Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Comparação Paralela de Sequências Biológicas Longas utilizando Unidades de Processamento Gráfico (GPUs)

Edans Flávius de Oliveira Sandes

Brasília
2011

Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Comparação Paralela de Sequências Biológicas Longas utilizando Unidades de Processamento Gráfico (GPUs)

Edans Flávius de Oliveira Sandes

Monografia apresentada como requisito parcial
para conclusão do Mestrado em Computação

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

Brasília
2011

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

Coordenador: Prof. Dr. Mauricio Ayala Rincón

Banca examinadora composta por:

Prof.ª Dr.ª Alba Cristina M. A. de Melo (Orientadora) — CIC/UnB
Prof. Dr. Siang Wun Song — IME/USP
Prof. Dr. Pedro de Azevedo Berger — CIC/UnB

CIP — Catalogação Internacional na Publicação

Sandes, Edans Flávius de Oliveira.

Comparação Paralela de Sequências Biológicas Longas utilizando Unidades de Processamento Gráfico (GPUs) / Edans Flávius de Oliveira Sandes. Brasília : UnB, 2011.
93 p. : il. ; 29,5 cm.

Tese (Mestrado) — Universidade de Brasília, Brasília, 2011.

1. Programação Paralela, 2. Comparação de Sequências Biológicas, 3. Unidades de Processamento Gráfico (GPUs)

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

Comparação Paralela de Sequências Biológicas Longas utilizando Unidades de Processamento Gráfico (GPUs)

Edans Flávius de Oliveira Sandes

Monografia apresentada como requisito parcial
para conclusão do Mestrado em Computação

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

Prof. Dr. Siang Wun Song
IME/USP

Prof. Dr. Pedro de Azevedo Berger
CIC/UnB

Prof. Dr. Mauricio Ayala Rincón
Coordenador do Mestrado em Computação

Brasília, 30 de junho de 2011

Agradecimentos

Primeiramente agraço a Deus por permitir que eu realize sonhos e alcance novos horizontes, pois sei que o homem não pode receber coisa alguma se antes não lhe for dada do céu.

Agradeço também à minha família, que por estar sempre ao meu lado formou grande parte do que sou. À minha noiva amada, Rejane, que sempre acreditou em mim e sempre me incentivou, agradeço por toda a compreensão, paciência e amor.

À minha orientadora, agradeço por todos os ensinamentos, conselhos e ajudas. Obrigado pelo tempo investido, sempre direcionando meus estudos e colocando ordem em minhas ideias, pois sei que sem sua vasta experiência eu não teria conseguido chegar até aqui. Aos professores deste departamento, agradeço por toda a minha formação. As lições que aprendi não foram apenas teorias, mas exemplos práticos de como aplicar conhecimento em situações concretas.

Aos amigos que ganhei durante os anos, obrigado por me propiciarem momentos alegres e descontraídos. Isso com certeza fez minha caminhada até aqui ser muito mais fácil. Gostaria de citar o nome de todos, mas seria injusto se me esquecesse de alguém. Então, aos amigos da universidade, da igreja, do trabalho, do colégio, vizinhos e muitos outros que conheci, agradeço-lhes pelo simples fato de terem existido em minha vida.

Resumo

A comparação de sequências biológicas é uma operação muito importante na Bioinformática. Embora existam métodos exatos para comparação de sequências, estes métodos usualmente são preteridos por causa da complexidade quadrática de tempo e espaço. De forma a acelerar estes métodos, muitos algoritmos em GPU foram propostos na literatura. Entretanto, todas estas propostas restringem o tamanho da sequência de busca de forma que a comparação de sequências genômicas muito longas não é possível. Neste trabalho, nós propomos e avaliamos o CUDAlign, um algoritmo em GPU capaz de comparar sequências biológicas longas com o método exato de Smith-Waterman com o modelo *affine gap*. O CUDAlign foi implementado em CUDA e testado em duas placas de vídeo, separadamente. Para sequências reais com tamanho entre 1 MBP (milhões de pares de bases) e 47 MBP, um desempenho aproximadamente constante em GCUPS (Bilhões de células atualizadas por segundo) foi obtida, mostrando o potencial de escalabilidade da nossa abordagem. Além disso, o CUDAlign foi capaz de comparar o cromossomo 21 humano e o cromossomo 22 do chimpanzé. Esta operação levou aproximadamente 18 horas na GeForce GTX 285, resultando em um desempenho de 23.87 GCUPS, valor muito próximo do desempenho máximo previsto (23.93 GCUPS). Até onde sabemos, esta foi a primeira vez que cromossomos grandes como esses foram comparados com um método exato.

Palavras-chave: Programação Paralela, Comparação de Sequências Biológicas, Unidades de Processamento Gráfico (GPUs)

Abstract

Biological sequence comparison is a very important operation in Bioinformatics. Even though there do exist exact methods to compare biological sequences, these methods are not often employed due to their quadratic time and space complexity. In order to accelerate these methods, many GPU algorithms were proposed in the literature. Nevertheless, all of them restrict the size of the query sequence in such a way that Megabase genome comparison is prevented. In this work, we propose and evaluate CUDAlign, a GPU algorithm that is able to compare Megabase biological sequences with an exact Smith-Waterman affine gap variant. CUDAlign was implemented in CUDA and tested in two GPU boards, separately. For real sequences whose size range from 1 MBP (Megabase Pairs) to 47 MBP, a close to uniform GCUPS (Giga Cells Updates per Second) was obtained, showing the potential scalability of our approach. Also, CUDAlign was able to compare the human chromosome 21 and the chimpanzee chromosome 22. This operation took approximately 18 hours on GeForce GTX 285, resulting in a performance of 23.87 GCUPS, very close to the maximum predicted performance (23.93 GCUPS). As far as we know, this is the first time such huge chromosomes are compared with an exact method.

Keywords: Parallel Programming, Biological Sequence Comparison, Graphical Processor Units (GPUs)

Sumário

1 Introdução **1**
1.1 Motivação . . . 1
1.2 Contribuições . . . 3
1.3 Estrutura do documento . . . 3

2 Comparação de Sequências Biológicas **4**
2.1 Needleman-Wunsch . . . 5
2.2 Smith-Waterman . . . 6
2.3 Gotoh . . . 8
2.4 Myers e Miller (Hirschberg) . . . 9
2.5 Fickett . . . 11

3 Comparação Paralela de Sequências Biológicas **13**
3.1 *Wavefront* . . . 13
3.2 SWMMX (Rognes e Seeberg) . . . 14
3.3 StripSW (Farrar) . . . 15
3.4 GPU-SW (W. Liu) . . . 15
3.5 SWCuda (Manavsky e Valle) . . . 16
3.6 CBESW (A. Wirawan) . . . 17
3.7 CudaSW++1.0 (Y. Liu) . . . 17
3.8 CudaSW++2.0 (Y. Liu) . . . 18
3.9 Comparação dos artigos . . . 19

4 Arquiteturas de Processadores Gráficos **21**
4.1 Larrabee . . . 22
4.2 Cell BE . . . 24
4.3 GPUs da NVidia . . . 25
4.3.1 Componentes de *Hardware* . . . 25
4.3.2 Linhas de Produtos da NVidia . . . 27
4.3.3 Componentes de *Software* . . . 28
4.3.4 Programação em CUDA . . . 30

5 Projeto do CUDAlign **33**
5.1 Técnicas de Paralelismo no CUDAlign . . . 34
5.1.1 Paralelismo externo . . . 35
5.1.2 Paralelismo interno . . . 36
5.2 Execução interna da *thread* . . . 37

5.3 Estruturas em memória 38
5.4 Otimizações 40
5.4.1 Delegação de Células 40
5.4.2 Divisão de Fases 41
5.4.3 *Loop Unrolling* 44
5.4.4 Tolerância a Falhas 45
5.5 Pseudocódigo 46
5.6 Número de Blocos, Threads e Registradores 47
5.7 Previsão de desempenho 50

6 Resultados Experimentais **52**
6.1 Informações de Compilação 52
6.2 Informações da Execução 52
6.2.1 Ambientes Utilizados 52
6.2.2 Sequências selecionadas 54
6.3 Parametrização 54
6.4 Previsão de desempenho 56
6.5 Dados experimentais 59
6.6 Análise dos dados 60

7 Conclusão e Trabalhos Futuros **65**
7.1 Conclusão 65
7.2 Trabalhos Futuros 66

Referências **67**

I Artigo Publicado sobre o CUDAlign **73**

Lista de Figuras

1.1 Crescimento do GenBank 2

2.1 Trecho do alinhamento entre duas amostras do vírus H1N1 5
2.2 Matriz de programação dinâmica do algoritmo Needleman-Wunsch . 7
2.3 Matriz de programação dinâmica do algoritmo Smith-Waterman . . 8
2.4 Dois níveis de recursão do algoritmo de Myers e Miller 10
2.5 Representação do algoritmo de Fickett 11

3.1 Execução em *wavefront* 14

4.1 Arquitetura do processador Larrabee 23
4.2 Arquitetura do processador Cell BE 24
4.3 Arquitetura CUDA - *Stream Processors* (SP) 26
4.4 Arquitetura CUDA - *Texture Processor Clusters* (TPC) 27
4.5 Arquitetura CUDA - Componentes de *hardware* da GT200 28
4.6 Arquitetura CUDA - Componentes de *software* 30
4.7 Arquitetura CUDA - Hierarquia de memória 31

5.1 Paralelismo Externo - Divisão da matriz em blocos 36
5.2 Bloco retangular com 3 *threads* 36
5.3 Vizinhança-K 38
5.4 Bloco não retangular com 3 *threads* 41
5.5 Cadeia de delegação de células entre os blocos 42
5.6 Problema de dependência de dados entre blocos 43
5.7 Execução do bloco dividida em duas fases 44
5.8 Barramento Horizontal dividido em várias linhas da matriz 46

6.1 Tempo de execução variando o número de blocos B 57
6.2 Tempo de execução da GTX 285 para vários tamanhos de sequência . 60
6.3 Tempo de execução em escala logarítmica 61
6.4 Desempenho das comparações em MCUPS 62

Lista de Tabelas

3.1 Artigos sobre o algoritmo Smith-Waterman com *Affine Gap* 20

4.1 Comparação entre placas NVIDIA 29
4.2 Comparação entre modelos da linha TESLA 29

5.1 Tamanho e tipo de memória utilizada para as estruturas do CUDAlign 40

6.1 Detalhes das GPUs utilizadas nos experimentos 53
6.2 Informações sobre as sequências reais utilizadas 54
6.3 Estatísticas de compilação 55
6.4 Tempo de execução variando o número de blocos B 58
6.5 Constantes da fórmula de previsão de desempenho 58
6.6 Previsão de desempenho das comparações 59
6.7 Dados das comparações utilizadas nos testes 59
6.8 Tempo de execução das comparações 61
6.9 Comparação entre BLAST e CUDAlign 63
6.10 Diferença entre o tempo de execução e o tempo previsto 63

Capítulo 1

Introdução

1.1 Motivação

Nos últimos anos, novas técnicas de sequenciamento de DNA foram desenvolvidas, permitindo um aumento significativo na velocidade de criação de informações genômicas. Por meio de novas tecnologias, o tamanho das bases de dados genômicas continua a crescer exponencialmente a cada ano. Uma das bases de dados usadas pelos biólogos é o GenBank [1], mantido pela *National Center for Biotechnology Information* (NCBI) [2], uma das divisões da *National Library of Medicine* (NLM) nos Estados Unidos. A Figura 1.1 ilustra o crescimento do GenBank nos últimos anos. A versão de abril de 2011 do GenBank possui mais de 126 bilhões de pares de bases de DNA provenientes das 135 milhões de sequências cadastradas [3]. O tamanho do GenBank duplica a cada 30 meses [4].

Devido ao crescimento do tamanho das bases genômicas, uma quantidade enorme de sequências precisa ser comparada entre si, de forma a inferir suas características funcionais e estruturais. Nesse cenário, o tempo gasto em cada comparação, assim como a acurácia dos resultados obtidos, são fatores fundamentais para determinar o sucesso dos projetos genomas.

A comparação de sequências é uma operação básica na Bioinformática que serve de subsídio para as análises biológicas. Em cada comparação, um escore indica o grau de similaridade entre as sequências e um alinhamento apresenta as regiões de similaridade entre elas [5]. Espaços (*gaps*) podem ser inseridos nas sequências para produzir melhores alinhamentos. Eventualmente, mais de um alinhamento pode ser produzido entre duas sequências, representando várias regiões similares.

Durante a comparação de sequências, determina-se como será o tratamento dos *gaps*. Uma solução simples é atribuir penalidades constantes para cada *gap* inserido. Entretanto, observa-se que quando os *gaps* ocorrem em posições consecutivas, os alinhamentos representam melhor a evolução biológica. Com isso, o modelo mais utilizado entre os biólogos é o *affine gap model* [6], onde a penalidade do primeiro *gap* é maior que a penalidade dos *gaps* seguintes.

Smith-Waterman (SW) [7] é um algoritmo baseado no conceito de maior subsequência comum (LCS - *longest common subsequence*). O SW utiliza técnicas de programação dinâmica para encontrar o alinhamento local ótimo entre duas subsequências de tamanhos m e n, utilizando tempo e memória na ordem de $O(mn)$.

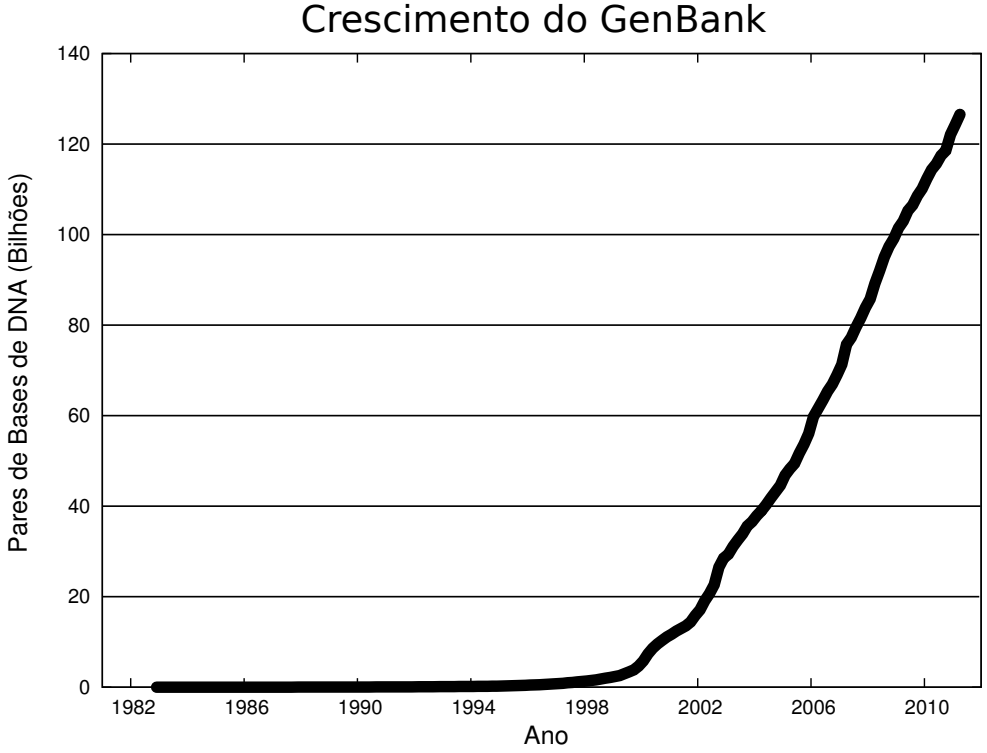


Figura 1.1: Crescimento do número de bases contidas no GenBank durante os anos de 1982 a 2011 [3].

Nesse algoritmo, uma matriz de tamanho $(m + 1) \times (n + 1)$ é calculada. O Smith-Waterman é muito acurado, mas ele necessita de bastantes recursos computacionais. Para diminuir essa exigência, várias heurísticas foram criadas para diminuir o tempo de processamento e o gasto de memória, mas com o prejuízo de o resultado poder não ser ótimo. Tornaram-se então necessárias pesquisas para acelerar os algoritmos exatos de comparação biológica. Uma das formas de aceleração é paralelizar os algoritmos entre vários nós de processamento.

Uma tendência recente na área de programação paralela é a utilização de placas gráficas (GPUs - *Graphics Processing Units*) para acelerar o tempo de execução de algoritmos. Além das funções gráficas usuais, as GPUs atuais são capazes de executar algoritmos de propósito geral, técnica chamada de GPGPU (*General Purpose computing on GPUs*). Devido à facilidade de aquisição dessas placas e à existência de linguagens de alto nível para programação em GPU, técnicas de GPGPU estão ganhando cada vez mais popularidade.

Na área de Bioinformática, algumas implementações de SW em GPU foram desenvolvidas e apresentaram resultados bastante satisfatórios [8–13]. Entretanto, todas elas impõem restrições no tamanho máximo das sequências de busca. Desta forma, não conhecemos nenhuma estratégia que seja capaz de comparar sequências de DNA com tamanho na ordem de 30 milhões de pares de bases (MBP).

O objetivo desta dissertação é propor e avaliar uma estratégia paralela em GPU

que seja capaz de comparar duas sequências de DNA utilizando o método exato de Smith-Waterman com o modelo *affine gap*. A solução, chamada de CUDAlign, deve ser suficientemente escalável para comparar sequências maiores que 30 milhões de pares de base (MBP).

