^2})$, a complexidade de tempo do estágio 2 é de $O(l\frac{mn}{p^2})$, onde l ($1 \leq l \leq p$) é um valor que depende no comprimento e na posição das faixas de recomputação. No melhor caso, onde todas as especulações foram corretas, temos que $l = 1$ e a complexidade de tempo será $O(\frac{mn}{p^2})$. No pior caso, onde todas as especulações foram incorretas, temos que $l = p$ e a complexidade de tempo será $O(\frac{mn}{p})$, que é igual à complexidade de tempo do PT (Seção 10.2.1).

Para estimar o valor l , observa-se que as faixas são normalmente pequenas, mas elas se apresentam longas em duas circunstâncias. A primeira ocorre em regiões com grande ocorrência de *mismatches* e *gaps*. Pequenas variações no alinhamento dentro destas regiões podem gerar maior quantidade de *matches* e produzir escores máximos locais que não fazem parte do alinhamento ótimo. Estas variações forçam que o alinhamento especulado siga por um caminho alternativo que, posteriormente, terá penalidades muito maiores, divergindo do alinhamento ótimo esperado. A segunda circunstância ocorre quando existe uma repetição de pequenos padrões de repetição de DNA, tanto causada por repetições em *tandem* ou repetições do caracteres “N” geradas por regiões não sequenciadas na montagem do genoma (quantidade que tende a ser reduzida em cada nova versão de montagem). O padrão de repetição do DNA pode produzir uma repetição do escore local máximo em várias posições da coluna intermediária, reduzindo assim a probabilidade de acertar a especulação.

Pseudocódigo O pseudocódigo para a estratégia *Incremental Speculative Traceback* é apresentado no Algoritmo 9. Sem perda de generalidade, assumiremos que o escore ótimo encontrado pelo estágio 1 encontra-se na última GPU. Para representar o algoritmo IST, as linhas 5 a 25 do Algoritmo 9 substituem as linhas 5 a 12 do Algoritmo 8. No Algoritmo 9, o melhor escore de cada coluna intermediária é utilizado por cada GPU como o primeiro *crosspoint* especulado. Em seguida, o laço de especulação (linhas 12 a 25) calcula o estágio 2 com diferentes coordenadas *crosspoint*, até que um *crosspoint* não especulado (i.e. correto) seja recebido.

Durante a fase de especulação, se o *crosspoint* especulado não tiver sido calculado (i.e. se ele não estiver no *cache*), o estágio 2 é executado (linha 14) e o resultado é inserido no *cache* (linha 15), caso contrário o resultado é obtido do *cache* (linha 17). A coordenada superior à esquerda armazenada em $list_2[0]$ (linha 19), calculada pelo estágio 2, é enviada para a GPU à esquerda (linha 21) e um novo *crosspoint* especulado é recebido da GPU à direita (linha 23).

A recuperação do alinhamento ótimo inicia-se na última GPU, que utiliza a coordenada do escore ótimo como o primeiro *crosspoint* (linha 6) e o define como sendo do tipo “não especulado” (linha 7). Sempre que uma GPU receber um *crosspoint* não especulado, o *crosspoint* enviado para a próxima GPU é marcado como sendo do mesmo tipo não especulado. Isto faz com que esta marcação seja cascadeada ao longo de todas as GPUs (linha 20), finalizando o laço de execução do estágio 2 (linha 25). Os estágios 3 e 4

são executados em *pipeline* após o término do estágio 2, assim como discutido *Pipelined Traceback*.

O Algoritmo 9 foi simplificado por questões de apresentação, mas na implementação real existe a limitação do tamanho do cache (*max_spec_tries*). Desta forma, quando o tamanho do *cache* atingir este limite, o IST para de especular novos *crosspoints* e aguarda por um *crosspoint* do tipo não especulado.

Algorithm 9 Speculative Traceback (*n*th GPU)

```

4: ...
5: if n is last GPU then
6:   crosspoint  $\leftarrow$  bestn
7:   crosspoint.speculated  $\leftarrow$  false
8: else
9:   crosspoint  $\leftarrow$  best score in last column
10:  crosspoint.speculated  $\leftarrow$  true
11: end if
12: repeat
13:   if Cache[crosspoint] is empty then
14:     list2  $\leftarrow$  STAGE2(crosspoint)
15:     Cache[crosspoint]  $\leftarrow$  list2
16:   else
17:     list2  $\leftarrow$  Cache[crosspoint]
18:   end if
19:   left_crosspoint  $\leftarrow$  list2[0]
20:   left_crosspoint.speculated  $\leftarrow$  crosspoint.speculated
21:   SENDn-1(left_crosspoint)
22:   if crosspoint.speculated then
23:     RECVn+1(crosspoint)
24:   end if
25: until crosspoint.speculated is false
26: ...

```

10.3 Resultados Experimentais

Nesta seção, apresentamos experimentos realizados no *XSEDE Keeneland Full Scale (KFS) system*, que é um *cluster* com 264 nós de computação do modelo HP SL250G8. Cada nó é equipado com dois processadores Intel Sandy Bridge (Xeon E5) com 8 núcleos cada, 32 GB de memória RAM e três GPUs do modelo NVIDIA Tesla M2090 (Tabela 3.2). Os nós se comunicam por meio de uma rede de interconexão Mellanox FDR InfiniBand de 40 Gbits/s e são conectados a um sistema de arquivos distribuído Lustre. Nos nossos testes, até 384 GPUs foram utilizadas. O CUDAlign 4.0 utilizou a linguagem de programação C++, CUDA 5.5 e as bibliotecas Pthreads e Sockets. Os *kernels* do CUDAlign foram executados com $B = 64$ blocos CUDA, $T = 128$ *threads* em cada bloco e cada *thread* processa $\alpha = 4$ linhas da matriz. De maneira similar ao CUDAlign 3.0 (Seção 9.4), utilizamos 8 MB de buffer na comunicação de colunas intermediárias entre as GPUs, espaço este suficiente para armazenar 1 milhão de células.

10.3.1 Sequências utilizadas nos testes

Os experimentos utilizaram sequências reais de DNA obtidas do repositório do National Center for Biotechnology Information (NCBI), disponível em *www.ncbi.nlm.nih.gov*. As comparações foram realizadas entre todos os cromossomos homólogos do homem (GRCh37) e do chimpanzé (panTro4), produzindo um conjunto de teste com 25 pares de sequências. Os tamanhos totais do genoma humano e do chimpanzé foram, respectivamente, 3,34 GBP (bilhões de pares de base) e 3,16 GBP.

Os números de acesso e os tamanhos dos cromossomos estão apresentados na Tabela 10.1. Os tamanhos variam de 26 MBP (milhões de pares de base) até 249 MBP, produzindo matrizes de programação dinâmica de 1,56 a 56,91 peta células. Por questões de validação, o escore local ótimo, o comprimento do alinhamento, cobertura e o percentual de *matches*, *mismatches* e *gaps* também estão apresentados na mesma tabela. Os parâmetros utilizados para o SW foram: *match*: +1; *mismatch* -3; *first gap*: -5; *extension gap*: -2. Em um primeiro momento, comparamos todos os cromossomos apenas para obter os alinhamentos ótimos, utilizando um número variado de GPUs em cada caso, mas sem mensurar o desempenho de forma detalhada.

Todos os alinhamentos locais ótimos estão apresentados na Figura 10.3, onde podemos ver que muitas comparações produziram um alinhamento local cobrindo praticamente toda a extensão da matriz, representando a alta similaridade entre os cromossomos homólogos dessas duas espécies.

Tabela 10.1: Cromossomos usados nos testes do CUDAlign 4.0.

| Chr. | Human (GRCh37) | | Chimp. (panTro4) | | Peta Cells | Escore | Tamanho | Cobert. | <i>Matches</i> | <i>Mismt.</i> | <i>Gaps</i> |
|---------|----------------|------|------------------|------|------------|-----------|-----------|---------|----------------|---------------|-------------|
| | Accession | Tam. | Accession | Tam. | | | | | | | |
| chr01 | NC_000001.10 | 249M | NC_006468.3 | 228M | 56.91 | 84608525 | 255117470 | 99,1% | 80,1% | 5,3% | 14,6% |
| chr02-A | NC_000002.11 | 243M | NC_006469.3 | 114M | 27.63 | 74861783 | 118554635 | 73,4% | 89,0% | 3,1% | 7,9% |
| chr02-B | | | NC_006470.3 | 248M | 60.20 | 93139254 | 135655977 | 53,2% | 90,5% | 2,1% | 7,4% |
| chr03 | NC_000003.11 | 198M | NC_006490.3 | 202M | 40.07 | 152598201 | 205950608 | 99,9% | 92,2% | 2,0% | 5,8% |
| chr04 | NC_000004.11 | 191M | NC_006471.3 | 193M | 36.99 | 81532618 | 107996353 | 54,6% | 92,7% | 1,9% | 5,4% |
| chr05 | NC_000005.9 | 180M | NC_006472.3 | 183M | 33.04 | 63924833 | 87400843 | 46,9% | 92,2% | 2,7% | 5,1% |
| chr06 | NC_000006.11 | 171M | NC_006473.3 | 173M | 29.54 | 120465367 | 176613042 | 99,4% | 90,6% | 2,9% | 6,5% |
| chr07 | NC_000007.13 | 159M | NC_006474.3 | 162M | 25.75 | 93144399 | 164099089 | 97,8% | 87,8% | 3,4% | 8,8% |
| chr08 | NC_000008.10 | 146M | NC_006475.3 | 144M | 21.07 | 101825467 | 138467654 | 92,7% | 92,2% | 2,2% | 5,6% |
| chr09 | NC_000009.11 | 141M | NC_006476.3 | 138M | 19.46 | 76043976 | 145206895 | 99,8% | 86,3% | 5,6% | 8,1% |
| chr10 | NC_000010.10 | 136M | NC_006477.3 | 134M | 18.10 | 80128465 | 141139131 | 99,9% | 87,1% | 3,4% | 9,6% |
| chr11 | NC_000011.9 | 135M | NC_006478.3 | 133M | 17.97 | 80183754 | 138964666 | 99,7% | 87,7% | 4,7% | 7,6% |
| chr12 | NC_000012.11 | 134M | NC_006479.3 | 134M | 17.97 | 49076981 | 67897406 | 49,2% | 91,8% | 2,5% | 5,7% |
| chr13 | NC_000013.10 | 115M | NC_006480.3 | 115M | 13.26 | 64071638 | 116494539 | 99,0% | 87,8% | 7,9% | 4,3% |
| chr14 | NC_000014.8 | 107M | NC_006481.3 | 107M | 11.44 | 82247932 | 108107078 | 98,4% | 92,9% | 1,8% | 5,3% |
| chr15 | NC_000015.9 | 103M | NC_006482.3 | 100M | 10.21 | 64957513 | 103625609 | 99,1% | 89,3% | 3,9% | 6,8% |
| chr16 | NC_000016.9 | 90M | NC_006483.3 | 90M | 8.13 | 45421118 | 95068007 | 99,8% | 84,5% | 4,8% | 10,7% |
| chr17 | NC_000017.10 | 81M | NC_006484.3 | 83M | 6.71 | 22218058 | 35392160 | 41,5% | 88,9% | 3,1% | 8,0% |
| chr18 | NC_000018.9 | 78M | NC_006485.3 | 77M | 5.98 | 46959759 | 61253576 | 77,3% | 93,1% | 2,0% | 4,9% |
| chr19 | NC_000019.9 | 59M | NC_006486.3 | 64M | 3.76 | 17297608 | 36386342 | 56,2% | 84,8% | 4,6% | 10,6% |
| chr20 | NC_000020.10 | 63M | NC_006487.3 | 62M | 3.89 | 40050427 | 65286930 | 99,9% | 88,2% | 2,6% | 9,2% |
| chr21 | NC_000021.8 | 48M | NC_006488.2 | 46M | 2.24 | 36006054 | 48579349 | 99,0% | 91,9% | 1,1% | 7,1% |
| chr22 | NC_000022.10 | 51M | NC_006489.3 | 50M | 2.55 | 31510791 | 51929087 | 98,9% | 88,5% | 3,8% | 7,7% |
| chrX | NC_000023.10 | 155M | NC_006491.3 | 157M | 24.35 | 59862909 | 157365502 | 95,4% | 82,1% | 7,1% | 10,8% |
| chrY | NC_000024.9 | 59M | NC_006492.3 | 26M | 1.56 | 1394673 | 2283191 | 6,0% | 88,1% | 2,0% | 10,0% |

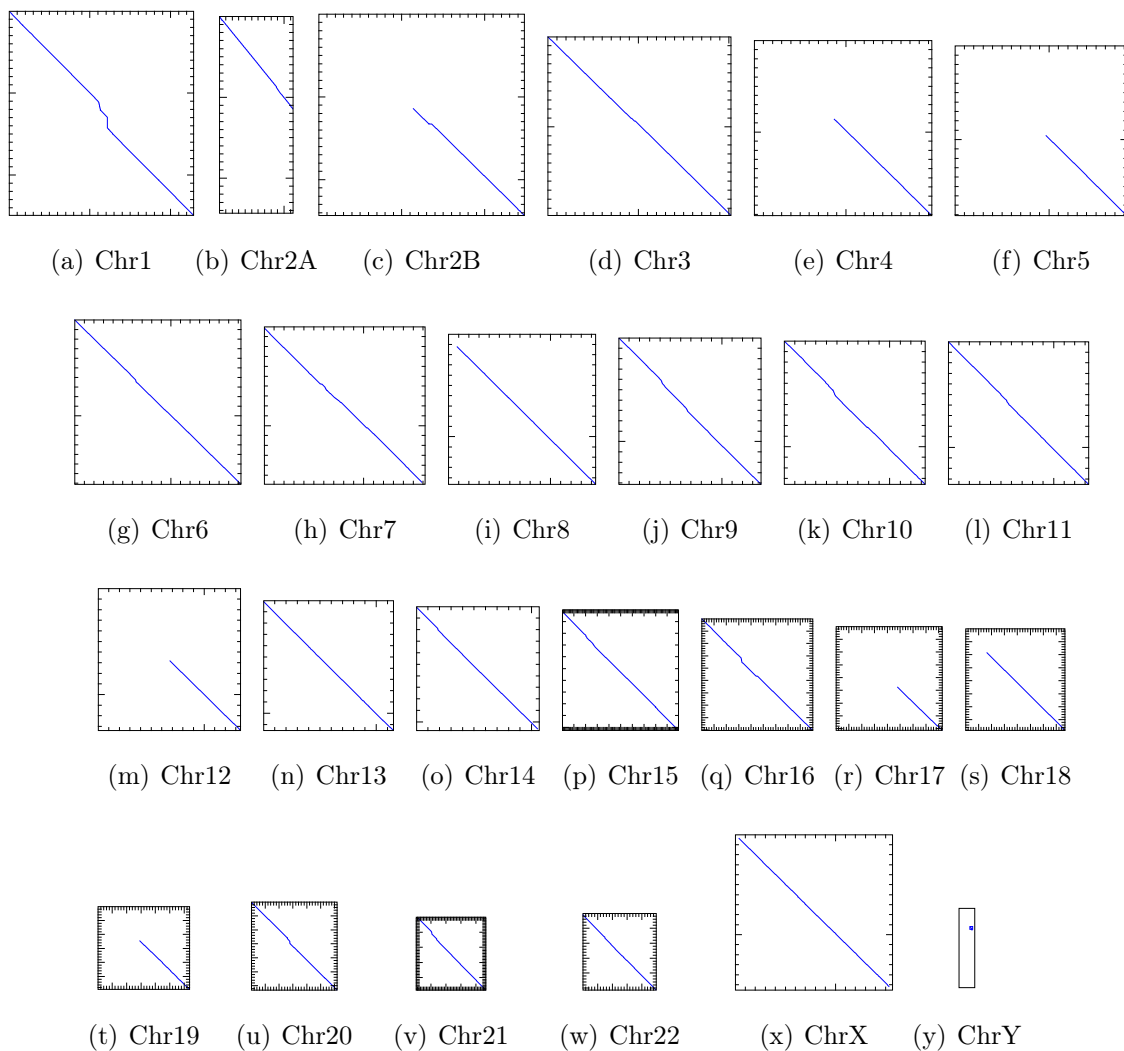


Figura 10.3: Alinhamentos locais ótimos entre os cromossomos homólogos do homem e do chimpanzé

10.3.2 Desempenho do *Pipelined Traceback*

Esta seção avalia o desempenho e a escalabilidade da estratégia PT (Seção 10.2.1) utilizando as sequências chr22 e chr16. O número de nós variou de 1 a 128 (3 a 384 GPUs). A Tabela 10.2 apresenta o resultado experimental da execução completa (*Total*) e o tempo dividido entre estágio 1 e o *traceback* (estágios 2 a 6). Conforme apresentado, os *speedups* obtidos com 128 nós para o chr22 e chr16 foram, respectivamente, $26,9\times$ e $29,7\times$ (21,0% e 23,2% de eficiência de paralelismo).

A separação do tempo total de execução mostra que o estágio 1 do CUDAlign possui uma escalabilidade muito melhor. Analisando apenas o estágio 1, o *speedup* obtido foi de $84,0\times$ (chr22) e $97,3\times$ (chr16) com 128 nós (65,6% e 76,0% de eficiência de paralelismo), resultando em um pico de desempenho de 8,3 e 9,7 TCUPS (Trilhões de células processadas por segundo) para o chr22 e chr16, respectivamente. Os resultados do estágio 1 para o chr22 e chr16 são consistentes aos obtidos no CUDAlign 3.0 (Capítulo 9).

Em contrapartida, a fase de *traceback* do PT não foi capaz de utilizar o ambiente

Tabela 10.2: Tempos de execução do chr22 e do chr16 (Estratégia PT).

| | Nós | Total | | Estágio 1 | | Traceback |
|-------|---------|--------|--------------|-----------|--------------|-----------|
| | /GPUs | Tempo | Spd. | Tempo | Spd. | Tempo |
| chr22 | 1/3 | 26660s | 1,0x | 25809s | 1,0x | 851s |
| | 2/6 | 13986s | 1,9x | 13220s | 2,0x | 766s |
| | 4/12 | 7342s | 3,6x | 6617s | 3,9x | 725s |
| | 8/24 | 4002s | 6,7x | 3348s | 7,7x | 654s |
| | 16/48 | 2362s | 11,3x | 1716s | 15,0x | 647s |
| | 32/96 | 1534s | 17,4x | 925s | 27,9x | 610s |
| | 64/192 | 1092s | 24,4x | 507s | 50,9x | 585s |
| | 128/384 | 993s | 26,9x | 307s | 84,0x | 686s |
| chr16 | 1/3 | 85173s | 1,0x | 81543s | 1,0x | 3629s |
| | 2/6 | 44403s | 1,9x | 41139s | 2,0x | 3265s |
| | 4/12 | 24634s | 3,5x | 21397s | 3,8x | 3237s |
| | 8/24 | 13767s | 6,2x | 10688s | 7,6x | 3079s |
| | 16/48 | 8192s | 10,4x | 5404s | 15,1x | 2788s |
| | 32/96 | 5377s | 15,8x | 2780s | 29,3x | 2597s |
| | 64/192 | 3693s | 23,1x | 1477s | 55,2x | 2216s |
| | 128/384 | 2870s | 29,7x | 838s | 97,3x | 2033s |

paralelo de maneira eficiente e, conseqüentemente, limitou a escalabilidade de toda a aplicação. Por exemplo, o percentual de tempo de execução da fase de *traceback* aumentou de 4% a 71% quando aumentamos o número de nós de 1 a 128 (3 a 384 GPUs). Claramente, a falta de escalabilidade é resultado da dependência serial entre as GPUs. Este efeito negativo do *traceback* é reduzido significativamente quando utilizamos a estratégia IST, como será visto na seção 10.3.3.

10.3.3 Impacto do *Incremental Speculative Traceback*

Esta seção avalia o impacto do *Incremental Speculative Traceback* (IST) sobre o desempenho do *traceback*. As avaliações experimentais foram feitas sobre 5 pares de cromossomos homólogos: chr22, chr16, chr13, chr8 e chr5. Estas sequências foram selecionadas de forma a prover uma variação razoável para o tamanho da matriz de programação dinâmica (2,55, 8,13, 13,26, 21,07 e 33,04 petacélulas, respectivamente).

Os tempos de execução e TCUPS para todos os estágios, para o estágio 1 e para o *traceback* (estágios 2-6) estão presentes na Tabela 10.3 (até 128 nós/384 GPUs), tanto para o *Incremental Speculative Traceback* (IST) como para o *Pipelined Traceback* (PT). Conforme apresentado, obteve-se até 10,37 TCUPS considerando todos os estágios e até 11,08 TCUPS considerando apenas o estágio 1. Como pode ser visto, TCUPS maiores foram obtidos para sequências maiores. As estratégias PT e IST executam o mesmo código para o estágio 1 e, desta forma, o tempo de execução deste estágio é muito similar (aproximadamente 1% de diferença), exceto para alguns casos destacados na Seção 10.3.5.

Os *hits* de especulação da estratégia IST para as comparações chr22, chr16, chr13, chr8 e chr5 (com 128 nós) foram 69,2%, 88,3%, 79,4%, 99,7% e 98,2% respectivamente. Considerando a alta taxa de efetividade de especulação, os resultados apresentam que o IST foi capaz de melhorar significativamente o desempenho do *traceback* para todas as sequências e para todas as quantidades de nós utilizadas. O aumento do desempenho do IST sobre o PT para a fase de *traceback* também está apresentado na Tabela 10.3. O

IST melhorou esta fase de $2,15\times$ a $2,81\times$, de $4,35\times$ a $5,03\times$ e de $3,85\times$ a $7,66\times$ para as comparações chr22, chr16 e chr13, respectivamente. Adicionalmente, a melhoria do *traceback* com o IST para as comparações chr8 e chr5 com 128 nós foi de $18,30\times$ e $21,03\times$ respectivamente. A respeito do tempo de execução total, o IST foi $2,93\times$ mais rápido que o PT para o chr8 com 128 nós. Deve ser notado que as especulações não introduzem um *overhead* no tempo de computação total visto que o IST executa durante o tempo que as GPUs estariam ociosas.

O ganho de desempenho no IST é relacionado à habilidade de especular os *crosspoints* corretamente, o que depende das características dos alinhamentos. A próxima seção apresenta uma análise detalhada destas características sobre o desempenho da estratégia IST.

Tabela 10.3: Tempo de Execução do PT e IST

| Cmp. | | Total | Estágio 1 | <i>Traceback</i> | | |
|-----------------------|-------|---------------|------------------------|-----------------------|-------|---------------|
| | | Tempo (TCUPS) | Tempo (TCUPS) | Tempo (PT/IST) | | |
| 16 nós (48 GPUs) | chr22 | PT | 2362s (1,08) | 1716s(1,49) | 647s | 2,15x |
| | | IST | 2001s (1,27) | 1701s(1,50) | 301s | |
| | chr16 | PT | 8192s (0,99) | 5404s(1,50) | 2788s | 4,35x |
| | | IST | 6063s (1,34) | 5422s(1,50) | 641s | |
| | chr13 | PT | 12825s (1,03) | 8921s(1,49) | 3904s | 7,66x |
| | | IST | 9375s (1,41) | 8865s(1,50) | 510s | |
| 32 nós (96 GPUs) | chr22 | PT | 1534s (1,66) | 925s(2,76) | 610s | 2,71x |
| | | IST | 1139s (2,24) | 914s(2,79) | 225s | |
| | chr16 | PT | 5377s (1,51) | 2780s(2,93) | 2597s | 4,60x |
| | | IST | 3323s (2,45) | 2758s(2,95) | 565s | |
| | chr13 | PT | 8113s (1,63) | 4536s(2,92) | 3577s | 4,90x |
| | | IST | 5212s (2,54) | 4482s(2,96) | 730s | |
| 64 nós (192 GPUs) | chr22 | PT | 1092s (2,34) | 507s(5,03) | 585s | 2,81x |
| | | IST | 766s (3,33) | 558s(4,57) | 208s | |
| | chr16 | PT | 3693s (2,20) | 1477s(5,50) | 2216s | 5,03x |
| | | IST | 1914s (4,25) | 1473s(5,52) | 441s | |
| | chr13 | PT | 5390s (2,46) | 2364s(5,61) | 3026s | 3,86x |
| | | IST | 3237s (4,10) | 2453s(5,41) | 785s | |
| 128 nós (384 GPUs) | chr22 | PT | 993s (2,57) | 307s(8,31) | 686s | 2,61x |
| | | IST | 569s (4,49) | 306s(8,33) | 263s | |
| | chr16 | PT | 2870s (2,83) | 838s(9,70) | 2033s | 4,94x |
| | | IST | 1305s (6,23) | 894s(9,10) | 412s | |
| | chr13 | PT | 4176s (3,18) | 1427s(9,29) | 2749s | 3,85x |
| | | IST | 2126s (6,24) | 1412s(9,39) | 715s | |
| | chr8 | PT | 6515s (3,24) | 2020s(10,43) | 4495s | 18,30x |
| | | IST | 2219s (9,50) | 1973s(10,68) | 246s | |
| | chr5 | PT | 6490s (5,09) | 2982s(11,08) | 3508s | 21,03x |
| | | IST | 3188s (10,37) | 3021s(10,94) | 167s | |

10.3.4 Efeitos das características do alinhamento no IST

Para avaliar o impacto das características dos alinhamentos sobre o IST, criamos uma linha do tempo para as execuções chr16, chr13 e chr8 com 128 nós (384 GPUs), que estão apresentados na Figura 10.4. A área cinza no topo dos gráficos representam a execução

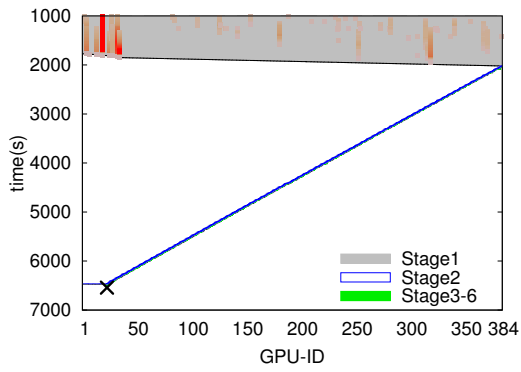
do estágio 1 e as faixas de recomputação representadas pelas linhas diagonais para o canto inferior esquerdo referem-se à fase de *traceback* (estágios 2 a 5). Nas linhas de tempo da estratégia PT, a área branca entre o estágio 1 e o *traceback* é o tempo ocioso entre as GPUs, consequência das dependências no estágio 2. Nas linhas de tempo do IST, a fase de especulação apresenta uma região azul à direita do estágio 1 (área cinza). Em ambos os casos, uma cruz (\times) no eixo y marca o tempo total de execução de cada comparação.

Os *tracebacks* com a estratégia PT (Figura 10.4) são apresentados como linhas diagonais para o canto inferior esquerdo. É possível observar facilmente a grande quantidade de tempo em que as GPUs ficam ociosas entre os estágios 1 e 2. Os estágios 3 a 4 são executados em *pipeline* e, normalmente, eles são executados em um intervalo bastante curto de tempo.

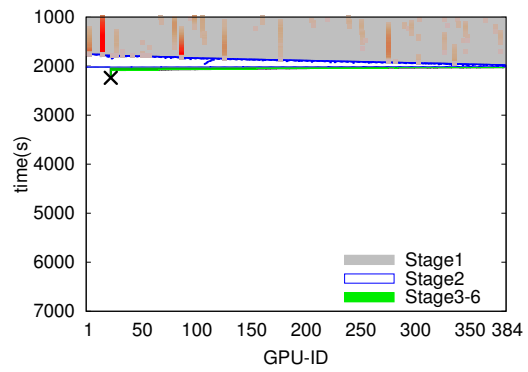
Os *tracebacks* com a estratégia IST (Figura 10.4) apresentam um comportamento diferente, onde o estágio 2 é apresentado como uma região iniciada logo abaixo do estágio 1 em cada GPU. Desta região, existem várias linhas diagonais apontando para o canto inferior esquerdo, representando as faixas de recomputação criadas pelo método de especulação incremental. Algumas destas faixas percorrem várias GPUs, como é possível ver na região central da comparação chr16 (Figura 10.4(f)) e nas primeiras GPUs da comparação chr13 (Figura 10.4(d)). Na comparação chr8, a especulação foi ainda mais eficiente (Figura 10.4(b)), produzindo um caminho de *traceback* praticamente horizontal, apontando para a esquerda, sendo capaz de reduzir o tempo de *traceback* de 4495s para 246s (18,30 \times).

No chr16 com a estratégia PT (Figura 10.4(e)) observa-se um maior intervalo de execução nos estágios 3-6 entre as GPUs 110 e 140. Este intervalo de tempo ocorreu por causa de uma grande região de *gaps* em torno da GPU 140, o que produziu partições desproporcionais que exigiram um maior tempo de execução no algoritmo Myers-Miller (estágio 4). Este tempo adicional foi propagado para outras GPUs (110-139) devido ao caminho de comunicação do estágio 4 (linha 15 no Algoritmo 8). O mesmo comportamento ocorreu no chr16 com IST (Figura 10.4(f)), mas a comunicação percorreu um maior número de GPUs (1-139) pois todas elas já tinham especulado corretamente as partições durante a fase de especulação do estágio 2 e, conseqüentemente, todas estas GPUs precisaram aguardar o resultado do estágio 4 da GPU 140. Assim que este resultado foi produzido, todas as GPUs encerraram o estágio 4 permitindo a execução dos estágios 5-6 pela primeira GPU.