1.2 Contribuições

As principais contribuições desta dissertação são as seguintes:

- A estratégia proposta é eficiente e escalável para comparação exata de sequências biológicas de DNA;
- A solução desenvolvida é capaz de comparar sequências com mais de 30 milhões de bases;
- A otimização "Delegação de células" mantém a GPU processando com o máximo paralelismo durante a maior parte do tempo de processamento;
- A otimização "Divisão de fases" permite que a delegação de células seja feita com maior desempenho e sem incoerências de dados;
- Uma fórmula de previsão de desempenho foi obtida e, por meio dela, é possível prever o tempo aproximado de execução para quaisquer tamanhos de sequência.
- Os cromossomos 21 humano (47 milhões de pares de bases) e o cromossomo 22 do chimpanzé (33 milhões de pares de bases) [14] foram comparados utilizando a estratégia proposta. Até onde temos conhecimento, esta foi a primeira vez que sequências com esses tamanhos foram comparadas com métodos exatos.

1.3 Estrutura do documento

O restante desta dissertação está organizado da seguinte forma. O Capítulo 2 apresenta os principais algoritmos que são utilizados para a comparação de sequências biológicas. Em seguida, o Capítulo 3 descreve os métodos e algoritmos paralelos para comparação de sequências. O Capítulo 4 detalha as arquiteturas das GPUs. O projeto do CUDAlign, que é a estratégia proposta nesta dissertação, é descrito no Capítulo 5. O Capítulo 6 mostra os resultados experimentais obtidos em duas GPUs. Por fim, o Capítulo 7 conclui o trabalho atual e lista os trabalhos futuros. O Anexo I possui o artigo publicado sobre o CUDAlign.

Capítulo 2

Comparação de Sequências Biológicas

Uma das tarefas mais básicas da Bioinformática é identificar se duas sequências biológicas (ou parte delas) são biologicamente relacionadas. Quando duas sequências são comparadas, procura-se evidenciar o quanto elas divergem através de um processo de mutação. O processo de mutação ocorre em uma sequência quando seus resíduos são alterados, ou ainda quando resíduos são inseridos ou removidos da sequência original [15].

Para identificar o grau de relacionamento entre duas sequências, aplicam-se procedimentos computacionais para identificar o melhor alinhamento. Um alinhamento é definido como um pareamento entre os resíduos de cada sequência. Em cada par, os resíduos podem ser equivalentes (*matches*) ou diferentes (*mismatches*) assim como podem ser adicionadas lacunas que representam inserções ou remoções de resíduos (*gaps*). Cada par apresenta uma pontuação (positiva ou negativa) e o somatório da pontuação de todos os pares compõem o escore total, que é o valor que quantifica a expressividade do alinhamento.

A Figura 2.1 é um trecho de um alinhamento entre duas amostras do vírus H1N1 (*Influenza A*). Observe nesta figura a presença de um grande número de pares com bases iguais (*matches*) e duas ocorrências de *gaps* (apresentados com o caractere '-').

Para identificar qual é o melhor alinhamento possível, alguns critérios devem ser definidos para se realizar o procedimento computacional:

1. **Tipo de alinhamento**. O alinhamento pode ser classificado em três tipos [15]: alinhamento global é aquele que inclui todos os resíduos das sequências; alinhamento local é aquele que permite excluir um prefixo ou sufixo das sequências, permitindo a análise de subsequências; alinhamento semi-global é o que permite excluir somente o prefixo ou somente o sufixo das sequências.

2. **Tipo de sequência**. Os resíduos das sequências biológicas podem ser de dois tipos: bases nitrogenadas, formando sequências de DNA ou RNA; ou aminoácidos, formando sequências de proteínas.

3. **Similaridade entre os resíduos**. Utiliza-se uma matriz de substituição sbt com valores que indicam quanto um resíduo x é propício de ser derivado do

```
GQ915018.1  1030  AGAGGCCTATTTGGGGCCATTGCCGGTTTCATTGAA-GGGGGGTGGACAGGG  1080
                  |||||||||||||||||||||||||||||||||||| |||||||| ||||||
CY046923.1  984   AGAGGCCTATTTGGGGCCATTGCCGGTTTCATTGAAAGGGGGGTG-ACAGGG  1034
```

Figura 2.1: Trecho do alinhamento entre duas amostras do vírus H1N1

resíduo y. Assim, constrói-se uma matriz com todas as possíveis substituições. Nas sequências de DNA/RNA, normalmente considera-se apenas o valor de *match* (quando $x = y$) e *mismatch* (quando $x \neq y$). Já em sequências de aminoácidos, utilizam-se matrizes com dimensão 20×20. Um exemplo de matriz de substituição para aminoácidos é a BLOSUM50 [16].

4. **Penalidades por *gaps***. Um *gap* em um alinhamento representa uma inserção ou remoção. Geralmente, quanto mais *gaps* forem apresentados em um alinhamento, mais distantes essas sequências se encontram na cadeia evolutiva [15]. Os principais tipos de penalidades são: o *linear gap*, em que um *gap* de tamanho k é penalizado por $-kG$, sendo G uma constante; e o *affine gap*, em que a penalização é calculada por $-G_{first} - (k-1)G_{ext}$, sendo G_{first} a penalidade por abrir um *gap* e G_{ext} a penalidade por estendê-lo [15]. O *affine gap* segue o raciocínio biológico de que uma deleção de vários resíduos deve ser considerada como uma única deleção, tornando um *gap* longo mais propício de ocorrer do que vários *gaps* pequenos. O *linear gap* é um caso específico do *affine gap*, quando $G_{ext} = G_{first}$.

Dadas as considerações acima, existem algoritmos específicos para identificar o melhor escore possível entre duas sequências, assim como para encontrar o alinhamento que gera esse melhor escore. O restante deste capítulo descreverá diversos algoritmos de alinhamento de sequências. Para todos eles, as entradas são dadas como duas sequências S_0 e S_1, com comprimentos $m = |S_0|$ e $n = |S_1|$. Além disso, as constantes G_{first} e G_{ext} são definidas, respectivamente, como a penalidade por abrir um *gap* e como a penalidade por estendê-lo. Em algoritmos que utilizam apenas *linear gap*, a constante G representa a penalidade do *gap*. Tanto a matriz de substituição (aminoácidos) como a penalidade de *match*/*mismatch* (nucleotídeos) serão representadas pela função $sbt(x, y)$.

2.1 Needleman-Wunsch

O algoritmo Needleman-Wunsch (NW) [17] obtém o alinhamento global ótimo entre as sequências S_0 e S_1, permitindo que *gaps* sejam inseridos para melhorar o alinhamento.

Esse algoritmo calcula uma matriz H onde o elemento $H_{i,j}$ representa o escore do melhor alinhamento entre os prefixos $S_0[1..i]$ e $S_1[1..j]$. Para computar essa matriz utiliza-se a técnica de programação dinâmica, em que soluções maiores são obtidas a partir de soluções menores [18]. 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 $H_{i,0} = -iG$ e $H_{0,j} = -jG$.

A fórmula de recorrência baseia-se no fato de que o valor $H_{i,j}$ de um alinhamento terminado nos resíduos $S_0[i]$ 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 resíduos $S_0[i]$ e $S_1[j]$; (2) alinhar $S_0[1..i]$ e $S_1[1..j-1]$ e inserir um *gap* em S_1; (3) alinhar $S_0[1..i-1]$ e $S_1[1..j]$ e inserir um *gap* em S_0. Desta forma, a recorrência é descrita pela equação 2.1.

$$H_{i,j} = \max \begin{cases} H_{i-1,j-1} & + & sbt(S_0[i], S_1[j]) \\ H_{i-1,j} & - & G \\ H_{i,j-1} & - & G \end{cases} \tag{2.1}$$

A equação é aplicada para preencher a matriz H, do canto superior esquerdo para o canto inferior direito (fase 1). Para encontrar o alinhamento, mantém-se um ponteiro em cada célula indicando qual célula vizinha originou este valor. Assim, percorre-se a sequência de ponteiros da matriz no sentido reverso, iniciando no canto inferior direito até chegar no canto superior esquerdo (fase 2 - *traceback*). 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 Figura 2.2 apresenta um exemplo da matriz de programação dinâmica para um alinhamento global ótimo.

O algoritmo descrito necessita de preencher toda a matriz de similaridade (fase 1), que contém $(m+1)\times(n+1)$ elementos. 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 similaridade entre duas sequencias de 1MBP (1 milhão de pares de bases) demandaria um uso de memória de 4TB (Tera bytes), caso cada célula ocupasse 4 bytes na memória.

2.2 Smith-Waterman

Uma situação muito mais comum na Bioinformática é buscar o melhor alinhamento entre subsequências de S_0 e S_1. Esse melhor alinhamento é chamado de alinhamento local ótimo e é encontrado através do algoritmo de Smith-Waterman (SW) [7].

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 similaridade. 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 nova

H(i,j)		A	C	T	T	C	C	A	G	A
	0	-5	-10	-15	-20	-25	-30	-35	-40	-45
A	-5	1	-4	-9	-14	-19	-24	-29	-34	-39
G	-10	-4	-1	-6	-11	-16	-21	-26	-28	-33
T	-15	-9	-6	0	-5	-10	-15	-20	-25	-30
T	-20	-14	-11	-5	1	-4	-9	-14	-19	-24
C	-25	-19	-13	-10	-4	2	-3	-8	-13	-18
C	-30	-24	-18	-15	-9	-3	3	-2	-7	-12
G	-35	-29	-23	-20	-14	-8	-2	1	-1	-6
G	-40	-34	-28	-25	-19	-13	-7	-4	2	-3
A	-45	-39	-33	-30	-24	-18	-12	-6	-3	3
G	-50	-44	-38	-35	-29	-23	-17	-11	-5	-2
G	-55	-49	-43	-40	-34	-28	-22	-16	-10	-7

Alinhamento Global Ótimo:

```
1  AGTTCCGGAGG  11
   | |||| ||
1  ACTTCCAGA--  9
```

Figura 2.2: Matriz de programação dinâmica gerada pelo alinhamento global entre as sequências S_0=AGTTCCGGAGG e S_1=ACTTCCAGA. Parâmetros: *Match*: +1; *Mismatch*: -2; *Gap*: -5. As células em destaque indicam o percorrimento reverso (*traceback*) capaz de produzir um alinhamento global ótimo.

equação de recorrência está descrita na Equação 2.2.

$$H_{i,j} = \max \begin{cases} H_{i-1,j-1} + sbt(S_0[i], S_1[j]) \\ H_{i-1,j} - G \\ H_{i,j-1} - G \\ 0 \end{cases} \tag{2.2}$$

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 inicia na posição $H_{i,j}$ onde ocorrer o maior valor da matriz. O percorrimento é feito até encontrar uma célula com valor zero. Dessa forma, obtém-se o melhor alinhamento desconsiderando prefixos e sufixos pouco significativos. A Figura 2.3 apresenta um exemplo da matriz de programação dinâmica para o alinhamento local.

H(i,j)		A	C	T	T	C	C	A	G	A
	0	-5	-10	-15	-20	-25	-30	-35	-40	-45
A	-5	2	0	0	0	0	0	0	0	0
G	-10	0	1	0	0	0	0	0	2	0
T	-15	0	0	3	2	0	0	0	0	1
T	-20	0	0	2	5	1	0	0	0	0
C	-25	0	2	0	1	7	3	0	0	0
C	-30	0	2	1	0	3	9	4	0	0
G	-35	0	0	1	0	0	4	8	6	1
G	-40	0	0	0	0	0	0	3	10	5
A	-45	0	0	0	0	0	0	2	5	12
G	-50	0	0	0	0	0	0	0	4	7
G	-55	0	0	0	0	0	0	0	2	3

Alinhamento Local Ótimo:

```
1  AGTTCCGGA  9
   | |||| ||
1  ACTTCCAGA  9
```

Figura 2.3: Matriz de programação dinâmica gerada pelo alinhamento local entre as sequências S_0=AGTTCCGGAGG e S_1=ACTTCCAGA. Parâmetros: *Match*: +1; *Mismatch*: -2; *Gap*: -5. As células em destaque indicam o percorrimento reverso (*traceback*) capaz de produzir um alinhamento local ótimo.

2.3 Gotoh

Uma generalização da Equação 2.1 pode ser feita substituindo a constante G por uma função genérica $\gamma(k)$ que retorna a penalidade de um *gap* com comprimento k. Obtemos então a Equação 2.3, que pode ser resolvida usando programação dinâmica com a complexidade aumentada para $O(n^3)$ [19].

$$H_{i,j} = \max \begin{cases} H_{i-1,j-1} + sbt(S_0[i], S_1[j]) & \\ H_{i-k,j} - \gamma(k), & k = 1, \ldots, i-1 \\ H_{i,j-k} - \gamma(k), & k = 1, \ldots, j-1 \end{cases} \tag{2.3}$$

Um caso particular ocorre quando temos a fórmula $\gamma(k) = -G_{first} - (k-1)G_{ext}$. Para este modelo, chamado de *affine gap*, o algoritmo desenvolvido por Gotoh [20] calcula o melhor alinhamento com complexidade de tempo $O(n^2)$.

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 2.4, 2.5 e 2.6. Adicionalmente, pode-se incluir uma condição para limitar o valor mínimo da matriz H em zero, assim como feito no algoritmo de SW para obter o melhor alinhamento local.

$$H_{i,j} = \max \begin{cases} H_{i-1,j-1} & + & sbt(S_0[i], S_1[j]) \\ E_{i,j} \\ F_{i,j} \\ 0 \end{cases} \tag{2.4}$$

$$E_{i,j} = \max \begin{cases} E_{i,j-1} & - & G_{ext} \\ H_{i,j-1} & - & G_{first} \end{cases} \tag{2.5}$$

$$F_{i,j} = \max \begin{cases} F_{i-1,j} & - & G_{ext} \\ H_{i-1,j} & - & G_{first} \end{cases} \tag{2.6}$$

2.4 Myers e Miller (Hirschberg)

A complexidade teórica dos algoritmos vistos anteriormente inviabilizam a obtenção do alinhamento entre sequências longas. A principal limitação ocorre na quantidade de memória necessária para armazenar toda a matriz. Por exemplo, para alinhar duas sequências com 1 milhão de bases, o algoritmo de SW necessita de 4 *terabytes* de memória. Para resolver este problema, algoritmos foram desenvolvidos para obter o alinhamento ótimo utilizando espaço linear, ou seja, na ordem de $O(m + n)$.

Se apenas o escore máximo for necessário, 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, apenas duas linhas (ou duas colunas) precisam ser armazenadas em memória, tornando o algoritmo linear 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 pelo alinhamento completo. Então, mesmo na comparação contra um banco, é importante obter os alinhamentos das sequências mais similares.

Hirschberg [21] desenvolveu um algoritmo para resolver o problema da Maior Subsequência Comum (LCS – *Longest Common Subsequence*) em espaço linear. Este algoritmo foi posteriormente adaptado por Myers e Miller [22] para transformar o algoritmo de Gotoh em uma solução em espaço linear.

Partindo de um alinhamento global, a idéia é encontrar o ponto médio pelo qual passa um alinhamento ótimo. Para encontrar este ponto, a computação é feita em duas partes. Na primeira, o alinhamento é feito por colunas até a coluna central $\frac{n}{2}$. Na segunda parte, o processamento é feito sobre as duas sequências invertidas S_0^r e S_1^r, até que a mesma coluna central $\frac{n}{2}$ seja calculada.

Os valores finais da coluna $\frac{n}{2}$ são armazenados nos vetores CC e DD. A diferença entre os dois vetores é que o vetor DD armazena o escore dos alinhamentos

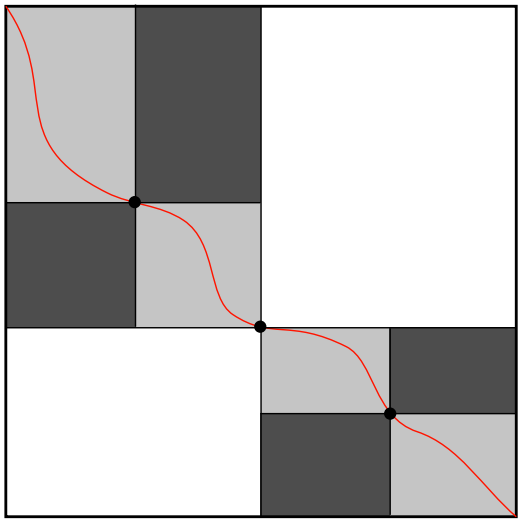


Figura 2.4: Dois níveis de recursão do algoritmo de Myers e Miller.

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 atravessa a coluna $\frac{n}{2}$ através da linha i terá escore $K_i = min\{CC_i + CC'_i, DD_i + DD'_i - G_{open}\}$, onde G_{open} é a penalidade por abrir um *gap* (i.e. $G_{open} = G_{first} - G_{ext}$). Observe que, quando somamos o valor dos dois vetores DD e DD', estamos em um caso onde existem *gaps* em ambas as direções, logo precisamos remover a penalidade G_{open} de um dos *gaps*.

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

Em seguida, realiza-se recursivamente o processamento de duas seções da matriz: $(0, 0)$ a $(i^*, \frac{n}{2})$ e $(i^*, \frac{n}{2})$ a (n, m). A cada passo, dividem-se as seções em trechos menores, até que o tamanho de cada seção seja trivial. A Figura 2.4 ilustra dois passos do algoritmo de Myers e Miller.

Haja vista que a cada passo recursivo a área de processamento cai pela metade, podemos 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 $T = \sum_{i=1}^{\log_2 m - 1} \frac{T_0}{2^k} \leq 2.T_0$, logo, o algoritmo continua na ordem de $O(m.n)$ em tempo, mas com o benefício de utilizar apenas $O(m + n)$ de memória.