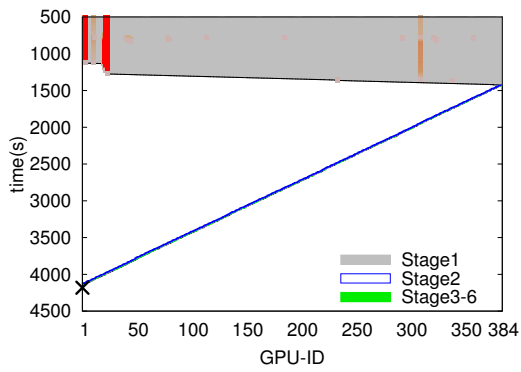
Conforme comentado na Seção 10.2.2, uma razão para a ocorrência de faixas de recomputação é a existência de regiões com uma grande incidência de N's, causadas por regiões não sequenciadas durante a montagem do genoma. A Figura 10.5 ilustra os tamanhos e posições destas regiões de "N" (em preto) de cada cromossomo. O comprimento e cobertura (%) das maiores regiões de N nos cromossomos humanos chr22, chr16, chr13, chr8 e chr5 são, respectivamente, 16 MBP (31%), 11 MBP (12%), 19 MBP (16%), 3 MBP (2,1%) e 3 MBP (1,7%). Observa-se que estas porcentagens de regiões de "N" e suas localizações influenciam bastante no speedup do método IST (Tabela 10.3). Por exemplo, o alinhamento do chr22 possui a maior região de "N" (em porcentagem), gerando o menor *speedup* do *traceback* (abaixo de 2,81 \times). Em contrapartida, os alinhamentos chr8 e chr5 apresentaram as menores regiões de "N", produzindo os maiores speedups (18,30x e 21,03x, respectivamente). Em relação aos alinhamentos do chr16 e chr13, ambos possuem regiões de "N" similares, mas o chr16 humano contém uma região evidente no meio do cromossomo e o chr13, no início. Adicionalmente, o chr13 possui uma região de "N" de 7 MBP



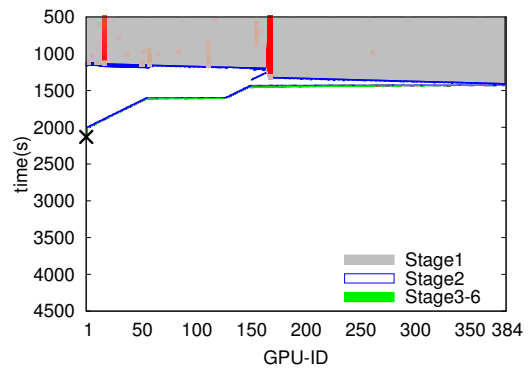
(a) Chr8 - PT



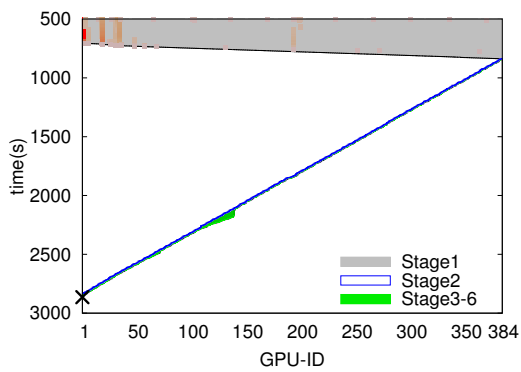
(b) Chr8 - IST



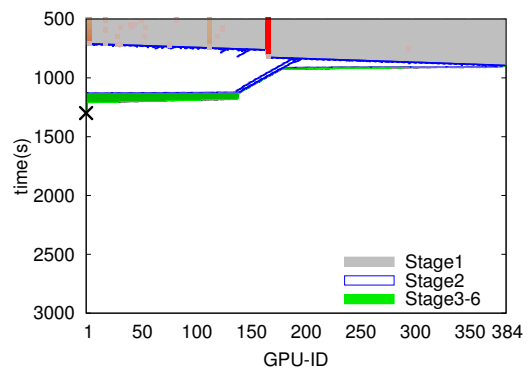
(c) Chr13 - PT



(d) Chr13 - IST



(e) Chr16 - PT



(f) Chr16 - IST

Figura 10.4: Linhas de tempo dos estágios

(6,5%) localizada no meio do cromossomo. Note que a maior faixa de recomputação no chr16 (Figura 10.4(f)) e no chr13 (Figura 10.4(d)) coincidem com essas regiões.

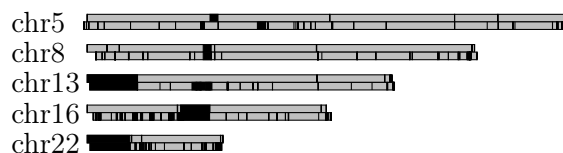


Figura 10.5: Regiões não sequenciadas (regiões de “N”) nos cromossomos homólogos (homem acima, chimpanzé abaixo)

10.3.5 Análise do uso do *Buffer* (Estágio 1)

A Figura 10.4 apresenta o uso do *buffer* de saída do estágio 1 (Seção 9.2) utilizando diferentes tons de vermelho. Cada coluna em vermelho indica um alto uso do *buffer*, resultado de uma taxa de desbalanceamento entre as GPUs. Quando uma GPU está mais lenta que a GPU à sua esquerda, seu *buffer* começa a ser preenchido até que, eventualmente, ele fique completamente cheio, cascadeando a lentidão para as GPUs anteriores. Por exemplo, no chr16 com IST (Figura 10.4(f)) existem linhas vermelhas próximas às GPUs centrais, indicando um desbalanceamento nestas GPUs. Este comportamento produziu uma diferença de 6% (55s) no tempo do estágio 1, comparando o tempo de execução do PT com o IST (Tabela 10.3). No chr13, tanto o IST (Figura 10.4(d)) como o PT (Figura 10.4(c)) apresentaram desbalanceamentos. Nestes casos, entretanto, o tempo de execução do estágio 1 do IST e do PT foi praticamente o mesmo, com apenas 1% de diferença.

Conforme discutido anteriormente, o IST e o PT executam o mesmo código do estágio 1 e as diferenças do tempo de execução deste estágio não estão relacionadas com a estratégia de *rollback*. O desbalanceamento pode ter sido causado pelos seguintes fatores: 1) sequências com regiões muito similares, gerando atualizações frequentes nas variáveis que armazenam o melhor escore e causando um pequeno *overhead* devido à execução destas instruções adicionais na GPU; 2) utilização concorrente de recursos com outras tarefas no *cluster*, o que pode gerar uma perda de desempenho no sistema de arquivo ou um aumento da latência de rede. Visto que o primeiro fator depende somente do conteúdo das sequências, este tipo de desbalanceamento deve ser reproduzível entre execuções iguais. Entretanto, o segundo fator é normalmente irreproduzível. Adicionalmente, o segundo fator tende a produzir impacto durante um maior tempo da execução, considerando que a concorrência entre tarefas é um fator sistêmico. Com essas considerações e analisando as linhas de tempo obtidas, podemos inferir que a maioria dos cenários de desbalanceamento foram causados por fatores irreproduzíveis, formados pela concorrência de recursos dentro do *cluster*.

10.4 Conclusão do Capítulo

O CUDAlign 4.0 permitiu a obtenção do alinhamento completo em múltiplas GPUs de maneira eficiente. Por meio dos mecanismos de *rollback* propostos para múltiplos nós (PT - *Pipelined Traceback* e IST - *Incremental Speculative Traceback*), foi possível obter o alinhamento completo de todos os cromossomos homólogos entre o homem e o chimpanzé, totalizando mais que 500 trilhões de células processadas. O método IST foi capaz de acelerar em até $21\times$ o tempo do *rollback* em comparação com o método PT. Utilizando

384 GPUs, foi possível obter um pico de desempenho de 10,37 TCUPS, acima do maior valor encontrado na literatura (Tabela 4.1). Este desempenho poderia ser ainda maior em um cenário ideal onde todas as GPUs tivessem exatamente o mesmo desempenho, sem nenhum tipo de contenção conforme foi observado no *cluster* utilizado para os testes (Seção 10.3.5). Visto que os efeitos observados nesse ambiente são muito difíceis de serem produzidos, método de balanceamento dinâmico de carga seria extremamente útil para corrigir eventuais problemas de contenção. Proporemos este método no Capítulo 11, onde verificaremos a eficácia do balanceamento dinâmico por meio de simulações.

Capítulo 11

Balanceamento de *Wavefront* em Múltiplas GPUs

Conforme visto no Capítulo 10, pequenas diferenças nas capacidades de processamento das GPUs podem causar variações no uso dos *buffers* até que, eventualmente, a execução seja bloqueada à espera de dados ou de espaço livre nos *buffers*. Nesta situação, o bloqueio na comunicação causará lentidão nos demais nós, gerando um efeito em cascata capaz de comprometer o desempenho de toda a execução do *wavefront*. No nosso contexto, dizemos que o *wavefront* está balanceado se todos os nós de processamento estiverem calculando a mesma quantidade de linhas por segundo, de maneira que não ocorra nenhum bloqueio na execução.

O objetivo deste capítulo é propor e avaliar uma solução baseada em agentes capaz de executar balanceamento dinâmico de carga durante o processamento do *wavefront* multinodo, com base em um conjunto de métricas. A avaliação do método é feita por meio de simulações, considerando cenários controlados de desbalanceamento de carga. Tanto a proposta como sua avaliação foram publicadas em [39].

A Seção 11.1 apresenta o projeto do sistema multiagentes. As Seções 11.2, 11.3 e 11.4 apresentam métricas que são utilizadas pelas abordagens de tomada de decisão dos agentes. A Seção 11.5 avalia, por meio de simulações, as abordagens propostas. Por fim, a Seção 11.6 apresenta uma discussão sobre o capítulo.

11.1 Projeto do Sistema Multiagentes

O projeto de um agente deve considerar quatro propriedades que formam o acrônimo **PAGE**, composto dos termos em inglês **P**ercepts (Percepções), **A**ctions (Ações), **G**oals (Objetivos) e **E**nvironment (Ambiente) [140]. As percepções descrevem os dados que um agente pode obter do ambiente. As ações são os comandos que um agente pode executar para alterar o ambiente. Os objetivos são as motivações que levam os agentes a executarem ações em direção a uma meta. O ambiente é onde o agente reside. Todas essas propriedades definem o “raciocínio” de um agente. Uma definição apropriada do PAGE é muito importante para compreender os objetivos dos agentes e sobre como será a interação entre eles e com o ambiente.

A Figura 11.1 apresenta a visão geral da arquitetura do sistema multiagentes proposto, onde cada nó N_1, N_2, \dots, N_k é composto por um agente reflexivo chamado de *agente*

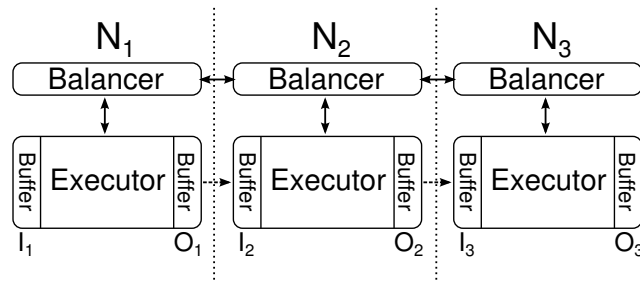


Figura 11.1: Arquitetura do sistema multiagentes proposto.

executor e um agente baseado em objetivo chamado de *agente balanceador*, conforme a classificação de agentes de [140]. Na Tabela 11.1 é descrito o PAGE dos agentes executor e balanceador.

Tabela 11.1: Percepções, ambiente, objetivos e ações (PAGE) dos agentes propostos

| Agente Executor | Agente Balanceador |
|---|--|
| Objetivos: Mantém e avalia a execução do processo | Objetivos: Otimiza a distribuição de colunas para reduzir o tempo de execução |
| Ambiente: O sistema operacional e a rede de comunicação. Este ambiente é parcialmente observável, estocástico, sequencial, dinâmico e contínuo. | Ambiente: O sistema multiagentes. O ambiente é parcialmente observável, estocástico, sequencial, dinâmico e contínuo. |
| Percepções: <ol style="list-style-type: none"> 1. número de linhas processadas por segundo; 2. número de células aguardando no <i>buffer</i>; 3. tempo ocioso aguardando dados; | Percepções: <ol style="list-style-type: none"> 1. estatísticas do agente executor; 2. intenções dos outros agentes balanceadores; |
| Ações: <ol style="list-style-type: none"> 1. Reiniciar o processo quando uma nova distribuição de colunas for requisitada. | Ações: <ol style="list-style-type: none"> 1. manter a distribuição de colunas; 2. balancear a distribuição de colunas; |

O agente executor é responsável por executar o algoritmo de *wavefront* e por manter o *buffer* de comunicação entre os nós. No início da execução, cada agente executor fica responsável por calcular um conjunto de colunas da matriz de programação dinâmica. O agente executor mensura o desempenho do cálculo da matriz de programação dinâmica e o estado dos *buffers*. O agente executor deve, periodicamente, identificar o número de linhas processadas por segundo, o número de células armazenadas nos *buffers* e o tempo desperdiçado aguardando dados ou espaço nos *buffers* (tempo de bloqueio). Decidimos que o agente executor não obterá estas estatísticas durante o começo ou final do processamento, pois as medições não serão acuradas por causa do preenchimento e esvaziamento do *wavefront*.

O agente balanceador é responsável por analisar o estado do agente executor e por identificar se é melhor manter ou rebalancear a distribuição de colunas. Se um dos balanceadores decidir rebalancear, ele irá negociar com todos os outros agentes para redistribuir as colunas e reduzir o tempo total de processamento. A redistribuição é feita considerando pesos dinâmicos em cada nó. O agente utiliza as ideias do modelo *Belief-Desire-Intention*/BDI (Crenças-Desejos-Intenções) [141] para separar o processo de deliberação do processo de redistribuição.

Propomos duas estratégias a serem utilizadas pelos agentes balanceadores para identificação da necessidade de rebalanceamento: 1) estratégia global, onde os agentes conhecem o estado de todos os outros agentes; 2) estratégia local, onde os agentes somente possuem conhecimento sobre os seus próprios estados. A diferença entre as duas estratégias será explicada nas Seções 11.2 a 11.4, onde estão definidas as métricas de execução do agente executor, as métricas de balanceamento avaliadas pelas estratégias global e local, o mecanismo de pesos e as fases de negociação dos agentes balanceadores.

11.2 Métricas de Execução

Considerando que os nós são descritos como N_1, N_2, \dots, N_k e que W e H são, respectivamente, o número de colunas e linhas da matriz de programação dinâmica (Seção 2.2.2), as métricas obtidas pelo agente executor são formalmente definidas conforme as seguintes variáveis, todas definidas localmente e individualmente para cada nó.

- c_i : número de colunas distribuídas ao nó de processamento N_i (onde $\sum_{i=1}^k c_i = W$);
- s_i : poder computacional do nó N_i , medido em número de linhas processadas por segundo;
- n_i : velocidade de comunicação atual do nó N_i , medidas em células enviadas por segundo;
- r_i : índice da última linha completamente processada pelo nó N_i (onde $r_i \geq r_{i+1}$ para cada nó $i < k$);
- b_i^{\leftarrow} : quantidade de células armazenadas no *buffer* de entrada do nó N_i ;
- b_i^{\rightarrow} : quantidade de células armazenadas no *buffer* de saída do nó N_i ;
- w_i^{\leftarrow} : tempo total em que o nó N_i esteve bloqueado aguardando por dados no *buffer* de entrada;
- w_i^{\rightarrow} : tempo total em que o nó N_i esteve bloqueado aguardando por espaço no *buffer* de saída;

O poder computacional s_1, s_2, \dots, s_k indica o potencial que cada nó possui. Considerando que um nó pode estar subutilizado ou sobrecarregado com outras tarefas, o poder computacional de um nó poderá ser maior que a velocidade efetiva de processamento da matriz.

11.3 Métricas de Balanceamento Global

Cada agente balanceador atualiza, periodicamente, suas próprias crenças sobre o balanceamento do *wavefront*. Desta forma, se um agente identificar que é melhor rebalancear

o *wavefront*, então todos os agentes balanceadores negociarão a redistribuição das colunas. A estratégia global de balanceamento utiliza o seguinte conjunto de fórmulas para quantificar o benefício que a nova redistribuição de colunas é capaz de produzir.

- $f_i(x)$: função de previsão de desempenho para o nó N_i medida em linhas por segundo, onde x é o número de colunas. Por exemplo, $f_1(100)$ retorna o número estimado de linhas processadas por segundo pelo nó N_1 considerando que o número de colunas atribuídas a ele seja de $c_1 = 100$. Para a simulação, a função de previsão de desempenho será escrita como $f_i(x) = k \cdot (s_i/x)$, onde k é uma constante da simulação e s_i o poder computacional do nó N_i .
- T_i : tempo restante estimado para a conclusão do processamento do nó N_i , assumindo que a distribuição de colunas será mantida igual à atual. Este tempo pode ser calculado conforme a Equação 11.1.

$$T_i = \frac{H - r_i}{s_i} \quad (11.1)$$

- \widehat{T} : tempo restante global estimado para a conclusão do processamento de todos os nós. Este valor é definido como o valor máximo do tempo restante de todos os nós (Equação 11.2).

$$\widehat{T} = \max_{1 \leq i \leq k} T_i \quad (11.2)$$

- $T'_i(x)$: tempo restante estimado para a conclusão do processamento do nó N_i , assumindo que a distribuição de colunas será alterada para x colunas. Este tempo pode ser calculado conforme a Equação 11.3.

$$T'_i(x) = \frac{H - r_i}{f_i(x)} + \alpha_i(x) \quad (11.3)$$

onde $\alpha_i(x)$ é o custo adicional estimado por reiniciar a computação no nó N_i se realocarmos x colunas a ele.

- $\widehat{T}'(c_1, c_2, \dots, c_k)$: é o tempo restante global estimado para a conclusão do processamento de todos os nós. Ele é definido como o tempo restante estimado máximo de todos os nós considerando uma distribuição de colunas (c_1, c_2, \dots, c_k) (Equação 11.4).

$$\widehat{T}'(c_1, c_2, \dots, c_k) = \max_{1 \leq i \leq k} T'_i(c_i) \quad (11.4)$$

- $B_i(x)$: é o benefício de tempo por realocar x colunas ao nó N_i . Se $B_i(x) > 0$, então dizemos que a nova alocação de colunas x é *localmente aceitável*, caso contrário ela é *localmente inaceitável*. O benefício de tempo para cada nó é calculado conforme a Equação 11.5.

$$B_i(x) = T_i - T'_i(x) \quad (11.5)$$

- $\widehat{B}(c_1, c_2, \dots, c_k)$: é o benefício global de tempo por realocar para uma nova distribuição de colunas. Se $\widehat{B}_i(c_1, c_2, \dots, c_k) > 0$, então dizemos que a nova alocação é *globalmente aceitável*, caso contrário ela é *globalmente inaceitável*. O benefício global é calculado pela Equação 11.6.

$$\widehat{B}(c_1, c_2, \dots, c_k) = \widehat{T} - \widehat{T}'(c_1, c_2, \dots, c_k) \quad (11.6)$$

Para identificar o benefício global de uma nova distribuição de colunas, o agente balanceador precisa conhecer as métricas de execução de todos os agentes executores. Para isso, os nós precisam continuamente trocar dados, o que é uma operação muito onerosa que pode gerar um grande *overhead* em cenários reais.

11.4 Métricas de Balanceamento Local

Em vez de calcular as equações globais definidas na Seção 11.3, a estratégia de balanceamento local identifica o estado de balanceamento do *wavefront* utilizando somente as variáveis que são locais em cada nó. Para quantificar o estado de balanceamento, a estratégia de balanceamento local utiliza os conceitos de estabilidade definidos pelas seguintes métricas:

- δ_i^{\leftarrow} e δ_i^{\rightarrow} : são as métricas de estabilidade dos *buffers*, onde δ_i^{\leftarrow} e δ_i^{\rightarrow} representam o número de linhas processadas por segundo que o nó N_i deveria adicionalmente possuir para parar o incremento do *buffer* de entrada (b_i^{\leftarrow}) e do *buffer* de saída (b_i^{\rightarrow}), respectivamente. Para calcular a estabilidade do *buffer* entre os tempos t_0 e t_1 , seja Δb_i^{\leftarrow} e Δb_i^{\rightarrow} a variação das métricas b_i^{\leftarrow} e b_i^{\rightarrow} entre os tempos t_0 e t_1 , e seja $\Delta t = t_1 - t_0$. Então, as métricas de estabilidade δ_i^{\leftarrow} e δ_i^{\rightarrow} são calculadas conforme as Equações 11.7 e 11.8.

$$\delta_i^{\leftarrow} = \frac{\Delta b_i^{\leftarrow}}{\Delta t} \quad (11.7)$$

$$\delta_i^{\rightarrow} = -\frac{\Delta b_i^{\rightarrow}}{\Delta t} \quad (11.8)$$

- ψ_i^{\leftarrow} e ψ_i^{\rightarrow} : são as métricas de estabilidade de bloqueio, onde ψ_i^{\leftarrow} e ψ_i^{\rightarrow} representam o número de linhas processadas por segundo que o nó N_i deveria adicionalmente possuir para parar o bloqueio do *buffer* de entrada (w_i^{\leftarrow}) e do *buffer* de saída (w_i^{\rightarrow}), respectivamente. Para calcular a estabilidade de entrada entre os tempos t_0 e t_1 , seja Δr_i , Δw_i^{\leftarrow} e Δw_i^{\rightarrow} as variações das métricas r_i , w_i^{\leftarrow} e w_i^{\rightarrow} entre os tempos t_0 e t_1 , considerando $\Delta t = t_1 - t_0$. Então, a métrica de estabilidade ψ_i^{\leftarrow} e ψ_i^{\rightarrow} são calculadas conforme as Equações 11.9 e 11.10.

$$\psi_i^{\leftarrow} = -\frac{\Delta r_i}{\Delta t} \cdot \frac{\Delta w_i^{\leftarrow}}{\Delta t - \Delta w_i^{\leftarrow}} \quad (11.9)$$

$$\psi_i^{\rightarrow} = \frac{\Delta r_i}{\Delta t} \cdot \frac{\Delta w_i^{\rightarrow}}{\Delta t - \Delta w_i^{\rightarrow}} \quad (11.10)$$

- π_i^{\leftarrow} e π_i^{\rightarrow} : são, respectivamente, as métricas de estabilidade de entrada e saída do nó N_i , sendo essas métricas simplesmente a soma da estabilidade do *buffer* e da estabilidade de bloqueio de cada nó, conforme mostram as Equações 11.11 e 11.12.

$$\pi_i^{\leftarrow} = \delta_i^{\leftarrow} + \psi_i^{\leftarrow} \quad (11.11)$$

$$\pi_i^{\rightarrow} = \delta_i^{\rightarrow} + \psi_i^{\rightarrow} \quad (11.12)$$

- Π_i : é a estabilidade local do nó N_i , que é a soma da estabilidade de entrada π_i^{\leftarrow} e a de saída π_i^{\rightarrow} de N_i (Equação 11.13). Um nó é dito *localmente estável* se a métrica Π_i é próxima de zero (menor que uma constante), caso contrário o nó torna-se *localmente instável*. Assim que a instabilidade local é detectada, o nó deve notificar os outros nós para negociarem a redistribuição de colunas.

$$\Pi_i = \pi_i^{\leftarrow} + \pi_i^{\rightarrow} = (\delta_i^{\leftarrow} + \psi_i^{\leftarrow}) + (\delta_i^{\rightarrow} + \psi_i^{\rightarrow}) \quad (11.13)$$

- $\hat{\Pi}$: é a estabilidade global do *wavefront*. Esta métrica é definida como o valor máximo das estabilidades locais de todos os nós (Equation 11.14). Um *wavefront* é dito *estável* se todos os nós forem *localmente estáveis*, caso contrário, se algum nó notificou a sua *instabilidade local*, o *wavefront* é chamado de *instável*.

$$\hat{\Pi} = \max_{1 \leq i \leq k} |\Pi_i| \quad (11.14)$$

11.4.1 Pesos de Balanceamento

Uma alocação de colunas é definida formalmente como sendo o conjunto (c_1, c_2, \dots, c_k) representando o número de colunas atribuídas a cada nó, onde a soma $\sum_{i=1}^k c_i$ é igual ao número total de colunas W da matriz de programação dinâmica. A redistribuição de colunas considera pesos em cada nó, conforme definições a seguir:

- **pesos**: Os pesos de redistribuição $(\gamma_1, \gamma_2, \dots, \gamma_k)$ atribuídos a cada nó consideram a diferença de desempenho dos nós para criar uma nova distribuição de colunas (c_1, c_2, \dots, c_k) . A nova distribuição de colunas é calculada pela Equação 11.15:

$$(c_1, c_2, \dots, c_k) = (W \frac{\gamma_1}{\gamma'}, W \frac{\gamma_2}{\gamma'}, \dots, W \frac{\gamma_k}{\gamma'}) \quad (11.15)$$

onde

$$\gamma' = \sum_{i=1}^k \gamma_i \quad (11.16)$$

- **pesos iniciais**: Os pesos iniciais podem ser atribuídos utilizando alguma métrica de desempenho de cada nó, como por exemplo o GFLOPS de cada nó ou o resultado de alguma ferramenta de *benchmark*. Se os pesos forem os mesmos $(\gamma_1 = \gamma_2 = \dots = \gamma_k)$, as colunas são igualmente distribuídas para cada nó.

- **pesos atualizados:** Assim que os nós decidem rebalancear, cada agente balanceador é responsável por definir um novo peso γ_i capaz de reduzir o tempo computacional. Para calcular os pesos desejados, a estratégia global utiliza a Equação 11.17:

$$\gamma_i = s_i \quad (11.17)$$

e a estratégia local utiliza a Equação 11.18:

$$\gamma_i = c_i \cdot (\Delta r_i - \psi_i^{\leftarrow}) \quad (11.18)$$

Para investigar o envolvimento de todos os agentes balanceadores durante a negociação, duas observações devem ser feitas.

Primeiro, um nó não pode mudar sua própria distribuição de colunas por si mesmo, visto que o número total de colunas deve ser sempre igual à largura W da matriz de programação dinâmica. Sendo assim, se um nó aumentar o número de colunas atribuídas a si, então a mesma quantidade de colunas deverá ser reduzida dos outros nós.

Segundo, visto que a distribuição de colunas é baseada em pesos de cada nó, então uma mudança de um único peso irá afetar todos os outros nós. A política de balanceamento é baseada nos pesos $(\gamma_1, \gamma_2, \dots, \gamma_k)$ e a distribuição de colunas é calculada utilizando as Equações 11.15 e 11.16. Então, supondo que o nó N_i aumente seu poder computação para o seu novo peso $\gamma_i + \beta$, a variável γ' (somatório de todos os pesos) irá aumentar em β e a nova distribuição de colunas será calculada pela Equação 11.19:

$$\left(W \frac{\gamma_1}{\gamma' + \beta}, W \frac{\gamma_2}{\gamma' + \beta}, \dots, W \frac{\gamma_i + \beta}{\gamma' + \beta}, \dots, W \frac{\gamma_k}{\gamma' + \beta} \right) \quad (11.19)$$

É possível notar que toda a distribuição de colunas irá sofrer alteração. Para um cenário mais complexo, é possível evitar algumas distribuições de colunas se existir um outro nó N_j que reduza seu peso na mesma proporção $-\beta$, de forma a manter a variável γ' inalterada. Entretanto, este cenário ocorre raramente e não será considerado nesta tese.

11.4.2 Negociação de Balanceamento

Quando um sistema baseado em agentes é projetado, um objetivo desejável é criar agentes autônomos com habilidade social. Isto inclui a capacidade de resolver conflitos entre os agentes sem um ponto de decisão central. No *wavefront* multinós, uma distribuição de colunas incorreta pode aumentar o tempo de execução de alguns nós e deixar outros nós mais ociosos. Sendo assim, uma negociação deve ser feita entre todos os agentes de forma a convencê-los de que a redistribuição produzirá um resultado globalmente aceitável e vantajoso.

Quatro fases de negociação foram projetadas para executar a redistribuição de balanceamento: Requisição, Concordância, Distribuição e Reinício. A Figura 11.2 apresenta o diagrama de sequência e a Figura 11.3 mostra o diagrama de estados durante a negociação de balanceamento. Esta negociação é aplicada tanto às estratégias globais como locais, conforme explicado a seguir:

- **Requisição:** Na Figura 11.2, considere que o nó N_3 possui a intenção de balancear as colunas entre os nós. Então, este nó envia uma requisição para o nó à esquerda,

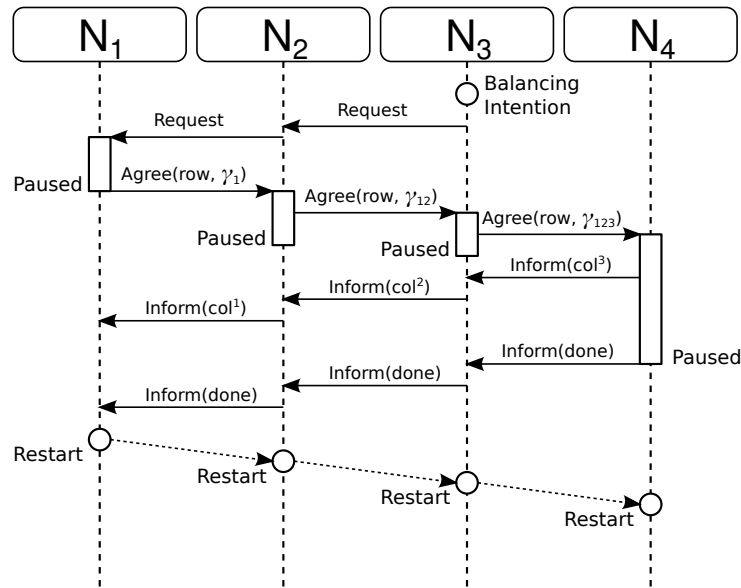


Figura 11.2: Negociação do Balanceamento.

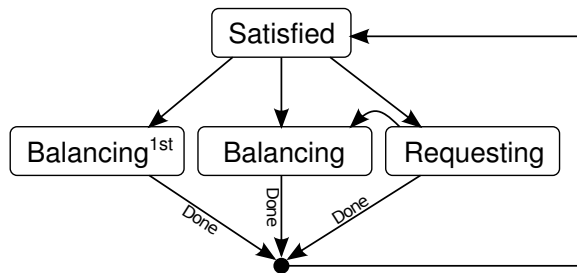


Figura 11.3: Diagrama de transição de estados.

que propaga a mensagem até que ele encontre o primeiro nó N_1 , que suspende a execução da matriz imediatamente. Durante a propagação da requisição, os nós N_3 a N_2 alteram seus estados para *Requesting* e o nó N_1 altera seu estado para *Balancing^{1st}*.

- **Concordância:** Assim que o nó N_1 suspender a execução da matriz, ele envia uma mensagem de concordância para o nó à direita (N_2), informando que a linha com índice row foi a última linha processada e informando o peso γ_1 . Então, o próximo nó inicia a suspensão do processamento até que a mesma linha row , do nó N_1 , seja processada. Antes mesmo que o nó termine de calcular a linha row , ele propaga a mensagem de concordância para o próximo nó. Em vez de enviar o peso do primeiro nó, o segundo nó envia a soma dos pesos $\gamma_{12} = \gamma_1 + \gamma_2$ para o próximo nó. Esta operação propaga a soma dos pesos anteriores, um a um, até o último nó N_4 , que recebe o mesmo índice row e a soma dos pesos de todos os nós $\gamma_{123} = \gamma_1 + \gamma_2 + \gamma_3$. Neste momento, o nó N_1 está no estado *Balancing^{1st}* e os demais estão no estado *Balancing*.
- **Distribuição:** Assim que o último nó recebe a mensagem de concordância, ele é

capaz de calcular a sua própria distribuição de colunas utilizando a Equação 11.15, considerando que a soma de todos os pesos é calculado pela fórmula $\gamma' = \gamma_{123} + \gamma_4$. Então, ele informa ao nó anterior o número de colunas $col^3 = W - c_4$ que continua desbalanceado. Visto que o nó N_3 possui a soma dos pesos anteriores γ_{12} , então a mesma operação de distribuição é realizada. O Nó N_3 calcula sua própria distribuição utilizando a Equação 11.15, mas considerando que o peso W é somente as colunas restantes col^3 e a soma de pesos é $\gamma' = \gamma_{12} + \gamma_3$. Este processo propaga até o nó N_1 , que simplesmente aloca todas as colunas restantes col^1 para si.

- **Reinício:** Assim que o último nó N_4 finaliza o cálculo da linha *row*, ele informa a conclusão para todos os nós anteriores, propagando esta informação (mensagem “done”) até o nó N_1 . Então, todos os nós alteram seu estado de volta para *Satisfied* e reiniciam o processamento da matriz de programação dinâmica. O *wavefront* irá propagar normalmente até que todos os nós reiniciem a sua execução, com a nova distribuição de colunas.

11.5 Resultados Experimentais

Para avaliar a proposta de balanceamento de carga, optou-se por implementar um simulador no ambiente JADE [142]. Com este ambiente, pode-se simular várias mudanças no poder computacional dos nós, sem a necessidade de um ambiente real. Neste trabalho, não foram simuladas mudanças na velocidade de comunicação, sempre considerando que a comunicação não é gargalo para os nós.

Para analisar as duas estratégias de balanceamento, projetamos um gráfico que representa a dinâmica de execução. A simulação produz como resultado este gráfico, permitindo que a análise do estado de cada *buffer* durante todo o tempo de simulação.

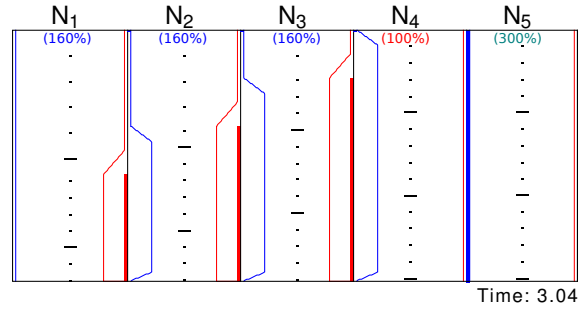
Primeiro, mostraremos uma simulação com distribuição estática de colunas, que considera somente o poder computacional inicial dos nós e assume que o poder computacional é mantido durante toda a execução da simulação. Em seguida, iremos apresentar as simulações com balanceamento dinâmico, considerando mudanças no poder computacional durante a simulação. Antes de apresentar os resultados da simulação, iremos explicar o gráfico do balanceamento.

11.5.1 Gráfico de Balanceamento

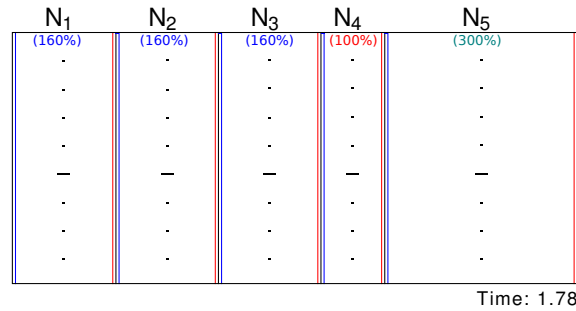
O gráfico de balanceamento foi criado para prover informações sobre a dinâmica do *wavefront* multinós. Nesta seção será explicado como analisar este gráfico.

Considere as Figuras 11.4(a) e 11.4(b) com distribuição estática de colunas. O retângulo externo representa a matriz de programação dinâmica e a distribuição de colunas é ilustrada pelas linhas verticais cruzando da primeira a última linha. Nestas figuras, existem 5 nós e a distribuição de colunas produz 5 faixas, cada uma para um nó.

Existem várias marcações com linhas horizontais dentro das raias. Cada marcação representa uma unidade de tempo de simulação, iniciando da linha superior até a linha inferior. As marcações mais largas representam uma unidade de tempo e as mais estreitas representam 0,2 unidade. Visto que os nós executam em paralelo utilizando o método de *wavefront* multinós, as linhas dos nós são processadas em tempos diferentes. Então,



(a) Cenário desbalanceado.



(b) Cenário balanceado.

Figura 11.4: Gráfico de balanceamento para *wavefront* estático.

a computação é mais rápida se as marcas de tempo estão separadas por uma distância maior. O tempo total da simulação é sempre apresentada abaixo da matriz.

Em ambos os lados de cada linha vertical intermediária, existem outras linhas que representam a quantidade de dados aguardando nos *buffers*, onde a linha à esquerda está associada com o *buffer* de saída e a linha à direita está associada com o *buffer* de entrada do próximo nó. Se o *buffer* de entrada estiver vazio ou o *buffer* de saída estiver cheio, então o processo irá bloquear e o nó se tornará ocioso. O estado de ociosidade é representado por uma linha mais espessa nos lados das linhas intermediárias, onde o lado à esquerda representa bloqueios no *buffer* de saída e o lado à direita representa bloqueios no *buffer* de entrada. O gráfico de balanceamento não expressa a porcentagem de tempo do *buffer* de entrada, mas o percentual de tempo ocioso é ignorado se ele for menor que 5%.

11.5.2 *Wavefront* multinodo estático

Nesta subseção, simulamos a distribuição estática em duas situações: 1) cenário desbalanceado, com distribuição uniforme de pesos; 2) cenário balanceado, com os pesos proporcionais ao poder computacional de cada nó.

A Figura 11.4(a) ilustra cinco nós com a mesma distribuição de colunas (20% de colunas para cada nó). Neste exemplo, o poder computacional de cada nó é, respectivamente, 160%, 160%, 160%, 100% and 300%, onde a porcentagem é utilizada para comparar o poder computacional dos nós. Visto que os nós N_1 , N_2 e N_3 são capazes de computar 60% mais linhas por segundo que o nó N_4 , então o *buffer* de entrada do nó N_4 cresce até o limite máximo. Quando o *buffer* fica cheio, então o *buffer* de saída do nó N_3 começa a crescer até, eventualmente, chegar ao seu limite, causando o bloqueio do processamento

do nó. Assim que o nó N_3 também se bloqueia, a sua velocidade de processamento começa a ser limitada pelo poder computacional do nó N_4 e o fluxo de dados começa a bloquear os *buffers* dos nós N_2 e, em seguida, do nó N_1 , assim como podemos ver na Figura 11.4(a). Visto que o nó N_5 é 3 vezes mais rápido que o nó N_4 , então ele aguarda, periodicamente, por dados em seu *buffer* de entrada, limitando a sua velocidade de computação ao poder computacional do nó N_4 . Embora o gráfico de *wavefront* não expresse a percentagem do tempo de bloqueio do *buffer* de entrada, sabe-se que o nó N_5 fica ocioso por $\frac{2}{3}$ do tempo, pois ele é 3 vezes mais rápido que o nó N_4 .

As marcações horizontais são muito úteis para identificar a posição no tempo e a velocidade de computação de cada nó. Por exemplo, se analisarmos as primeiras duas marcas menores dos nós N_1 , N_2 e N_3 na Figura 11.4(a), nota-se que até aquele momento estes nós estavam bastante sincronizados. Entretanto, se analisarmos o tempo após a terceira marcação menor, podemos ver que os nós N_1 , N_2 e N_3 não estão mais sincronizados. Isto pode ser consequência de uma série de operações de bloqueio no *buffer* de saída, causadas pela menor velocidade do nó N_4 . Visto que os nós não estão sincronizados, podemos observar que os nós N_1 , N_2 , N_3 and N_4 terminaram em momentos bastante diferentes.

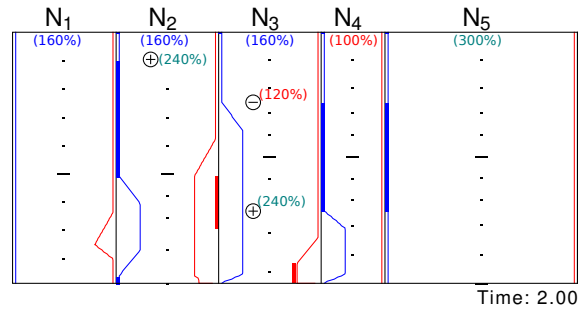
Os nós N_4 e N_5 estão com velocidades praticamente idênticas durante toda a computação. Visto que podemos ver uma linha espessa na coluna intermediária, então podemos concluir que N_5 está sincronizado com N_4 devido ao bloqueio causado pela diferença de velocidade entre eles. Os nós N_4 e N_5 terminaram a computação praticamente ao mesmo tempo, com um tempo total de simulação de 3,04 unidades de tempo.

O segundo cenário é ilustrado na Figura 11.4(b), o qual mostra o gráfico de balanceamento para o cenário balanceado, considerando o poder computacional inicial de cada nó. Visto que N_5 é 3 vezes mais rápido que N_4 e praticamente 2 vezes mais rápido que os nós N_1 , N_2 e N_3 , então ele processa uma área maior que os outros. Podemos ver que o *wavefront* multinós é bastante estável, sem incremento do uso de *buffers* nem tempos de bloqueio. O tempo total de execução foi 1,78 unidade de tempo, uma redução de 41% comparado com o cenário desbalanceado (Figura 11.4(a)). Pode-se observar que, se a distribuição de colunas não for feita considerando o poder computacional dos nós, o tempo de execução dos nós será maior por causa da grande quantidade de bloqueios nos *buffers* de comunicação.

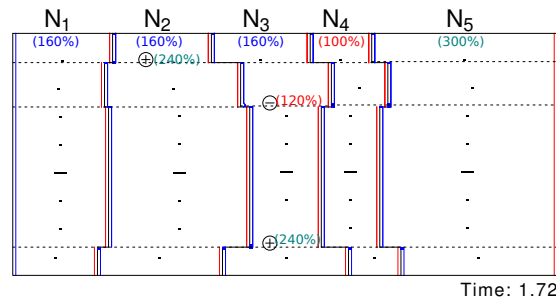
11.5.3 Balanceamento dinâmico no *Wavefront* multinodo

Visto que os nós podem aumentar ou reduzir seu poder computacional durante a execução, então o *wavefront* deve se adaptar para as diversas situações. A primeira simulação, ilustrada nas Figuras 11.5(a), 11.5(b) e 11.5(c), apresenta o cenário onde 5 nós iniciam com poder computacional de 160%, 160%, 160%, 100% e 300%, respectivamente. Então, à 0,20 unidade de tempo o nó N_2 aumenta seu poder computacional para 240%, à 0,50 unidade de tempo o nó N_3 reduz seu poder computacional para 120% e à 1,50 unidade de tempo o nó N_3 aumenta seu poder computacional para 240%. O gráfico de balanceamento apresenta o aumento como o símbolo ‘+’ e a redução como o símbolo ‘-’.

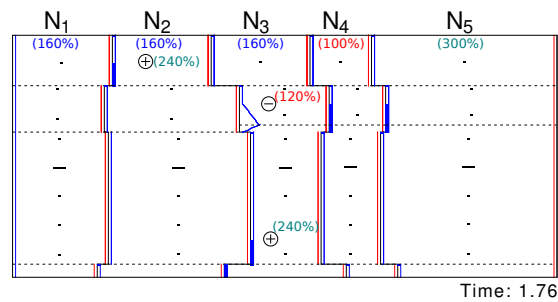
No cenário de desbalanceamento (Figura 11.5(a)), o balanceador ignora a instabilidade. Nesta situação, podemos observar que o *buffer* de entrada do nó N_2 inicia o bloqueio à 0,20 unidade de tempo, o *buffer* de entrada do nó N_3 começa a encher à 0,50 unidade de tempo (propagando a instabilidade para os nós N_2 e N_1), o *buffer* de saída do



(a) Cenário desbalanceado.



(b) Cenário com Balanceamento Global.



(c) Cenário com Balanceamento Local.

Figura 11.5: Balanceamento com três mudanças no poder computacional.

nó N_2 começa a bloquear à 1,00 unidade de tempo e todo o caminho de *buffers* entre os nós N_1 e N_3 começa a se esvaziar quando o nó N_3 aumenta seu poder computacional, à 1,50 unidade de tempo. Podemos observar também que, por causa da redução do poder computacional do nó N_3 , os nós N_4 e N_5 ficam parcialmente ociosos, até o momento em que o nó N_3 aumenta seu poder computacional. A partir deste momento, o *buffer* de entrada do nó N_4 começa a encher. O tempo total de execução foi de 2,0 unidades de tempo.

Na Figura 11.5(b), pode-se observar o cenário de balanceamento global, em que o agente balanceador conhece exatamente o poder computacional e o benefício de efetuar a decisão de balanceamento. Estes agentes podem ser considerados “oráculos” no sentido de conhecer o potencial computacional exato de todos os nós, a todo momento. Os balanceadores foram configurados para efetuar a decisão de balanceamento assim que eles detectarem um benefício de, no mínimo, 0,01 unidade de tempo. Na Figura 11.5(b), podemos ver três redistribuições logo em seguida das alterações de poder computacional.

Quando um nó aumenta seu poder, a distribuição de colunas deste nó aumenta proporcionalmente, caso contrário ela diminui. É possível notar que não existe instabilidade no *wavefront*, os *buffers* não estão nem aumentando nem diminuindo e não existe nenhum bloqueio de execução. Este cenário é praticamente perfeito, executando em 1,72 unidade de tempo, um ganho de 14% quando comparado com o cenário desbalanceado.

Na Figura 11.5(c), consideramos o balanceamento local. Pode-se notar que existe uma pequena demora entre a ocorrência da instabilidade e a decisão de balanceamento. Embora esse *delay* seja configurável e pode ser reduzido a valores bem pequenos, o aumento da sensibilidade de detecção do desbalanceamento pode gerar um grande custo adicional caso o ambiente apresente instabilidades muito curtas. A distribuição de colunas no balanceamento local é idêntica ao do cenário global, mas em caso de um ambiente real, existe a grande vantagem de não haver trocas intensas de mensagens nem um agente central de decisão. O tempo total de execução foi de 1,76 unidade de tempo, um ganho de 12% comparado com o cenário desbalanceado e uma perda de 2% comparado com o cenário de balanceamento global.

11.6 Conclusão do Capítulo

Neste capítulo foi proposto um sistema multiagentes capaz de balancear dinamicamente a execução do *wavefront* em múltiplos nós, permitindo acelerar o tempo de computação em caso de contenções ou desbalanceamento do ambiente. Por meio de seus agentes executores e balanceadores, algumas métricas foram desenvolvidas para identificar o desbalanceamento e encontrar a carga ideal para cada um dos nós. Duas estratégias foram propostas, uma utilizando informações globais (centralizada) e outras apenas informações locais de cada nó (descentralizada). As duas estratégias apresentaram um desempenho muito próximo, tornando a estratégia local preferida, visto que ela não precisa de uma comunicação tão intensa de dados. Embora os experimentos tenham sido apresentados em ambiente simulado, a implementação e avaliação deste mecanismo de balanceamento de carga no CUDAlign serão desenvolvidas em trabalhos futuros.

Capítulo 12

Multi-platform Architecture for Sequence Aligners (MASA)

O CUDAlign é uma variante do algoritmo Smith-Waterman (SW) otimizada para a arquitetura CUDA da NVIDIA. Durante o desenvolvimento desta tese, observou-se que algumas das otimizações empregadas no CUDAlign poderiam ser aplicadas para outras plataformas de hardware, tais como GPUs de outros fabricantes, CPUs, FPGAs, e plataformas de software, tais como OpenCL, OpenMP e OmpSs. Com esta observação, verificou-se que a arquitetura do CUDAlign poderia ser modificada para permitir o desacoplamento entre o código independente da plataforma e o código específico. A arquitetura resultante foi chamada de *Multi-platform Architecture for Sequence Aligners* (MASA), que é a contribuição deste capítulo.

O principal objetivo do MASA é prover um *framework* flexível e customizável que simplifique o desenvolvimento de ferramentas similares ao CUDAlign, permitindo o alinhamento de sequências de qualquer tamanho com métodos exatos em outras plataformas. A arquitetura MASA possui um conjunto de módulos e otimizações que pode ser reutilizado em diferentes ambientes. Além disso, devido a sua característica modular, esta arquitetura permite que novas otimizações possam ser aplicadas de uma só vez para diversas plataformas de hardware e software.

Neste capítulo, serão apresentadas a análise do código do CUDAlign (Seção 12.1), a arquitetura MASA e seus componentes (Seção 12.2), a interface de programação (API) (Seção 12.3), uma descrição sobre como criar extensões do MASA para diferentes arquiteturas (Seção 12.4), quatro implementações de extensões (Seção 12.5) e os resultados experimentais obtidos (Seção 12.6). Ao final, apresentaremos a conclusão do capítulo (Seção 12.7).

12.1 Análise do Código do CUDAlign

O código fonte do CUDAlign possui aproximadamente 90% de código independente de plataforma (portável) e 10% de código específico da arquitetura (não portável). A parte não portável está relacionada principalmente ao cálculo das equações de recorrência executada pelos *kernels* CUDA, assim como todas as funções da biblioteca CUDA (alocação de memória, transferência de dados para GPU, entre outros). Embora a menor parte do código esteja relacionada com a parte não portável, o tempo de execução do CUDAlign se

concentra justamente nesta parte. O código independente de plataforma possui funções de gerenciamento, tais como as operações de entrada/saída, estatísticas, comunicação, validação e gerenciamento dos estágios.

Para migrar o CUDAlign para outra plataforma, apenas o código específico da plataforma precisa ser reescrito para o novo ambiente. Visto que esta parte do código é pequena comparada com o restante do código MASA, o esforço de migração é proporcionalmente pequeno. Cada nova customização será chamada de “extensão”, sendo composta pela união da parte específica da plataforma (não portátil) com a parte independente de plataforma (portátil) compilada em uma biblioteca estática. Cada extensão produz um arquivo binário executável que pode ser executado na plataforma escolhida.

A estratégia de paralelização do CUDAlign depende do hardware da GPU, sendo então uma característica muito específica da plataforma. Entretanto, a arquitetura MASA permite que as diferentes extensões desenvolvam a sua técnica de paralelismo utilizando como base duas estratégias distintas: diagonal ou genérica. Com a estratégia em diagonal (Figura 8.13(c)), a extensão processa a matriz com um *wavefront* convencional, aplicando técnicas de paralelismo entre os blocos de uma mesma antidiagonal. Com a estratégia genérica, a extensão possui liberdade de processar os blocos da matriz em qualquer ordem de processamento (Figura 8.13(g)), desde que respeitadas as dependências entre os blocos. A estratégia genérica permite o uso de mecanismos de processamento em *dataflow*.

Visto que o *Block Pruning* (BP) (Capítulo 8) é a otimização mais relevante do CUDAlign, foi decidido que a implementação desta otimização deveria ser feita de maneira independente de plataforma, com flexibilidade para ser utilizada tanto para o processamento em diagonal (Figura 8.13(c)) como para uma ordem genérica de processamento (Figura 8.13(g)). Na versão atual do MASA, o BP foi desenvolvido para alinhamentos locais, assim como nas versões anteriores do CUDAlign. O BP para outros tipos de alinhamento será investigado em trabalhos futuros.

O projeto do MASA foi desenvolvido de forma a utilizar todas as funcionalidades do CUDAlign, em todas as suas versões anteriores (Capítulos 5 a 10). Entre estas funcionalidades podemos citar: o armazenamento de linhas especiais simultaneamente em disco e memória; execução em múltiplos nós [36] com suporte a ambientes heterogêneos [37]; múltiplas iterações do estágio 3 para reduzir o tamanho das partições recebidas pelo estágio 4, entre outras funcionalidades. O projeto do MASA também considera que futuras funcionalidades possam ser facilmente aplicadas a todas as extensões desenvolvidas para outras plataformas, bastando uma nova compilação. Entre as novas funcionalidades inseridas durante o desenvolvimento do MASA podemos citar a produção de vários tipos de alinhamento ótimo, incluindo os alinhamentos locais, globais e semiglobais. O código fonte do MASA está disponível na internet com licença de código livre (Anexo I).

12.2 Arquitetura MASA

A arquitetura MASA foi projetada em 5 módulos, conforme ilustrado na figura 12.1. Os módulos Gerência de Dados, Comunicação, Estatísticas e Gerenciamento dos Estágios são reutilizados em todas as extensões. O *Aligner* é o módulo responsável por executar o cálculo da matriz de programação dinâmica para os estágios 1 a 3. O *Aligner*, por sua vez, é dividido em 3 módulos. Os módulos *Tipo de Processamento* e *Block Pruning* são customizáveis de acordo com as características de cada plataforma. O cálculo da matriz

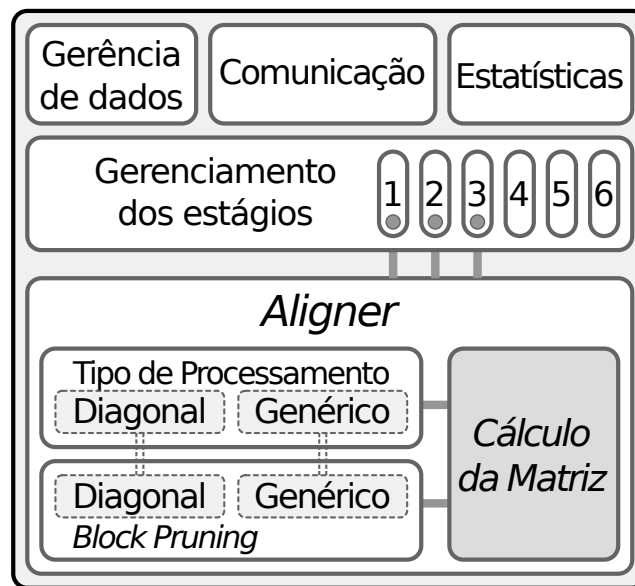


Figura 12.1: Arquitetura MASA

de programação dinâmica é a função mais computacionalmente utilizada no MASA, sendo então o módulo passível de sofrer as otimizações específicas para a plataforma utilizada. A seguir descreveremos a função de cada um dos módulos.

Aligner: Módulo responsável por executar a equação de recorrência de SW ou NW, contendo o código específico e otimizado para cada plataforma. Embora este módulo seja responsável por executar a seção não portátil do código, a arquitetura MASA permite que o *Aligner* reutilize alguns recursos não dependentes da plataforma, facilitando o desenvolvimento do código. A depender da escolha entre os tipos de processamento por diagonal ou genérico, diferentes técnicas de *Block Pruning* (Capítulo 8) podem ser utilizadas pelo *Aligner*. Além disso, uma versão básica do algoritmo de SW/NW é fornecida no MASA de maneira portátil, podendo ou não ser reutilizada em alguns ambientes. Como será visto na Seção 12.3, o reuso destes recursos é realizada por meio de hierarquia de classes utilizando o paradigma de orientação a objetos.

Gerência de dados: Este módulo é responsável por gerenciar a entrada e saída de dados do MASA, tais como a leitura das sequências de entrada, processamento dos parâmetros por linha de comando, armazenamento de linhas e colunas especiais (em disco e memória), geração de arquivos binários e textuais com o alinhamento, inicialização e leitura de diretórios, restauração de *checkpoints*, armazenamento de *crosspoints*, entre outros.

Estatísticas: Este módulo provê informações sobre o tempo de execução gasto em cada estágio, uso de memória e disco, porcentagem de blocos descartados com o BP, entre outras informações.

Gerenciamento dos Estágios: Este módulo é responsável por coordenar a execução dos estágios 1 a 3 utilizando o *Aligner* e por executar os estágios 4 a 6 em CPU, conforme inicialmente proposto e implementado no CUDAlign 2.0 (Capítulo 6) e aprimorado no CUDAlign 2.1 (Capítulo 7), 3.0 (Capítulo 9) e 4.0 (Capítulo 10). Durante a execução dos estágios 1 a 3, a matriz de programação dinâmica é dividida em partições (Figura 12.2). Cada partição é enviada para o *Aligner* contendo a primeira linha e a primeira coluna

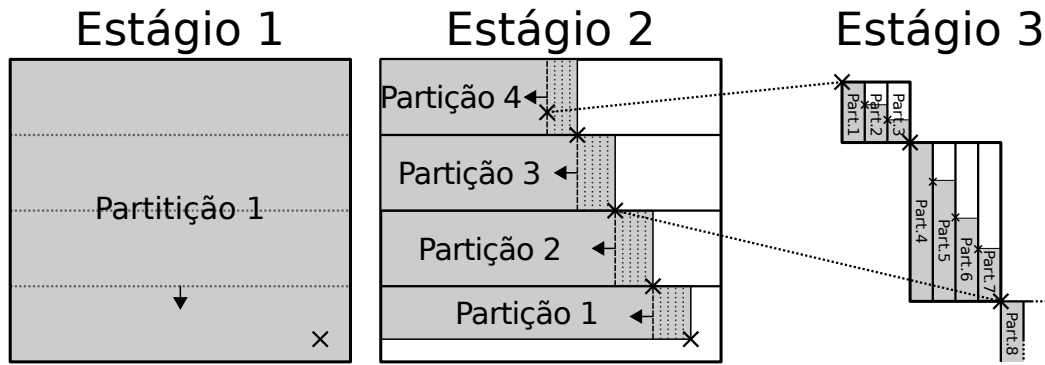


Figura 12.2: Processamento em partições.

desta partição. Por sua vez, o *Aligner* provê como saída a última linha e a última coluna da partição, a célula com o maior escore desta partição e as linhas e colunas especiais. No estágio 1, a partição enviada para o *Aligner* coincide com o tamanho total da matriz, podendo haver um subparticionamento caso o tamanho da matriz for maior do que o tamanho máximo suportado pelo *Aligner*. Nos estágios 2 e 3, existem várias partições menores, sendo elas definidas pelas fronteiras exercidas pelas linhas especiais salvas nos estágios anteriores. Nesses estágios, assim que as células da última linha e da última coluna da partição são recebidas, o módulo busca pelo *crosspoint* utilizando o procedimento de *matching* por objetivo (Seção 6.1.2) e, assim que o *crosspoint* é encontrado, o cálculo da partição é encerrado e uma nova partição é enviada para o *Aligner*. Este procedimento é executado até que as extremidades do alinhamento sejam encontradas em cada estágio. Os estágios 4 a 6 não utilizam o código especializado do *Aligner*, sendo então executados em CPU com o mesmo código reutilizável em todas as extensões.

Comunicação: Este módulo é responsável pela comunicação entre os diversos nós de processamento, permitindo a execução de múltiplas GPUs ou múltiplos dispositivos em um *wavefront* distribuído. Os conceitos utilizados neste módulo são aqueles descritos no CUDA 3.0, conforme apresentado no Capítulo 9.

Block Pruning (BP): A otimização BP foi implementada de maneira não dependente de plataforma, com alguns dos algoritmos descritos na Seção 8.4.2. O *Diagonal BP* (Figura 12.3(a)) funciona para a paralelização feita por diagonais e o *Generic BP* (Figura 12.3(b)), para a paralelização genérica. O *Generic BP* foi implementado tanto com complexidade linear $O(B_h + B_v)$ quanto quadrática $O(B_h \times B_v)$, onde B_h é o número de blocos na dimensão horizontal e B_v é o número de blocos na dimensão vertical (Seção 8.4.2).

Tipo de Processamento: Visto que as equações de recorrência de SW, NW e Gotoh possuem a mesma dependência de dados (i.e. o cálculo da célula $H_{i,j}$ depende das células $H_{i-1,j}$, $H_{i,j-1}$ e $H_{i-1,j-1}$), a arquitetura MASA provê, de maneira portátil, duas formas básicas de se processar a matriz: por diagonal e de maneira genérica. Em ambos os métodos, a matriz é dividida em um *grid* de blocos, mantendo o mesmo padrão de dependência de dados existente nas células.