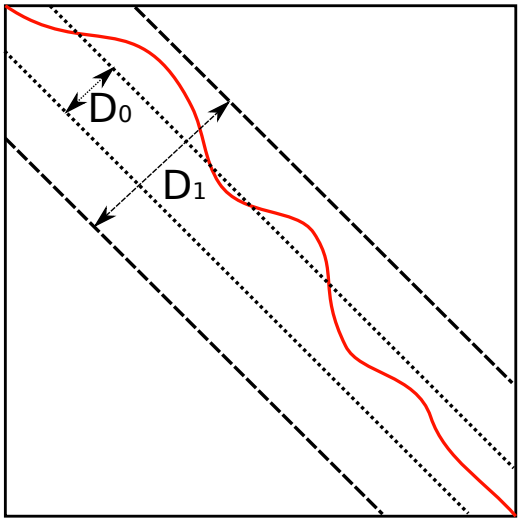


Figura 2.5: Representação do algoritmo de Fickett. O alinhamento não está inicialmente contido na área delimitada pelo valor D_0, mas está dentro da área delimitada por D_1.

2.5 Fickett

Fickett desenvolveu uma otimização [23] capaz de reduzir o tempo de execução de um alinhamento para $O(k.n)$, onde k é o número de *gaps* no 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 é baseado no problema da distância de edição, em que calcula-se uma matriz d de maneira similar ao algoritmo de NW. Entretanto, quanto maior o valor d_{mn}, mais distante (i.e. menos similar) estão as duas sequências.

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 aplicar este conceito:

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 [24];
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 um alinhamento 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 $D_2 > D_3 > \ldots$ até que seja possível encontrar um alinhamento.

Dado o valor limite D, a matriz é computada da seguinte forma. Calculam-se inicialmente as células $d_{1,1}, \ldots, d_{1,L_1}$, 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 $d_{2,1}, \ldots, d_{2,L_2}$, sendo L_2 o menor valor tal que $d_{2,L_2} \geq D$ e $L_2 > L_1$. Assim, temos 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 nas próximas linhas. Sendo assim, podemos generalizar a computação da seguinte forma: calculados os elementos $d_{i,K_i}, \ldots, d_{i,L_i}$ da linha i, a linha $i+1$ é calculada para os elementos $d_{i+1,K_{i+1}}, \ldots, d_{i+1,L_{i+1}}$, 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.

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 2.5 ilustra o funcionamento do algoritmo de Fickett.

Capítulo 3

Comparação Paralela de Sequências Biológicas

A comparação de sequências biológicas muitas vezes é uma tarefa que consome muitos recursos computacionais. Dependendo do tamanho das sequências ou do número de sequências comparadas, o tempo de processamento pode levar vários dias. Esse fato é ainda agravado pois os bancos de sequências públicas crescem exponencialmente. O GenBank [1], por exemplo, é um banco de sequências que duplica de tamanho a cada 30 meses [4].

Para aumentar o desempenho das ferramentas de comparação, a área de computação de alto desempenho vem propondo técnicas específicas para a bioinformática, incluindo hardwares especializados e algoritmos paralelos. Este capítulo visa apresentar técnicas de paralelismo e algoritmos capazes de melhorar o tempo de comparação de sequência, sem utilizar hardwares especializados em operações de Bioinformática.

3.1 *Wavefront*

Nos algoritmos de comparação de sequências vistos no Capítulo 2, a maior parte do tempo de execução é gasta calculando a matriz de similaridade e esta é a parte dos algoritmos que usualmente é paralelizada. Uma das estratégias tradicionalmente utilizadas para paralelizar o cálculo da matriz de similaridade é o método *wavefront*, que se baseia em processar as células em ondas, uma anti-diagonal por vez. O padrão de acesso observado nos algoritmos de comparação biológica é semelhante ao do algoritmo Smith-Waterman (Seção 2.2), de forma que 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 3.1 ilustra o método *wavefront*. No começo da computação, apenas a células 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

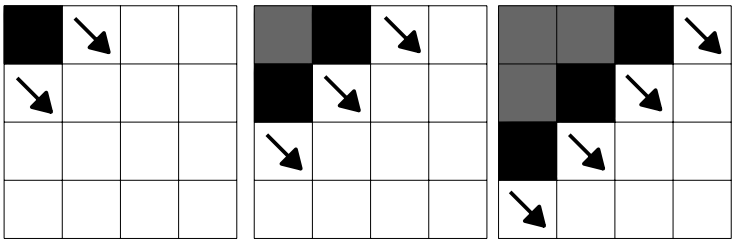


Figura 3.1: Execução em *wavefront*. Três iterações estão ilustradas, sendo que cada iteração calcula uma diagonal. Células pretas representam as células que estão sendo processadas na diagonal atual, as células cinzas são células já processadas e células brancas serão processadas nas iterações seguintes.

tamanho máximo. O paralelismo decresce nas últimas diagonais, até chegar na última célula no canto inferior direito.

3.2 SWMMX (Rognes e Seeberg)

O artigo de Rognes e Seeberg [25] apresenta uma implementação do SW para comparação de aminoácidos com *affine gap* utilizando o conjunto de instruções MMX/SSE. O paralelismo é obtido no sentido da sequência S_0 (vertical), e não no sentido do *wavefront* (diagonal). Para isso, empregou-se a optimização SWAT [26], que aproveita o fato de a matriz F do *affine gap* conter normalmente valores próximos de zero. Com isso, a matriz principal H acessa as matrizes E e 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, salvando muito tempo de processamento. Entretanto, se o limite 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 $sbt(S_0[i], x)$ 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. 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 do MMX/SSE. Cada operação 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.

A comparação foi realizada entre o banco *Swiss-Prot* e 11 sequências selecionadas (as mesmas usadas pelo artigo do BLAST 2 [24]), com tamanho entre 189 a 567 aminoácidos. A penalidade dos *gaps* foi $-10 - 1k$, sendo k o tamanho da sequência de *gaps*. 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.

3.3 StripSW (Farrar)

Farrar [27] apresenta uma implementação de SW com *affine gap* utilizando o conjunto de instruções SSE2, aplicando o paralelismo no sentido vertical da matriz. Assim como no SWMMX (Seção 3.2), 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 o vetor F seja feita no *loop* externo, e não no *loop* interno como era no SWMMX.

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, 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 2.998 MCUPS, utilizou-se uma penalidade de *gap* bastante elevada: $-40 - 2k$, sendo k o tamanho da sequência de *gaps*.

3.4 GPU-SW (W. Liu)

O artigo de W. Liu [8] implementa dois algoritmos em GPU: Smith-Waterman com *affine gap* e o ClustalW [28] para alinhamentos múltiplos de sequências (MSA).

O SW implementado retorna apenas os escores ótimos entre uma sequência e um banco, deixando o alinhamento final para a CPU. O percorrimento da matriz é feito paralelamente com o método *wavefront* (diagonal). 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 amenizar este problema, as sequências são dispostas em ordem crescentes 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 resíduos e o tamanho de um lote em 4.096 sequências.

No alinhamento múltiplo de sequências (MSA), todas as sequências da entrada são alinhadas duas a duas com o SW e, em seguida, executa-se o algoritmo ClustalW [28] para obter o alinhamento múltiplo. Para isso, o algoritmo SW foi adaptado para informar também o número de *matches* ocorridos no alinhamento local

ótimo. Por causa dessa adaptação, uma matriz extra N_A precisa ser calculada, o que exige o uso de mais uma textura de duas dimensões. Para diminuir a área supérflua das texturas, o particionamento dos lotes pode ser feito de duas formas, a depender do tamanho da maior sequência ($maxlen$): se $maxlen > threshold$, utiliza-se o método *multiquery*; caso contrário, executa-se o método *multibatch*. O método *multibatch* particiona as sequências *subject* em vários lotes com $step$ sequências e cada lote é alinhado paralelamente contra uma única sequência de busca. Já o método *multiquery* alinha várias sequências de busca paralelamente com todo o lote.

Outro artigo de W. Liu [9] apresenta uma fórmula para prever o tempo de execução da sua implementação de SW em GPU. Os fatores que determinam a fórmula são classificados em *data-constant*, *data-linear* e *computation-dependent*.

Os resultados do SW 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. Os melhores resultados foram obtidos utilizando $threshold$ entre 500 e 900 e $step$ entre 50 a 160. A linguagem usada para programação da GPU foi a OpenGL Shading Language (GLSL[29, 30]).

3.5 SWCuda (Manavsky e Valle)

No artigo de Manavsky e Valle [10] é apresentada uma implementação do algoritmo Smith-Waterman em GPU, utilizando o ambiente CUDA [31, 32] da NVidia. O algoritmo utiliza *affine gap* 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 percorrimento 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 3.2) 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 por tamanho 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.

3.6 CBESW (A. Wirawan)

No artigo de A. Wirawan [11], o SW com *affine gap* foi implementado em um processador Cell BE do Playstation 3 (Seção 4.2). A otimização de Farrar (Seção 3.3) 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). Cada coluna é dividida em segmentos e cada segmento é processado paralelamente. O Cell BE não possui operações aritmética saturada 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 resultado comparativo do desempenho é dado em MCUPS, que indica o número médio de células da matriz processadas por segundo. O CBESW atingiu pico de 3.646 MCUPS na comparação da sequência de tamanho 852. Comparando com as implementações SSEARCH e Striped SW (Seção 3.3) 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 3.5), executado em uma GeForce 8800GTX, foi obtido um *speedup* de 3x.

3.7 CudaSW++1.0 (Y. Liu)

O artigo [12] apresenta uma proposta que executa o SW 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. O paralelismo é obtido por meio da técnica de wavefront, em que as células das diagonais são processadas em paralelo. Para tratar o problema de as diagonais possuírem um número distinto de células, a implementação considera que cada diagonal possui virtualmente o mesmo número de células.

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 com-

parar sequências menores que uma constante. Na segunda fase, o método intra-tarefas é executado, comparando as sequências restantes. A constante selecionada como tamanho divisor das duas fases foi de 3.072.

Para comparação da maioria das sequências, faz-se necessário o uso da memória global para salvar os resultados intermediários de uma *thread*. Entretanto, a memória global é a memória mais lenta da GPU, o que demanda que seu acesso seja feito de maneira otimizada. A arquitetura da GPU permite que o acesso seja mais rápido caso todas as *threads* acessem a memória em um padrão ordenado. Sendo assim, as *threads* foram agrupadas considerando os detalhes da arquitetura da GPU, de forma a obter maior desempenho no acesso à memória.

Os métodos inter-tarefas e intra-tarefas possuem agrupamentos de *threads* com responsabilidades distintas. Na abordagem inter-tarefas, cada *thread* calcula a comparação de uma sequência do banco, mas na abordagem intra-tarefas, todas as *threads* de um bloco calculam a comparação de uma mesma sequência. Desta forma, usando inter-tarefas, as *threads* de um bloco devem acessar um segmento da memória global representando várias sequências, e no método intra-tarefas, um segmento deve representar dados da mesma sequência.

O CUDASW++ foi testado em uma GPU simples (GTX 280) e uma GPU com dois processadores (GTX 295). A GTX 280 possui 30 SMs totalizando 240 SPs e 1GB de RAM. A GTX 295 possui duas GPUs na mesma placa, totalizando 60 SMs, 480 SPs e 1.8 GB de RAM. 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 GCPUS. 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).

3.8 CudaSW++2.0 (Y. Liu)

Na versão 2.0 do CudaSW++[13], duas implementações do SW com *affine 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 3.3).

A implementação foi testada 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.

3.9 Comparação dos artigos

A Tabela 3.1 apresenta um resumo comparativo entre os artigos discutidos neste capítulo. A coluna "Estratégia" resume a abordagem empregada e em que plataforma a solução foi testada. A coluna "Banco" apresenta qual versão do banco de proteínas *Swiss-Prot* foi utilizada, indicando também o número de sequências e o número total de aminoácidos contidos no banco.

A coluna "Tam. Max." informa qual o tamanho da maior sequência de busca comparada. Nos artigos listados na tabela, o tamanho das sequências de busca utilizadas nos testes em GPU chegam a no máximo 5.478 resíduos. Limitar o tamanho da sequência de busca permite que as estruturas de dados sejam armazenadas nas memórias mais rápidas das GPUs. Embora isso acelere o processamento, sequências maiores são impedidas de serem comparadas.

Os dados da coluna "Saída" indicam que todos os algoritmos retornam apenas o escore de similaridade, sem retornar o alinhamento completo. Já que as sequências comparadas são pequenas, o alinhamento completo é obtido pela CPU apenas das melhores sequências.

A coluna "MCUPS" registra o desempenho médio medido em milhões de células atualizadas por segundo. O desempenho obtidos nos artigos que usaram GPUs variaram de 700 MCUPS a 29.700 MCUPS, sendo que o poder de processamento das placas gráficas utilizadas por cada uma das soluções é distinto. Por fim, a coluna "*Speedup*" apresenta as comparações de desempenho apresentadas nos artigos.

Artigo	**Estratégia**	**Entrada**		**Saída**	**MCUPS**	***Speedup***
		Banco	**Tam. Max.**			
3.2SWMMX (Rognes)	Paralelismo na coluna com otimização MMX/SSE (SWAT). CPU: Intel Pentium III 500MHz	Swiss-Prot 38.0 80.000 seqs 29.1mi aa	567	Escore	150	SSEARCH: 6x OSEARCH: 13x
3.3StripSW (Farrar)	Paralelismo na coluna com otimização SSE2 (SWAT) e acesso com padrão *striped*. CPU: Intel Xeon Core 2 Duo 2.0GHz	Swiss-Prot 49.1 208.005 seqs 75.8mi aa	567	Escore	2.998	SWMMX: 3x Wozniak: 8x
3.4GPU-SW (W. Liu)	Paralelismo na diagonal e em todas as sequências do banco. Outro algoritmo realiza MSA, aplicando SW dois a dois. GPUs: NVIDIA 7900 GTX e ATI X1900 XT.	Swiss-Prot 46.3 176.469 seqs 63.9mi aa	4.095	Escore	$\sim$ 700	SSEARCH: 10x OSEARCH: 16x
3.5SWCuda (Manavsky)	Distribuição das sequências do banco entre *threads*. GPU: 2 x NVIDIA 8800 GTX	Swiss-Prot 51.3 250.296 seqs 91.7mi aa	567	Escore	3.612	Farrar: 3x Liu: 18x SSEARCH: 30x BLAST: 2.4x
3.6CBESW (Wirawan)	Paralelismo na coluna (Farrar) e distribuição das sequências do banco entre processadores. CPU: CellBE com 6 SPEs (Playstation3).	Swiss-Prot 55.2 362.782 seqs 130.5mi aa	852	Escore	3.646	SSEARCH: 30x StripSW: 1.64x CUDASW: 3x
3.7CudaSW++ 1.0 (Y. Liu)	Paralelismo na diagonal e distribuição das sequências do banco entre processadores. GPU: NVIDIA GTX 280 e NVIDIA GTX 295.	Swiss-Prot 56.6 405.506 seqs 146.2mi aa	5.478	Escore	16.087	SWPS3: 4.73x CUDASW: 10.27x
3.8CudaSW++ 2.0 (Y. Liu)	Paralelismo na coluna com otimização SWAT e acesso com padrão *striped*. GPU: NVIDIA GTX 280 e NVIDIA GTX 295.	Swiss-Prot 56.6 405.506 seqs 146.2mi aa	5.478	Escore	29.700	CudaSW++1.0: 1.77x

Tabela 3.1: Artigos sobre o algoritmo Smith-Waterman com *Affine Gap*

Capítulo 4

Arquiteturas de Processadores Gráficos

As Unidades de Processamento Gráfico (GPU - *Graphics Processing Unit*) são microprocessadores com funções otimizadas para acelerar a renderização gráfica. A aceleração se deve 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 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) [33], no qual as rotinas gráficas eram todas embarcadas, não permitindo que programadores criassem rotinas personalizadas.

As gerações seguintes de GPUs (1980-1990)[33] flexibilizaram o *pipeline* permitindo que sequências de instruções executassem em algum dos estágios do *pipeline* das GPUs [34, 35]. 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 criadas para permitir a codificação dos *shaders*. A primeira linguagem que se tem conhecimento, datada de 1984, é a Shade Trees Language [36], 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 [37], que permitiu que estruturas de programação fossem utilizadas. Em 1990, a Pixar publicou a RenderMan Shading Language (RSL) [38] para criação de filmes em três dimensões.

Em 1998, foi proposta a linguagem *pfman* [39], similar a RenderMan. Esta linguagem foi a primeira com compilação destinada a processadores gráficos [40], em especial para a plataforma PixelFlow [41].

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) [33].

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 [42], Sh[43, 44] e CUDA [32].

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 [45]. O paralelismo é obtido por meio dos vários núcleos interconectados no interior da GPU. As GPUs mais modernas chegam a possuir centenas de núcleos.

Assim como os processadores Cell BE [46] e Larrabee [47], as GPUs modernas encontram-se classificadas como arquiteturas paralelas *manycore* [48]. Estas arquiteturas são aquelas que possuem suporte a inúmeros núcleos em um único processador, cuja infraestrutura (interconexão, hierarquia de memória, etc) é desenhada para ser altamente escalável independentemente do número de núcleos [49]. As plataformas *manycore* diferem-se das CPUs 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 [48, 50, 51].

Uma arquitetura *manycore* pode possuir núcleos homogêneos e heterogêneos [48]. 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 inúmeras *threads* [48]. Além disso, podem existir núcleos de propósito específico para aceleração de tarefas em *hardware*, como acontece nas placas gráficas.

Por meio de arquiteturas *manycore*, inúmeras áreas de pesquisa já apresentam resultados bastante expressivos com algoritmos paralelos [52]. Dentre as áreas de pesquisa que utilizam arquiteturas *manycore* podemos citar a Física [53, 54], a Criptografia [55, 56] e a Bioinformática [10, 12, 57–59].