No processamento por diagonal, o cálculo da matriz inicia-se no bloco superior esquerdo e propaga diagonalmente para o restante dos blocos (Figura 8.13(c)). Os blocos da mesma diagonal podem ser processados em paralelo, de acordo com a abordagem adotada pelo *Aligner* em cada plataforma. A estratégia de processamento por diagonal é

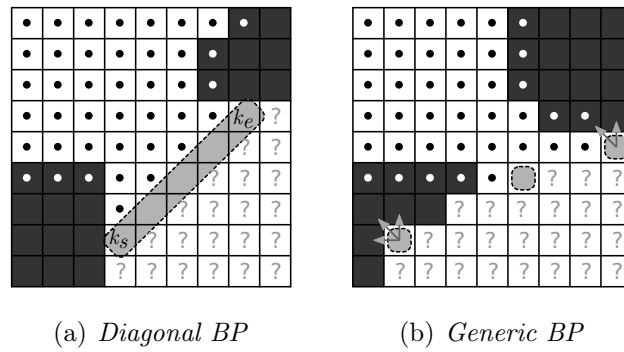


Figura 12.3: Estratégias de *Block Pruning* implementadas no MASA. Blocos com um círculo foram completamente processados, blocos de cor preta foram considerados *prunable*, blocos tracejados são os que estão sendo processado no dado momento e blocos com interrogação ainda não foram processados. O *Diagonal BP* processa blocos na mesma diagonal em paralelo. O *Generic BP* processa os blocos em qualquer ordem, mesmo se estiverem em diagonais distintas.

internamente associada ao *Diagonal BP*, facilitando a aplicação desta otimização em um *Aligner* que utilize esta abordagem.

Já no processamento genérico, a ordem de processamento dos blocos é de inteira responsabilidade do código específico de plataforma, permitindo que o *Aligner* processe a matriz em qualquer ordem de processamento, desde que respeitado o padrão de dependência entre os blocos. Esse método provê maior flexibilidade ao *Aligner*, permitindo inclusive que a matriz seja processada por diagonais. O padrão de processamento genérico foi proposto para permitir que o MASA operasse com processamento por *dataflow*. Neste modelo, cada bloco é considerado um nó no *dataflow*, sendo que ele se torna pronto para ser calculado assim que todas as suas dependências de dados forem resolvidas. O método de *dataflow* permite reduzir as barreiras de sincronização quando comparado com o método em diagonal.

12.3 MASA-API

A arquitetura MASA foi implementada utilizando a linguagem de programação C++, baseada no paradigma de orientação a objetos. A API do MASA está apresentada na Figura 12.4 como um diagrama de classes. A classe IALIGNER é a interface entre o módulo de gerenciamento de estágios e as implementações dos *Aligners*. Cada extensão do MASA deve criar a sua própria classe *Aligner* que implemente os métodos virtuais do IALIGNER. O *Aligner* comunica-se com o módulo de gerenciamento de estágios através da interface IMANAGER, que possui quatro tipos de métodos: métodos GET, que retornam parâmetros de alinhamento; métodos MUST, que ditam o comportamento de execução do *Aligner*; métodos RECEIVE, que transferem ao *Aligner* as linhas e colunas iniciais da partição; e os métodos DISPATCH, que enviam valores de células do *Aligner* para o módulo de gerenciamento dos estágios. Alguns dos métodos do IMANAGER estão listados na Tabela 12.1.

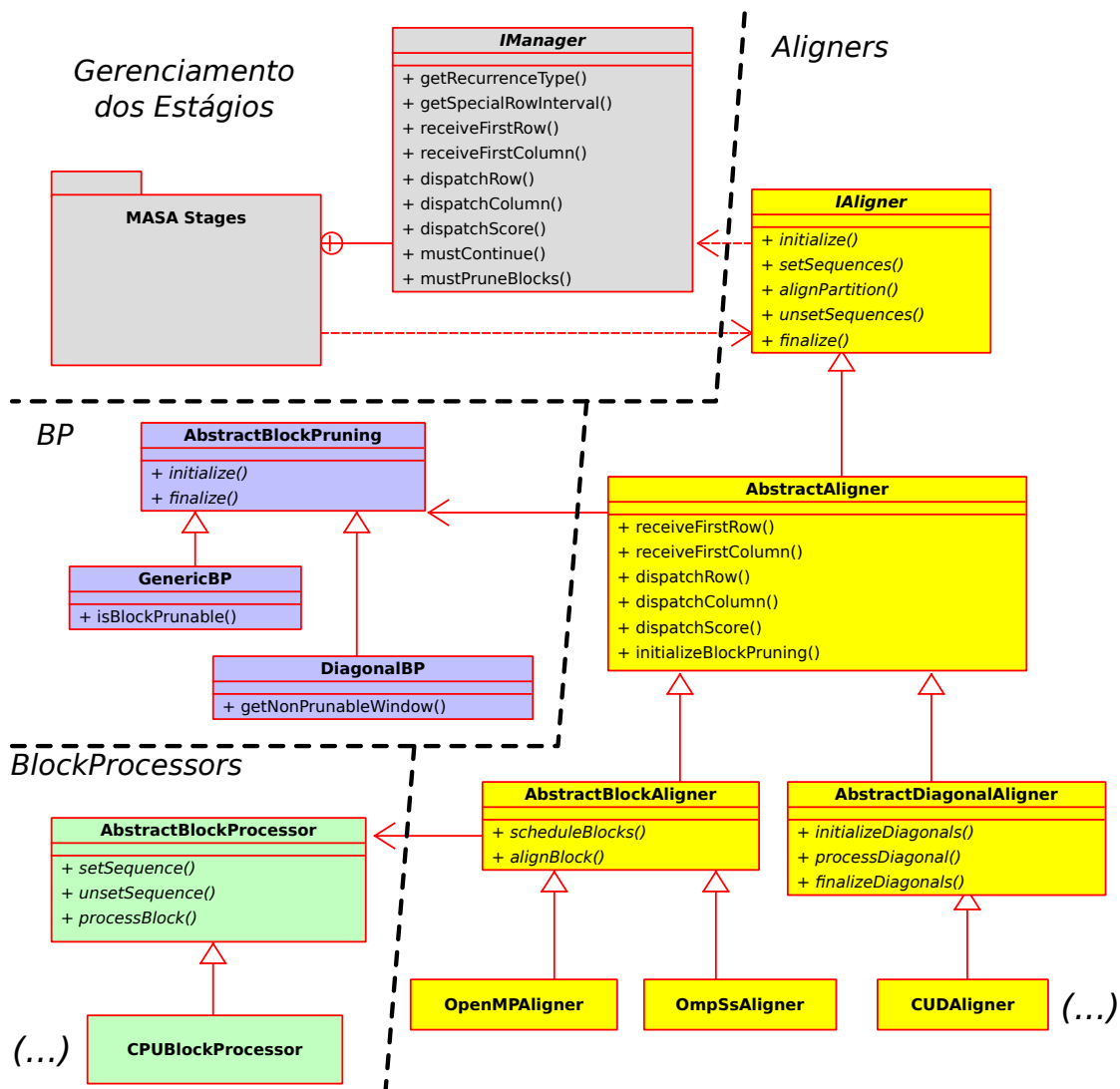


Figura 12.4: MASA-API: Diagrama de Classes.

Tabela 12.1: Alguns métodos do IManager

| Name | Description |
|-----------------------|--|
| GETRECURRENTYPE | Tipo de Recorrência: SW ou NW |
| GETSPECIALROWINTERVAL | Menor distância entre linhas especiais |
| RECEIVEFIRSTROW | Recebe as células a primeira linha da partição |
| RECEIVEFIRSTCOLUMN | Recebe as células da primeira coluna da partição |
| DISPATCHROW | Transfere as linhas especiais de uma partição |
| DISPATCHCOLUMN | Transfere as colunas especiais de uma partição |
| DISPATCHSCORE | Transfere o melhor escore encontrado em cada bloco |
| MUSTCONTINUE | Se for <i>false</i> , o <i>Aligner</i> deve finalizar a execução |
| MUSTPRUNEBLOCKS | Define se o <i>Aligner</i> deve executar o <i>Block Pruning</i> |

Visto que o *Aligner* estende a classe `IALIGNER`, ele deve implementar todos os métodos virtuais para poder ser instanciado. Os métodos virtuais do `IALIGNER` estão relacionados com o ciclo de vida do *Aligner*. Os métodos `IALIGNER::INITIALIZE` e `IALIGNER::FINALIZE` são executados apenas uma vez durante todo o ciclo de vida do *Aligner* e eles devem alocar e desalocar todos os recursos necessários durante a execução (e.g. hardware de aceleração, memória, etc.). Em seguida, cada iteração de um estágio executa uma vez o método `IALIGNER::SETSEQUENCE` para definir a região e a direção das sequências utilizadas neste estágio e o `IALIGNER::UNSETSEQUENCE` para permitir que o *Aligner* desaloque recursos temporariamente alocados em cada iteração. Para cada estágio, uma ou mais partições são alinhadas, de forma que o *Aligner* executa esta tarefa no método `IALIGNER::ALIGNPARTITION`.

Para simplificar a implementação de um `IALIGNER`, a arquitetura MASA possui uma hierarquia de classes com diferentes tipos de *Aligners*, conforme representado na Figura 12.4. A classe `ABSTRACTALIGNER` encapsula os métodos da interface `IMANAGER` e inicializa o *grid* e as operações do *Block Pruning*. O *Block Pruning* (BP) é gerenciado pela classe `ABSTRACTBLOCKPRUNING` e suas subclasses, mostradas na Figura 12.4 com as classes de cor azul. Desta forma, existem dois tipos de *Aligners*: `ABSTRACTBLOCKALIGNER` e `ABSTRACTDIAGONALALIGNER`, representando respectivamente as estratégias de paralelismo genérica e por diagonal.

A classe `ABSTRACTBLOCKALIGNER` calcula a matriz com base em blocos retangulares, usando a estratégia `GENERICBP`. Cada subclasse do `ABSTRACTBLOCKALIGNER` (ex.: `OpenMP` e `OmpSs`) deve implementar o seu próprio escalonador para executar os blocos em paralelo. A execução da equação de recorrência em cada bloco é feita por uma subclasse de `ABSTRACTBLOCKPROCESSOR`, representada na Figura 12.4. A subclasse `CPUBLOCKPROCESSOR` processa os blocos utilizando as equações de NW/SW em CPU. Outros processadores de blocos poderiam, por exemplo, utilizar FPGA ou instruções SSE para processar os blocos.

A classe `ABSTRACTDIAGONALALIGNER` calcula a matriz de programação dinâmica utilizando a estratégia de *wavefront*, processando uma diagonal por vez com a estratégia `DIAGONALBP`. A classe `CUDALIGNER` estende esta classe e, sempre que o método `PROCESSDIAGONAL` é chamado, o `CUDALIGNER` executa uma nova diagonal de blocos na GPU, utilizando a arquitetura CUDA. Embora o `ABSTRACTBLOCKALIGNER` também possa implementar um escalonador capaz de processar em diagonais, o `ABSTRACTDIAGONALALIGNER` permite que uma diagonal inteira seja processada de uma só vez em GPU, em vez de utilizar um processamento bloco a bloco. Sendo assim, a arquitetura CUDA paraleliza o cálculo dos blocos utilizando seu próprio escalonador. O `ABSTRACTDIAGONALALIGNER` também permite o processamento de blocos em formatos de paralelogramos.

12.4 Criação de uma extensão MASA

Para criar uma nova extensão, o programador deve desenvolver uma classe *Aligner* (implementando diretamente a interface `IALIGNER` ou estendendo uma das classes abstratas da hierarquia de classes) e fornecer uma instância desta classe para o ponto de entrada do MASA. Então, o código independente de plataforma processa os argumentos de linha de comando, lê as sequências e coordena a execução dos estágios. Quando as equações de recorrência de SW ou NW precisarem ser executadas, o *Aligner* é invocado (método

`alignPartition`). Este recebe as coordenadas da partição assim como a primeira linha e a primeira coluna utilizadas pelas equações de recorrência. O *Aligner* utiliza as funções do MASA (originalmente providas pela interface *IManager*) para informar o melhor escore encontrado, assim como as linhas especiais e a última linhas e coluna da partição.

O Algoritmo 10 ilustra uma versão simplificada de uma implementação baseada no `ABSTRACTBLOCKALIGNER`. A classe concreta (com todos os métodos virtuais implementados) será chamada simplesmente de `ALIGNER`. A inicialização básica de memória, o particionamento do *grid* e a inicialização do *Block Pruning* (BP) são transparentemente feitos pela inicialização do `ABSTRACTBLOCKALIGNER`. Então, o método `scheduleBlocks` específico (linhas 2-6) itera pelos blocos do *grid*, por diagonais, e invoca o método `alignBlock` para cada bloco (linha 4). O método `alignBlock` recebe do MASA a primeira linha (linha 10) ou coluna (linha 11) dos blocos vizinhos. O teste do BP é feito pelo MASA (linha 12) e, caso o bloco não seja *pruned*, o `ALIGNER` executa o método `processBlock` (linha 13) para calcular a equação de recorrência. O melhor escore encontrado é enviado ao MASA (linha 14). As linhas especiais (linha 16) e a última coluna da partição (linha 17) também são enviadas. O ponto de entrada do programa (linhas 20-23) cria uma instância do `ALIGNER` que é fornecida por parâmetro para o ponto de entrada do MASA (linha 22), utilizando o `CPUBLOCKPROCESSOR` como processador de blocos padrão (linha 21).

O método `processBlock` é implementado no `ABSTRACTBLOCKALIGNER` e ele essencialmente lê a primeira linha e coluna dos vetores `block.row` e `block.col`, calcula a equação de recorrência de SW/NW (utilizando o processador de blocos fornecido) e armazena a última linha e coluna nos mesmos vetores `block.row` e `block.col`. As linhas e colunas dos blocos são encadeadas de maneira que a última linha/coluna de um bloco seja a primeira linha/coluna do bloco vizinho. O método `processBlock` é intensivo computacionalmente e este código é bastante adequado para receber otimizações específicas de plataforma. Além disso, outras ferramentas de terceiros que realizam comparação de sequências podem ser adaptadas ao MASA. Isso pode ser feito por meio da reimplementação do método `processBlock` utilizando o código da ferramenta de terceiros, com modificações para se adequarem aos parâmetros de entrada e saída deste método.

12.5 Extensões MASA

Nesta seção, apresentaremos quatro extensões, que usam diferentes ferramentas e modelos de programação (OpenMP [143], OmpSs [144, 145] e CUDA [74]) para diferentes plataformas de *hardware* (multicores, GPUs, Intel Phi). A Figura 12.5 ilustra as quatro extensões implementadas e quais componentes do MASA foram utilizados em cada uma delas. Já a Figura 12.6 apresenta as classes utilizadas em cada extensão. Cada uma das extensões utiliza o modelo de *affine gap* (Seção 2.2.3) e apresenta alguma modificação no Algoritmo 10. Para apresentar uma ideia em termos de linha de código (excluindo linhas em branco e comentários), o código dependente de plataforma possui 116 (OpenMP), 187 (OmpSs) e 1.493 (CUDAalign) linhas, sendo que a parte independente de plataforma contém mais que 15.000 linhas. Nas seções a seguir, cada uma das extensões será detalhada.

Algorithm 10 Pseudocódigo do *Aligner* baseado em bloco

```
1: procedure ALIGNER::SCHEDULEBLOCKS
2:   for each diagonal do
3:     for each block in diagonal do
4:       ALIGNBLOCK(block)
5:     end for
6:   end for
7: end procedure
8:
9: procedure ALIGNER::ALIGNBLOCK(block)
10:  if block está na 1ª linha then block.row ← RECEIVEFIRSTROW
11:  if block está na 1ª coluna then block.col ← RECEIVEFIRSTCOLUMN
12:  if Not ISBLOCKPRUNED(block) then
13:    block.score := PROCESSBLOCK(block)
14:    DISPATCHSCORE(block.score)
15:  end if
16:  if isSpecialRow(block.row) then DISPATCHROW(block.row)
17:  if block está na última coluna then DISPATCHCOLUMN(block.col)
18: end procedure
19:
20: procedure MAIN(args)
21:   processor = new CPUBlockProcessor()
22:   MASA::ENTRYPOINT(args, new Aligner(processor))
23: end procedure
```

12.5.1 MASA-OpenMP/CPU

A extensão MASA-OpenMP/CPU utiliza OpenMP para calcular em paralelo os blocos de uma mesma diagonal. Para isso, a diretiva “`#pragma omp parallel for schedule(dynamic,1)`” foi inserida antes da linha 3 do Algoritmo 10. O escalonamento dinâmico (`dynamic`) foi utilizado pois o tamanho do laço paralelo é normalmente maior que o número de *threads*. A função `processBlock` executa a implementação padrão do SW/NW em CPU, sem utilizar vetorização (SIMD).

12.5.2 MASA-OpenMP/Phi

A extensão MASA-OpenMP/Phi emprega a mesma estratégia de paralelização do MASA-OpenMP/CPU, onde *threads* independentes processam cada diagonal em paralelo. Ela também utiliza o “dynamic scheduler” do OpenMP para proporcionar melhor desempenho à execução. O modo de execução nativa do Intel Phi foi utilizado nesta extensão de forma que toda a aplicação (incluindo a parte independente de plataforma) executasse dentro do coprocessador. Este modo de execução foi possível pois o Intel Phi executa um *kernel* Linux que provê os serviços necessários para esta abordagem.

Visto que o Intel Phi é equipado com vetores SIMD de 512-bits, tentou-se modificar o código do *Aligner* para vetorizar a execução de células dentro do loop do método `processBlock`. Com essa modificação, utilizamos as ferramentas de compilação da Intel para vetorizar automaticamente as instruções dentro do laço mais interno do algoritmo. Entretanto, esta modificação não apresentou efeitos positivos ao se comparar com a versão gerada por compilação cruzada do MASA-OpenMP/CPU. Este baixo desempenho ocorreu pois o código modificado possui um grande número de instruções com operações condicionais no laço interno, quando comparado com a versão original, o que já foi visto como limitador em outros trabalhos [146]. Como solução deste problema, soluções como a proposta por Farrar (Seção 4.2.2) seriam mais recomendadas. Sendo assim, neste trabalho utilizamos a versão com compilação cruzada do MASA-OpenMP/CPU para mostrar

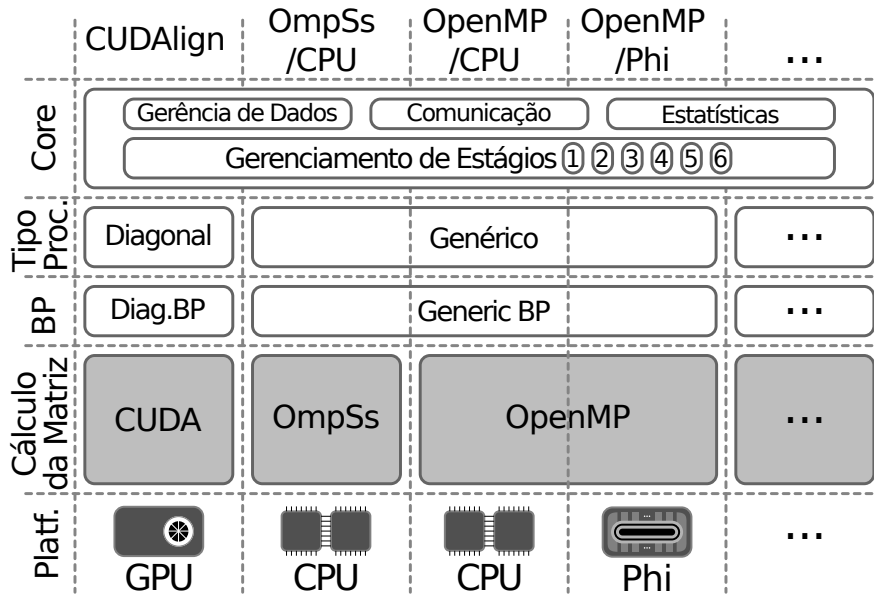


Figura 12.5: Extensões MASA

a flexibilidade da arquitetura MASA, deixando para trabalhos futuros as otimizações específicas do Intel Phi, tais como visto em [131]. Ainda assim, o desempenho do Intel Phi foi comparável com as outras soluções em CPU, conforme apresentaremos na Seção 12.6.

12.5.3 MASA-OmpSs/CPU

A extensão MASA-OmpSs/CPU utiliza o ambiente de programação OmpSs [145], que emprega um paralelismo baseado em fluxo de dados (*dataflow*). O OmpSs propõe um modelo de programação unificado para sistemas heterogêneos, incorporando ideias do OpenMP e adicionando suporte para paralelismo assíncrono e irregular. As aplicações OmpSs são descritas como fluxos de dados que formam um grafo de dependência entre as tarefas. O OmpSs utiliza várias otimizações, como por exemplo a movimentação automatizada de dados realizada pelo compilador e por bibliotecas de execução. A paralelização de aplicações no OmpSs é feita por meio de diretivas de compilação (“#pragma”), que devem ser utilizadas pelo programador para definir as porções do código que representam tarefas no *dataflow*. As dependências de dados entre as tarefas também são definidas nestas diretivas, formando um fluxo de entrada e saída de dados entre as diversas regiões do código. O OmpSs também fornece várias políticas de escalonamento de tarefas.

No MASA-OmpSs/CPU, as seções que invocam os métodos `processBlock`, `dispatchRow` e `dispatchColumn` foram todas anotadas com tarefas OmpSs, de forma que o compilador seja capaz de criar estruturas apropriadas para executar cada tarefa. As diretivas utilizadas também indicam os dados de entrada e saída de cada tarefa, criando as dependências entre as tarefas que garantirão a corretude da execução.

Considerando o Algoritmo 10, cada chamada ao método `alignBlock` (linhas 9-18) cria até três tipos tarefas: (1) tarefas `processBlock` (linhas 12-15); (2) tarefas `dispatchRow`

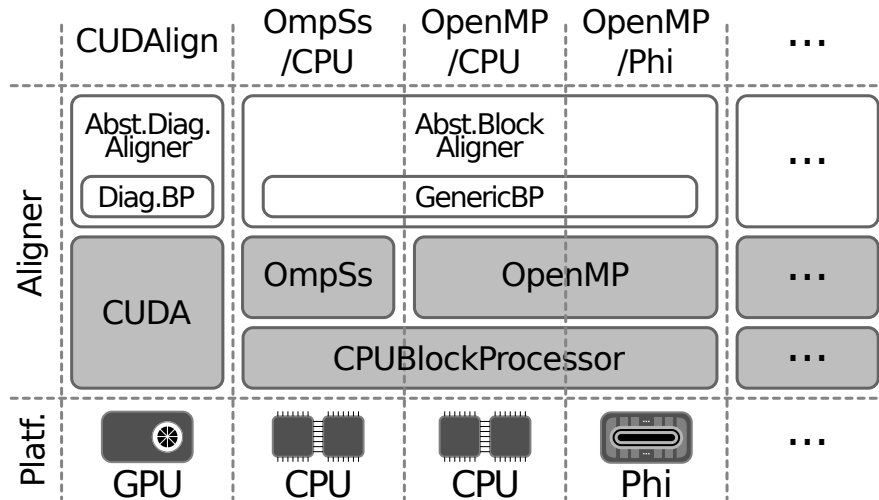


Figura 12.6: Extensões MASA - Hierarquia de Classes

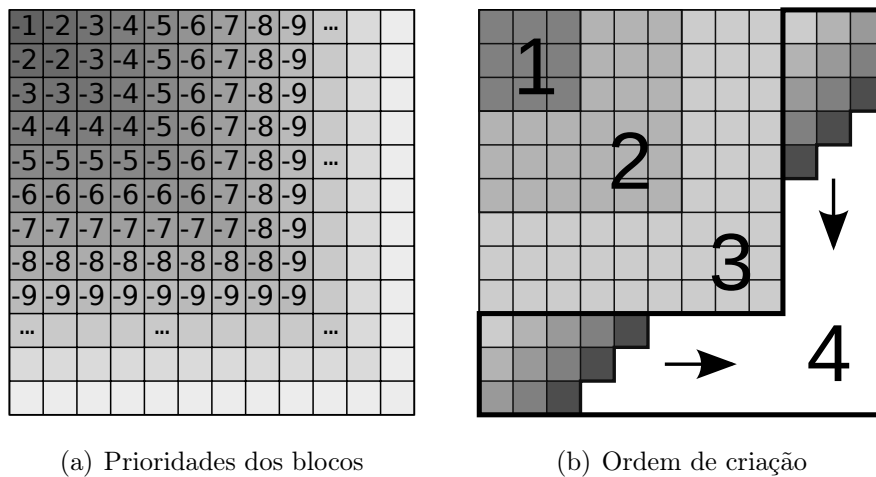


Figura 12.7: Prioridades e ordem de criação das tarefas no OmpSs.

(linha 16); and (3) `dispatchColumn` (linhas 17). A dependência entre as tarefas é criada considerando os vetores `block.row` e `block.col`.

Visto que o OmpSs é capaz de executar blocos em ordem genérica (Figura 8.13(g)) e que sabemos que o processamento por quadrados apresenta o melhor desempenho no *Block Pruning* (Seção 8.5.2), definimos um conjunto de prioridades para criar uma ordem preferencial de execução por quadrados (Figura 12.7(a)), beneficiando a eficácia da otimização *Block Pruning* conforme visto na Seção 8.6. Considerando o bloco (bx, by) , sua prioridade é definida como $\min(-bx, -by)$. O mecanismo *hysteresis throttle* do OmpSs [147] foi habilitado para limitar o número de tarefas no grafo de execução de tarefas, reduzindo assim a quantidade de memória utilizada pelo OmpSs. Quando o número de tarefas por *thread* chega em um dado limite, a criação de tarefas é suspensa até que o número de tarefas volte a um dado patamar. Para prevenir uma perda de paralelismo durante a suspensão criada pelo *hysteresis throttle*, a ordem de criação precisou ser alterada para seguir a ordem de prioridade definida. Sendo assim, as tarefas são criadas em raias (Figura 12.7(b)), cuja

largura é igual ao número de *threads* definida na execução. Por sua vez, o número de *threads* é igual ao número de núcleos no ambiente de execução. Dentro de cada raia, a criação de blocos segue a criação por diagonais (quarta raia na Figura 12.7(b)).

12.5.4 MASA-CUDAlign

Visto que o CUDAlign foi criado antes da arquitetura MASA, seu código original foi basicamente mantido o mesmo, com algumas modificações para se adequar à hierarquia de classes. Em vez de utilizar o ABSTRACTBLOCKALIGNER, o MASA-CUDAlign utiliza o ABSTRACTDIAGONALALIGNER como classe base. Neste caso, o método `processDiagonal` é chamado para cada diagonal e a classe CUDALIGNER invoca uma chamada de *kernel* por diagonal. Todas as alocações de memória e transferência de dados são feitas pelo CUDALIGNER, exigindo um maior controle sobre os dados quando comparado com as extensões baseadas no ABSTRACTBLOCKALIGNER. As chamadas aos métodos do MASA (i.e. métodos *dispatch* e *receive*) são sempre feitas pela CPU, normalmente precedidas por chamadas ao método `cudaMemcpy` para transferir de/para a GPU.

12.6 Resultados Experimentais

O MASA foi implementado em C/C++ e as extensões MASA (Seção 12.5) utilizaram CUDA 4.1, OpenMP 3.0 e OmpSs 1.0. Assim como no CUDAlign 3.0 (Seção 9.4), o Cluster Minotauro do Barcelona Supercomputing Center (Seção 9.4), foi utilizado nos testes. O Minotauro é composto por 128 nós, cada um com dois processadores Intel Xeon E5649 com 6 núcleos cada e duas GPUs NVIDIA Tesla M2090. Nos nossos testes, utilizamos apenas um nó do Minotauro. Para o MASA-OpenMP/CPU e o MASA-OmpSs/CPU, utilizamos os 12 núcleos disponíveis em um nó. O MASA-CUDAlign utilizou uma única GPU do nó. Para a execução do MASA-Phi, utilizamos o coprocessador Intel Xeon Phi SE10P cedido pelo programa XSEDE.

12.6.1 Sequências utilizadas nos testes

Tabela 12.2: Sequências utilizadas nos testes.

| Comp. | Sequência 1 | | Sequência 2 | | Score Ótimo | | | |
|-------|-------------|---------|-------------|---------|-------------|----------|------------|-----------|
| | Accession | Tamanho | Accession | Tamanho | Local | Overlap | Semiglobal | Global |
| 10K | AF133821.1 | 10K | AY352275.1 | 10K | 5091 | 4279 | 3594 | 2981 |
| 50K | NC_001715.1 | 57K | AF494279.1 | 57K | 52 | 2 | -60479 | -63880 |
| 150K | NC_000898.1 | 162K | NC_007605.1 | 172K | 18 | 1 | -201600 | -208667 |
| 500K | NC_003064.2 | 543K | NC_000914.1 | 536K | 48 | 1 | -554606 | -585725 |
| 1M | CP000051.1 | 1M | AE002160.2 | 1M | 88353 | 62525 | -588728 | -1189459 |
| 3M | BA000035.2 | 3M | BX927147.1 | 3M | 4226 | 0 | -2655752 | -2662376 |
| 5M | AE016879.1 | 5M | AE017225.1 | 5M | 5220960 | 5220960 | 5220955 | 5220950 |
| 7M | NC_005027.1 | 7M | NC_003997.3 | 5M | 172 | 2 | -6020449 | -8201748 |
| 10M | NC_017186.1 | 10M | NC_014318.1 | 10M | 10235188 | 10235188 | 10235188 | 10235188 |
| 23M | NT_033779.4 | 23M | NT_037436.3 | 25M | 9063 | 0 | -26746584 | -27446770 |
| 47M | NC_000021.7 | 47M | BA000046.3 | 32M | 27206434 | 27179500 | -484675 | -572719 |

Comparamos sequências reais de DNA (Tabela 12.2) obtidas do National Center for Biotechnology Information (NCBI) em www.ncbi.nlm.nih.gov. O comprimento das sequências varia de 10 KBP (milhares de pares de base) até 47 MBP (Milhões de pares de base).

Tabela 12.3: Tempos de execução, GCUPS e número de blocos *pruned* para as extensões do MASA (alinhamento local). Valores em negrito são os melhores resultados em cada comparação. O símbolo “~” indica valores desprezíveis.

| Comp. | SRA | Ext | Estágios (seg.) | | | | | Total | | Pruned | Tamanho dos Blocos |
|-------|------|------------|-----------------|-------|-------|-------|------|--------------|--------------|--------------|--------------------|
| | | | 1 | 2 | 3 | 4 | 5+6 | Tempo | GCUPS | | |
| 10K | 1M | CUDAAlign | ~ | ~ | ~ | 0.2 | ~ | 0.6 | 0.16 | 27.3% | 512×321 |
| | | OmpSs/CPU | 0.1 | ~ | ~ | ~ | ~ | 0.3 | 0.41 | 39.0% | 418×428 |
| | | OpenMP/CPU | 0.2 | 0.1 | ~ | ~ | ~ | 0.2 | 0.57 | 37.8% | 418×428 |
| | | OpenMP/Phi | 0.7 | 0.4 | 0.2 | 0.2 | ~ | 1.6 | 0.06 | 42.8% | 129×129 |
| 50K | 3M | CUDAAlign | 0.2 | ~ | ~ | ~ | ~ | 1.4 | 2.31 | 0.0% | 512×271 |
| | | OmpSs/CPU | 1.7 | 0.2 | ~ | ~ | ~ | 1.9 | 1.72 | 0.0% | 1024×1024 |
| | | OpenMP/CPU | 1.8 | 0.2 | ~ | ~ | ~ | 2.1 | 1.58 | 0.0% | 1024×1024 |
| | | OpenMP/Phi | 3.2 | 0.8 | ~ | ~ | ~ | 4.1 | 0.78 | 0.0% | 129×129 |
| 150K | 5M | CUDAAlign | 1.3 | ~ | ~ | ~ | ~ | 1.9 | 14.39 | 0.0% | 512×335 |
| | | OmpSs/CPU | 13.0 | 1.3 | ~ | ~ | ~ | 14.4 | 1.93 | 0.0% | 1024×1024 |
| | | OpenMP/CPU | 13.8 | 1.4 | ~ | ~ | ~ | 15.2 | 1.83 | 0.0% | 1024×1024 |
| | | OpenMP/Phi | 17.8 | 1.8 | ~ | ~ | ~ | 19.8 | 1.41 | 0.0% | 168×159 |
| 500K | 50M | CUDAAlign | 10.3 | ~ | ~ | ~ | ~ | 11.6 | 25.00 | 0.0% | 512×1047 |
| | | OmpSs/CPU | 134.1 | 0.9 | ~ | ~ | ~ | 135.2 | 2.15 | 0.0% | 1024×1024 |
| | | OpenMP/CPU | 136.0 | 1.0 | ~ | ~ | ~ | 137.1 | 2.12 | 0.0% | 1024×1024 |
| | | OpenMP/Phi | 160.1 | 12.9 | ~ | ~ | ~ | 173.3 | 1.68 | 0.0% | 266×263 |
| 1M | 250M | CUDAAlign | 34.4 | 1.2 | 0.9 | 4.7 | 0.1 | 41.9 | 26.75 | 11.0% | 512×2095 |
| | | OmpSs/CPU | 453.8 | 22.4 | 1.8 | 3.3 | 0.1 | 481.7 | 2.33 | 12.6% | 1024×1024 |
| | | OpenMP/CPU | 462.2 | 23.4 | 1.1 | 1.0 | 0.1 | 488.1 | 2.30 | 11.9% | 1024×1024 |
| | | OpenMP/Phi | 538.1 | 33.8 | 11.1 | 21.6 | 1.0 | 605.9 | 1.85 | 12.0% | 511×525 |
| 3M | 1G | CUDAAlign | 331 | ~ | 0.3 | ~ | ~ | 333 | 31.04 | 0.1% | 512×6411 |
| | | OmpSs/CPU | 4748 | 46.9 | ~ | 0.1 | ~ | 4796 | 2.15 | 0.2% | 1024×1024 |
| | | OpenMP/CPU | 4724 | 48.7 | ~ | 0.1 | ~ | 4773 | 2.16 | 0.2% | 1024×1024 |
| | | OpenMP/Phi | 5275 | 79.9 | 0.5 | 1.1 | ~ | 5357 | 1.93 | 0.2% | 1024×1024 |
| 5M | 3G | CUDAAlign | 455 | 16.7 | 9.6 | 51.2 | 1.5 | 536 | 51.02 | 53.7% | 512×10212 |
| | | OmpSs/CPU | 4296 | 213.6 | 46.8 | 36.8 | 1.6 | 4595 | 5.95 | 66.5% | 1024×1024 |
| | | OpenMP/CPU | 5449 | 213.8 | 11.3 | 9.5 | 1.2 | 5685 | 4.81 | 57.5% | 1024×1024 |
| | | OpenMP/Phi | 6053 | 448.1 | 117.3 | 154.5 | 11.2 | 6785 | 4.03 | 57.5% | 1024×1024 |
| 7M | 3G | CUDAAlign | 1190 | ~ | ~ | ~ | ~ | 1191 | 31.35 | 0.0% | 512×10209 |
| | | OmpSs/CPU | 17080 | 109.4 | ~ | ~ | ~ | 17190 | 2.17 | 0.0% | 1024×1024 |
| | | OpenMP/CPU | 17044 | 111.3 | ~ | ~ | ~ | 17157 | 2.18 | 0.0% | 1024×1024 |
| | | OpenMP/Phi | 18504 | 176.1 | ~ | ~ | ~ | 18682 | 2.00 | 0.0% | 1024×1024 |
| 10M | 5G | CUDAAlign | 1730 | 57.8 | 18.9 | 102.7 | 3.1 | 1914 | 54.74 | 53.3% | 512×19993 |
| | | OmpSs/CPU | 16572 | 903.3 | 182.2 | 163.0 | 2.2 | 17824 | 5.88 | 66.5% | 1024×1024 |
| | | OpenMP/CPU | 20838 | 928.9 | 36.7 | 41.6 | 3.1 | 21849 | 4.80 | 57.5% | 1024×1024 |
| 23M | 10G | CUDAAlign | 17785 | ~ | ~ | 0.2 | ~ | 17787 | 31.75 | 0.0% | 512×47936 |
| 47M | 50G | CUDAAlign | 28029 | 288 | 61.2 | 341.6 | 15.6 | 28738 | 53.58 | 48.1% | 512×91688 |

Os parâmetros do SW/NW foram ma (*match*): +1; mi (*mismatch*) -3; G_{open} : -5; G_{ext} : -2. Para validação, os escores ótimos obtidos durante os testes também estão mostrados na Tabela 12.2 para alinhamentos locais, *overlap*, semiglobais e globais. Utilizando a Definição 2.1.20, consideraremos, para os resultados desta seção, alinhamentos semiglobais como do tipo 2/2 e do tipo 3/3 (*overlap* [5]).

12.6.2 Tempo de execução para alinhamentos locais

Nos nossos testes, o tamanho padrão dos blocos para o MASA-OpenMP/CPU and MASA-OmpSs/CPU foi de 1024×1024, sendo este tamanho reduzido automaticamente se o número de blocos em uma diagonal for menor que $2 \times p$, onde p é o número de núcleos. Para o MASA-CUDAAlign, utilizamos $T = 128$ threads, $B = 512$ blocos e $\alpha = 4$, resultando em um tamanho de bloco de $\alpha \cdot T \times \frac{n}{B} = 512 \times \frac{n}{512}$, onde n é a largura da partição (tamanho da sequência 2). Para o MASA-OmpSs/Phi, nós empiricamente definimos o tamanho do

bloco para cada comparação, incrementando o tamanho do bloco até que não houvesse grandes efeitos no desempenho.

A Tabela 12.3 apresenta o tamanho do *Special Rows Area* (SRA) utilizado, o tempo gasto em cada estágio, o tempo de execução total, o GCUPS, o percentual de blocos *pruned* e o tamanho dos blocos no estágio 1. O tempo total inclui também o tempo de inicialização e o tempo gasto com entrada e saída em todos os estágios. Algumas implementações não executaram todas as comparações por causa de restrições de tempo nos ambientes. O SRA foi definido para armazenar até 4GB em memória RAM, sendo que os bytes restantes foram armazenados em disco.

O MASA-CUDAlign apresentou o melhor desempenho para as sequências com mais de 50KBP, variando o GCUPS de 2,31 até 54,74. Estes resultados são comparáveis com os apresentados na Seção 7.2, observando-se que a placa M2090 é um pouco mais rápida que a GTX 560 Ti (Tabelas 3.1 e 3.2). Este primeiro conjunto de resultados mostra que a arquitetura MASA não inseriu *overhead* considerável no tempo de execução total. Apenas na comparação de 10KBP o MASA-CUDAlign obteve resultados inferiores às outras extensões, principalmente por causa da falta de paralelismo necessário para utilizar a GPU em sua capacidade máxima.

O GCUPS variou de 0,57 a 4,81 (MASA-OpenMP/CPU), 0,41 a 5,95 (MASA-OmpSs/CPU) and 0,06 a 4,03 (MASA-OpenMP/Phi). As extensões MASA-OmpSs/CPU, MASA-OpenMP/CPU e MASA-OpenMP/Phi podem ser otimizadas com o uso da versão *striped* (Seção 4.2.2) que permite o uso eficiente de instruções vetoriais para acelerar a execução da equação de recorrência. Assim, este tipo de otimização será tratado em trabalhos futuros.

Conforme esperado, as comparações com sequências mais similares (Tabela 12.2) apresentam maior eficácia de *pruning*. As sequências de 10K, 5M, 10M e 47M são muito similares, com eficácia de *pruning* acima de 40%. As sequências com 1M possuem similaridade média, com *pruning* acima de 10%. As demais sequências são pouco similares, apresentando *pruning* abaixo de 1%.

A comparação entre o tempo de execução do MASA-OmpSs/CPU (generic BP + paralelismo por *dataflow*) e do MASA-OpenMP/CPU (generic BP + paralelismo em diagonal) mostra que a primeira extensão é mais eficiente na maioria dos casos. O melhor desempenho do MASA-OmpSs/CPU deve-se principalmente a maior eficácia de *pruning* combinada com a execução paralela flexível da estratégia em *dataflow*. Por exemplo, nas comparações de 5M e 10M, o MASA-OmpSs/CPU processou 21% menos blocos que o MASA-OpenMP/CPU, reduzindo assim o tempo de execução do estágio 1 na mesma proporção. Para a comparação de 1M, o GCUPS do MASA-OmpSs/CPU também foi melhor que o MASA-OpenMP/CPU, mas em apenas 1% visto que as sequências não são tão similares e a taxa de *pruning* foi praticamente a mesma.

A melhora na eficácia de *pruning* do MASA-OmpSs/CPU deve-se à ordem de execução dos blocos definida por prioridades, conduzindo o *wavefront* em formato de quadrado (*square wave*). O *square wave* privilegia os blocos na diagonal principal, que apresentam os melhores escores caso as sequências sejam bastante similares (Seção 8.5.2). A ordem de execução da matriz está mostrada na Figura 12.8, onde é possível ver um tom de cinza diferente para cada 1000 blocos processados. Os primeiros 100×100 blocos (marcados com um quadrado na figura) foram executados de maneira similar ao *square wave*, respeitando o conjunto de prioridades de cada bloco. Após os 100×100 primeiros blocos, podemos ver

que as raias usadas durante a criação das tarefas são muito mais evidentes. Isto ocorre pois o sistema de *hysteresis throttle* suspende a criação de tarefas assim que o número de tarefas enfileiradas atinge um valor máximo, forçando que as *threads* processem os blocos vizinhos, respeitando o *wavefront* em cada raia. Assim que as *threads* começam a processar os blocos em um formato diagonal em cada raia, este padrão tende a persistir até o final do cálculo da matriz.

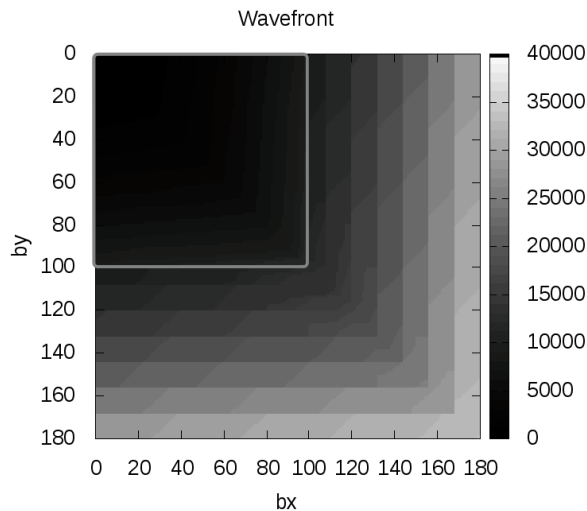


Figura 12.8: Padrão de processamento do MASA-OmpSs/CPU.

Embora o block *pruning* e a estratégia de paralelização do MASA-OpenMP/CPU e do MASA-OpenMP/Phi sejam os mesmos, a porcentagem de blocos *pruned* varia em alguns casos. Isso é devido à diferença do tamanho dos blocos nestas situações específicas. Em especial, a comparação de 10K apresentou uma melhor eficácia de *pruning* para o MASA-OpenMP/Phi, visto que esta extensão possui blocos bem menores.

12.6.3 Resultados de pruning para *perfect match*

Esta seção apresenta uma análise detalhada do *Block Pruning* para as estratégias de paralelização utilizadas para os casos de *perfect match* (duas sequências idênticas) usando alinhamento local. As Figuras 12.9(a), 12.9(b) e 12.9(c) mostra, respectivamente, as áreas de *pruning* para o MASA-CUDAlign (55,2%), MASA-OpenMP/CPU (57,3%) e MASA-OmpSs/CPU (66,2%) quando comparando a sequência CP000051.1 (1 MBP) com ela mesma (*perfect match*). A linha na diagonal representa o alinhamento ótimo e ele divide a área de *pruning* nos lados esquerdo e direito. O MASA-OmpSs/CPU apresenta a maior área de pruning em ambos os lados. Tanto o MASA-CUDAlign e o MASA-OpenMP/CPU processam a matriz por diagonais, mas o MASA-CUDAlign possui blocos mais largos, produzindo um *wavefront* menos inclinado e com diferente área de *pruning*. Comparando o MASA-CUDAlign com o MASA-OpenMP/CPU, o MASA-CUDAlign possui uma maior área de *pruning* no lado esquerdo e o MASA-OpenMP/CPU, no lado direito. Entretanto, a diferença no lado direito é mais significativa que a do lado esquerdo, produzindo uma maior área total de *pruning* no MASA-OpenMP/CPU.

A Figura 12.10 apresenta a área de *pruning* do MASA-CUDAlign para o *perfect match* com sequências maiores (23 MBP, 32 MBP and 47 MBP), onde o formato do *pruning* é praticamente idêntico, com 53% de blocos *pruned*. O formato mostrados na Figura 12.10 (23 MBP, 32 MBP e 47 MBP) é um pouco diferente que a forma apresentada na Figura 12.9(a) (1 MBP), especialmente no canto superior direito e na linha central. Esta diferença ocorre por causa da diferença dos tamanhos dos blocos, que altera a inclinação da ângulo do *wavefront*.

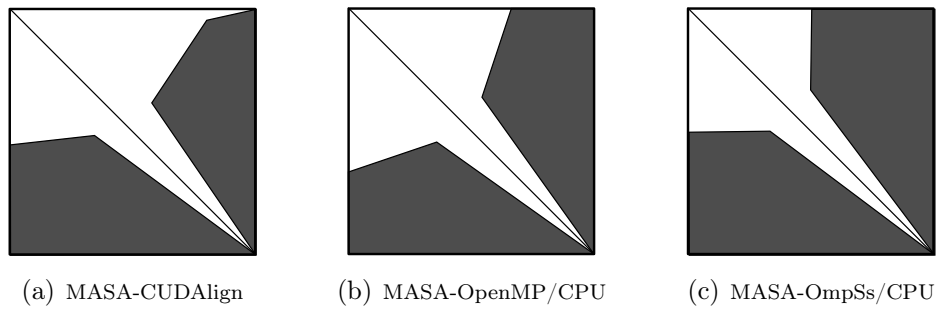


Figura 12.9: Área de *pruning* (cinza) comparando a sequência CP000051.1 (1 MBP) com ela mesma (*perfect match*).

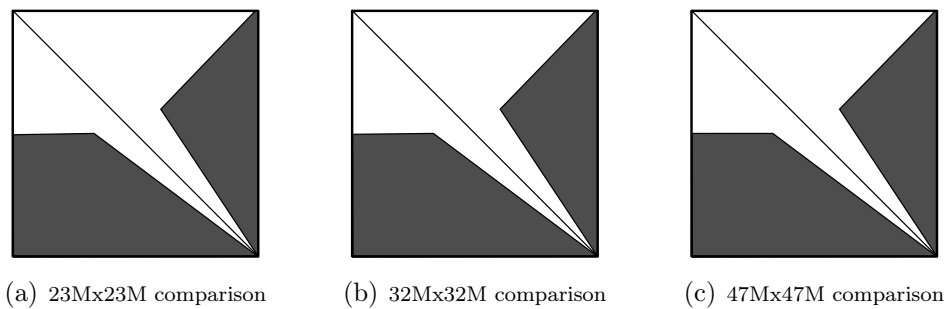


Figura 12.10: Outras áreas de *pruning* (cinza) com *perfect match*, utilizando o MASA-CUDAlign.

12.6.4 Tempo de execução para alinhamentos globais e semiglobais

Nesta seção, comparamos os alinhamentos globais, semiglobais e *overlap* com o alinhamento local, considerando a implementação do MASA-CUDAlign. Utilizando a Definição 2.1.20, estamos considerando alinhamentos semiglobais como do tipo 2/2, alinhamentos *overlap* como 3/3, alinhamentos globais como ++ e locais como */*. Para uma comparação justa, o *Block Pruning* foi desabilitado e uma nova execução foi feita para os quatro tipos de alinhamento. Em alguns casos, o alinhamento em *overlap* pode ser bastante curto, residindo muito próximo dos cantos da matriz, limitando o resultado a valores positivos.

A Tabela 12.4 apresenta os tempos de execução e o GCUPS para o MASA-CUDAlign sem o *Block Pruning*, para os quatro tipos de alinhamento (local, *overlap*, semiglobal e global). Visto que os alinhamentos global, semiglobal e *overlap* utilizam a equação de

recorrência NW, que não possui uma cláusula condicional para evitar valores negativos, o estágio 1 apresentou desempenho melhor que a equação de recorrência do alinhamento local (SW). Esta diferença pode ser vista na Tabela 12.4, onde o alinhamento local é aproximadamente 3% mais lento no estágio 1 que os outros tipos de alinhamento, na maioria dos casos.

O tempo de execução dos estágios 2 a 6 depende do tamanho do alinhamento ótimo produzido. O alinhamento global ótimo tende a residir próximo da diagonal principal da matriz e seu comprimento é, no mínimo, o tamanho da maior sequência. Então, o alinhamento global é normalmente maior que os outros alinhamentos, produzindo um maior tempo de *traceback*. O alinhamento local das sequências de 47M (Figura 12.12(b)) e o alinhamento *overlap* residem em uma diagonal deslocada, sendo mais curtos que o alinhamento do alinhamento global e semiglobal, os quais inserem vários *gaps* antes do início da sequência 1. Entretanto, nas comparações de 5M e 10M, todos os tipos de alinhamento produzem praticamente o mesmo resultado, gerando um tempo de *traceback* praticamente idêntico.

A Figura 12.11 apresenta as formas dos alinhamentos local, *overlap*, semiglobal e global para a comparação de 1M. Nesta figura, é possível ver que os diferentes tipos de alinhamento residem em diferentes bordas da matriz. O alinhamento local (Figura 12.11(a)) inicia-se no meio da matriz e termina na borda direita. O alinhamento *overlap* (Figura 12.11(b)) inicia-se na borda esquerda e termina na borda inferior. O alinhamento semiglobal (Figura 12.11(c)) inicia-se na borda esquerda e termina na borda direita, representando todos os caracteres da segunda sequência. Por fim, o alinhamento global (Figura 12.11(d)) inicia-se no canto superior esquerdo e termina no canto inferior direito, produzindo um alinhamento maior que todos os outros alinhamentos, embora possuindo o menor escore entre eles. A diferença do posicionamento das extremidades do alinhamento geram diferentes escore ótimos (Tabela 12.2), i.e, 88353 (local), 62525 (*overlap*), -588728 (semiglobal) e -1189459 (global).

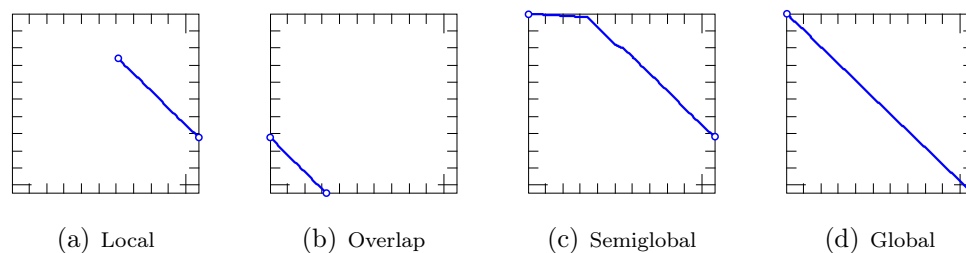


Figura 12.11: Alinhamentos ótimos da comparação de 1M (CP000051.1 e AE002160.2)

Tabela 12.4: Tempos de execução e GCUPS para o MASA-CUDAlign sem BP (alinhamentos do tipo local, *overlap*, semiglobal e global)

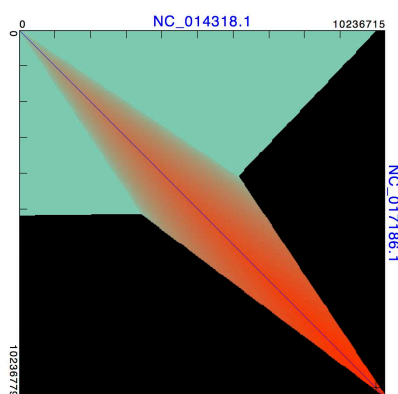
| Comp. | SRA | Tipo | Estágios (seg.) | | | | | Total | |
|-------|------|--------|-----------------|-------|------|-------|------|-------|-------|
| | | | 1 | 2 | 3 | 4 | 5+6 | Tempo | GCUPS |
| 10K | 1M | Local | ~ | ~ | ~ | 0,2 | ~ | 1,3 | 0,1 |
| | | Overl. | ~ | ~ | ~ | 0,2 | ~ | 0,6 | 0,2 |
| | | Semi | ~ | ~ | ~ | 0,2 | ~ | 1,6 | 0,1 |
| | | Global | ~ | ~ | ~ | 0,2 | ~ | 1,5 | 0,1 |
| 50K | 3M | Local | 0,2 | ~ | ~ | ~ | ~ | 0,4 | 7,1 |
| | | Overl. | 0,2 | ~ | ~ | ~ | ~ | 1,2 | 2,7 |
| | | Semi | 0,2 | ~ | ~ | 0,5 | ~ | 2,0 | 1,6 |
| | | Global | 0,2 | 0,1 | ~ | 0,7 | ~ | 2,3 | 1,4 |
| 150K | 5M | Local | 1,2 | ~ | ~ | ~ | ~ | 2,0 | 14,2 |
| | | Overl. | 1,2 | ~ | ~ | ~ | ~ | 2,6 | 10,6 |
| | | Semi | 1,2 | 0,4 | 0,5 | 2,2 | ~ | 5,8 | 4,8 |
| | | Global | 1,2 | 0,4 | 0,6 | 1,6 | ~ | 5,4 | 5,2 |
| 500K | 50M | Local | 10,3 | ~ | ~ | ~ | ~ | 10,7 | 27,2 |
| | | Overl. | 10,0 | ~ | ~ | ~ | ~ | 11,3 | 25,8 |
| | | Semi | 10,0 | 1,3 | 0,9 | 4,6 | ~ | 18,0 | 16,2 |
| | | Global | 9,9 | 1,4 | 1,0 | 5,6 | 0,1 | 19,1 | 15,3 |
| 1M | 250M | Local | 37,5 | 1,2 | 0,9 | 4,7 | 0,1 | 45,2 | 24,8 |
| | | Overl. | 36,5 | 0,8 | 0,6 | 3,3 | 0,1 | 43,0 | 26,1 |
| | | Semi | 36,4 | 2,2 | 1,8 | 10,2 | 0,2 | 51,7 | 21,7 |
| | | Global | 36,2 | 2,5 | 2,0 | 10,5 | 0,3 | 52,9 | 21,2 |
| 3M | 1G | Local | 332 | ~ | ~ | 0,3 | ~ | 333 | 31,0 |
| | | Overl. | 321 | ~ | ~ | ~ | ~ | 322 | 32,0 |
| | | Semi | 321 | 11,2 | 6,0 | 37,7 | 1,1 | 378 | 27,3 |
| | | Global | 320 | 11,1 | 6,0 | 37,7 | 0,9 | 377 | 27,4 |
| 5M | 3G | Local | 871 | 17,5 | 9,6 | 51,3 | 1,6 | 952 | 28,7 |
| | | Overl. | 846 | 17,1 | 9,6 | 51,3 | 4,0 | 928 | 29,4 |
| | | Semi | 846 | 17,2 | 9,7 | 51,4 | 2,8 | 928 | 29,5 |
| | | Global | 844 | 16,9 | 9,6 | 51,5 | 1,7 | 925 | 29,6 |
| 7M | 3G | Local | 1189 | 0,2 | ~ | ~ | ~ | 1191 | 31,4 |
| | | Overl. | 1152 | 0,2 | ~ | ~ | ~ | 1154 | 32,4 |
| | | Semi | 1152 | 19,6 | 9,2 | 45,2 | 4,0 | 1232 | 30,3 |
| | | Global | 1150 | 22,0 | 10,9 | 72,7 | 2,1 | 1259 | 29,7 |
| 10M | 4G | Local | 3317 | 113,5 | 18,7 | 103,0 | 3,4 | 3557 | 29,5 |
| | | Overl. | 3210 | 113,8 | 18,8 | 105,8 | 11,6 | 3463 | 30,3 |
| | | Semi | 3211 | 114,9 | 18,9 | 102,1 | 5,4 | 3454 | 30,3 |
| | | Global | 3208 | 114,8 | 18,9 | 102,2 | 3,4 | 3449 | 30,4 |
| 23M | 10G | Local | 17793 | 0,2 | ~ | 0,3 | ~ | 17796 | 31,7 |
| | | Overl. | 17259 | 0,2 | ~ | ~ | ~ | 17263 | 32,7 |
| | | Semi | 17260 | 392,7 | 42,8 | 203,4 | 6,6 | 17908 | 31,5 |
| | | Global | 17253 | 407,7 | 44,7 | 245,6 | 8,6 | 17962 | 31,4 |
| 47M | 50G | Local | 48498 | 445,1 | 61,2 | 339,5 | 10,8 | 49357 | 31,2 |
| | | Overl. | 47043 | 440,3 | 61,2 | 339,8 | 10,9 | 47898 | 32,1 |
| | | Semi | 47042 | 477,7 | 62,9 | 371,2 | 13,5 | 47970 | 32,1 |
| | | Global | 47026 | 475,5 | 62,6 | 373,4 | 10,9 | 47950 | 32,1 |

12.6.5 Resultados dos alinhamentos

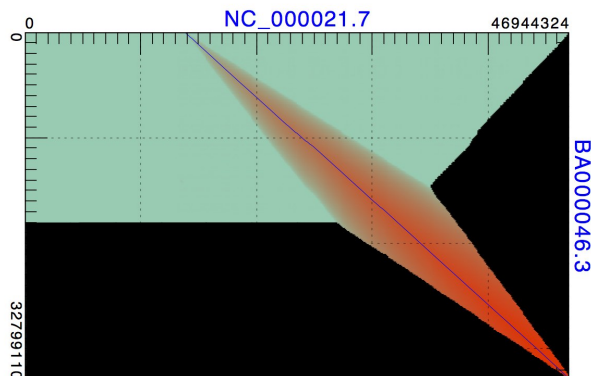
Nesta seção, mostramos alguns resultados de alinhamentos obtidos com a arquitetura MASA.

10M - *Amycolaptosis mediterranei*

A Figura 12.12(a) mostra o alinhamento ótimo local entre a estirpe S699 (NC_017186.1) e U32 (NC_014318.1) da espécie *Amycolaptosis mediterranei*. O *Amycolaptosis mediterranei* produz um importante antibiótico (ricamycin) e é extensamente estudado na literatura [148] [149]. Na Figura 12.12(a), as regiões verde e laranja representam as células da matriz que foram calculadas e a região preta representa as células *pruned*. Como pode ser visto, o resultado é praticamente um *match* perfeito. O percentual de *matches*, *mismatches* e *gaps* são, respectivamente, 99,996%, 0,003% e 0,001%. Estes resultados estão consistentes com o reportado em [148]. O escore local ótimo obtido foi 10.235.188.



(a) 10M - *A. mediterranei*



(b) chr21 - human x chimpanzee

Figura 12.12: Alinhamento local ótimo de algumas comparações. A área verde (cinza claro) apresenta baixo escore, a área laranja (cinza escuro) apresenta escore alto e área preta indica área com blocos *pruned*.

Cromossomo 21 - Homem × Chimpanzé

A análise entre os cromossomos 21 do homem e do chimpanzé é um tópico bastante ativo de pesquisa e novos genes estão sendo descobertos de maneira cada vez mais rápida [150]. Recentemente, uma pesquisa [151] efetuou uma nova análise genética deste cromossomo em relação ao estudo da síndrome de Down.

A Figura 12.12(b) apresenta o gráfico do alinhamento local ótimo entre os cromossomos 21 e a Figura 12.13 mostra uma pequena parte do arquivo texto contendo o alinhamento. Na Figura 12.12(b), o cromossomo 21 humano (NC_000021.7) está no eixo x e o cromossomo 21 do chimpanzé (BA000046.3) está no eixo y . O alinhamento ótimo está apresentado como uma linha azul e ele se inicia na posição 13,841,681 (NC_000021.7) e 1 (BA000046.3). O percentual de *matches*, *mismatches* e *gaps* são, respectivamente, 94.380%, 1.537% and 4.082%.

quando possível (Figura 12.15). O tamanho das dez maiores regiões e as suas localizações dentro do alinhamento ótimo estão apresentadas na Tabela 12.5. Estas regiões podem ser utilizadas por biólogos como regiões de interesse para análise biológicas mais detalhadas.