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, a Intel e a IBM (em conjunto com Sony e Toshiba). Este capítulo tratará das arquiteturas adotadas por estes fabricantes. Em especial, a plataforma CUDA (*Compute Unified Device Architecture*), da NVidia, será mais aprofundada por ser a arquitetura escolhida para implementação desta dissertação.

4.1 Larrabee

Em agosto de 2008, a Intel Corporation publicou um artigo descrevendo o Larrabee [47], uma arquitetura *manycore* para processamento visual. Em dezembro de 2009, a Intel anunciou o cancelamento da produção do Larrabee, embora o projeto ainda continue e a plataforma de desenvolvimento será disponibilizada para pesquisa e desenvolvimento[60].

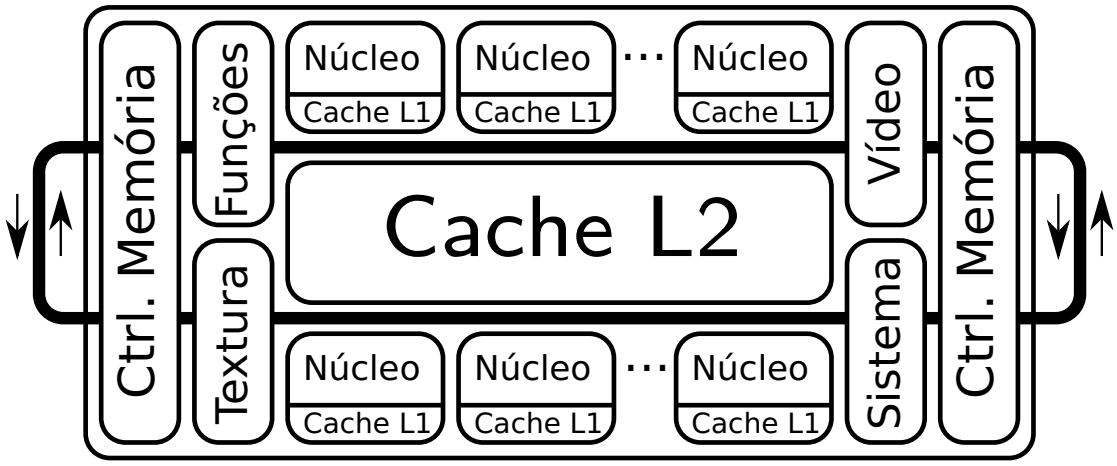


Figura 4.1: Arquitetura interna do Larrabee. Rede de interconexão em anel liga vários núcleos sem execução fora de ordem, mantendo um cache L2 coerente.

A arquitetura original é composta por núcleos de CPUs que executam o conjunto de instruções x86 do Pentium, acrescido de operações vetoriais e instruções escalares especializadas. Embora o Larrabee tenha sido inicialmente desenvolvido com o propósito de competir no segmento de processadores gráficos, sua arquitetura também permite a programação em GPGPU.

A Figura 4.1, adaptada de [61], ilustra a arquitetura interna do Larrabee.

Ao contrário das CPUs mais recentes, os núcleos do Larrabee não utilizam execução de instruções fora de ordem. A decisão de utilizar núcleos sem execução fora de ordem foi justificada comparando o *throughput* de duas configurações: (a) 2 núcleos com execução fora de ordem e (b) 10 núcleos sem execução fora de ordem. Ambas as configurações possuem área física e consumo de energia semelhantes, mas o resultado da configuração (b) foi 40x maior (em FLOPS) que a configuração (a)[47].

O núcleo escolhido para o Larrabee foi baseado no processador Pentium, adicionado o suporte para executar 4 *threads*, com registradores independentes, o que permite um maior aproveitamento do *hardware*. Por exemplo, enquanto uma *thread* aguarda dados da memória durante *cache misses*, outra *thread* pode entrar em execução neste núcleo.

Além disso, também foi adicionada no núcleo uma unidade de processamento vetorial (CPU) com 16 elementos. Cada unidade é capaz de executar instruções em ponto fixo ou ponto flutuante, de precisão simples ou dupla. A Intel publicou um artigo [62] apresentando o conjunto de instruções LRBni (*Larrabee new instructions*), uma extensão do x86 para operações vetoriais.

Outra característica da arquitetura Larrabee é a presença de caches L1 para cada núcleo e cache L2 coerente, havendo uma rede em anel entre os núcleos para manter coerência de dados entre os caches. Desta forma, todos os núcleos do Larrabee possuem acesso transparente ao espaço completo da memória RAM.

Além dos núcleos, dos caches e do anel de interligação, adicionam-se ao processador as interfaces de vídeo e de sistema, assim como unidades especiais dedicadas

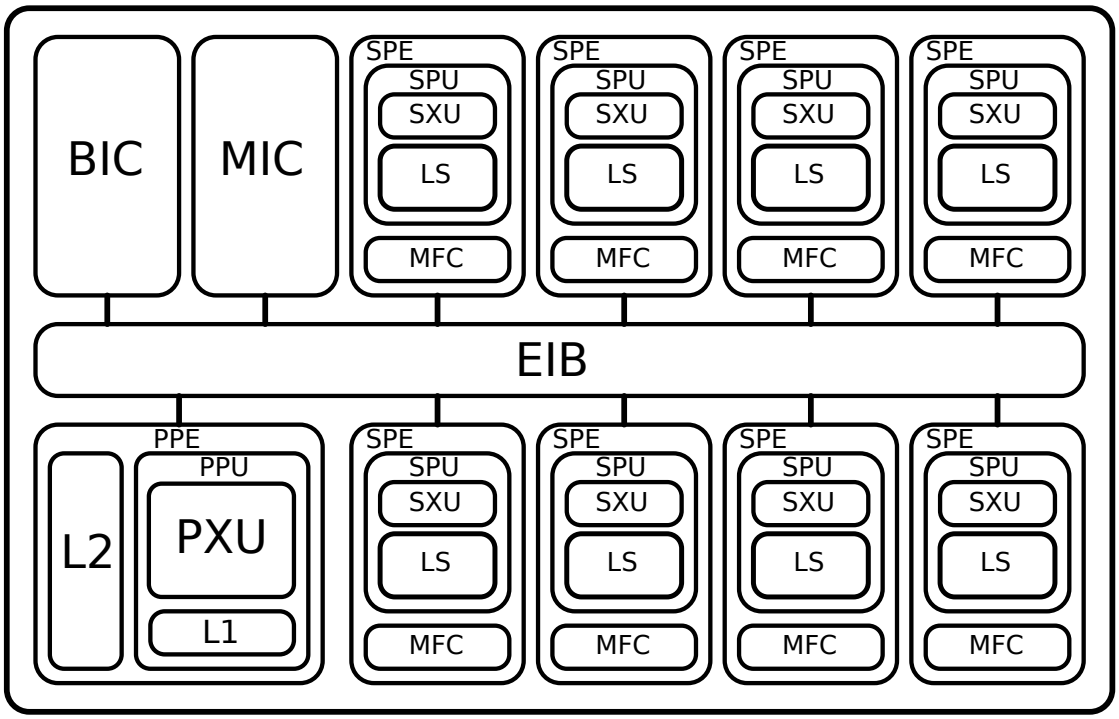


Figura 4.2: Arquitetura interna do Cell BE

para processamento de texturas e de outras funções fixas para processamento gráfico.

4.2 Cell BE

Cell Broadband Engine (Cell BE) é o nome da arquitetura de processadores desenvolvida em conjunto pela IBM, Toshiba e Sony [46]. Esta arquitetura foi construída para suportar aplicações que demandam alto poder computacional, incluindo aplicações científicas, simulações, cálculos físicos e jogos. A Figura 4.2 ilustra os componentes internos da arquitetura Cell BE.

O *Power Processor Element* (PPE) é um núcleo baseado na arquitetura PowerPC de 64-bits, responsável por executar as tarefas de propósito geral. O PPE é compatível com toda a especificação da arquitetura PowerPC, podendo executar sistemas operacionais de 32-bit e 64-bit, assim como rodar aplicações desenvolvidas para esta arquitetura. O PPE contém uma unidade central chamada Power Processor Unit (PPU) e um cache L2. Já o PPU é composto por uma unidade de controle chamada Power Execution Unit (PXU) e um cache L1.

Os *Synergistic Processor Elements* (SPEs) são os núcleos responsáveis por otimizar tarefas computacionalmente intensivas, que atuam sobre vários elementos de dados através de instruções do tipo *single-instruction multiple-data* (SIMD). Um processador Cell BE possui 8 SPEs. Cada SPE executa uma *thread* independente.

Um SPE é composto por uma *Synergistic Processor Unit* (SPU), estrutura responsável pela execução das instruções, e por um *Synergistic Memory Flow Control-*

ler (MFC), unidade responsável por controlar a transferência de dados e sincronização com a memória e com o barramento de interligação. Cada SPU possui acesso coerente à memória compartilhada, incluindo à área de memória mapeada para o espaço de entrada e saída. Além disso, uma SPU contém uma área de armazenamento local (*Local Storage* - LS).

Tanto o PPE como os SPEs são integrados e compartilham diversas estruturas, tais como tabelas de paginação, endereçamento virtual e funções de interrupção.

As demais estruturas do Cell BE são: *Memory Interface Controller* (MIC), unidade responsável pela gerência das controladores de memória Rambus XDR; *Bus Interface Controller* (BIC), barramento de comunicação com componentes de sistema; *Element Interconnect Bus* (EIB), barramento responsável pela interligação de todos os componentes do processador.

O Playstation 3, principal console de jogos da Sony, utiliza o Cell BE como núcleo central de processamento, auxiliado pelo processador gráfico *Reality Synthesizer*, desenvolvido pela NVidia em conjunto com a Sony. No Playstation 3, o Cell BE possui 7 SPEs sendo que 6 são utilizados para as aplicações e 1 para o sistema operacional.

Outro utilizador da arquitetura Cell BE é o RoadRunner, o sétimo supercomputador mais rápido do mundo em Novembro de 2010 [63]. No RoadRunner, existem 12.960 processadores PowerXCell 8i, que é uma versão mais rápida do Cell BE. A capacidade total de processamento do RoadRunner é maior que 1 Petaflops.

4.3 GPUs da NVidia

Compute Unified Device Architecture (CUDA) [32] é uma 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.

Até o momento, existem três gerações de arquiteturas da NVidia CUDA: a série G80, a série GT200 e a série GF100 (Fermi). Cada arquitetura também possui um versionamento, chamado de *compute capability*, composto por dois números x.y (versão *major* e *minor* respectivamente), sendo 1.y para as séries G80 e GT200 e 2.y para a GF100. Variações no número y representam modificações menores na mesma arquitetura, podendo haver novas funcionalidade [32].

Esta seção apresentará a arquitetura de *hardware* das GPUs da NVidia. Um destaque maior será dado à série GT200, visto que as placas utilizadas para os testes experimentais desta dissertação pertencem a esta série de arquitetura.

4.3.1 Componentes de *Hardware*

Cada GPU da NVidia é composta por vários *Stream Processors* (SP), que são as menores unidades de execução da arquitetura. Cada SP possui duas Unidades Lógicas Aritméticas (ALUs) e uma Unidade de Ponto Flutuante (FPU). As instruções

são executadas em ordem e não existe cache. A Figura 4.3 ilustra um SP e seus componentes.

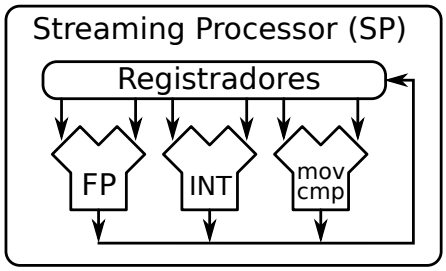


Figura 4.3: *Stream Processors* (SP) [64]

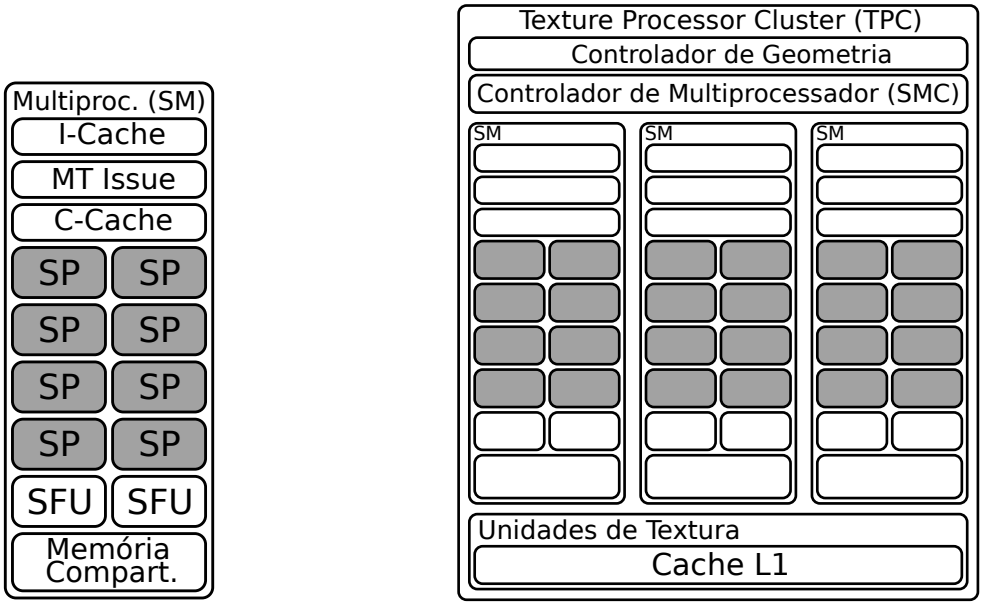
Um *Stream Multiprocessor* (SM) é um agrupamento de SPs, acompanhados de *Special Function Units* (SFUs), uma unidade de controle *MultiThread* (MT issue unit), caches de instrução e de dados e uma memória compartilhada. Cada SM possui 8 SPs e 2 SFUs nas séries G80 e GT200 e 32 SPs e 4 SFUs na série GF100.

O SM executa funções especiais através das suas SFUs, que também possuem unidades de multiplicação para ponto flutuante, sendo que apenas na série GF100 as operações de ponto flutuante são calculadas com precisão dupla. A *MT issue unit* é responsável pelo controle das *threads* e distribuição de instruções para todas as SPs e SFUs. A memória compartilhada possui tamanho de 16KB (G80 e GT200) ou 48KB (GF100) e pode ser utilizada tanto para leitura como para escrita. A figura 4.4(a) descreve um *Stream Multiprocessor*.

Um outro nível de agrupamento existe por meio dos *Texture Processor Clusters* (TPCs). Cada TPC é composto por dois SMs na arquitetura G80 e três SMs na arquitetura GT200, sendo que na arquitetura GF100 não existe este nível de agrupamento, de forma que a função de um TPC é realizada por cada SM. Sabendo que cada SM possui oito SPs, o número de SPs em um TPC é 16 no G80 e 24 no GT200. Além dos SMs, um TPC possui unidades controladoras e unidades de memória para texturas, assim como uma cache para acesso a esta unidade. A Figura 4.4(b) apresenta um TPC da arquitetura GT200 e seus componentes.

Por fim, os vários TPCs (ou os vários SM na arquitetura GF100) geram a unidade de processamento gráfico. Cada placa pode possuir um número diferente de TPCs. Por exemplo, a placa GeForce 9800 GTX é composta por 8 TPCs (totalizando 16 SMs e 128 SPs) e a placa Geforce GTX 280 por 10 TPCs (totalizando 30 SMs e 240 SPs). A Figura 4.5 ilustra a arquitetura GT200 com 8 TPCs (totalizando 24 SMs e 192 SPs).

A última geração da arquitetura CUDA, a GF100 (Fermi) [65], possui ainda outras modificações quando comparadas às linhas G80 e GT200, dentre as quais podemos citar: hierarquia com Cache L1 configurável e Cache L2 unificado; maior quantidade de memória compartilhada por bloco; suporte a *error-correcting code* (ECC) para proteção contra erro de memória em aplicativos sensíveis; otimizações das operações em ponto flutuante com precisão dupla. Além disso, a arquitetura Fermi possui suporte a até 512 núcleos, organizados em 16 SMs com 32 núcleos cada, o que mais que duplica a quantidade de núcleos das GPUs anteriores. A arquitetura Fermi possui capacidade para endereçar até 6 GB de memória RAM.



(a) *Stream Multiprocessor* (SM) [64]

(b) *Texture Processor Cluster* (TPC) [64]

Figura 4.4: Componentes da arquitetura CUDA. Um multiprocessador (a) possui 8 *Stream Processors* (SP). Um *Texture Processor Clusters* (b) é um agrupamento de 3 SMs (24 SPs) na arquitetura GT200.

4.3.2 Linhas de Produtos da NVidia

A NVIDIA comercializa GPUs em várias linhas de placas, destinadas a diferentes ramos do mercado. Dentre elas, podemos citar a linha GeForce (destinada ao ramo de jogos em computadores pessoais), a linha Quadro (focada no mercado de criação gráfica profissional e *Computer-Aided Design* - CAD) e a linha Tesla (criada para GPGPU). A seguir cada uma destas linhas serão brevemente descritas.

GeForce

A linha GeForce, mesmo com placas usualmente mais baratas que as das demais linhas, chegam a possuir capacidade computacional na ordem de Gigaflops. Esta linha é direcionada para computadores pessoais, em especial para aceleração gráfica de jogos. A Tabela 4.1 apresenta a arquitetura, a quantidade de núcleos e o poder computacional em GigaFlops de algumas placas da linha GeForce. Por possuir um bom custo-benefício, a linha GeForce é muito utilizada em pesquisas acadêmicas.

Quadro

A linha Quadro da NVidia 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.