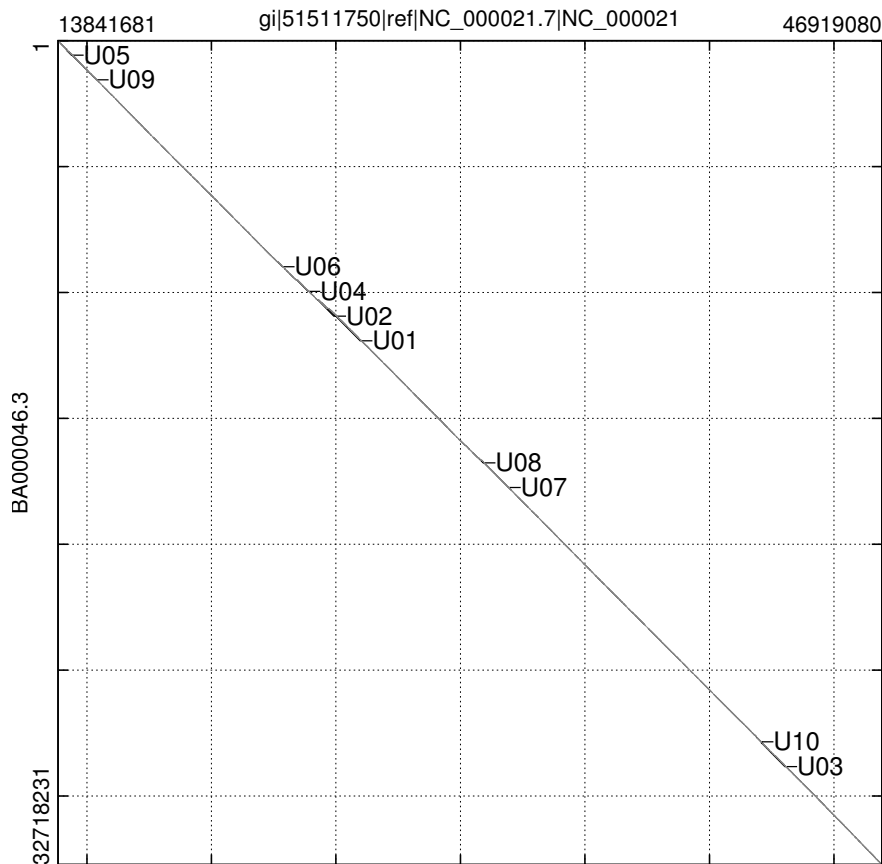


Figura 12.15: Maiores regiões *unmatched* entre os cromossomos 21 do homem e chimpanzé.

12.7 Conclusão do Capítulo

A arquitetura MASA (*Multi-platform Architecture for Sequence Aligners*) permitiu a portabilidade do CUDAlign para diferentes arquiteturas de *hardware* e *software*. Para avaliar a capacidade de portabilidade, quatro extensões foram criadas para diferentes plataformas de *hardware* e ambientes de *software*: MASA-CUDAlign, MASA-OmpSs/CPU, MASA-OpenMP/CPU and MASA-OpenMP/Phi. As três últimas extensões foram portadas com a criação de menos de 200 linhas de código, fornecendo um indicativo sobre o esforço de se portar o CUDAlign para outras plataformas. Além das extensões apresentadas nesta tese, a extensão MASA-OpenCL foi desenvolvida ao longo do Mestrado do aluno Marco Antônio C. de Figueiredo Jr. [41], do mesmo grupo de trabalho do autor desta tese. Neste trabalho, foi possível obter um desempenho de 179,2 GCUPS em uma placa AMD Radeon R9 280X [42]. Outro projeto também está iniciando para a criação de uma extensão para FPGA. Por meio da arquitetura MASA, pretende-se criar uma infraestrutura comum para alinhamento de sequências, permitindo agregar várias implementações em plataformas distintas para acelerar ainda mais o desempenho de comparações de sequências longas. O

Tabela 12.5: Tamanhos e localização das maiores regiões *unmatched*

| Nome | Tamanho | Loc. S_0 | Loc. S_1 |
|------|---------|-------------------|-------------------|
| U01 | 59072 | 10630194-10631376 | 24654791-24713863 |
| U02 | 56421 | 10280845-10288485 | 24240387-24296799 |
| U03 | 55857 | 27853079-27908936 | 42082089-42129406 |
| U04 | 46315 | 9515853-9518073 | 23422491-23468803 |
| U05 | 39683 | 337151-376789 | 14172065-14193379 |
| U06 | 26872 | 8777783-8777920 | 22640592-22667464 |
| U07 | 25757 | 16668297-16668303 | 30834341-30860098 |
| U08 | 24390 | 16569464-16569473 | 30703011-30727401 |
| U09 | 24251 | 408670-408670 | 14224960-14249211 |
| U10 | 18456 | 27632628-27651084 | 41875088-41879528 |

principal benefício da arquitetura MASA é que a grande maioria das funcionalidades foram criadas de maneira portátil a todas as extensões, permitindo que essas funcionalidades sejam aplicadas a diversas plataformas com pouco ou nenhum esforço adicional.

Parte III
Conclusão

Capítulo 13

Conclusões e Trabalhos Futuros

Nesta tese de doutorado, foram propostos e avaliados métodos para permitir o alinhamento ótimo de duas sequências longas de DNA utilizando plataformas de alto desempenho. A principal plataforma investigada foi a Unidade de Processamento Gráfico (GPU) da NVIDIA, o que resultou no desenvolvimento da ferramenta CUDAlign. A primeira versão desta ferramenta (CUDAlign 1.0 [122]) foi produzida durante o mestrado do autor desta tese, sendo que esta primeira versão era capaz de utilizar apenas uma única GPU e somente informava o escore ótimo local.

13.1 Resultados Obtidos

No escopo desta tese, propusemos estratégias paralelas e otimizações que permitiram que as funcionalidades do CUDAlign 1.0 fossem expandidas, criando as versões 2.0 [34], 2.1 [35], 3.0 [36][37] e 4.0 [38], o que permitiu que múltiplas GPUs fossem utilizadas em conjunto para produzir escores e alinhamentos ótimos. A Tabela 13.1 apresenta um resumo com o desempenho máximo obtido nos testes de cada uma das versões.

Tabela 13.1: Desempenho das versões do CUDAlign propostas nesta tese

| Versão | Saída | Desempenho | Ambiente | Tam. Máx. |
|--------------|-------------|-------------|-------------|-----------|
| CUDAlign 2.0 | Alinhamento | 23,1 GCUPS | 1 × GTX 285 | 47 MBP |
| CUDAlign 2.1 | Alinhamento | 50,7 GCUPS | 1 × GTX 560 | 59 MBP |
| CUDAlign 3.0 | Escore | 1,73 TCUPS | 64 × M2090 | 249 MBP |
| CUDAlign 4.0 | Alinhamento | 10,37 TCUPS | 384 × M2090 | 249 MBP |

No CUDAlign 2.0 [34], o alinhamento completo pode ser obtido com uma única GPU utilizando 6 estágios. O primeiro estágio foi feito com base no CUDAlign 1.0, com a diferença de que algumas linhas especiais são salvas em disco. Os estágios 2 e 3 processam a matriz em sentidos alternados, de forma a encontrar pontos (*crosspoints*) onde o alinhamento ótimo cruza as linhas especiais. O estágio 4 executa o algoritmo de Myers-Miller para reduzir o espaço entre os *crosspoints*, até que o tamanho das partições formadas por estes pontos seja suficientemente pequeno. O estágio 5 alinha cada uma dessas pequenas partições e o estágio 6 permite a visualização do alinhamento completo. Os resultados experimentais do CUDAlign 2.0 foram obtidos em uma placa NVIDIA GeForce GTX 285,

onde foi possível alinhar os cromossomos 21 do homem e do chimpanzé (47 MBP \times 33 MBP) em 18 horas e 30 minutos, com um desempenho maior que 23 milhões de células processadas por segundo (GCUPS).

No CUDAlign 2.1 [35], a otimização *Block Pruning* (BP) foi proposta, implementada e avaliada no escopo do estágio 1, com uma única GPU. Esta otimização identifica blocos da matriz de programação dinâmica que jamais contribuirão para o alinhamento ótimo. Desta forma, estes blocos são descartados e o tempo de execução é reduzido. Para sequências muito parecidas, foi mostrado que o tempo de execução é reduzido em mais de 50%. Utilizando uma placa NVIDIA GeForce GTX 560 Ti, os cromossomos 21 do homem e do chimpanzé (47 MBP \times 33 MBP) foram alinhados em 8 horas e 26 minutos, com um desempenho maior que 50 GCUPS. A otimização BP permitiu uma redução de 51% no tempo de comparação desta sequência.

Nesta tese, um estudo mais detalhado do *Block Pruning* permitiu mensurar, por meio de simulações e fórmulas, a eficácia desta otimização considerando diferentes fatores. Por exemplo, quanto maior a semelhança das sequências, mais notável é a eficácia de *pruning*. Além disso, a ordem de processamento da matriz e o ângulo do *wavefront* também influenciam na eficácia, sendo que o processamento por quadrados é o mais efetivo, enquanto o processamento por quadrados invertidos apresenta o menor desempenho do *Block Pruning*. Os parâmetros das equações de recorrência também influenciam esta otimização, sendo que observamos uma maior eficácia de *pruning* quando aumentamos os valores absolutos dos *gap*. Além disso, várias propriedades da matriz foram formalizadas nesta tese, permitindo restringir os valores possíveis de uma célula da matriz dado o valor de uma célula de referência.

O CUDAlign 3.0 [36] permitiu que várias GPUs fossem utilizadas em conjunto para obter o escore ótimo. Neste contexto, os resultados experimentais foram obtidos com o *cluster* Minotauro, hospedado no Barcelona Supercomputing Center (BSC). Executando o CUDAlign 3.0 neste *cluster*, os cromossomos 1 do homem e do chimpanzé (249 MBP \times 228 MBP) foram comparados pela primeira vez na história com um método exato, pelo que temos conhecimento. Esta comparação foi executada em 9 horas e 9 minutos utilizando 64 GPUs homogêneas e dedicadas, atingindo 1726 GCUPS. Devido a previsibilidade de execução deste algoritmo no Minotauro, foi possível estimar o tempo de execução e o *speedup* das execuções com um erro menor que 0,45% e 5%, respectivamente. Outro trabalho desenvolvido ao longo desta tese apresentou resultados em ambientes heterogêneos [37], com desempenho de 140 GCUPS com 3 GPUs. Em ambiente simulado [39], também foram desenvolvidas técnicas de balanceamento dinâmico de carga que permitem a redistribuição de carga caso as GPUs apresentem desempenho variável ao longo da execução.

O CUDAlign 4.0 permitiu que múltiplas GPUs fossem utilizadas para obter o alinhamento completo. Visto que a fase de *traceback* é eminentemente sequencial, desenvolvemos um método especulativo chamado de *Incremental Speculative Traceback* (IST). Neste método, o *traceback* é especulado enquanto as GPUs estão ociosas após o término do primeiro estágio. Os resultados experimentais obtidos com o cluster Keeneland Full Scale (KFS) mostraram que o IST foi capaz de reduzir em até $21\times$ o tempo de execução do *traceback*. Com este método, foi possível, pela primeira vez, obter o alinhamento local ótimo de todos os pares de cromossomos homólogos do homem e do chimpanzé. Utilizando 384 GPUs, o alinhamento dos cromossomos 5 foi obtido em 53 minutos, com um desempenho de 10,37

Tabela 13.2: Comparação do CUDAlign 4.0 com outras abordagens

| Solução | Saída | GCUPS | Ambiente | Tam. Máx. |
|---------------------|---------------|---------------|--------------------|--------------------|
| CUDAlign 4.0 | Alinh. | 10.370 | 384 × M2090 | 249.250.621 |
| SW-Rivyera | Escore | 3.040 | 128 × FPGA | 100 |
| Parallel-LTDP | Alinh. | 900 | 16 × Intel Xeon | 800 |
| CUDASW++3.0 | Escore | 196 | GTX690+i7 | 35.213 |
| SWAPHI-LS | Escore | 114 | 4 × Xeon Phi | 50.000.000 |

trilhões de células atualizadas por segundo (TCUPS).

Comparando o CUDAlign 4.0 com as melhores abordagens da literatura que temos conhecimento em cada plataforma (Tabela 13.2), podemos observar que esta versão evoluiu o estado da arte em três aspectos. Em termos de desempenho, o CUDAlign 4.0 obteve três vezes mais GCUPS que a solução SW-Rivyera (Seção 4.4.4). No aspecto dos dados de entrada, foi possível comparar sequências com mais de 249 milhões de bases, aumentando em quase cinco vezes o tamanho da maior sequência comparada com o SWAPHI-LS (Seção 4.4.6). Em relação ao número de GPUs utilizadas simultaneamente, não temos conhecimento de nenhuma abordagem que tenha utilizado mais de 10 GPUs. O CUDAlign 4.0 foi testado com até 384 GPUs, aumentando expressivamente o poder computacional.

Durante a evolução das versões, foi observado que várias das otimizações do CUDAlign poderiam ser empregadas em outras plataformas, em especial a otimização *Block Pruning* (BP). Por causa disso, um esforço foi realizado para criar uma arquitetura flexível que pudesse ser utilizada em outras plataformas. Foi então criada a arquitetura MASA (*Multi-Platform Architecture for Sequence Aligners*). Esta plataforma permitiu que o CUDAlign e o *Block Pruning* fossem portados para o OpenMP (para CPU e para o Intel Phi) e para o OmpSs, este último permitindo a execução com o modelo de *dataflow*. Com o OmpSs, a eficácia do *Block Pruning* foi sensivelmente incrementado por meio de uma especificação correta de prioridade nos blocos, de forma a considerar as propriedades teóricas do *Block Pruning* estudadas nesta tese. Um outro trabalho foi desenvolvido ao longo do Mestrado do aluno Marco Antônio C. de Figueiredo Jr. [41], do mesmo grupo de trabalho do autor desta tese. No trabalho deste aluno, o CUDAlign foi portado para o OpenCL com resultados expressivos em uma única GPU, chegando a 179,2 GCUPS com uma placa AMD Radeon R9 280X [42].

Analisando todos os resultados obtidos durante o desenvolvimento desta tese, podemos concluir que o CUDAlign mostrou-se uma ferramenta eficiente, capaz de comparar e alinhar sequências com mais de 200 milhões de bases. Todos os objetivos específicos desta tese (Seção 1.3) foram atingidos, sendo que a principal contribuição foi evoluir o estado da arte para que os alinhamentos ótimos sejam obtidos em tempo razoável, atividade que até então era considerada inviável. Em especial, todos os cromossomos humanos foram comparados com seus cromossomos homólogos do genoma do chimpanzé, permitindo um potencial avanço na área de comparação genômica, disponível para ser investigado por biólogos. O código fonte foi disponibilizado para este fim e encontra-se em <https://github.com/edanssandres/MASA-CUDAlign> (Anexo I).

13.2 Trabalhos Futuros

Nesta tese, o *Block Pruning* (BP) foi desenvolvido com um escopo voltado para alinhamentos locais e em um único nó de processamento. Para trabalhos futuros, pretendemos formalizar e integrar o BP em ambientes com múltiplos nós e para alinhamentos globais e semi-globais, conforme descrito no Capítulo 8. Planejamos também estender a análise teórica do BP para alinhamentos deslocados da diagonal principal, pois esse cenário foi observado em comparações reais.

Nesta tese, o *Block Pruning* inicia o processamento considerando que o melhor escore conhecido é zero (pior caso). Se utilizarmos uma heurística rápida que dê uma estimativa do alinhamento, podemos utilizar esse escore obtido como valor inicial do BP. Isso aumentaria ainda mais a eficácia do algoritmo para sequências similares. Por fim, outro trabalho futuro seria aplicar o BP durante a comparação de uma sequência de busca com um banco de sequências. Neste cenário, o maior escore encontrado entre os pares de sequências já processados poderia ser utilizado como valor inicial nos pares de sequência seguintes. Desta forma, comparações com escore baixo poderiam ser mais rapidamente descartadas.

Em relação ao MASA, pretendemos implementar novas funcionalidades e novas extensões. Por exemplo, almejamos criar extensões MASA que utilizem instruções SSE, utilizando as ideias desenvolvidas por Farrar [110] em CPU e Intel Phi. Ainda mais, pretendemos criar extensões MASA otimizadas para FPGA. Utilizando as extensões para diferentes plataformas, podemos executar testes envolvendo diversos tipos de ambientes heterogêneos e híbridos. Para permitir o uso de múltiplos *clusters* distribuídos, utilizando links de menor capacidade, seria interessante aplicar compressão de dados durante a comunicação via *socket* entre nós de processamento. Essa mesma compressão também poderia ser aplicada ao salvar as linhas especiais em disco ou memória.

A respeito da divisão dos estágios no MASA-CUDAlign, existem atualmente parâmetros que regulam o tamanho ideal das partições em cada estágio e o número de linhas/colunas especiais que precisam ser salvas. Pretendemos fazer um estudo desses parâmetros para determinar, automaticamente, os valores que produzam o melhor desempenho. Além disso, o estágio 4 (Myers-Miller) pode ser otimizado considerando que a maioria das partições apresenta um alinhamento com um escore alto em sua diagonal principal, permitindo que otimizações como Fickett (Seção 2.2.7) sejam aplicadas com o Myers-Miller. Por fim, o estágio 5 também poderia ser otimizado, aproveitando o fato de que as partições possuem um tamanho pequeno e restrito. Com isso, poderíamos precomputar os resultados com algoritmos similares ao dos quatro russos [152].

Para uma melhor estabilidade na execução com múltiplos nós, implementaremos as técnicas propostas nesta tese para balanceamento dinâmico de carga e avaliaremos seu comportamento em ambientes reais. Um outro trabalho futuro seria desenvolver mecanismos que permitam ingresso e saída de nós ao longo da execução. Com esse mecanismo, poderíamos criar um grid colaborativo para a tarefa de alinhamento de sequências. A disponibilização de um *web server* na internet para execução de alinhamentos ótimos também será considerada em trabalhos futuros.

Embora o CUDAlign tenha permitido que todo o genoma humano fosse comparado com o do chimpanzé utilizando algoritmos exatos, necessitamos de um estudo especializado feito por biólogos para interpretar os resultados. Sendo assim, pretendemos que

trabalhos futuros incluam a investigação biológica dos alinhamentos ótimos obtidos pelo CUDAlign 4.0.

Referências

- [1] M. Schatz and B. Langmead. The DNA data deluge. *Spectrum, IEEE*, 50(7):28–33, 2013. 1
- [2] W. R. Pearson. Training for bioinformatics and computational biology. *Bioinformatics*, 17(9):761–762, 2001. 1
- [3] N. M. Luscombe, D. Greenbaum, M. Gerstein, et al. What is bioinformatics? a proposed definition and overview of the field. *Methods of information in medicine*, 40(4):346–358, 2001. 1
- [4] D. M. Mount. *Bioinformatics - sequence and genome analysis (2. ed.)*. Cold Spring Harbor Laboratory Press, 2004. 1, 8, 9
- [5] R. Durbin, S. Eddy, A. Krogh, and G. Mitchison. *Biological sequence analysis*. Cambridge University Press, 2002. 1, 9, 171
- [6] D. Gusfield. *Algorithms on strings, trees, and sequences: computer science and computational biology*. Cambridge University Press, 1997. 1, 9
- [7] S. Aluru. *Handbook of computational molecular biology*. CRC Press, 2005. 1
- [8] S. B. Needleman and C. D. Wunsch. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *Journal of Molecular Biology*, 48(3):443–453, 1970. 2, 17
- [9] T. F. Smith and M. S. Waterman. Identification of common molecular subsequences. *Journal of Molecular Biology*, 147(1):195–197, 1981. 2, 18
- [10] O. Gotoh. An improved algorithm for matching biological sequences. *Journal of Molecular Biology*, 162(3):705–708, 1982. 2, 20
- [11] E. W. Myers and W. Miller. Optimal alignments in linear space. *Computer applications in the Biosciences*, 4(1):11–17, 1988. 2, 3, 22
- [12] D. S. Hirschberg. A linear space algorithm for computing maximal common subsequences. *Communications of the ACM*, 18(6):341–343, 1975. 2, 3, 21, 80
- [13] W. R. Pearson and D. J. Lipman. Improved tools for biological sequence comparison. *Proceedings of the National Academy of Sciences*, 85(8):2444–2448, 1988. 2, 8
- [14] S. F. Altschul, W. Gish, W. Miller, E. W. Myers, and D. J. Lipman. Basic local alignment search tool. *Journal of Molecular Biology*, 215(3):403–410, 1990. 2, 8, 24

- [15] K. Compton and S. Hauck. Reconfigurable computing: A survey of systems and software. *ACM Computing Surveys*, 34(2):171–210, 2002. 3
- [16] T. Chen, R. Raghavan, J. N. Dale, and E. Iwata. Cell Broadband Engine Architecture and Its First Implementation: A Performance View. *IBM Journal of Research and Development*, 51(5):559–572, 2007. 3
- [17] D. B. Kirk and W.-m. W. Hwu. *Programming Massively Parallel Processors: A Hands-on Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2010. 3
- [18] J. Jeffers and J. Reinders. *Intel Xeon Phi Coprocessor High Performance Programming*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2013. 3
- [19] J. Owens, M. Houston, D. Luebke, S. Green, J. Stone, and J. Phillips. GPU computing. *Proceedings of the IEEE*, 96(5):879–899, 2008. 3
- [20] General-Purpose computation on Graphics Processing Units. <http://gpgpu.org/>, 2011. 3, 27
- [21] R. J. Lipton and D. Lopresti. A systolic array for rapid string comparison. In *Proceedings of the Chapel Hill Conference on Very Large Scale Integration*, pages 363–376, 1985. 3
- [22] S. U. C. S. D. K. S. Laboratory, A. Galper, and D. Brutlag. *Parallel Similarity Search and Alignment with the Dynamic Programming Method*. Report (Stanford University. Medical Computer Science. Knowledge Systems Laboratory). Knowledge Systems Laboratory, Medical Computer Science, Stanford University, 1990. 3
- [23] S. Rajko and S. Aluru. Space and time optimal parallel sequence alignments. *IEEE Transactions on Parallel and Distributed Systems*, 15(12):1070–1081, 2004. 3, 38
- [24] C. Chen and B. Schmidt. Computing large-scale alignments on a multi-cluster. *IEEE International Conference on Cluster Computing*, page 38, 2003. 3
- [25] R. B. Batista, A. Boukerche, and A. C. M. A. Melo. A parallel strategy for biological sequence alignment in restricted memory space. *Journal of Parallel and Distributed Computing*, 68(4):548–561, 2008. 3
- [26] A. Wozniak. Using video-oriented instructions to speed up sequence comparison. *Computer Applications in the Biosciences*, 13(2):145–150, 1997. 3
- [27] T. Rognes. Faster smith-waterman database searches with inter-sequence simd parallelisation. *BMC Bioinformatics*, 12(1):221, 2011. 3, 40
- [28] S. Manavski and G. Valle. CUDA compatible GPU cards as efficient hardware accelerators for Smith-Waterman sequence alignment. *BMC Bioinformatics*, 9(Suppl 2), 2008. 3, 27, 42, 43

- [29] Y. Liu, B. Schmidt, and D. Maskell. CUDASW++2.0: enhanced Smith-Waterman protein database search on CUDA-enabled GPUs based on SIMT and virtualized simd abstractions. *BMC Research Notes*, 3(1):93, 2010. 3, 44
- [30] S. Sarkar, G. Kulkarni, P. Pande, and A. Kalyanaraman. Network-on-chip hardware accelerators for biological sequence alignment. *IEEE Transactions on Computers*, 59(1):29–41, 2010. 3, 47
- [31] W. Zhang, P. Ciclitira, and J. Messing. Pacbio sequencing of gene families — a case study with wheat gluten genes. *Gene*, 533(2):541 – 546, 2014. 4
- [32] NVIDIA. *NVIDIA CUDA Architecture*. NVIDIA, 2010. 4, 33
- [33] E. F. O. Sandes. Comparação Paralela de Sequências Biológicas Longas utilizando Unidades de Processamento Gráfico (GPUs). Master’s thesis, Universidade de Brasília, Brasília, Brasil, 2011. 5, 45, 50, 56, 57
- [34] E. F. O. Sandes and A. C. M. A. Melo. Smith-Waterman alignment of huge sequences with GPU in linear space. In *IEEE International Parallel Distributed Processing Symposium*, pages 1199–1211, 2011. 5, 6, 59, 182, 201
- [35] E. F. O. Sandes and A. C. M. A. Melo. Retrieving smith-waterman alignments with optimizations for megabase biological sequences using gpu. *IEEE Transactions on Parallel and Distributed Systems*, 24(5):1009–1021, 2013. 5, 6, 46, 70, 80, 182, 183, 201
- [36] E. F. O. Sandes, G. Miranda, A. C. M. A. Melo, X. Martorell, and E. Ayguade. CUDAlign 3.0: Parallel Biological Sequence Comparison in Large GPU Clusters. In *IEEE/ACM Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, pages 160–169, 2014. 5, 6, 118, 160, 182, 183, 201
- [37] E. F. O. Sandes, G. Miranda, , A. C. M. A. Melo, X. Martorell, and E. Ayguadé. Fine-grain parallel megabase sequence comparison with multiple heterogeneous GPUs. In *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPOPP ’14, pages 383–384 (short paper), 2014. 5, 6, 160, 182, 183, 202
- [38] E. F. O. Sandes, G. Miranda, X. Martorell, E. Ayguadé, G. Teodoro, and A. C. M. A. Melo. Genome Wide Alignment in GPU Cluster with Incremental Speculative Traceback. *Submitted to IEEE Transactions on Parallel and Distributed Systems*, 2015. 5, 6, 182, 202
- [39] E. F. O. Sandes, C. G. Ralha, and A. C. M. A. Melo. An agent-based solution for dynamic multi-node wavefront balancing in biological sequence comparison. *Expert Systems with Applications*, 41(10):4929 – 4938, 2014. 5, 6, 146, 183, 201
- [40] E. F. O. Sandes, G. Miranda, X. Martorell, E. Ayguadé, G. Teodoro, and A. C. M. A. Melo. MASA: a multi-platform architecture for sequence aligners with block pruning. *ACM Transactions on Parallel Computing (accepted)*, 2015. 6, 201

- [41] M. A. C. de Figueiredo Jr. MASA-OpenCL: Comparação Paralela de Sequências Biológicas Longas em GPU. Master's thesis, Universidade de Brasília, Brasília, Brasil, 2015. 6, 179, 184
- [42] M. A. C. de Figueiredo Jr., E. F. O. Sandes, and A. C. M. A. Melo. Parallel megabase dna sequence comparison with opencl. In *22st International Conference on High Performance Computing, HiPC 2015, Bangalore, India, December 16-19, 2015 (accepted)*, pages 1–10, 2015. 6, 179, 184
- [43] E. F. O. Sandes, A. Boukerche, and A. C. M. A. Melo. Parallel Exact Pairwise Biological Sequence Comparison: Algorithms, Platforms and Classification. *Submitted to ACM Computing Surveys*, 2015. 6, 36, 202
- [44] A. L. Delcher, S. Kasif, R. D. Fleischmann, J. Peterson, O. White, and S. L. Salzberg. Alignment of whole genomes. *Nucleic acids research*, 27(11):2369–2376, 1999. 8
- [45] A. L. Delcher, A. Phillippy, J. Carlton, and S. L. Salzberg. Fast algorithms for large-scale genome alignment and comparison. *Nucleic acids research*, 30:2478–2483, 2002. 8
- [46] S. Kurtz, A. Phillippy, A. Delcher, M. Smoot, M. Shumway, C. Antonescu, and S. Salzberg. Versatile and open software for comparing large genomes. *Genome Biology*, 5(2):R12+, 2004. 8
- [47] B. Chevreux. *MIRA: an Automated Genome and EST Assembler*. PhD thesis, University of Heidelberg, Heidelberg, Germany, 2005. 9
- [48] A. Cornish-Bowden. Nomenclature for incompletely specified bases in nucleic acid sequences: recommendations 1984. *Nucleic acids research*, 13(9):3021–3030, 1985. 10
- [49] S. Henikoff and J. G. Henikoff. Amino acid substitution matrices from protein blocks. *Proceedings of the National Academy of Sciences of the United States of America*, 89(22):10915–10919, 1992. 11
- [50] Y. Liu and B. Schmidt. GSWABE: Faster GPU-accelerated sequence alignment with optimal alignment retrieval for short dna sequences. *Concurrency and Computation: Practice and Experience*, 27(4):958–972, 2015. 13, 46
- [51] Y. Choi. A fast computation of pairwise sequence alignment scores between a protein and a set of single-locus variants of another protein. In *Proceedings of the ACM Conference on Bioinformatics, Computational Biology and Biomedicine, BCB '12*, pages 414–417. ACM, 2012. 13
- [52] Y. Chen, B. Schmidt, and D. Maksell. An FPGA aligner for short read mapping. In *Field Programmable Logic and Applications (FPL), 2012 22nd International Conference on*, pages 511–514, 2012. 13
- [53] E. V. Denardo. *Dynamic Programming: Models and Applications*. Denardo, 2003. 17