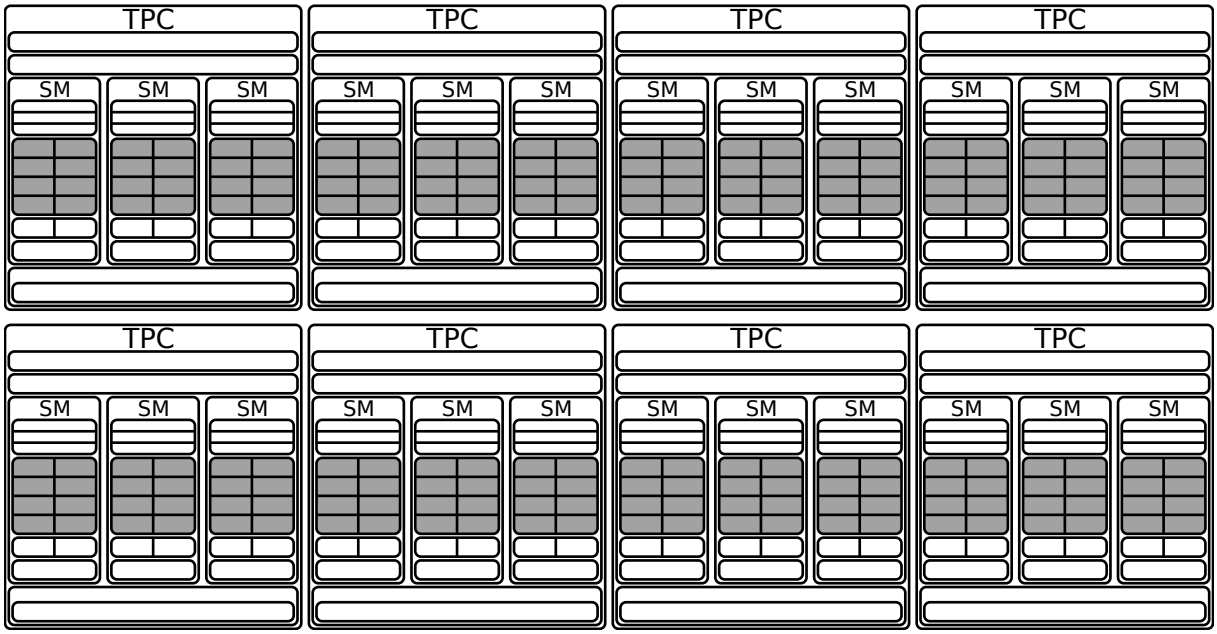


Figura 4.5: Arquitetura GT200 [64]

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. Por ter um foco para aplicações de propósito geral, algumas placas possuem até 4GB de memória dedicada.

Até o momento, três séries do NVIDIA TESLA foram lançadas: TESLA 8, TESLA 10 e TESLA 20. A série TESLA 8, baseada na arquitetura G80, compreende os modelos C870, D870 e S870, sendo que o C870 é uma placa com processador único e 1.5GB de memória RAM, o D870 é um *Desktop* com duas placas C870 e o S870 é um servidor com quatro placas C870. A série TESLA 10, baseada na arquitetura GT200, abrange os modelos C1060, M1060 e S1070, sendo a C1060 a placa com processador único e 4GB de memória RAM, o M1060 é um servidor com uma placa C1060 e o S1070 um servidor com quatro placas C1060. Por fim, a série TESLA 20, baseada na arquitetura Fermi, possui os modelos C2050/C2070, M2050/M2070 e S2050. Os modelos C2050/C2070 e M2050/M2070 são placas (série C) e servidores (série M) com opções de 3GB ou 6GB e o modelo S2050 é um servidor com quatro placas da arquitetura Fermi, totalizando 12GB de memória.

A Tabela 4.2 apresenta um comparativo entre todos os modelos do NVIDIA TESLA.

4.3.3 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 4.6 ilustra as camadas de *software* da arquitetura CUDA.

Modelo	Arquitetura	Núcleos	GFlops
GTX590	GF100	2x480	2304
GTX580	GF100	512	1581
GTX570	GF100	480	1405
GTX480	GF100	480	1345
GTX470	GF100	448	1089
GTX465	GF100	352	855
GTX460	GF100	336	907
GTX295	GT200	2x240	1788
GTX285	GT200	240	1066
GTX275	GT200	240	1011
GTX260	GT200	216	875
GTS250	GT200	128	470
9800 GTX+	G80	128	470
9800 GT	G80	112	336
9600 GT	G80	64	208
9500 GT	G80	32	134
9400 GT	G80	8	67

Tabela 4.1: Comparação entre algumas placas NVIDIA GeForce [66, 67].

Modelo	Nº de GPUs	Arquitetura	Memória	GFlops
C870	1	G80	1,5 GB	518
D870	2	G80	3 GB	1037
S870	4	G80	6 GB	2074
C1060	1	GT200	4 GB	933
M1060	1	GT200	4 GB	933
S1070	4	GT200	16 GB	4147
C2050	1	GF100	3 GB	1288
C2070	1	GF100	6 GB	1288
M2050	1	GF100	3 GB	1288
M2070	1	GF100	6 GB	1288
S2050	4	GF100	12 GB	5152

Tabela 4.2: Comparação entre modelos da linha TESLA [66].

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, exite um *driver*, que executa em modo usuário, que provê uma interface de programação (API) de baixo nível. Dentro desta API, 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

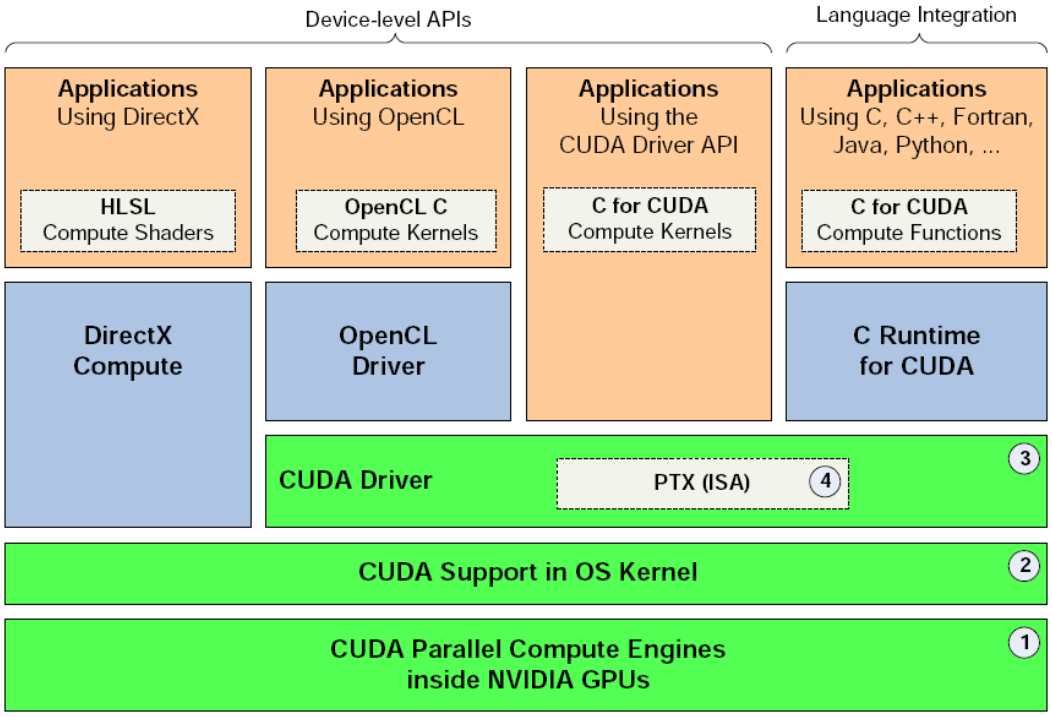


Figura 4.6: Componentes de *software* da arquitetura CUDA [31]

CUDA, que possui chamadas de mais alto nível e bibliotecas com funções mais específicas, como transformadas de Fourrier (FFT) e operações de álgebra linear (CUBLAS); uma camada OpenCL, que é um framework desenvolvido pelo grupo Kronos para aplicações paralelas em ambientes heterogêneos (GPUs, CPUs, etc); camadas para suporte CUDA em outras linguagens, tais como Java, Python e .NET. Além disso, OpenGL e DirectX também possuem integração com a arquitetura CUDA.

4.3.4 Programação em CUDA

Um programa desenvolvido para a arquitetura CUDA [31, 32, 68] possui capacidade de executar instruções na placa de vídeo. O procedimento executado na GPU é chamado de *kernel*. Várias instâncias de um *kernel* são executadas em paralelo através de um conjunto de *threads*. Este conjunto de *threads* é organizado em blocos de *threads* e em grids de blocos de *threads*. Os blocos podem possuir 1, 2 ou 3 dimensões enquanto os grids podem possuir 1 ou 2 dimensões.

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.

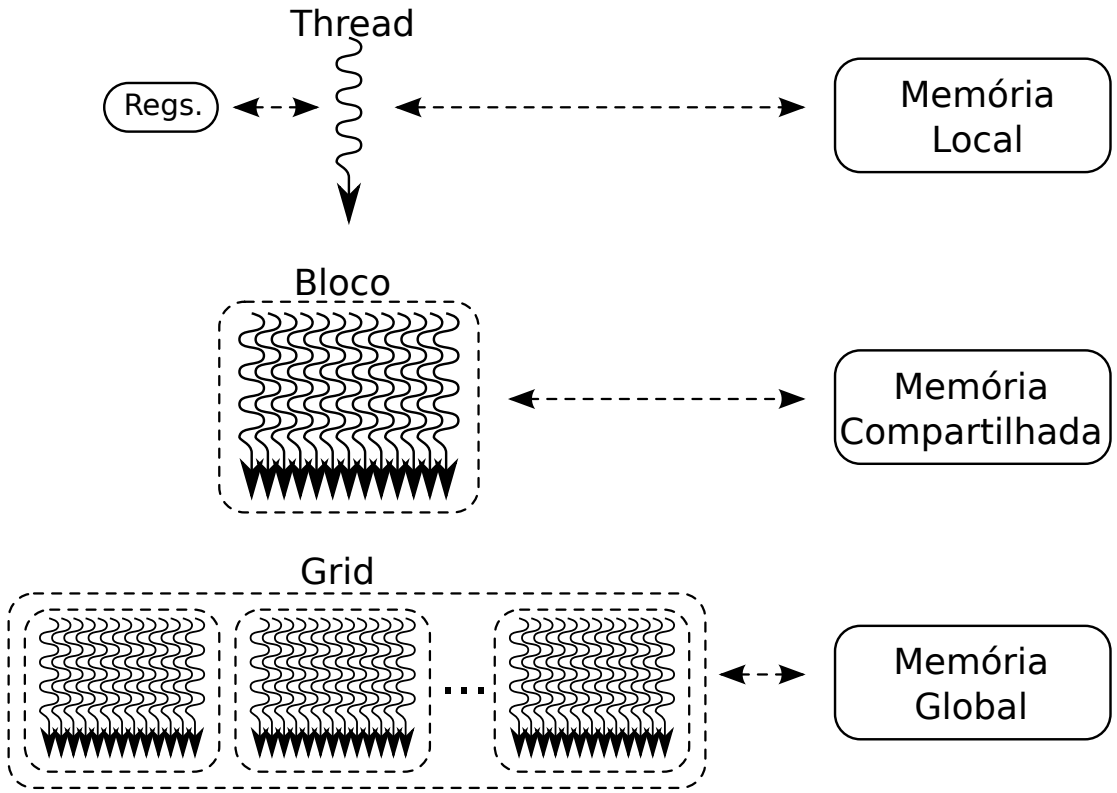


Figura 4.7: Hierarquia de memória da arquitetura CUDA.

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 cautelosa para não gerar inconsistências.

A Figura 4.7 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 rápida de tamanho 64KB; e a textura, que é uma memória 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 dimensões. Não existem mecanismos de coerência de cache em texturas, então uma operação de escrita na área de memória global mapeada para uma textura pode gerar incoerências no cache.

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 (SIMD). 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. 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. A 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 feito 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 $\lll T, B \ggg$. Este par (B, T) é chamado de configuração de execução.

No Programa 4.1, 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`$\lll 1, N \ggg$.