- [54] M. Waterman, T. Smith, and W. Beyer. Some biological sequence metrics. *Advances in Mathematics*, 20(3):367–387, 1976. 20
- [55] A. Driga, P. Lu, J. Schaeffer, D. Szafron, K. Charter, and I. Parsons. FastLSA: A Fast, Linear-Space, Parallel and Sequential Algorithm for Sequence Alignment. *Algorithmica*, 45(3):337–375, 2006. 22
- [56] J. W. Fickett. Fast optimal alignment. *Nucleic Acids Research*, 12(1):175–179, 1984. 23
- [57] K. Chao, W. R. Pearson, and W. Miller. Aligning two sequences within a specified diagonal band. *Computer Applications in the Biosciences*, 8(5):481–487, 1992. 25
- [58] A. Lefohn, M. Houston, J. Andersson, U. Assarsson, C. Everitt, K. Fatahalian, T. Foley, J. Hensley, P. Lalonde, and D. Luebke. Beyond programmable shading (parts i and ii). In *ACM SIGGRAPH 2009 Courses*, SIGGRAPH '09, pages 1–312. ACM, 2009. 26, 27
- [59] D. Blythe. Rise of the graphics processor. *Proceedings of the IEEE*, 96(5):761–778, 2008. 26
- [60] N. England. A graphics system architecture for interactive application-specific display functions. *IEEE Computer Graphics and Applications*, 6:60–70, 1986. 26
- [61] H. Fuchs, J. Poulton, J. Eyles, T. Greer, J. Goldfeather, D. Ellsworth, S. Molnar, G. Turk, B. Tebbs, and L. Israel. Pixel-planes 5: a heterogeneous multiprocessor graphics system using processor-enhanced memories. In *Proceedings of the 16th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '89, pages 79–88. ACM, 1989. 26
- [62] B. Corrie and P. Mackerras. Data shaders. In *Proceedings of the 4th conference on Visualization '93*, VIS '93, pages 275–282. IEEE Computer Society, 1993. 26
- [63] M. McGuire, G. Stathis, H. Pfister, and S. Krishnamurthi. Abstract shade trees. In *Proceedings of the 2006 symposium on Interactive 3D graphics and games*, I3D '06, pages 79–86. ACM, 2006. 26
- [64] P. Lalonde and E. Schenk. Shader-driven compilation of rendering assets. In *Proceedings of the 29th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '02, pages 713–720. ACM, 2002. 26
- [65] R. L. Cook. Shade trees. *ACM SIGGRAPH Computer Graphics*, 18:223–231, 1984. 26
- [66] K. Perlin. An image synthesizer. *ACM SIGGRAPH Computer Graphics*, 19:287–296, 1985. 27
- [67] P. Hanrahan and J. Lawson. A language for shading and lighting calculations. *ACM SIGGRAPH Computer Graphics*, 24:289–298, 1990. 27

- [68] M. Olano and A. Lastra. A shading language on graphics hardware: the pixelflow shading system. In *Proceedings of the 25th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '98, pages 159–168. ACM, 1998. 27
- [69] P. Lalonde and E. Schenk. Shader-driven compilation of rendering assets. *ACM Transactions on Graphics*, 21:713–720, 2002. 27
- [70] S. Molnar, J. Eyles, and J. Poulton. Pixelflow: High-speed rendering using image composition. In *Computer Graphics*, pages 231–240, 1992. 27
- [71] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan. Brook for GPUs: Stream computing on graphics hardware. *ACM Transactions on Graphics*, 23:777–786, 2004. 27
- [72] M. D. McCool, Z. Qin, and T. S. Popa. Shader metaprogramming. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, HWWS '02, pages 57–68, 2002. 27
- [73] M. McCool, S. Du Toit, T. Popa, B. Chan, and K. Moule. Shader algebra. *ACM Transactions on Graphics*, 23:787–795, 2004. 27
- [74] NVIDIA. *NVIDIA CUDA Programming Guide 3.2*. NVIDIA, 2010. 27, 28, 33, 34, 35, 166
- [75] M. Flynn. Some Computer Organizations and Their Effectiveness. *IEEE Transactions on Computers*, 21(9):948–960, 1972. 27
- [76] M. Gschwind, H. P. Hofstee, B. Flachs, M. Hopkins, Y. Watanabe, and T. Yamazaki. Synergistic processing in cell's multicore architecture. *IEEE Micro*, 26(2):10–24, 2006. 27
- [77] L. Seiler, D. Carmean, E. Sprangle, T. Forsyth, M. Abrash, P. Dubey, S. Junkins, A. Lake, J. Sugerman, R. Cavin, R. Espasa, E. Grochowski, T. Juan, and P. Hanrahan. Larrabee: a many-core x86 architecture for visual computing. *ACM Transactions on Graphics*, 27(3):1–15, 2008. 27
- [78] D. S. Scott. Intel many integrated core architecture for hpc. In *Proceedings of the ATIP/A*CRC Workshop on Accelerator Technologies for High-Performance Computing: Does Asia Lead the Way?*, ATIP '12, pages 1–29, 2012. 27
- [79] S. Borkar. Thousand core chips: a technology perspective. In *Proceedings of the 44th annual Design Automation Conference*, DAC '07, pages 746–749. ACM, 2007. 27
- [80] A. Vajda. *Programming Many-Core Chips*. Springer Publishing Company, Incorporated, 1st edition, 2011. 27
- [81] Many-core processor. <http://software.intel.com/en-us/articles/many-core-processor/>, 2011. 27
- [82] K. Fatahalian and M. Houston. GPUs: A closer look. *Queue*, 6:18–28, 2008. 27

- [83] M. D. Hill and M. R. Marty. Amdahl’s law in the multicore era. *IEEE COMPUTER*, 41(7):33–38, 2008. 27
- [84] E. Hermann, B. Raffin, F. Faure, T. Gautier, and J. Allard. Multi-GPU and multi-CPU parallelization for interactive physics simulations. In *Proceedings of the 16th international Euro-Par conference on Parallel processing: Part II*, Euro-Par’10, pages 235–246. Springer-Verlag, 2010. 27
- [85] J. R. da Silva, Jr., E. W. Clua, P. A. Pagliosa, and A. Montenegro. Fluid simulation with rigid body triangle accuracy collision using an heterogeneous GPU/CPU hardware system. In *Proceedings of the 2010 ACM SIGGRAPH symposium on Interactive 3D Graphics and Games*, I3D ’10, pages 1–20. ACM, 2010. 27
- [86] C. Li, H. Wu, S. Chen, X. Li, and D. Guo. Efficient implementation for md5-rc4 encryption using GPU with CUDA. In *Proceedings of the 3rd international conference on Anti-Counterfeiting, security, and identification in communication*, ASID’09, pages 167–170. IEEE Press, 2009. 27
- [87] N. Costigan and P. Schwabe. Fast elliptic-curve cryptography on the cell broadband engine. In *Proceedings of the 2nd International Conference on Cryptology in Africa: Progress in Cryptology*, AFRICACRYPT ’09, pages 368–385. Springer-Verlag, 2009. 27
- [88] Y. Liu, W. Huang, J. Johnson, and S. Vaidya. GPU accelerated Smith-Waterman. In *Computational Science – ICCS 2006*, pages 188–195. Springer, 2006. 27, 41
- [89] W. Liu, B. Schmidt, G. Voss, A. Schroder, and W. Muller-Wittig. Bio-sequence database scanning on a GPU. *IPDPS*, page 274, 2006. 27
- [90] Y. Liu, D. Maskell, and B. Schmidt. CUDASW++: optimizing Smith-Waterman sequence database searches for CUDA-enabled graphics processing units. *BMC Research Notes*, 2(1):73, 2009. 27, 43
- [91] L. Ligowski and W. Rudnicki. An efficient implementation of Smith-Waterman algorithm on GPU using CUDA, for massively parallel scanning of sequence databases. In *IEEE International Workshop on High Performance Computational Biology (Hi-COMB 2009)*, 2009. 27, 42
- [92] NVIDIA. *NVIDIA’s Next Generation CUDA Compute Architecture: Fermi*. NVIDIA, 2009. 28
- [93] C. M. Wittenbrink, E. Kilgariff, and A. Prabhu. Fermi GF100 GPU Architecture. *IEEE Micro*, 31(2):50–59, 2011. 28, 29
- [94] NVIDIA. *NVIDIA’s Next Generation CUDA Compute Architecture: Kepler GK110*. NVIDIA, 2009. 28, 30
- [95] NVIDIA. *NVIDIA GeForce GTX 680 Whitepaper*. NVIDIA, 2012. 28
- [96] NVIDIA. *NVIDIA GeForce GTX 980 Whitepaper*. NVIDIA, 2014. 28, 30, 31

- [97] R. Smith. The NVIDIA geforce GTX 750 Ti and GTX 750, 2014. 29
- [98] Video card reviews and specifications. <http://www.gpureview.com>, 2011. 32
- [99] NVIDIA website. <http://www.nvidia.com/>, 2010. 32
- [100] J. E. Stone, D. Gohara, and G. Shi. OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems. *IEEE Des. Test*, 12(3):66–73, 2010. 32
- [101] K. Opencl and A. Munshi. The OpenCL Specification Version: 1.0 Document Revision: 48. 32
- [102] NVIDIA. *NVIDIA CUDA C Best Practices Guide*. NVIDIA, 2010. 33
- [103] A. Lashgar, A. Baniasadi, and A. Khonsari. Dynamic warp resizing: Analysis and benefits in high-performance SIMT. In *Computer Design (ICCD), 2012 IEEE 30th International Conference on*, pages 502–503, 2012. 34
- [104] D. A. Benson, I. Karsch-Mizrachi, D. J. Lipman, J. Ostell, and E. W. Sayers. Genbank. *Nucleic Acids Research*, 38:D46–51, 2010. 36
- [105] G. F. Pfister. *In search of clusters: the coming battle in lowly parallel computing*. Prentice-Hall, Inc., 1995. 37
- [106] S. Aluru, N. Futamura, and K. Mehrotra. Parallel biological sequence comparison using prefix computations. *Journal of Parallel and Distributed Computing*, 63(3):264–272, 2003. 38
- [107] T. Rognes and E. Seeberg. Six-fold speed-up of Smith-Waterman sequence database searches using parallel processing on common microprocessors. *Bioinformatics*, 16(8):699–706, 2000. 38, 39
- [108] P. Green. Swat. <http://www.phrap.org/phredphrap/swat.html>. 38
- [109] S. F. Altschul, T. L. Madden, A. A. Schäffer, J. Zhang, Z. Zhang, W. Miller, and D. J. Lipman. Gapped blast and psiblast: a new generation of protein database search programs. *Nucleic Acids Research*, 25(17):3389–3402, 1997. 39
- [110] M. Farrar. Striped Smith-Waterman speeds database searches six times over other simd implementations. *Bioinformatics*, 23(2):156–161, 2007. 39, 185
- [111] S. Maleki, M. Musuvathi, and T. Mytkowicz. Parallelizing dynamic programming through rank convergence. In *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPOPP '14, pages 219–232. ACM, 2014. 40
- [112] M. A. Hoepfner. NCBI bookshelf: books and documents in life sciences and health care. *Nucleic Acids Research*, 41(Database-Issue):1251–1260, 2013. 41
- [113] W. Liu, B. Schmidt, G. Voss, and W. Muller-Wittig. Streaming algorithms for biological sequence alignment on GPUs. *IEEE Transactions on Parallel and Distributed Systems*, 18(9):1270–1281, 2007. 41

- [114] J. D. Thompson, D. G. Higgins, and T. J. Gibson. CLUSTAL W: improving the sensitivity of progressive multiple sequence alignment through sequence weighting, position-specific gap penalties and weight matrix choice. *Nucleic Acids Research*, 22:4673–680, 1994. 41, 42
- [115] D. B. J. Kessenich and R. Rost. The OpenGL Shading Language, Document Revision 59, technical report. <http://www.opengl.org/documentation/ogls1.html>, 2005. 41
- [116] R. J. Rost. *OpenGL(R) Shading Language (2nd Edition)*. Addison-Wesley Professional, 2006. 41
- [117] W. Liu, B. Schmidt, and W. Müller-Wittig. Performance analysis of general-purpose computation on commodity graphics hardware: A case study using bioinformatics. *J. VLSI Signal Process. Syst.*, 48(3):209–221, 2007. 42
- [118] A. Khajeh-Saeed, S. Poole, and J. B. Perot. Acceleration of the Smith-Waterman algorithm using single and multiple graphics processors. *Journal of Computational Physics*, 229(11):4247–4258, 2010. 43
- [119] D. A. Bader, K. Madduri, J. R. Gilbert, V. Shah, J. Kepner, T. Meuse, and A. Krishnamurthy. Designing scalable synthetic compact applications for benchmarking high productivity computing systems. *Cyberinfrastructure Technology Watch*, 2006. 43
- [120] M. O. W. B. Doug Hains, Zach Cashero and S. Rajopadhye. Improving CUDASW++, a parallelization of Smith-Waterman for CUDA enabled devices. In *IEEE International Workshop on High Performance Computational Biology (Hi-COMB 2011)*, pages 490–501, 2011. 44
- [121] Y. Liu, A. Wirawan, and B. Schmidt. CUDASW++ 3.0: Accelerating Smith-Waterman protein database search by coupling CPU and GPU SIMD instructions. *BMC Bioinformatics*, 14:117, 2013. 44
- [122] E. F. O. Sandes and A. C. M. A. Melo. CUDAlign: using GPU to accelerate the comparison of megabase genomic sequences. In *15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP*, pages 137–146. ACM, 2010. 45, 50, 51, 52, 53, 54, 55, 57, 65, 182
- [123] F. Ino, Y. Kotani, and K. Hagihara. Harnessing the power of idle gpus for acceleration of biological sequence alignment. In *IEEE International Symposium on Parallel Distributed Processing.*, pages 1–8, 2009. 46
- [124] F. Ino, Y. Munekawa, and K. Hagihara. Sequence homology search using fine grained cycle sharing of idle gpus. *IEEE Transactions on Parallel and Distributed Systems*, 23(4):751–759, 2012. 46
- [125] M. Korpar and M. Sikic. SW#-GPU-enabled Exact Alignments on Genome Scale. *Bioinformatics*, 29:2494–2495, 2013. 46, 80
- [126] A. Wirawan, C. K. Kwoh, T. H. Nim, and B. Schmidt. CBESW: Sequence Alignment on the Playstation 3. *BMC Bioinformatics*, 9:377, 2008. 46

- [127] A. Sarje and S. Aluru. Parallel genomic alignments on the cell broadband engine. *IEEE Transactions on Parallel and Distributed Systems*, 20(11):1600–1610, 2009. 47
- [128] L. Wienbrandt. Bioinformatics Applications on the FPGA-Based High-Performance Computer RIVYERA. In W. Vanderbauwhede and K. Benkrid, editors, *High-Performance Computing Using FPGAs*, pages 81–103. Springer, 2013. 47
- [129] H. T. Kung. Why Systolic Architectures? *Computer*, 15(1):37–46, 1982. 47
- [130] L. Wang, Y. Chan, X. Duan, H. Lan, X. Meng, and W. Liu. Xsw: Accelerating biological database search on xeon phi. In *Parallel Distributed Processing Symposium Workshops (IPDPSW), 2014 IEEE International*, pages 950–957, 2014. 48
- [131] Y. Liu and B. Schmidt. Swaphi: Smith-waterman protein database search on xeon phi coprocessors. In *Application-specific Systems, Architectures and Processors (ASAP), 2014. 25th IEEE International Conference on*, pages 184–185, 2014. 48, 168
- [132] A. Boukerche, A. C. M. A. Melo, E. Flavius, and M. Ayala-Rincon. An exact parallel algorithm to compare very long biological sequences in clusters of workstations. *Cluster Computing*, 10(2):187–202, 2007. 70, 80
- [133] P. Hart, N. Nilsson, and B. Raphael. A formal basis for the heuristic determination of minimum cost paths. *Systems Science and Cybernetics, IEEE Transactions on*, 4(2):100–107, 1968. 80
- [134] R. E. Korf. Depth-first iterative-deepening: An optimal admissible tree search. *Artif. Intell.*, 27(1):97–109, 1985. 80
- [135] R. E. Korf. A divide and conquer bidirectional search: First results. In *Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence, IJCAI 99*, pages 1184–1191, 1999. 80
- [136] R. E. Korf. Divide-and-conquer frontier search applied to optimal sequence alignment. In *National Conference on Artificial Intelligence*, pages 910–916, 2000. 80
- [137] T. Yoshizumi, T. Miura, and T. Ishida. A* with partial expansion for large branching factor problems. In *Proceedings of the Seventeenth National Conference on Artificial Intelligence and Twelfth Conference on Innovative Applications of Artificial Intelligence*, pages 923–929. AAAI Press, 2000. 80
- [138] A. Davidson. A fast pruning algorithm for optimal sequence alignment. In *Bioinformatics and Bioengineering Conference*, pages 49–56, 2001. 80
- [139] Maxima, a Computer Algebra System. Version 5.34.1. <http://maxima.sourceforge.net/>, 2014. 106
- [140] S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach (3rd Edition)*. Prentice Hall, 2009. 146, 147

- [141] M. E. Bratman. *Intention, Plans, and Practical Reason*. CSLI Publications, 1999. 148
- [142] F. L. Bellifemine, G. Caire, and D. Greenwood. *Developing Multi-Agent Systems with JADE*. John Wiley & Sons, 2007. 154
- [143] L. Dagum and R. Menon. OpenMP: an industry standard API for shared-memory programming. *Computational Science & Engineering, IEEE*, 5(1):46–55, 1998. 166
- [144] J. Bueno, L. Martinell, A. Duran, M. Farreras, X. Martorell, R. M. Badia, E. Ayguade, and J. Labarta. Productive cluster programming with ompss. In *Proceedings of the 17th International Conference on Parallel Processing - Volume Part I, Euro-Par'11*, pages 555–566. Springer-Verlag, 2011. 166
- [145] A. Duran, E. Ayguade, R. M. Badia, J. Labarta, L. Martinell, X. Martorell, and J. Planas. OmpSs: a Proposal for Programming Heterogeneous Multicore Architectures. *Parallel Processing Letters*, 21(2):173–193, 2011. 166, 168
- [146] X. Tian, H. Saito, S. Preis, E. Garcia, S. Kozhukhov, M. Masten, A. Cherkasov, and N. Panchenko. Practical simd vectorization techniques for intel xeon phi coprocessors. In *Parallel and Distributed Processing Symposium Workshops PhD Forum (IPDPSW), 2013 IEEE 27th International*, pages 1149–1158, 2013. 167
- [147] B. S. Center. Throttling policies. <http://pm.bsc.es/ompss-docs/user-guide/run-programs-plugin-throttle.html>, 2013. 169
- [148] M. Verma et al. Whole genome sequence of the rifamycin B-producing strain amycolatopsis mediterranei S699. *Journal of Bacteriology*, 193(19):5562–5563, 2011. 177
- [149] W. Zhao et al. Complete genome sequence of the rifamycin SV-producing amycolatopsis mediterranei U32 revealed its genetic characteristics in phylogeny and metabolism. *Cell Research*, 10:1096–1108, 2010. 177
- [150] M. Scarpato, R. Esposito, D. Evangelista, M. Aprile, M. R. Ambrosio, A. C. C. Angelini, and V. Costa. Analysis of Expression on Human Chromosome 21, ALE-HSA21: a Pilot Integrated Web Resource. *Database - Journal of Biological Databases and Curation*, 2014, 2014. 177
- [151] A. Letourneau, F. A. Santoni, X. Bonilla, M. R. Sailani, D. Gonzalez, J. Kind, and C. Chevalier. Domains of genome-wide gene expression dysregulation in down syndrome. *Nature*, 508:345–350, 2014. 177
- [152] V. L. Arlazarov, E. A. Dinic, M. A. Kronrod, and I. A. Faradžev. On economical construction of the transitive closure of a directed graph. *Soviet Mathematics—Doklady*, 11(5):1209–1210, 1970. 185

Anexo I

Repositórios com o código fonte do MASA

MASA-Core: <https://github.com/edanssandres/MASA-Core.git>
MASA-CUDAlign: <https://github.com/edanssandres/MASA-CUDAlign.git>
MASA-OmpSs: <https://github.com/edanssandres/MASA-OmpSs.git>
MASA-OpenMp: <https://github.com/edanssandres/MASA-OpenMP.git>
MASA-Serial: <https://github.com/edanssandres/MASA-Serial.git>

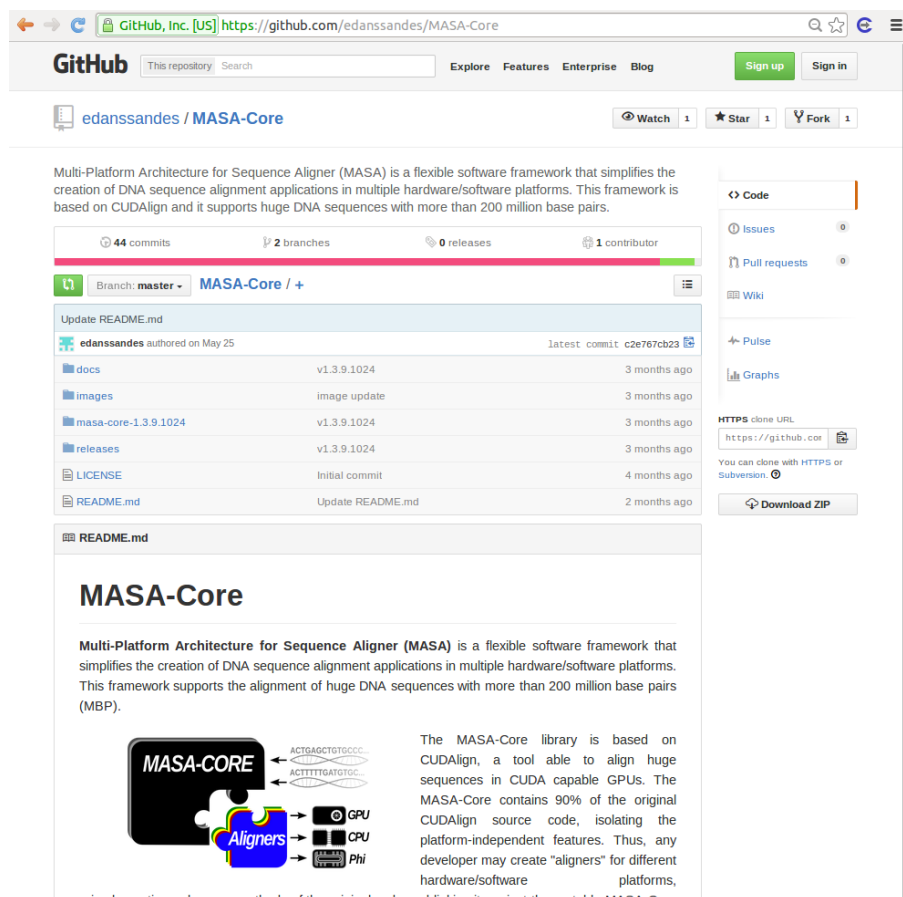


Figura I.1: Captura de tela da página do repositório do MASA-Core (GitHub)

Anexo II

Script de simulação do *Block Pruning* (Gnuplot)

```
#####
# Autor: Edans Sandes
#
# Execução: gnuplot -e "op1=3;op2=4;op3=1;ang=30;psi=90;" sim.gnuplot
# Parâmetros: (*default)
# op1: 1* Perfect Match/PM_r
#       2 Perfect Match/PM^1
#       3 Semi-Perfect Match/PM^p
# op2: 1* Row-by-Row      4 Angle
#       2 Col-by-Col      5 Square
#       3 Diagonal        6 Anti-Square
#
# ang: 30* angle          psi: 90* similarity(0%-100%)
# g: 3* gap              m: 1000* matrix width
# ma: 1* match           n: 1000* matrix height
# mi: 3* mismatch
#
#####

if (!exists("m")) m=1000
if (!exists("n")) n=1000

set terminal gif enhanced font "arial,10" size n,m
set output 'out.gif'

if (!exists("borders")) borders=0.02

set border 4095 front linecolor -1 linewidth 2
if (borders==0) unset border
set lmargin at screen 0+borders*m/n;
set rmargin at screen 1-borders*m/n;
set bmargin at screen 0+borders;
set tmargin at screen 1-borders;

set view map
set isosamples m*1, m*1
set samples m*1,m*1
unset surface
```

```

set style data pm3d
set style function pm3d
set ticslevel 0
set tics scale 4
set xtics 100 format ""
set ytics 100 format ""
if (borders==0) unset xtics
if (borders==0) unset ytics

set title ""
set xlabel ""
set xrange [ 1 : n ] noreverse nowriteback
set ylabel ""
set yrange [ 1 : m ] reverse nowriteback
set cbrange [0:1]
set pm3d implicit at b
set pm3d corners2color c3
set palette negative nops_allcF maxcolors 0 gamma 1.5 gray
unset colorbox

min(a,b)=(a<b)?a:b
max(a,b)=(a>b)?a:b

if (!exists("op1")) op1=1
if (!exists("op2")) op2=1
if (!exists("ang")) ang=30
if (!exists("g")) g=3
if (!exists("ma")) ma=1
if (!exists("mi")) mi=3
if (!exists("psi")) psi=90

angrad=ang/180.0*pi

if (op1==1) h(x,y) = min(x,y)*ma; else \
    if (op1==2) h(x,y) = max(0,min(x,y)*ma-abs(x-y)*g); else
    if (op1==3) h(x,y) = max(0,min(x,y)*(ma*psi-mi*(100-psi))/100.0-abs(x-y)*g); else

hmax(x,y)=h(x,y)+min(m-y,n-x)*ma

best1(x,y)=h(y,y)           # row-by-row
best2(x,y)=h(x,x)           # col-by-col
best3(x,y)=h((x+y)/2,(x+y)/2) # diagonal
best4(x,y)=h((y*cos(angrad)+x*sin(angrad))/(cos(angrad)+sin(angrad)),\
    (y*cos(angrad)+x*sin(angrad))/(cos(angrad)+sin(angrad))) # ang
best5(x,y)=h(max(x,y),max(x,y)) # square
best6(x,y)=h(min(x,y),min(x,y)) # anti-square

if (op2==1) best(x,y) = best1(x,y); else \
    if (op2==2) best(x,y) = best2(x,y); else \
    if (op2==3) best(x,y) = best3(x,y); else \
    if (op2==4) best(x,y) = best4(x,y); else \
    if (op2==5) best(x,y) = best5(x,y); else \
    best(x,y) = best6(x,y);

f(x,y)=(hmax(x,y)<=best(x,y))?1:0;
plot f(x,y) notitle

```

Anexo III

Artigos decorrentes desta tese

III.1 Artigos completos publicados em periódicos internacionais

- **CUDAlign 2.1 – Capes Qualis CC - A1 (JCR 2,173):**

Retrieving Smith-Waterman Alignments with Optimizations for Megabase Biological Sequences Using GPU. *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, pp 1009–1021. IEEE, 2013 [35].

- **Sistema Multiagentes – Capes Qualis CC - A1 (JCR 1,965):**

An agent-based solution for dynamic multi-node wavefront balancing in biological sequence comparison. *Expert Systems with Applications (ESWA)*, 41(10):4929 – 4938, Elsevier, 2014 [39].

- **Arquitetura MASA - (periódico novo, ainda sem JCR):**

MASA: a Multi-Platform Architecture for Sequence Aligners with Block Pruning. *ACM Transactions on Parallel Computing (TOPC)*. ACM, 2015 [40]. *Paper aceito, aguardando publicação.*

III.2 Artigos completos publicados em conferências internacionais

- **CUDAlign 2.0 – Capes Qualis CC - A1:**

Smith-Waterman Alignment of Huge Sequences with GPU in Linear Space. *IEEE International Parallel & Distributed Processing Symposium (IPDPS)*, pp 1199–1211. IEEE, 2011 [34].

- **CUDAlign 3.0 (em ambientes homogêneos) – Capes Qualis CC - A1:**

CUDAlign 3.0: Parallel Biological Sequence Comparison in Large GPU Clusters. *IEEE/ACM Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, pp 160–169, IEEE/ACM, 2014 [36].

III.3 Artigos resumidos publicados em conferências internacionais

- **CUDAAlign 3.0 (em ambientes heterogêneos) – Capes Qualis CC - A2:**
Fine-grain parallel megabase sequence comparison with multiple heterogeneous GPUs. Symposium on Principles and Practice of Parallel Programming (PPoPP), pp 383–384, ACM, 2014 [37].

III.4 Artigos completos submetidos a periódicos internacionais (em processo de revisão)

- **CUDAAlign 4.0 – Capes Qualis CC - A1 (JCR 2,173)**
Genome Wide Alignment in GPU Cluster with Incremental Speculative Traceback. IEEE Transactions on Parallel and Distributed Systems (TPDS). IEEE, 2015 [38].
- **Revisão Bibliográfica – Capes Qualis CC - A1 (JCR 4,043)**
Parallel Exact Pairwise Biological Sequence Comparison: Algorithms, Platforms and Classification. ACM Computing Surveys. ACM, 2015 [43].

III.5 Primeira página dos artigos

(em ordem de publicação)

Smith-Waterman Alignment of Huge Sequences with GPU in Linear Space

Edans Flavius de O. Sandes, Alba Cristina M. A. de Melo

Department of Computer Science

University of Brasilia (UnB)

Brasilia, Brazil

{edans,albam}@cic.unb.br

Abstract—Cross-species chromosome alignments can reveal ancestral relationships and may be used to identify the peculiarities of the species. It is thus an important problem in Bioinformatics. So far, aligning huge sequences, such as whole chromosomes, with exact methods has been regarded as unfeasible, due to huge computing and memory requirements. However, high performance computing platforms such as GPUs are being able to change this scenario, making it possible to obtain the exact result for huge sequences in reasonable time. In this paper, we propose and evaluate a parallel algorithm that uses GPUs to align huge sequences, executing the Smith-Waterman algorithm combined with Myers-Miller, with linear space complexity. In order to achieve that, we propose optimizations that are able to reduce significantly the amount of data processed and that enforce full parallelism most of the time. Using the GTX 285 Board, our algorithm was able to produce the optimal alignment between sequences composed of 33 Millions of Base Pairs (MBP) and 47 MBP in 18.5 hours.

I. INTRODUCTION

In the last decade, genome projects have produced a huge amount of new biological data. In order to better understand a newly sequenced organism, biologists compare its sequence against millions of other sequences, in order to infer properties. Sequence comparison is, thus, one of the most important mechanisms in Bioinformatics. One of the first exact methods to globally compare two sequences is Needleman-Wunsch (NW) [1]. It is based on dynamic programming (DP) and has time and space complexity $O(mn)$, where m and n are the sizes of the sequences. The NW algorithm was modified by Smith-Waterman (SW) [2] to deal with local alignments. In SW, a linear gap function was used. Nevertheless, in the nature, gaps tend to occur together. For this reason, the affine gap model is often used, where the penalty for opening a gap is higher than the penalty for extending it. Gotoh [3] modified the SW algorithm, without changing time and space complexity, to include affine gap penalties.

One of the most restrictive characteristics of SW and its variants is the quadratic space needed to store the DP matrices. For instance, in order to compare two 30 MBP sequences, we would need at least 3.6 PB of memory. This fact was observed by Myers-Miller [4], that proposed the use of Hirschberg's algorithm [5] to compute global alignments

in linear space. The algorithm uses a divide and conquer technique that recursively splits the DP matrix to obtain the optimal alignment.

In the last years, Graphics Processing Units (GPUs) have received a lot of attention because of their TFlops peak performance and their availability in PC desktops. In the Bioinformatics research area, there are some implementations of SW in GPUs [6, 7, 8, 9, 10, 11, 12, 13], that were able to obtain the similarity score with very good speedups. Nevertheless, with the exception of CUDAlign 1.0 [13], all of them define a maximum size for the query sequence. That means that two huge sequences cannot be compared in such implementations.

As far as we know, the only strategies that are able to retrieve the alignment in GPUs are [6] and [12]. Since both of them execute in quadratic space, the sizes of the sequences to be compared is severely restricted.

In this paper, we propose and evaluate CUDAlign 2.0, a new algorithm using GPU that is able to retrieve the alignment of huge sequences with the SW algorithm, using the affine gap model. Our implementation is only bound to the total available global memory in the GPU and the available disk space in the desktop. Our algorithm receives two input sequences and provides the optimal alignment as output. It runs in 6 stages, where the first three stages are executed in GPU and the last three stages run in CPU. The first stage executes SW [2] to retrieve the best score and its position in the DP matrices, as in CUDAlign 1.0 [13]. Also, some special rows are saved to disk. The goal of stages 2, 3 and 4 is to retrieve points where the optimal alignment occurs in special rows/columns, thus creating small sub-problems. In stage 5, the alignments for each sub-problem are obtained and concatenated to generate the full alignment. In stage 6, the alignment can be visualized. The proposed algorithm was implemented in CUDA and C++ and executed in the GTX 285 board. With our algorithm, we were able to retrieve the alignment between the human chromosome 21 and the chimpanzee chromosome 22, with respectively 47 MBP and 33 MBP, in 18.5 hours, using reasonable disk area and GPU memory.

The rest of this paper is organized as follows. In Section II, we present the Smith-Waterman and the Myers and Miller algorithms. In Section III, related work is discussed. Section

Retrieving Smith-Waterman Alignments with Optimizations for Megabase Biological Sequences Using GPU

Edans Flavius de O. Sandes and Alba Cristina M.A. de Melo, *Senior Member, IEEE*

Abstract—In Genome Projects, biological sequences are aligned thousands of times, in a daily basis. The Smith-Waterman algorithm is able to retrieve the optimal local alignment with quadratic time and space complexity. So far, aligning huge sequences, such as whole chromosomes, with the Smith-Waterman algorithm has been regarded as unfeasible, due to huge computing and memory requirements. However, high-performance computing platforms such as GPUs are making it possible to obtain the optimal result for huge sequences in reasonable time. In this paper, we propose and evaluate CUDAlign 2.1, a parallel algorithm that uses GPU to align huge sequences, executing the Smith-Waterman algorithm combined with Myers-Miller, with linear space complexity. In order to achieve that, we propose optimizations which are able to reduce significantly the amount of data processed, while enforcing full parallelism most of the time. Using the NVIDIA GTX 560 Ti board and comparing real DNA sequences that range from 162 KBP (Thousand Base Pairs) to 59 MBP (Million Base Pairs), we show that CUDAlign 2.1 is scalable. Also, we show that CUDAlign 2.1 is able to produce the optimal alignment between the chimpanzee chromosome 22 (33 MBP) and the human chromosome 21 (47 MBP) in 8.4 hours and the optimal alignment between the chimpanzee chromosome Y (24 MBP) and the human chromosome Y (59 MBP) in 13.1 hours.

Index Terms—Bioinformatics, sequence alignment, parallel algorithms, GPU

1 INTRODUCTION

BIOINFORMATICS is an interdisciplinary field that involves computer science, biology, mathematics, and statistics [1]. One of its main goals is to analyze biological sequence data and genome content in order to obtain the function/structure of the sequences as well as evolutionary information.

Once a new biological sequence is discovered, its functional/structural characteristics must be established. The first step to achieve this goal is to compare the new sequence with the sequences that compose genomic databases, in search of similarities. This comparison is made thousands of times in a daily basis, all over the world. Sequence comparison is, therefore, one of the most basic operations in Bioinformatics. As output, a sequence comparison operation produces similarity scores and alignments. The score is a measure of similarity between the sequences and the alignment highlights the similarities/differences between the sequences. Both are very useful and often are used as building blocks for more complex problems such as multiple sequence alignment and secondary structure prediction.

Smith and Waterman (SW) [2] proposed an exact algorithm that retrieves the optimal score and local alignment between two sequences. It is based on Dynamic Programming (DP) and has time and space complexity $O(mn)$, where m and n are the sizes of the sequences. In SW, a linear gap function was used. Nevertheless, in the nature, gaps tend to occur together. For this reason, the affine gap model is often used, where the penalty for opening a gap is higher than the penalty for extending it. Gotoh [3] modified the SW algorithm to include affine gap penalties.

One of the most restrictive characteristics of SW and its variants is the quadratic space needed to store the DP matrices. For instance, in order to compare two 33 MBP (Million Base Pairs) sequences, we would need at least 4.3 PB of memory. This fact was observed by Hirschberg [4], who proposed a linear space algorithm to compute the Longest Common Subsequence (LCS). Hirschberg's algorithm was later modified by Myers and Miller (MM) [5] to compute global alignments in linear space.

Another restrictive characteristic of the SW algorithm is that it is usually slow due to its quadratic time complexity. In order to accelerate the comparison between long sequences, heuristic tools such as LASTZ [6] and MUMMER [7] were created. They use seeds (LASTZ) and suffix trees (MUMMER) to scan the sequences, providing a big picture of the main differences/similarities between them. On the other hand, Smith-Waterman provides the optimal local alignment, where the regions of differences/similarities are much more accurate, as well as the gapped regions that represent inclusion/deletion of bases. Therefore, we claim that both kinds of tools should be used in a complementary way: first, MUMMER or LASTZ would be executed and

- E.F. de O. Sandes is with the Department of Computer Science, University of Brasilia, Campus Darcy Ribeiro, PO Box 4466, Asa Norte, Brasilia-DF CEP 70910-900, Brazil. E-mail: edans@cic.unb.br.
- A.C.M.A. de Melo is with the Department of Computer Science, University of Brasilia, Campus Darcy Ribeiro, PO Box 4466, Asa Norte, Brasilia-DF CEP 70910-900, Brazil. E-mail: albammm@cic.unb.br.

Manuscript received 16 Nov. 2011; revised 29 Apr. 2012; accepted 4 June 2012; published online 22 June 2012.

Recommended for acceptance by S. Aluru.

For information on obtaining reprints of this article, please send e-mail to: tpds@computer.org, and reference IEEECS Log Number TPDS-2011-11-0838. Digital Object Identifier no. 10.1109/TPDS.2012.194.

CUDAlign 3.0: Parallel Biological Sequence Comparison in Large GPU Clusters

Edans F. de O. Sandes*, Guillermo Miranda[†], Alba C. M. A. de Melo*, Xavier Martorell^{†‡}, Eduard Ayguadé^{†‡}

*University of Brasilia (UnB)

{edans, albamm}@cic.unb.br

[†]Barcelona Supercomputing Center (BSC)

{guillermo.miranda, xavier.martorell, eduard.ayguade}@bsc.es

[‡]Universitat Politècnica de Catalunya (UPC)

{xavim, eduard}@ac.upc.edu

Abstract—This paper proposes and evaluates a parallel strategy to execute the exact Smith-Waterman (SW) biological sequence comparison algorithm for huge DNA sequences in multi-GPU platforms. In our strategy, the computation of a single huge SW matrix is spread over multiple GPUs, which communicate border elements to the neighbour, using a circular buffer mechanism. We also provide a method to predict the execution time and speedup of a comparison, given the number of the GPUs and the sizes of the sequences. The results obtained with a large multi-GPU environment show that our solution is scalable when varying the sizes of the sequences and/or the number of GPUs and that our prediction method is accurate. With our proposal, we were able to compare the largest human chromosome with its homologous chimpanzee chromosome (249 Millions of Base Pairs (MBP) x 228 MBP) using 64 GPUs, achieving 1.7 TCUPS (Tera Cells Updated per Second). As far as we know, this is the largest comparison ever done using the Smith-Waterman algorithm.

I. INTRODUCTION

In comparative genomics, biologists need to compare their sequences against other organisms in order to infer functional and structural properties. Sequence comparison is, therefore, one of the most basic operations in Bioinformatics [1], usually solved using heuristic methods due to the excessive execution times of their exact counterparts.

The exact algorithm to execute pairwise comparisons is the one proposed by Smith-Waterman (SW) [2], which is based on Dynamic Programming (DP), with quadratic time and space complexities. The SW algorithm is normally executed to compare (a) two DNA sequences or (b) a protein sequence (query sequence) to a genomic database, which is composed of several protein sequences. Both cases have been parallelized in the literature. In the first case, a single SW matrix is calculated and all the Processing Elements (PEs) participate in this calculation (fine-grained computation). Since there are data dependencies, neighbour PEs communicate in order to exchange border elements. For Megabase DNA sequences, the algorithm calculates a huge matrix with several Petabytes. In the second case, multiple small SW matrices are calculated usually without communication between the PEs (coarse-grained computation). With the current genomic databases, often hundreds of thousands

SW matrices are calculated in a single $query \times database$ comparison.

GPUs (Graphics Processing Units) are highly parallel architectures which execute data parallel problems usually much faster than a general-purpose processor. For this reason, they have been considered to accelerate SW, with many versions already available, executing on a single GPU [3–7]. More recently, several approaches have been proposed to execute SW in multiple GPUs [8–12].

Very few GPU strategies [3, 12] allow the comparison of Megabase sequences longer than 10 Million Base Pairs (MBP). SW# [12] is able to use 2 GPUs in a single Megabase comparison to calculate the Myers-Miller [13] linear space variant of SW. CUDAlign [3] executes in a single GPU and obtains the alignment of Megabase sequences with a combined SW and Myers-Miller strategy. When compared to SW# (1 GPU), CUDAlign (1 GPU) presents better execution times for huge sequences [12].

In this paper, we propose and evaluate CUDAlign 3.0, an evolution of CUDAlign 2.1 [3] which executes the first stage of the SW algorithm in a fine-grained parallel way, comparing Megabase DNA sequences in multiple GPUs. In CUDAlign 3.0, we faced the challenge of distributing the computation of a huge DP matrix among several GPUs, with low impact on the performance. In the proposed strategy, GPUs are logically arranged in a linear way so that each GPU calculates a subset of columns of the SW matrix, sending the border column elements to the next GPU.

Due to the data dependencies of the SW recurrence relation, a slowdown in the communication between any 2 GPUs will slowdown the whole matrix computation [14]. To tackle this problem, we decided that computation must be overlapped with communication, so asynchronous CPU threads will send/receive data to/from neighbor GPUs while the GPU continues computing.

Sequence comparisons that deal with Megabase sequences can take hours or even days to complete. In this scenario, we developed a method to predict the execution time and speedup of a comparison, given the number of the GPUs and the sizes of the sequences.

CUDAlign 3.0 was implemented in CUDA, C++ and

Fine-Grain Parallel Megabase Sequence Comparison with Multiple Heterogeneous GPUs

Edans F. de O. Sandes

University of Brasilia
edans@cic.unb.br

Guillermo Miranda

Barcelona Supercomputing Center
guillermo.miranda@bsc.es

Alba C. M. A. Melo

University of Brasilia
alba@cic.unb.br

Xavier Martorell

Universitat Politècnica de Catalunya
Barcelona Supercomputing Center
xavier.martorell@bsc.es

Eduard Ayguadé

Universitat Politècnica de Catalunya
Barcelona Supercomputing Center
eduard.ayguade@bsc.es

Abstract

This paper proposes and evaluates a parallel strategy to execute the exact Smith-Waterman (SW) algorithm for megabase DNA sequences in heterogeneous multi-GPU platforms. In our strategy, the computation of a single huge SW matrix is spread over multiple GPUs, which communicate border elements to the neighbour, using a circular buffer mechanism that hides the communication overhead. We compared 4 pairs of human-chimpanzee homologous chromosomes using 2 different GPU environments, obtaining a performance of up to 140.36 GCUPS (Billion of cells processed per second) with 3 heterogeneous GPUS.

Categories and Subject Descriptors D.1.3 [Programming Techniques]: Concurrent Programming; J.3 [Life and Medical Sciences]: Biology and Genetics

Keywords GPU; Biological Sequence Comparison; Smith-Waterman;

1. Introduction

Smith-Waterman (SW) [4] is an exact algorithm based on the longest common subsequence (LCS) concept, that uses dynamic programming to find local alignments between two sequences. SW is very accurate but it needs a lot of computational resources. GPUs (Graphics Processing Units) have been considered to accelerate SW, but very few GPU strate-

gies [1, 3] allow the comparison of Megabase sequences longer than 10 Million Base Pairs (MBP). SW#[1] uses 2 GPUs to execute a Myers-Miller [2] linear space variant of SW. CUDAlign [3] uses a single GPU to execute a combined strategy with SW and Myers-Miller. When compared to SW#(1 GPU), CUDAlign (1 GPU) presents better execution times for huge sequences [1].

In this work, we modified the most computational intensive stage of CUDAlign, parallelizing the computation of a single huge DP matrix among heterogeneous GPUs in a fine-grained way. In the proposed strategy, GPUs are logically arranged in a linear way so that each GPU calculates a subset of columns of the SW matrix, sending the border column elements to the next GPU. Experimental results collected in 2 different environments show performance of up to 140 GCUPS (Billion of cells processed per second) using 3 heterogeneous GPUS. With this performance, we are able to compare real megabase sequences in reasonable time.

2. Proposed Multi-GPU Strategy

We modified the first stage of CUDAlign [3] to parallelize computation of a single huge DP matrix among many heterogeneous GPUs. The parallelization is done using a multi-GPU wavefront method, where the GPUs are logically arranged in a linear way, i.e, the first GPU is connected to the second, the second to the third and so on. Each GPU computes a range of columns of the DP matrix and the GPUs transfer the cells of their last column to the next GPU. In a scenario composed of heterogeneous GPUs, assigning the same number of columns to all GPUs is not a good choice. In this case, the slowest GPU would determine the processing rate of the whole wavefront. To avoid this, we statically distribute the columns proportionally to the computational power of each GPU. This distribution can be obtained from sequence comparison benchmarks that determine each GPU

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).

PPoPP '14, February 15–19, 2014, Orlando, Florida, USA.

Copyright is held by the owner/author(s).

ACM 978-1-4503-2656-8/14/02.

<http://dx.doi.org/10.1145/2555243.2555280>



Contents lists available at ScienceDirect

Expert Systems with Applications

journal homepage: www.elsevier.com/locate/eswa

An agent-based solution for dynamic multi-node wavefront balancing in biological sequence comparison



Edans Flavius de O. Sandes, Célia Ghedini Ralha*, Alba Cristina M.A. de Melo

Computer Science Department, Institute of Exact Sciences, University of Brasília, P.O. Box 4466, Zip Code 70.904-970 Brasília, Brazil

ARTICLE INFO

Keywords:

Agent-based approach
 Bioinformatics algorithms
 Heterogeneous multi-node wavefront method
 Parallel biological sequence comparison
 Smith–Waterman algorithm

ABSTRACT

Many parallel and distributed strategies were created to reduce the execution time of bioinformatics algorithms. One well-known bioinformatics algorithm is the Smith–Waterman, that may be parallelized using the wavefront method. When the wavefront is distributed across many heterogeneous nodes, it must be balanced to create a synchronous data flow. This is a very challenging problem if the nodes have variable computational power. This paper presents an agent-based solution for parallel biological sequence comparison applications that use the multi-node wavefront method. In our approach, autonomous agents are able to identify unbalanced computations and dynamically rebalance the load among the nodes. Two strategies were developed to the balancer agent in order to identify if the computations are balanced, one using global information and other using only local information. The global strategy demands a huge amount of data transfers, incurring in more communication, whereas the local strategy can decide about the balancing status using only local information. The results show that the balancing gains of strategies are very close. Thus, the local strategy is preferred, since it can be implemented in real wavefront balancers with almost the same benefits as the global strategy.

© 2014 Elsevier Ltd. All rights reserved.

1. Introduction

Bioinformatics is an interdisciplinary field that involves computer science, biology, mathematics and statistics (Mount, 2004). One of its main goals is to analyze biological sequence data and genome content in order to obtain the function/structure of the sequences as well as evolutionary information. Once a new biological sequence is discovered, its functional and structural characteristics must be established. The first step to achieve this goal is to compare the new sequence with the sequences with known characteristics, in search of similarities. This comparison is made thousands of times in a daily basis, all over the world. Sequence comparison is, therefore, one of the most basic and important operations in bioinformatics.

In order to cope with the sequence comparison computational challenge, Smith and Waterman (1981) proposed an exact algorithm to compare two sequences, retrieving the optimal score and alignment between two sequences. The Smith–Waterman algorithm (SW) is based on dynamic programming with time and space complexity $O(mn)$, where m and n are the sequence sizes. In the literature, there are two types of Smith–Waterman computations:

fine-grained and coarse-grained. The coarse-grained computation is usually applied to protein search and, in this case, one protein sequence (query sequence) is compared to a genomic database, which is composed of several protein sequences. The protein sequences are small, composed of thousands of characters. Therefore, several small SW matrices are computed. In the fine-grained approach, two DNA sequences are usually compared. DNA sequences can be very big, with hundreds of millions of characters. In this case, a single huge SW matrix is computed, with hundreds of Terabytes or even Petabytes, taking several hours or even days to complete. The multi-node wavefront method is employed for fine-grained comparisons.

Many efforts have been made to reduce the execution time of biological sequence comparison applications. Parallel versions for Smith–Waterman were created for clusters, FPGAs and GPUs. Even though the FPGA designs are highly parallel and fine-grained, they execute in one single FPGA processor element (Yamaguchi, Tsoi, & Luk, 2011; Zhou, Dou, & Xia, 2012). The approaches that execute in multiple processing elements are either fine-grained using clusters (Sathe & Shrimankar, 2013; Boukerche, Batista, de Melo, Scarel, & de Souza, 2012) and GPUs (Sandes, Miranda, Melo, Martorell, & Ayguade, 2014), or coarse-grained using GPUs (Liu, Wirawan, & Schmidt, 2013; Ino, Munekawa, & Hagihara, 2012). The fine-grained multi-node approaches in the literature target dedicated environments and distribute statically the multi-node wavefront calculations either evenly, since they execute in homogeneous

* Corresponding author. Tel.: +55 6182976665; fax: +55 6131073661.

E-mail addresses: ghedini@cic.unb.br, celiaghedini@gmail.com (C. Ghedini Ralha).

MASA: a Multi-Platform Architecture for Sequence Aligners with Block Pruning

Edans F. de O. Sandes, Department of Computer Science, University of Brasilia, Brazil
Guillermo Miranda, Barcelona Supercomputing Center, Spain
Xavier Martorell, Barcelona Supercomputing Center and Universitat Politecnica de Catalunya, Spain
Eduard Ayguade, Barcelona Supercomputing Center and Universitat Politecnica de Catalunya, Spain
George Teodoro, Department of Computer Science, University of Brasilia, Brazil
Alba C. M. A. de Melo, Department of Computer Science, University of Brasilia, Brazil

Biological sequence alignment is a very popular application in Bioinformatics used routinely worldwide. Many implementations of biological sequence alignment algorithms have been proposed for multicores, GPUs, FPGAs and CellBEs. These implementations are platform-specific and porting them to other systems requires considerable programming effort. This paper proposes and evaluates MASA, a flexible and customizable software architecture that enables the execution of biological sequence alignment applications with three variants (local, global and semi-global) in multiple hardware/software platforms with block pruning, which is able to reduce significantly the amount of data processed. To attain our flexibility goals, we also propose a generic version of block pruning and developed multiple parallelization strategies as building blocks, including a new asynchronous dataflow based parallelization, which may be combined to implement efficient aligners in different platforms. We provide four MASA aligner implementations for multicores (OmpSs and OpenMP), GPU (CUDA) and Intel Phi (OpenMP), showing that MASA is very flexible. The evaluation of our generic block pruning strategy shows that it significantly outperforms the previously proposed block pruning, being able to prune up to 66.5% of the cells when using the new dataflow based parallelization strategy.

Categories and Subject Descriptors: D.1.3 [Programming Techniques]: Parallel Programming

Additional Key Words and Phrases: Biological Sequence Alignment, Parallel Algorithms, GPU, multicores, Intel Phi

1. INTRODUCTION

The astonishing evolution of DNA sequencing techniques is producing an overwhelming number of new biological sequence data to be analyzed. Also, the number of laboratories that are analyzing the sequences is quickly increasing. To cope with this, Life Sciences laboratories are facing the challenge of producing accurate results in a very short time.

Once a new biological sequence is produced, its functional/structural characteristics must be established. In order to do that, the newly discovered sequences are compared against other sequences, looking for similarities. This is done in a daily basis, all over the world.

Biological sequence comparison is, therefore, a very important operation in Bioinformatics [Mount 2004]. It produces (a) a score, indicating the similarity between the sequences and, optionally, (b) an alignment, which highlights the regions of similarities/differences between the sequences.

Biological sequences can be DNA, RNA or protein sequences. Protein and RNA sequences are rather small and their sizes range from hundreds to tens of thousands of residues (amino acids and nucleotide bases, respectively). On the other hand, DNA sequences can be very long, often composed of Millions of base pairs (Mbp).

There are three types of comparisons: (a) global, where all the characters of the sequences belong to the alignment; (b) local, where a subset of the characters belongs to the alignment and (c) semi-global, where the head/tail of the sequences is discarded. Depending on the analysis, the biologists may choose among the types of the sequences

CUDAlign 4.0: Genome Wide Alignment in GPU Cluster with Incremental Speculative Traceback

Edans F. de O. Sandes, Guillermo Miranda, Xavier Martorell,
Eduard Ayguade, George Teodoro, and Alba C. M. A. de Melo, *Senior Member, IEEE*

Abstract—This paper proposes and evaluates CUDAlign 4.0, a parallel strategy to obtain the optimal alignment of huge DNA sequences in multi-GPU platforms, using the exact Smith-Waterman (SW) algorithm. In the first phase of CUDAlign 4.0, a huge Dynamic Programming (DP) matrix is computed by multiple GPUs, which asynchronously communicate border elements to the right neighbor in order to find the optimal score. After that, the traceback phase of SW is executed. The efficient parallelization of the traceback phase is very challenging because of the high amount of data dependency, which particularly impacts the performance and limits the application scalability. In order to obtain a multi-GPU highly parallel traceback phase, we propose and evaluate a new parallel traceback algorithm called Incremental Speculative Traceback (IST), which pipelines the traceback phase, speculating incrementally over the values calculated so far, producing results in advance. With CUDAlign 4.0, we were able to calculate SW matrices with up to 60 Peta cells, obtaining the optimal local alignments of all Human and Chimpanzee homologous chromosomes, whose sizes range from 26 Millions of Base Pairs (MBP) up to 249 MBP. As far as we know, this is the first time such genome wide comparison was made with the SW exact method. We also show that the IST algorithm is able to reduce the traceback time from $2.15\times$ up to $21.03\times$, when compared with the baseline traceback algorithm. The human \times chimpanzee chromosome 5 comparison (180 MBP \times 183 MBP) attained 10.35 TCUPS (Tera Cells Updated per Second) using 384 GPUs. In this case, CUDAlign 4.0 was able to produce the optimal alignment in 53 minutes and 7 seconds, with a speculation hit ratio of 98.2%.

Index Terms—Bioinformatics, sequence alignment, parallel algorithms, GPU



1 INTRODUCTION

In comparative genomics, biologists need to compare sequences of organisms in order to infer functional and structural properties. Sequence comparison is, therefore, one of the most basic operations in Bioinformatics [1], usually solved using heuristic methods due to the excessive computation demands of the exact methods.

The Smith-Waterman (SW) [2] is an exact algorithm to compute pairwise comparisons. It is based on Dynamic Programming (DP) and has quadratic time and space complexities. The SW algorithm is divided in two phases, where the first phase is responsible to calculate a DP matrix in order to obtain the **optimal score** between the sequences and the second phase (traceback) obtains the **optimal alignment**. SW is usually executed to compare (a) two DNA sequences or (b) a protein sequence (query sequence) to a genomic database with several protein sequences. Both approaches have been parallelized in the literature. In the first case, a single SW matrix is calculated and all the Processing Elements (PEs) cooperate in this calculation, and PEs communicate to exchange border elements (fine-grained computation). For Megabase DNA sequences, a huge DP matrix with

several Petabytes is computed. In the second case, multiple small SW matrices are calculated usually without communication between the PEs (coarse-grained computation). With the current genomic databases, often hundreds of thousands SW matrices are calculated in a single *query* \times *database* comparison.

GPUs (Graphics Processing Units) are highly parallel architectures that may execute data parallel problems much faster than a general-purpose processor. A number of works have already examined the use of GPUs to accelerate SW computation to obtain the optimal score. Some of them use only one GPU [3, 4, 5, 6, 7, 8], whereas several approaches have been recently proposed to execute SW in multiple GPUs [9, 10, 11, 12, 13, 14, 15].

Very few strategies [16, 4, 5, 15] are able to produce the optimal alignment of Megabase sequences longer than 1 Million Base Pairs (MBP). These strategies use linear space memory techniques to obtain the alignment with a reasonable amount of memory. As far as we know, there are no implementations of SW that presented the full optimal alignment between sequences longer than 60 MBP.

CUDAlign is a parallel application that is able to execute the SW algorithm in GPU for huge DNA sequences and it has evolved to many versions with incremental optimizations [3, 4, 5, 9, 10]. CUDAlign 2.1 [5] was able to obtain the alignment of sequences up to 33MBP in single GPU. In [5], the first phase of SW is executed with the wavefront method combined with a new strategy called block pruning optimization and the second phase of SW executes a modified Myers-Miller algorithm [17]

- E. Sandes, G. Teodoro, and A. Melo are with the Department of Computer Science, University of Brasilia, Brasilia, DF, Brazil.
E-mail: {edans,teodoro,albammm}@cic.umb.br
- G. Miranda, X. Martorell, E. Ayguade are with the Barcelona Supercomputing Center.
Email: {guillermo.miranda, xavier.martorell, eduard.ayguade}@bsc.es

Parallel Exact Pairwise Biological Sequence Comparison: Algorithms, Platforms and Classification

EDANS FLAVIUS DE OLIVEIRA SANDES, University of Brasilia, Brazil

AZZEDINE BOUKERCHE, University of Ottawa, Canada

ALBA CRISTINA MAGALHAES ALVES DE MELO, University of Brasilia, Brazil

Many Bioinformatics applications, such as exact pairwise biological sequence comparison, demand a great quantity of computing resources and, therefore, are excellent candidates to run in high performance computing (HPC) platforms. In the last two decades, a large number of HPC-based solutions were proposed for this problem that run in different platforms, targeting different types of comparisons with slightly different algorithms, making very difficult the comparative analysis of these approaches. This paper proposes a classification of parallel exact pairwise sequence comparison solutions in order to highlight their main characteristics in a unified way. Then, we discuss several HPC-based solutions, including clusters of multicores and accelerators such as CellBEs (Cell Broadband Engines), FPGAs (Field Programmable Gate Arrays), GPUs (Graphics Processing Units) and Intel Xeon Phi as well as hybrid solutions, which combine two or more platforms, providing the actual landscape of the main proposals in this area. Finally, we present open questions and perspectives in this research field.

Categories and Subject Descriptors: D.1.3 [Programming Techniques]: Parallel Programming

Additional Key Words and Phrases: parallel algorithms, biological sequence comparison, FPGA, GPU, multicores, CellBE, Intel Phi

1. INTRODUCTION

In the last decades, we have observed huge advancements in Genetics. Nowadays, there are several genomic databases, publicly accessible through the internet, which contain the biological sequences of a myriad of organisms, including the human. The analysis of the contents of these sequences can lead to important discoveries concerning the identification of genes that can cause diseases, the identification of regions that can be used to silence genes and the determination of evolutionary events, among others [Mount 2004]. The fantastic advancements that we are witnessing today and that we will surely witness in the next decades would not have occurred without Bioinformatics tools. These tools use automated search procedures to unveil the structure/function of the recently discovered sequences.

Pairwise Biological Sequence Comparison aims to compare two sequences, in search of similarities, and it is a crucial operation in many Bioinformatics tools. In the present days, pairwise comparisons are executed thousands of times every day, in research and industrial projects in all continents.

The most widely used exact algorithms to compare two sequences of sizes n and m , respectively, are edit distance [Levenshtein 1966], NW [Needleman and Wunsch 1970], SW [Smith and Waterman 1981], Gotoh [Gotoh 1982] and MM [Myers and Miller 1988]. All these algorithms use dynamic programming and execute in quadratic time ($O(nm)$). Since the genomic databases are growing in an exponential rate and DNA sequences can have thousands of millions of Base Pairs (BP), executing exact algorithms can take days or even weeks. For this reason, the BLAST [Altschul et al. 1990] heuristic algorithm was proposed, among others. Even though results are obtained quickly with these algorithms, they do not aim to obtain the optimal result. In the present paper, we discuss exact pairwise sequence comparison algorithms, which produce the optimal result.

In the last decades, there have been huge advancements in the area of High Performance Computing (HPC). Due to technological achievements and sophisticated component design, many computers nowadays are able to attain Teraflops (Trillions