Programa 4.1 Exemplo de código em CUDA.

```
// 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 5

Projeto do CUDAlign

Nos últimos anos, novas tecnologias permitiram um aumento significativo na velocidade de sequenciamento genômico. Por causa disso, as bases de dados genômicos crescem cada vez mais, assim como o número de operações sobre dessas bases. Sendo assim, o processo de comparação deve ser otimizado para executar no menor tempo possível.

Na literatura, as ferramentas que conhecemos que fazem comparação em GPU (Capítulo 3) foram projetadas para comparar uma única sequência com um banco de sequências, utilizando técnicas de paralelismo para acelerar o tempo de processamento. As sequências utilizadas como entrada dessas implementações são normalmente sequências de aminoácidos, que raramente ultrapassam o tamanho de 50 mil resíduos. Dessa forma, essas implementações possuem códigos otimizados para tratar sequências pequenas, não permitindo que duas sequências longas de DNA sejam comparadas (estas podem possuir tamanho na ordem de 30 milhões de bases).

O objetivo desta dissertação é propor uma ferramenta, chamada CUDAlign, que seja capaz de comparar sequências longas de DNA utilizando o algoritmo Smith-Waterman com *affine gap* (Seção 2.2) e com memória linear. A plataforma escolhida para a sua implementação foi a arquitetura CUDA da NVidia (Seção 4.3).

Para permitir o seu uso em diversas situações práticas, alguns requisitos do projeto foram definidos para tornar o CUDAlign suficientemente flexível:

- **Limite no tamanho das sequências**: O único limite que deve existir é a capacidade de memória global da placa de vídeo. Até onde sabemos, as implementações atuais do algoritmo Smith-Waterman (SW) em GPU (Capítulo 3) restringem o tamanho das sequências de entrada para valores abaixo de 60.000 resíduos, o que permite que memórias mais rápidas da GPU possam ser utilizadas no lugar da memória global. Entretanto, sequências maiores não podem ser comparadas nessas implementações.
- **Valor máximo do escore de similaridade**: O valor do escore poderá alcançar números da ordem de 2^{32}, sendo necessário calcular a matriz com precisão de 32 bits. Esse requisito é consequência do fato de que as sequências podem possuir milhões de bases, o que resultaria em um escore bastante elevado. Por exemplo, a comparação de duas sequências idênticas com 100 MBP geraria um escore de 100.000.000, considerando que um *match* possua escore $+1$.

Nesse caso, não seria possível calcular a matriz com precisão de 8 ou 16 bits sem gerar *overflow* ou causar imprecisão nos valores das células da matriz.

- **Memória Linear**: A quantidade de memória utilizada deverá ser da ordem $O(m + n)$, sendo m e n os tamanhos das duas sequências. Caso se utilizasse algoritmos com memória quadrática $O(mn)$, sequências longas jamais poderiam ser comparadas. Por exemplo, a comparação de sequências de 30 MBP necessitaria de 3.6 Petabytes de memória para armazenar toda a matriz, o que é um valor inviável nas arquiteturas atuais.
- **Tempo de execução na ordem de GCUPS**: A implementação deve processar na ordem de bilhões de células por segundo (GCUPS - *Giga Cells Updates per Second*). Essa escolha se deve ao fato de que outras implementações de SW em GPU atingem desempenho maiores que 1 GCUPS, tornando um bom referencial de desempenho (Seção 3.9). Como exemplo, se duas sequências de 10 MBP fossem comparadas com um desempenho de 10 GCUPS, seriam necessárias menos de 3 horas para realizar a tarefa, o que é um tempo extremamente viável considerando o enorme tamanho da matriz calculada.
- **Escalabilidade**: A implementação deve ser escalável, mantendo a proporção de desempenho (na ordem de GCUPS) mesmo para sequências longas, desde que, obviamente, os tamanhos das sequências estejam dentro da capacidade de memória global da placa de vídeo.
- **Recuperação por *Checkpoint***: Em caso de interrupção na execução, deve ser possível retomar a execução a partir de um estado anteriormente salvo (*checkpoint*). Dessa forma, evita-se que interrupções não programadas (queda de energia, travamento da máquina, etc.) reiniciem completamente o processamento. Outra vantagem em permitir recuperação por *checkpoint* é a possibilidade de parcelar a execução em vários períodos, o que flexibiliza o uso do CUDAlign em situações nas quais o hardware só esteja disponível em horários restritos.

O restante deste capítulo irá apresentar os detalhes do projeto do CUDAlign, incluindo a forma na qual o paralelismo é explorado e as otimizações que permitem a execução eficiente por meio de GPUs. Para simplificar as explicações que se seguem, as matrizes H, E e F (utilizadas no cálculo do *affine gap* - Seção 2.3) serão consideradas como componentes de uma única matriz de programação dinâmica. Apenas faremos menção dos componentes H, E e F quando forem relevantes para a explicação.

5.1 Técnicas de Paralelismo no CUDAlign

O CUDAlign foi projetado utilizando a técnica de *wavefront* (Seção 3.1) para obter paralelismo. Essa técnica consiste em processar iterativamente cada anti-diagonal da matriz, de forma que cada célula da mesma anti-diagonal possa ser processada paralelamente. O mesmo conceito pode ser aplicado agrupando células em blocos,

formando anti-diagonais de blocos que também podem ser processados paralelamente.

No CUDAlign, 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 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. Caso a divisão $\frac{n}{C}$ não seja exata, o número de colunas relativo ao resto da divisão deverá ser redistribuído nos blocos, o que pode gerar blocos com uma coluna a mais. Caso m não seja divisível por R, a última fileira de blocos no final da matriz será completada com linhas de preenchimento, que servirão para totalizar o número R de linhas por bloco. Considerando essas observações, será assumido, daqui em diante, que n é divisível por C e m é divisível por R.

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 de execução em CUDA (Seção 4.3.4), representada por $\lll B, T \ggg$ 4.3.4. 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 resíduos ($m = 36$ e $n = 36$). Além disso, suponha que tenhamos 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. Este exemplo está ilustrado na Figura 5.1.

Feito este primeiro agrupamento, os blocos são novamente agrupados em anti-diagonais. A anti-diagonal D_k ($k = 0, 1, 2, ...$) 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.

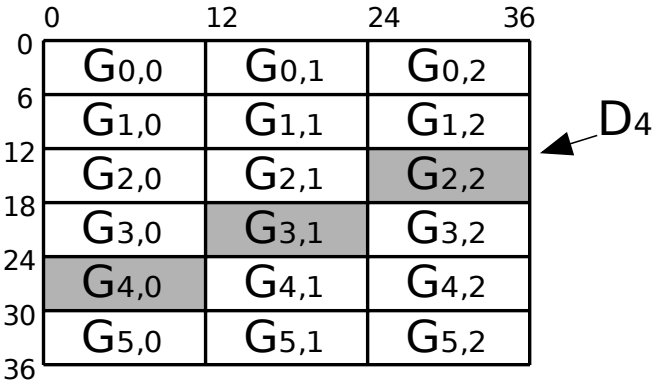


Figura 5.1: Divisão da matriz em blocos. A diagonal externa D_4 está com blocos em destaque.

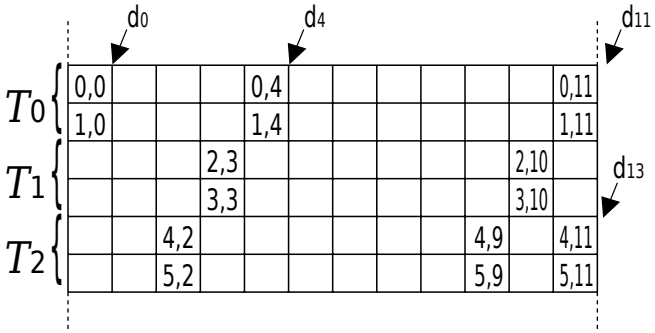


Figura 5.2: Bloco retangular com 3 *threads*. Cada *thread* é responsável por 2 linhas do bloco. As diagonais internas d_0, d_4, d_{11} e d_{13} estão indicadas na figura.

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 $(k = 0, 1, 2, ...)$ 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 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.

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 Execução interna da *thread*

O CUDAlign executa uma *thread* para cada grupo de α linhas do bloco. A execução interna de uma *thread* é feita da esquerda para a direita, sempre em grupos de α células verticais. Ao final de cada iteração, executa-se uma chamada à primitiva de barreira `__syncthreads()`, de forma a sincronizar a execução de cada diagonal interna. Sendo assim, as *threads* processam um grupo de α células e, em seguida, aguardam todas as demais *threads* terminarem o cálculo da diagonal interna.

Para uma *thread* processar um grupo de α células, ela precisa ler da memória os valores das α células à esquerda, da célula imediatamente superior e da célula da diagonal. As seguintes definições serão utilizadas para representar as células manipuladas em cada iteração:

- K^{α}_{cur} **- Células atuais**: Chamaremos de K^{α}_{cur} o conjunto de α células $(K_{cur.1}, K_{cur.2}, \ldots, K_{cur.\alpha})$ verticalmente agrupadas que estão sendo calculadas por uma *thread* em uma determinada iteração. A ordem de cálculo deve ser obrigatoriamente de cima para baixo, ou seja, no sentido da célula $K_{cur.1}$ à célula $K_{cur.\alpha}$.
- K^{α}_{left} **- Células anteriores**: As células K^{α}_{left} são o conjunto de α células $K = (K_{left.1}, K_{left.2}, \ldots, K_{left.\alpha})$ verticalmente agrupadas que foram calculadas por uma *thread* na iteração anterior. As células K_{left} encontram-se imediatamente à esquerda das células K_{cur}.
- K_{up} **- Célula superior**: A célula imediatamente acima da célula $K_{cur.1}$ é chamada de K_{up}.
- K_{diag} **- Célula diagonal**: A célula imediatamente acima da célula $K_{left.1}$ é chamada de K_{diag}. Essa célula encontra-se à diagonal da célula $K_{cur.1}$.
- **Vizinhança-K**: A agregação $K = (K^{\alpha}_{cur}, K^{\alpha}_{left}, K_{diag}, K_{up})$ é chamada de vizinhança-K e representa todas as dependências de memória em uma determinada iteração. A Figura 5.3 ilustra as células da vizinhança-K.

Durante a execução de uma *thread*, sucessivos grupos de α células são calculados em cada iteração, ou seja, cada iteração apresenta uma nova vizinhança-K. Para calcular as células K^{α}_{cur}, as células K^{α}_{left}, K_{up} e K_{diag} precisam ser corretamente lidas da memória. Para cada nova iteração, os valores de K^{α}_{left} são obtidos dos valores K^{α}_{cur} anteriores, assim como o valor de K_{diag} é obtido de K_{up} da iteração anterior.

Através do agrupamento de α células, a transferência de valores entre as variáveis K^{α}_{cur} e K^{α}_{left} pode ser feita de maneira agrupada, o que permite a otimização do acesso à memória. Sabendo que cada célula armazena 32 bits e que a palavra das GPUs agrupa 4 componentes de 32 bits (totalizando 128 bits), melhores resultados podem ser percebidos caso α seja múltiplo de 4.

As células da vizinhança-K são intensivamente utilizadas e precisam ser armazenadas em registradores. O valor K_{up} deve ser lido da *thread* superior. Caso a *thread* superior esteja no mesmo bloco, carrega-se o valor de K_{up} da memória compartilhada, mas se a *thread* superior estiver em um outro bloco, a leitura de

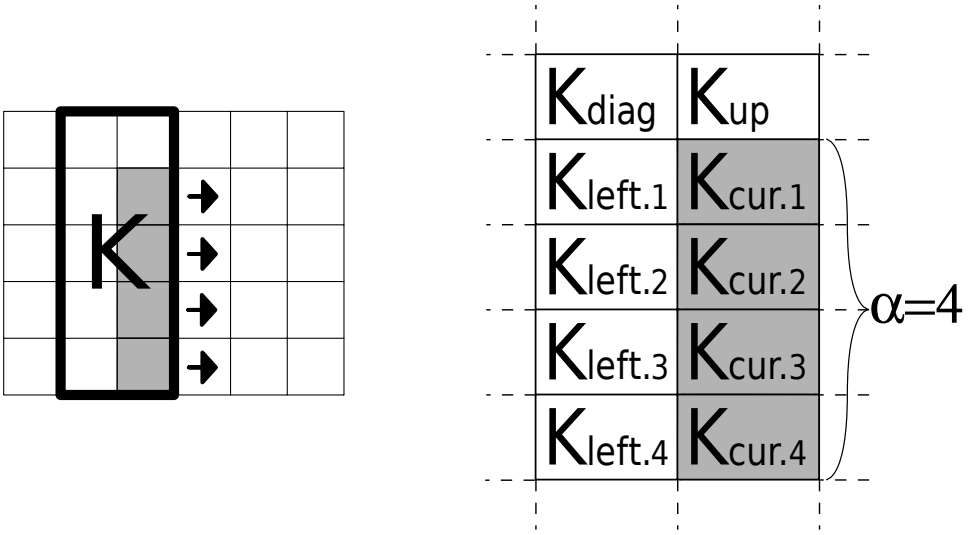


Figura 5.3: Vizinhança-K

K_{up} deverá ser feita por meio da memória global, pois blocos diferentes não podem compartilhar dados pela memória compartilhada.

De maneira similar, a célula inferior da vizinhança-K (i.e. $K_{cur.\alpha}$) precisa ser armazenada na memória para que ela possa ser lida pela *thread* inferior. Se a *thread* inferior estiver no mesmo bloco, o valor é armazenado na memória compartilhada, mas se a *thread* inferior estiver em outro bloco, a memória global é utilizada.

A área da memória global destinada para trocar os valores de $K_{cur.\alpha}$ (bloco superior) para K_{up} (bloco inferior) será chamada de *barramento horizontal*. O tamanho deste vetor deve ser proporcional ao tamanho da sequência $|S_1| = n$. Sabendo que a primeira linha da matriz de programação dinâmica é preenchida com zeros, o barramento horizontal deve ser iniciado com zeros.

Ao final do processamento de um bloco, as *threads* precisam passar os valores das últimas vizinhanças-K para as *threads* do próximo bloco à direita. No início da execução de um bloco, os valores de K^{α}_{left} e K_{diag} são obtidos, respectivamente, das últimas células K^{α}_{cur} e K_{up} do bloco imediatamente à esquerda.

Esta transferência também deve ser feita através da memória global, pois *threads* de blocos distintos não conseguem transferir valores por meio da memória compartilhada [32]. A área da memória global destinada à transferência destes valores será chamada de *barramento vertical*. O tamanho deste vetor deve ser proporcional ao número de *threads* T multiplicado pelo número de blocos B.

5.3 Estruturas em memória

Nesta seção estão descritas as estruturas armazenadas em memória e suas principais características. Ao se falar de matriz de programação dinâmica (DP), cada célula possui três componentes: H, F e E (Seção 2.3). Cada componente possui 32 *bits* (4 *bytes*).

As estruturas armazenadas em memória pelo CUDAlign são as seguintes:

- **Sequências**: As sequências são armazenadas em textura (memória somente-leitura). Cada nucleotídeo ocupa um *byte* em memória. Para evitar o acesso indevido à memória no final da matriz (caso de indivisibilidade pelo tamanho dos blocos), acrescentam-se $B \times T$ *bytes* nulos no final das sequências. O total de memória ocupada pelas duas sequências é de $m + n + 2 \times B \times T$ *bytes*.

- **Vizinhança-K**. A vizinhança-K é armazenada nos registradores de cada *thread*. Para calcular as células K^{α}_{cur} de uma vizinhança-K, os seguintes componentes precisam estar em memória: os componentes H e E das células K^{α}_{left}; o componente H da célula K_{diag} e os componentes H e F da célula K_{up}. Os resultados das células K^{α}_{cur} são armazenados em registradores, sendo que apenas o componente H das α células K_{cur} precisam de novos registradores. Os demais componentes são armazenados nos mesmos registradores usados para leitura. Sendo assim, uma vizinhança-K necessita de $(3 \times \alpha + 3)$ registradores de 32 *bits*.

- **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. A estratégia utilizada pelo CUDAlign foi de utilizar dois vetores na memória compartilhada, alternando a função de leitura e escrita. Desta forma não há a necessidade de sincronizar os momentos de leitura e de escrita à memória compartilhada. Como existem 2 vetores com 2 componentes por célula, são necessários $16 \times T$ *bytes* de memória compartilhada por bloco.

- **Barramento Horizontal**: Os valores da última linha de cada bloco são armazenados nesta área da memória global para que o bloco imediatamente inferior carregue 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 (memória cache somente-leitura e não-coerente). Como a leitura de um valor do barramento é feita sempre antes da escrita (devido ao deslocamento da diagonal interna), o cache não apresenta dados incoerentes. Apenas os valores relativos aos componentes H e F da matriz precisam ser salvos no barramento horizontal, pois os *gaps* que representam o componente E não interferem no bloco imediatamente inferior. Cada componente ocupa 4 *bytes* e o barramento horizontal precisa ter o tamanho da sequência S_1. Logo, o tamanho total do barramento horizontal é de $8 \times n$ *bytes*.

- **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. Cada *thread* deve armazenar: os componentes H e E das células K^{α}_{cur}; o componente H da célula K_{up} e o componente F da célula $K_{cur.\alpha}$. Cada componente ocupa 4 *bytes*. Sendo assim, o barramento vertical precisa de $8 \times \alpha + 8$ *bytes* por *thread*. Sabendo que o CUDAlign executa $B \times T$ *threads*, o tamanho total do barramento vertical é de $(8 \times \alpha + 8) \times B \times T$ *bytes*.

Estrutura	Tamanho	Tipo
Sequências	$m + n + 2 \times B \times T$ *bytes*	Textura
Vizinhança-K	$(3 \times \alpha + 3)$ *registradores*	Registradores
Memória Compartilhada	$16 \times T$ *bytes*	Memória Compartilhada
Barramento Horizontal	$8 \times n$ *bytes*	Memória Global + Textura
Barramento Vertical	$(8 \times \alpha + 8) \times B \times T$ *bytes*	Memória Global

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

A Tabela 5.1 apresenta um resumo das estruturas em memória. O total de memória global utilizada pelo CUDAlign é de $9 \times n + m + (8 \times \alpha + 10) \times B \times T$, o que permite concluir que o algoritmo utiliza memória na ordem de $O(m + n)$. Visto que o CUDAlign permite a entrada de sequências muito grandes, o tamanho do barramento horizontal torna-se a maior restrição do CUDAlign. Podemos então estimar que a quantidade de memória global utilizada pelo CUDAlign 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.

5.4 Otimizações

O desempenho de um programa na arquitetura CUDA é determinado pelo paralelismo permitido pela plataforma. Entretanto, detalhes de implementações podem influenciar bastante na capacidade de explorar o paralelismo do hardware. Esta seção irá detalhar vários pontos que foram analisados para melhorar o desempenho do CUDAlign.

5.4.1 Delegação de Células

Conforme visto nas seções 5.1.1 e 5.1.2, 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 apresentada na Seção 5.1.2, 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, foi proposta uma otimização chamada de *delegação de células*. Em vez de processar todas

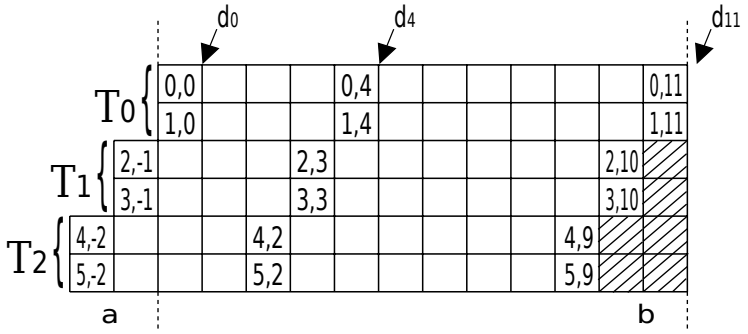


Figura 5.4: Bloco não retangular com 3 *threads*. O processamento continua enquanto o paralelismo da diagonal interna for máximo (até a diagonal d_{11}). Células hachuradas indicam as células pendentes, que serão processadas por outro bloco na próxima diagonal externa. Células com coluna negativa indicam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

as células de um bloco, o CUDAlign 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. Mais precisamente, a *thread* T_k deixa αk células pendentes e o total de células pendentes para o bloco é calculada pela fórmula 5.1.

$$\sum_{k=0}^{T-1} \alpha k = \frac{\alpha}{2} T(T-1) \tag{5.1}$$

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.

Observe que as células das colunas mais à direita da matriz também são delegadas, mas em vez de serem computadas pelos blocos imediatamente à direita, elas são processadas pelos blocos mais à esquerda da próxima diagonal externa. O mesmo ocorre no final da matriz, sendo que alguns blocos de preenchimento precisam ser criados para completar o processamento de toda a matriz. Por causa da delegação de células, uma diagonal externa adicional precisa ser criada para o último bloco, aumentando o número de diagonais externas para $|D| = B + \frac{m}{\alpha T}$.

5.4.2 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 que o escalonador da GPU pode executar os blocos em

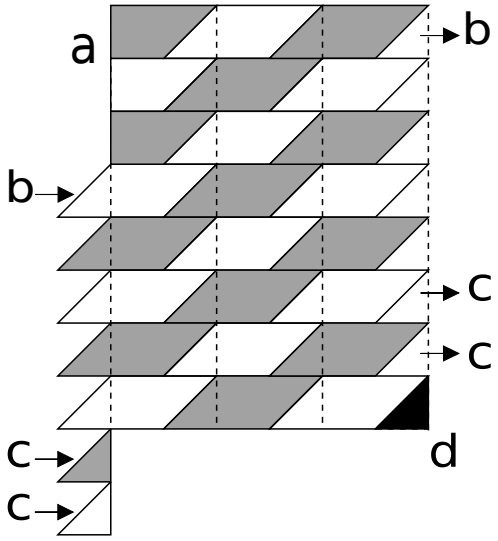


Figura 5.5: Delegação de células entre os blocos. A região (a) indica os blocos iniciais da matriz. O bloco (b) à direita da figura delega células para o bloco (b) à esquerda da matriz. Os blocos à direita indicados por (c) delegam células para blocos que preenchem o lado esquerdo (c) da matriz. Um bloco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

qualquer ordem, em paralelo ou em série, podendo até mesmo distribuí-los em diferentes placas de vídeo. 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 entres 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, esse procedimento é feito dividindo a execução do bloco em duas fases: a fase curta e a fase longa.

- **Fase Curta**: O objetivo dessa fase é terminar de processar todas as células pendentes. Para isso, processam-se as $T - 1$ primeiras diagonais internas

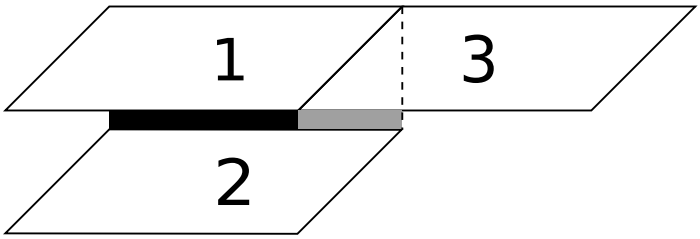


Figura 5.6: Problema de dependência de dados entre blocos. Caso o bloco 2 execute antes do bloco 3, o bloco 2 receberá valores indeterminados da área cinza.

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. Áreas com final $.1$ indicam as fases curtas e áreas com final $.2$ indicam as fases longas. No exemplo apresentado pela figura, temos 4 invocações de *kernels*, feitas na seguintes ordem:

1. calcula-se a área 1.1;
2. calcula-se a área 1.2;
3. calculam-se as áreas 2.1 e 3.1;
4. calculam-se as áreas 2.2 e 3.2;

Como pode ser observado na figura 5.7, o problema da dependência fica resolvido, pois a divisão de fases cria uma forma de sincronismo entre os blocos. Desta forma, a cada cálculo de um bloco todas as suas dependências já foram calculadas previamente.

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 condição $n \geq 2B(T-1)$ seja satisfeita, conforme demonstram as Inequações 5.2. Por simplicidade, esta condição será arredondada para apenas $n \geq 2BT$.

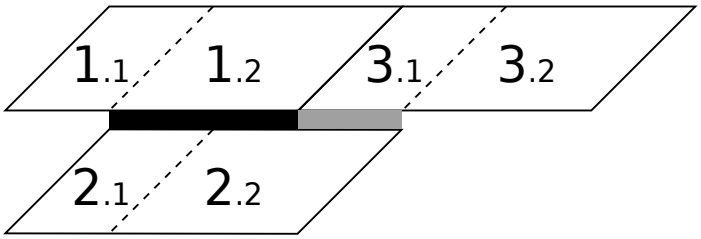


Figura 5.7: Execução do bloco dividida em duas fases. A fase curta possui sufixo $.1$ e a fase longa possui sufixo $.2$.

$$\begin{aligned} \frac{n}{B} - (T-1) &\geq T-1 \\ \frac{n}{B} &\geq 2(T-1) \\ n &\geq 2B(T-1) \end{aligned} \tag{5.2}$$

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 não existe dependência de dados com um único bloco. Sendo assim, o CUDAlign é capaz de alinhar sequências de qualquer tamanho, bastando alguns ajustes dos parâmetros B e T, caso necessário.

5.4.3 *Loop Unrolling*

Como foi explicado na Seção 5.2, o processamento de uma *thread* é feito da esquerda para direita, sendo que a vizinhança-K é constantemente atualizada para representar as células que serão processadas em cada iteração. O pseudocódigo apresentado no Algoritmo 1 descreve simplificadamente o laço principal. Uma descrição bem mais detalhada do algoritmo completo será dada na Seção 5.5 e por agora será explicado apenas o necessário para o entendimento do *loop unrolling*.

A função $K^{\alpha}_{cur} = SW(K^{\alpha}_{left}, K_{diag}, K_{up})$ recebe como parâmetros as células anteriores (K^{α}_{left}), a célula diagonal (K_{diag}) e a célula superior (K_{up}), retornando as células atuais K^{α}_{cur} como saída. A célula superior K_{up} é obtida pela função $LoadUp$. A cada iteração, realizam-se as atribuições $K^{\alpha}_{left} \leftarrow K^{\alpha}_{cur}$ e $K_{diag} \leftarrow K_{up}$, pois as células K^{α}_{cur} processadas na iteração anterior tornam-se as células anteriores K^{α}_{left} da iteração atual.

Para evitar que várias atribuições $K^{\alpha}_{left} \leftarrow K^{\alpha}_{cur}$ e $K_{diag} \leftarrow K_{up}$ sejam feitas, o laço que percorre todas as diagonais internas é desenrolado (técnica de *loop unrolling*) conforme descrito no Algoritmo 2. Observe que a saída da primeira chamada à

Algorithm 1 Laço Original

1: **for** $d_k = 0..|d| - 1$ **do**
2: K_{up} := LoadUp
3: K^{α}_{cur} := SW($K^{\alpha}_{left}, K_{diag}, K_{up}$)
4: $K^{\alpha}_{left} := K^{\alpha}_{cur}$
5: $K_{diag} := K_{up}$
6: **end for**

função SW é entrada da segunda chamada e vice-versa. Obviamente as células K'_{up} e K''_{up} precisam ser carregadas da memória antes do cálculo da função SW.

Algorithm 2 Laço Desenrolado

1: $K''_{up} := K_{diag}$
2: $K''^{\alpha}_{cur} := K^{\alpha}_{left}$
3: **for** $d_k = 0..\frac{|d|}{2} - 1$ **do**
4: K'_{up} := LoadUp
5: K'^{α}_{cur} := SW($K''^{\alpha}_{cur}, K''_{up}, K'_{up}$)
6: K''_{up} := LoadUp
7: K''^{α}_{cur} := SW($K'^{\alpha}_{cur}, K'_{up}, K''_{up}$)
8: **end for**

Utilizando o *loop unrolling*, reduz-se o número de iterações pela metade e elimina-se a necessidade das atribuições $K^{\alpha}_{left} \leftarrow K^{\alpha}_{cur}$ e $K_{diag} \leftarrow K_{up}$ em todas as iterações. O mesmo conceito é aplicado alternando os vetores de leitura e escrita da memória compartilhada.

Devido à diminuição de instruções executadas ao longo de todas as iterações, o *loop unrolling* reduz o tempo de execução de um bloco. Deve-se apenas ter cuidado na hora de dividir o número de iterações, pois caso o número original seja ímpar, uma iteração adicional sem *loop unrolling* deverá ser feita.

5.4.4 Tolerância a Falhas

Um dos requisitos propostos para o CUDAlign é a capacidade de recuperar a execução em caso de interrupção do programa. A recuperação da execução não precisa ser feita para o estado exatamente anterior à interrupção, mas aceita-se que um estado anterior próximo seja utilizado. Desta forma, pode-se limitar a perda caso ocorra uma parada não programada sem, contudo, onerar excessivamente o desempenho com o salvamento contínuo dos estados de execução.

Para ser possível a recuperação de um estado anterior, salva-se periodicamente o melhor escore, a sua posição de ocorrência e alguns segmentos de células da matriz. Duas estratégias poderiam ser adotadas para salvar os segmentos da matriz:

- **Armazenamento completo dos barramentos**: Nesta estratégia, os barramentos horizontal e vertical (Seção 5.3) são completamente salvos em disco. Para recuperar a execução, carregam-se os barramentos e reinicia-se o processamento a partir da respectiva diagonal externa.

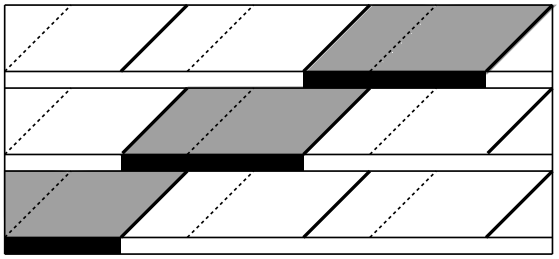


Figura 5.8: Barramento Horizontal dividido em várias linhas da matriz. Para salvar uma linha completa da matriz são necessárias várias iterações (i.e. várias diagonais externas).

- **Armazenamento de uma linha completa**: Esta abordagem salva uma linha inteira da matriz de programação dinâmica. Deve-se atentar ao fato de que o barramento horizontal representa células de diversas linhas da matriz, pois ele armazena dados dos vários blocos de uma diagonal externa, o que pode ser visto na Figura 5.8. Para salvar uma linha completa da matriz, deve-se salvar segmentos do barramento horizontal durante várias diagonais externas. Para recuperar a execução, carrega-se esta linha na memória e inicia-se o processamento como se esta linha fosse a primeira linha da matriz.

A estratégia utilizada pelo CUDAlign foi de salvar uma linha completa da matriz, pois desta forma permite-se que o trabalho salvo possa ser continuado por uma instância do CUDAlign com diferentes número de blocos ou *threads*, permitindo inclusive que diferentes GPUs sejam utilizadas. Caso fosse utilizada a estratégia de salvar os barramentos, todos os parâmetros de execução (número de blocos, de *threads*, etc) deveriam ser mantidos após a interrupção.

Outra vantagem observada em salvar a linha completa é que trabalhos futuros poderão aproveitar o mesmo mecanismo para salvar várias linhas especiais, o que permitiria utilizar variantes do algoritmo Myers-Miller para obter o alinhamento completo.

5.5 Pseudocódigo

O algoritmo 3 apresenta um pseudocódigo para os *kernel*s do CUDAlign. Esse código é executado com as configurações de execução $\lll B, T \ggg$, ou seja, o *kernel* será executado por T *threads* em paralelo para cada um dos B blocos.

O procedimento chamado *kernel* é invocado com o parâmetro D_k (linha 1), que representa qual a diagonal externa está sendo atualmente processada ($0 \leq D_k \leq (\frac{m}{\alpha T} + B - 1)$).

Na linha 2, obtêm-se as coordenadas (i, j) da célula superior esquerda a partir da qual inicia-se o processamento desta *thread*. Se j for negativo (linha 3), os valores devem ser ajustados para que as células delegadas sejam recebidas de αBT linhas acima (linhas 4-5). Na linha 7, os valores das primeiras células são inicializados com os valores das células delegadas pelo bloco anterior. Os nucleotídeos da

sequência relativos às α linhas desta *thread* são carregados na variável b^{α} na linha 8.

As diagonais internas são calculadas no laço principal (linhas 9-26). A condição na linha 10 garante que as células que estão fora da matriz não sejam processadas. Na linha 11, a célula K_{up} é carregada da *thread* superior, que armazenou a célula $K_{cur.\alpha}$ na iteração anterior. Como explicado na Seção 5.2, a célula K_{up} será lida da memória global (barramento horizontal) se a *thread* for a primeira do bloco (i.e. `threadIdx.x` for zero), caso contrário a célula será lida da memória compartilhada.

A computação da célula, baseada na equação de recorrência do Smith-Waterman (SW), é realizada na linha 12 para todas as células K^{α}_{cur}. Sempre que uma célula é calculada, verifica-se se um novo escore máximo foi encontrado (linha 14). Caso algum dos valores em K^{α}_{cur} possua o melhor escore, o valor máximo e a sua posição são ambos atualizados na memória global. Nas linhas 15-16 a inicialização da próxima iteração é feita. Na linha 18, o índice j é incrementado, representando a nova coluna em processamento. Caso j exceda o limite da matriz, a *thread* ajusta a posição (i, j) para a primeira coluna (linha 20) e para αBT linhas abaixo (linha 21). Adicionalmente, o algoritmo reinicializa os valores de K (linha 22) e os nucleotídeos b^{α} (linha 23).

Todas as *threads* do bloco são sincronizadas na linha 25, garantindo a consistência da memória compartilhada.

Após todas as diagonais internas serem processadas, delegam-se na linha 27 alguns valores de K para serem processados pelo próximo bloco. A saber, os valores delegados são os componentes H e E das células de K_{cur}, o componente H da célula K_{up} e o componente F da célula $K_{cur.\alpha}$. Estes valores são armazenados na memória global (barramento vertical), de forma que a execução do próximo bloco possa continuar a computação das células delegadas.

Embora o Algoritmo 3 não mostre a divisão em fases curta e longa (Seção 5.4.2), a implementação define dois *kernels* distintos, sendo que na fase curta (kernel.1) o laço principal é feito de $d_k = 0..T - 2$ e na fase longa (kernel.2) a iteração é feita de $d_k = T - 1..|d| - 1$. Além disso, a fase longa não precisa das duas condicionais das linhas 3-6 e 19-24, sendo estas linhas removidas. A implementação também considera o *loop unrolling* descrito na Seção 5.4.3.

O Algoritmo 4 apresenta o pseudocódigo da implementação em CPU. Nas linhas 2-3, as dimensões do grid e dos blocos são inicializados. Por estarmos tratando apenas uma dimensão de *threads*, nós definimos como 1 as outras duas dimensões. O laço das linhas 4-7 iteram através de cada diagonal externa $0 \leq D_k \leq |D| + B - 1$. Cada iteração executa uma fase curta (linha 5) e uma fase longa (linha 6). Como resultado, o Algoritmo 4 retorna o melhor escore a a posição correspondente (linhas 8-9), que foi anteriormente armazenada pela linha 14 do Algoritmo 3.

5.6 Número de Blocos, Threads e Registradores

Após implementado o código do CUDAlign, alguns parâmetros precisam ser ajustados para obtermos um melhor desempenho da aplicação. A escolha dos valores

Algorithm 3 Process Block (GPU)

```
1: procedure KERNEL(D_k)
2:     (i, j) := GETCOORD(D_k, blockIdx.x, threadIdx.x)
3:     if j < 0 then
4:         i := i − αBT
5:         j := |S_1| − j
6:     end if
7:     K := GETDELEGATEDCELL(i)
8:     b^α := S_0[i..i + α − 1]
9:     for d_k = 0..|d| − 1 do
10:        if i ≥ 0 or i < m then
11:            K_up := LOADUP(threadIdx.x)
12:            K^α_cur := SW(b^α, S_1[j], K^α_left, K_diag, K_up)
13:            STOREDOWN(threadIdx.x + 1, K_cur.α)
14:            UPDATEMAXSCORE(K^α_cur, i, j)
15:            K^α_left := K^α_cur
16:            K_diag := K_up
17:        end if
18:        j := j + 1
19:        if j = |S_1| then
20:            j := 0
21:            i := i + B.αT
22:            K := STARTNEWLINE
23:            b^α := S_0[i..i + α − 1]
24:        end if
25:        __syncthreads()
26:    end for
27:    DELEGATECELL(K, i)
28: end procedure
```

ideais é normalmente feita empiricamente, mas algumas regras na escolha podem ser definidas *a priori*. Esta seção mostrará os critérios para a escolha do número B de blocos, o número T de *threads* por bloco e quantidade de registradores por *thread*. O parâmetro B é definido em tempo de execução, mas os valores de T e o número de registradores por *thread* são fixados em tempo de compilação.

No CUDAlign, cada bloco possui um tempo de execução praticamente idêntico. Na arquitetura CUDA, um bloco fica residente no mesmo multiprocessador até que ele termine. Sendo assim, o desempenho pode ser comprometido caso o número de blocos B não seja múltiplo do número de multiprocessadores de uma placa. Por exemplo, a GTX285 possui 30 multiprocessadores. Se nessa placa 31 blocos forem executados e supondo que cada multiprocessador execute um único bloco, então 30 blocos serão executados por vez, o que deixa o último bloco em uma execução separada. Nesse caso teórico, o tempo para processar 31 blocos será muito maior do que o tempo para processar 30 blocos. Por esse motivo, recomenda-se que o número de blocos seja múltiplo do número de multiprocessadores.

Algorithm 4 Process Grid (CPU)

```
1: function PROCESSGRID
2:     grid := (B, 1, 1)
3:     threads := (T, 1, 1)
4:     for D_k := 0..|D| + B − 1 do
5:         KERNEL.1<<< grid, threads >>>(D_k);
6:         KERNEL.2<<< grid, threads >>>(D_k);
7:     end for
8:     (max, max_pos) := GETMAXSCORE
9:     return (max, max_pos)
10: end function
```

O número de registradores e a quantidade de memória compartilhada demandados por um *kernel* limitam o número de *threads* que podem ser executadas em um bloco. Por exemplo, supondo que um *kernel* necessite de 32 registradores e sabendo que a placa GTX285 possui 16.384 registradores por multiprocessador, então cada multiprocessador poderá executar até 512 *threads* concorrentemente.

A arquitetura também permite que um único multiprocessador seja responsável por processar vários blocos simultaneamente, através de um mecanismo de escalonamento. Este mecanismo permite esconder a latência de algumas instruções de Entrada/Saída, pois enquanto um conjunto de *threads* aguarda algum dado da memória, outro conjunto de *threads* poderá ser executado. O guia de melhores práticas do CUDA [68] recomenda que mais de um bloco seja executado no mesmo multiprocessador e que, no total, existam ao menos 192 *threads* ativas no mesmo multiprocessador.

O número máximo de blocos que podem ser executados concorrentemente em um multiprocessador depende do número de *threads*, do número de registradores e a quantidade de memória compartilhada demandados por um *kernel*. Por exemplo, um *kernel* que utilize 32 registradores e 128 *threads* poderá executar quatro blocos em um multiprocessador na GTX285, pois $4 \times 32 \times 128 = 16384$, que é o número máximo de registradores em cada multiprocessador desta GPU. Caso um registrador a mais fosse utilizado, não seria mais possível executar quatro blocos no multiprocessador, pois $4 \times 33 \times 128 = 16896 > 16384$, o que excede o número de registradores permitidos.

Cuidado análogo deve ser tomado com a memória compartilhada. Por exemplo, na GTX285 o tamanho da memória compartilhada é de 16KB por multiprocessador. Conforme visto na seção 5.3, a quantidade de memória compartilhada utilizada pelo CUDAlign é de $16.T$. Isso faz com que o número de *threads* que podem executar em um mesmo multiprocessador seja 1024. Caso se queira executar 4 blocos por multiprocessador, então o número máximo de *threads* por bloco será reduzido a 256.

A quantidade de registradores utilizados por um *kernel* pode ser limitada pela diretiva `__launch_bounds__(maxThreads, minBlocks)`, onde o parâmetro `maxThreads` significa o número máximo de *threads* executado por um bloco e `minBlocks` o número mínimo de blocos que se deseja executar simultaneamente no

mesmo multiprocessador. A diretiva `__launch_bounds__` é utilizada como heurística pelo compilador, que utilizará os parâmetros para determinar o número de registradores.

A seguir estão elencados os principais critérios adotados nas escolhas dos parâmetros B e T:

- o número T de *threads* por bloco deve ser maior que 64 e múltiplo de 32;
- o número de *threads* ativas em um multiprocessador deve ser no mínimo 192.
- um multiprocessador deve executar M blocos concorrentemente, sendo $M \geq 2$;
- o número B de blocos deve ser múltiplo do número SM de multiprocessadores multiplicado pelo número M de blocos concorrentes em um multiprocessador;

5.7 Previsão de desempenho

Esta seção apresenta um método para prever o tempo de execução do CUDAlign para duas sequências de tamanhos m e n. Algumas constantes desconhecidas serão utilizadas e seus valores serão obtidos nos resultados experimentais. Obviamente, estas constantes terão valores diferentes para cada ambiente testado, restando a esta seção apenas encontrar um modelo de fórmula que permita obter, aproximadamente, o tempo de execução de uma comparação.

Um bloco precisa processar $\frac{n}{B}$ diagonais internas. Devido à otimização de delegação de células (Seção 5.4.1), cada diagonal interna possui geralmente o mesmo tamanho, exceto nos primeiros e últimos blocos da matriz. Sendo assim, o tempo de execução t_i de uma diagonal interna pode ser considerado aproximadamente constante. Além do tempo de processamento, as chamadas aos *kernels* das fases longa e curta consomem um tempo adicional k_i, considerando o *overhead* de todas as transferências de dados entre CPU e GPU, assim como rotinas de inicialização e destruição de cada *kernel*. Sendo assim, o tempo total de execução de um único bloco é de $t_B = \frac{n}{B}.t_i + k_i$.

Exceto para as primeiras e últimas diagonais externas, o número de blocos em uma diagonal é sempre B. Com isso, o tempo de execução de uma diagonal externa é aproximadamente $t_e = \frac{B.t_B}{P} + k_e = \frac{n.t_i + B.k_i}{P} + k_e$, sendo P uma constante que indica o grau de paralelismo permitido pela GPU e k_e o tempo adicional relacionado às tarefas de inicialização, destruição e transferência de informações entre cada diagonal externa. A estimativa de execução de uma diagonal externa pode ser escrita simplificadamente como $t_e = a_1 + a_2.n$, sendo constantes os valores $a_1 = \frac{B.k_i}{P} + k_e$ e $a_2 = \frac{t_i}{P}$.

O número de diagonais externas é $|D| = B + \frac{m}{\alpha T}$. Portanto, o tempo gasto para calcular todas as diagonais externas é de aproximadamente $(B + \frac{m}{\alpha T}).t_e = (B + \frac{m}{\alpha T}).(a_1 + a_2.n)$. Esta fórmula pode ser simplificadamente reescrita como $b_1 + b_2 m + b_3 n + b_4 mn$, sendo constantes os valores $b_1 = a_1 B$, $b_2 = \frac{a_1}{\alpha T}$, $b_3 = a_2 B$ e $b_4 = \frac{a_2}{\alpha T}$.

Existe também um tempo $t_s = (n + m).c_s$ para a leitura das sequências de entrada, sendo c_s uma constante que indica a velocidade de leitura, e um tempo adicional k_{cpu} para iniciar o processamento em CPU.

Somando todos os tempos estimados, o tempo total para a comparação de duas sequências de tamanhos m e n pode ser representado pela fórmula $t(m,n) = (b_1 + b_2m + b_3n + b_4mn) + (n+m).c_s + k_{cpu}$. Esta fórmula pode ser simplesmente reescrita conforme a Equação 5.3, sendo constantes os valores $c_1 = b_1 + k_{cpu}$, $c_2 = b_2 + c_s$, $c_3 = b_3 + c_s$ e $c_4 = b_4$. Por meio desta fórmula, conclui-se também que o algoritmo executa com tempo de execução na ordem de $O(mn)$.

$$t(m,n) = c_1 + c_2m + c_3n + c_4mn \tag{5.3}$$

Uma métrica para mensurar o desempenho de uma implementação de Smith-Waterman é o número de células atualizadas por segundo (CUPS - *Cells Update per Second*). Para obtermos uma previsão do desempenho em CUPS, basta dividirmos o número de células da matriz ($m \times n$) pelo tempo de execução $t(m,n)$, conforme descrito na Equação 5.4.

$$CUPS(m,n) = \frac{mn}{t(m,n)} = \frac{mn}{c_1 + c_2m + c_3n + c_4mn} \tag{5.4}$$

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 fórmula $CUPS_{max} = \dfrac{1}{c_4}$, conforme demonstrado na Equação 5.5.

$$\begin{aligned} CUPS_{max} &= \lim_{m,n\to(\infty,\infty)} \frac{mn}{c_1 + c_2m + c_3n + c_4mn} \\ &= \lim_{m\to\infty} m \left(\lim_{n\to\infty} \frac{n}{c_1 + c_2m + n(c_3 + c_4m)} \right) \\ &= \lim_{m\to\infty} m \left(\frac{1}{c_3 + c_4m} \right) = \lim_{m\to\infty} \frac{m}{c_3 + c_4m} = \frac{1}{c_4} \end{aligned} \tag{5.5}$$

Capítulo 6

Resultados Experimentais

Neste capítulo estão descritos os experimentos realizados para avaliar o desempenho e a escalabilidade do CUDAlign.

6.1 Informações de Compilação

A versão utilizada do ambiente de desenvolvimento (SDK) do CUDA foi a 3.2. Os compiladores utilizados foram o `nvcc` 3.2 e o `g++` 4.4.3. Para a compilação dos *kernels*, utilizou-se o parâmetro `nvcc --gpu-architecture=sm_11`, que permite a execução do código em placas NVidia com *compute capability* a partir da versão 1.1.

6.2 Informações da Execução

Nesta seção, estão descritos os ambientes utilizados (*hardware* e *software*) e as sequências reais escolhidas como entradas da execução dos testes.

6.2.1 Ambientes Utilizados

Dois ambientes foram utilizados para a avaliação do desempenho da solução. Cada ambiente possui duas placas de vídeo, mas apenas as placas GTX 285 e GTX 260 foram utilizadas nos testes.

- **Ambiente Nº1**: Possui uma máquina dedicada para experimentos em CUDA. O hardware encontra-se no LAICO (LAboratório de sistemas Integrados e Concorrentes) do Departamento de Ciência da Computação da Universidade de Brasília (CIC/UnB). As especificações de Hardware e Software desta máquina são as seguintes:
  - CPU: Intel Pentium(R) Dual-Core E5400 2.70 GHz
  - Cache: L1 (32K/32K); L2 (2048K)
  - Placa Mãe: Asus P5KPL/1600
  - Memória: 2x2048
  - Disco: SAMSUNG HD322HJ (320G) 7200 RPM

  - GPU Nº1: GeForce GTX 285 1024M (PCI-E 4x) - Dedicada
  - GPU Nº2: GeForce 9600 GT 512M (PCI-E 16x) - Monitor Conectado
  - Sistema Operacional: Linux version 2.6.32-25-generic x86_64
  - Distribuição: Ubuntu 10.04.1 LTS
  - Driver da NVidia: 260.24
  - Versão do CUDA *toolkit*: 3.1

- **Ambiente Nº2**: Estação de usuário não dedicada. Embora a máquina não seja dedicada para os experimentos, a execução dos testes sempre foi feita fora do ambiente gráfico. As especificações de Hardware e Software desta máquina são as seguintes:

  - CPU: AMD Athlon(tm) 64 X2 Dual Core Processor 6000+ 3.00 GHz
  - Cache: L1 (64K/64K); L2 (1024K)
  - Placa Mãe: Asus M2N-E SLI
  - Memória: 2x1024
  - Disco: Seagate ST31500341AS (1500G) 7200 RPM
  - GPU Nº1: GeForce GTX 260 896M (PCI-E 8x) - Monitor Conectado
  - GPU Nº2: GeForce 8600 GT 512M (PCI-E 8x) - Dedicada
  - Sistema Operacional: Linux version 2.6.32-27-generic x86_64
  - Distribuição: Ubuntu 10.04.1 LTS
  - Driver da NVidia: 260.19.29
  - Versão do CUDA *toolkit*: 3.1

	GTX 285	GTX 260
Memória Global	1024M	896M
Clock	1.48GHz	1.35GHz
Compute Capability	1.3	1.3
Nº de Multiprocessadores (MP)	30	27
Memória Compartilhada[1]	16KB	16KB
Número de registradores[1]	16384	16384
Memória Constante	64KB	64KB
GFLOPS Teórico [2]	1066	875

Tabela 6.1: Detalhes das GPUs utilizadas nos experimentos.

[1]Quantidade por multiprocessador
[2]Valores teóricos obtidos pela fórmula: $Clock \times (MP \times 8) \times 3$.

6.2.2 Sequências selecionadas

Várias sequências reais foram selecionadas para os experimentos. A Tabela 6.2 apresenta o nome das sequências, o tamanho real e o identificador de acesso que permite encontrá-las no site do National Center for Biotechnology Information (NCBI) [2]. As comparações selecionadas foram agrupadas em pares nessa tabela.

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

Tabela 6.2: Informações sobre as sequências reais utilizadas. Tamanhos variam de 162 KBP a 47 MBP.

6.3 Parametrização

A escolha dos parâmetros B (número de blocos), T (número de *threads* por bloco) e R (número de registradores por *kernel*) são essenciais para atingir um bom desempenho. Embora na Seção 5.6 tenham sido apresentadas algumas recomendações para a seleção destes parâmetros, a análise empírica não deve ser descartada. Nesta seção, as placas GTX 260 e GTX 285 serão utilizadas para a avaliação desses parâmetros.

Dois modos de compilação serão analisados. Na compilação sem redução de registradores, a quantidade original de registradores por *thread* é mantida. Na compilação com redução, a quantidade de registradores é limitada através da diretiva `__launch_bounds__`[1]. Por meio da redução de registradores, um maior grau

[1]A diretiva `__launch_bounds__` exige dois parâmetros: o número máximo de threads e o número desejável de blocos residentes em um mesmo multiprocessador. Para o primeiro parâmetro, foi utilizado o próprio valor de T. O segundo parâmetro foi manualmente ajustado, para cada teste, de forma a obter um número de registradores menor ou igual a 32. O parâmetro de compilação `nvcc -gpu-architecture=sm_11` indica que a arquitetura adotada possui *compute capability* igual a 1.1, que possui 8192 registradores. Sendo assim, os parâmetros escolhidos foram $(64, 4)$ e $(128, 2)$ para as escolhas de $T = 64$ e $T = 128$, respectivamente. Embora seja ideal que 32 registradores fossem utilizados, o compilador utiliza heurísticas que podem reduzir um pouco mais o número de registradores.

	fase curta					
	regs.	regs./B	smem.	Bl. resid.	Thr. resid	ocup.
$T = 32$	36	2560	732	6	192	18.8%
$T = 64$		2560	1372	6	384	37.5%
$T = 96$		4608	2012	3	288	28.1%
$T = 128$		4608	2652	3	384	37.5%
$T = 32$	31	2048	732	8	256	25.0%
$T = 64$		2048	1372	8	512	50.0%
$T = 96$		4096	2012	4	384	37.5%
$T = 128$		4096	2652	4	512	50.0%
	fase longa					
	regs.	regs./B	smem.	Bl. resid.	Thr. resid	ocup.
$T = 32$	36	2560	700	6	192	18.8%
$T = 64$		2560	1340	6	384	37.5%
$T = 96$		4608	1980	3	288	28.1%
$T = 128$		4608	2620	3	384	37.5%
$T = 32$	30	2048	700	8	256	25.0%
$T = 64$		2048	1340	8	512	50.0%
$T = 96$		4096	1980	4	384	37.5%
$T = 128$		4096	2620	4	512	50.0%

Tabela 6.3: Informações sobre número de registradores e de memória compartilhada obtidas nas estatísticas de compilação. As estatísticas apresentadas são iguais para as placas GTX 260 e GTX 285, pois ambas possuem a mesma versão de arquitetura. Os valores foram obtidos variando o número de threads T entre 32 e 128.

de paralelismo pode ser obtido, embora o tempo de execução de uma única thread seja aumentado por causa do maior uso de memória local [32].

A Tabela 6.3 apresenta, para os *kernels* das fases longa e curta, informações sobre a compilação[2] utilizando vários valores de T, com e sem redução de registradores. As informações contidas na tabela são: o número de registradores por *thread*, o número de registradores por bloco, a memória compartilhada utilizada por bloco, o número de blocos e de *threads* residentes em um mesmo multiprocessador e a taxa de ocupação das *threads* em um multiprocessador (considerando o máximo de 1024 *threads*).

O valor R de registradores por *kernel* é originalmente 36 para ambos os *kernels*. Quando a diretiva `__launch_bounds__` é utilizada, o número R de registradores diminui para 31 e 30 nos *kernels* das fases longa e curta, respectivamente.

O número de registradores alocados para um bloco de *threads* (coluna "regs./B" da Tabela 6.3) é arredondado segundo alguns critérios de granularidade definidos em [32]. Por causa desse arredondamento, observa-se que o número de registradores por bloco é o mesmo para todo valor de T no intervalo $[1..64]$, assim como no

[2]Para obter o número de registradores utilizados por cada *kernel* e a memória compartilhada utilizada por um bloco, utilizou-se o parâmetro `nvcc -ptxas-options=-v` durante a compilação.

intervalo $[65..128]$. Isso permite inferir que os melhores valores de T tendem a ser múltiplos de 64 (duas *warps* de 32 *threads* [32]) .

As placas GTX 260 e GTX 285 possuem 16,384 registradores por multiprocessador. Considerando o número de registradores alocados por bloco, o número máximo de blocos residentes em um mesmo multiprocessador é de 6 quando $T = 32$ ou $T = 64$, e de 3 quando $T = 96$ ou $T = 128$. Por meio da redução de registradores, estes valores sobem para 8 e 4, respectivamente, o que permite um maior nível de paralelismo. Sabendo que estas placas suportam até 1024 threads residentes no mesmo multiprocessador, a taxa de ocupação chega a 50% quando utilizamos um *kernel* com redução de registradores e com número de *threads* igual a $T = 128$ ou $T = 64$.

Os tempos de execução para a comparação $1044K \times 1073K$ variando B de 27 a 512, estão apresentados na Figura 6.1. Nestas figuras é possível perceber que os picos ocorrem nos valores de B que são múltiplos do número SM de multiprocessadores, que na GTX 260 é $SM = 27$ e na GTX 285 é $SM = 30$. Os *kernels* com redução de registradores apresentam melhores resultados, devido à capacidade maior de paralelismo.

A Tabela 6.4 apresenta o tempo de execução apenas para os valores de B múltiplos do número de multiprocessadores: 27 para a GTX 260 e 30 para GTX 285. Os melhores tempos encontram-se nos picos onde o número de blocos residentes em todos os multiprocessadores torna-se máximo[3]. Os melhores picos estão em destaque ('*') na Tabela 6.4. Dentre os picos em destaque ('*'), o melhor tempo obtido sempre é o que possui menor valor de B, justificado pelo menor *overhead* de comunicação entre os blocos.

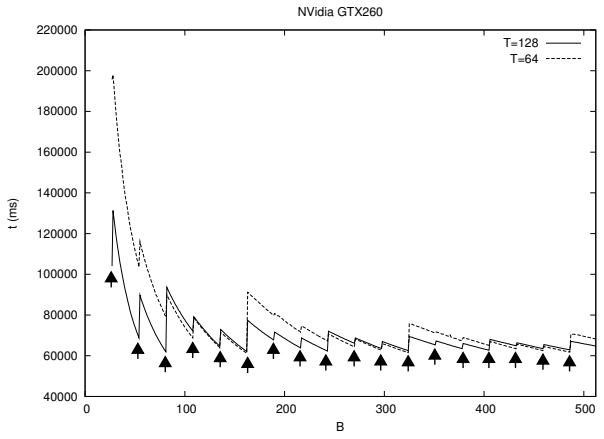
Por meio dos dados apresentados, conclui-se que a melhor escolha de B é o número de multiprocessadores multiplicado pelo número máximo de blocos residentes em cada multiprocessador. Além disso, o kernel com redução de registradores apresenta melhor desempenho do que sem redução. Deste modo, a partir de agora serão considerados apenas os *kernels* com redução de registradores e o número de blocos será $B = 216$ para a GTX 260 e $B = 240$ para a GTX 285.

6.4 Previsão de desempenho

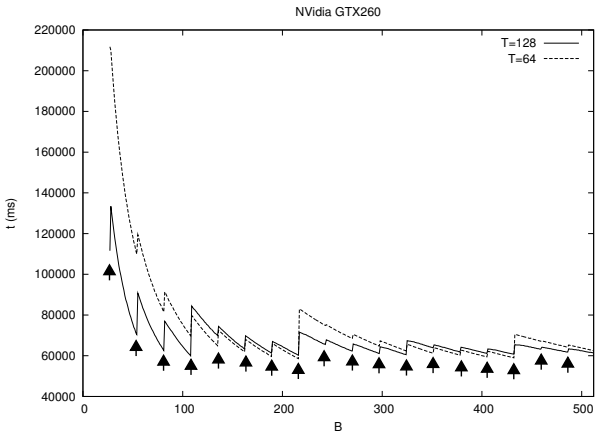
Conforme descrito na Seção 5.7, o tempo de comparação $t(m, n)$ para sequências de tamanhos m e n pode ser estimado através da fórmula $t(m, n) = c_1 + c_2m + c_3n + c_4mn$, sendo c_1, c_2, c_3 e c_4 constantes específicas para cada ambiente de execução.

Para determinar se essa fórmula é aproximadamente válida para qualquer tamanho de sequência, um experimento foi realizado variando o tamanho de ambas as entradas de 100.000 a 3.600.000 de bases, com um intervalo de 500.000 bases. Estas entradas foram criadas através do truncamento das sequências da

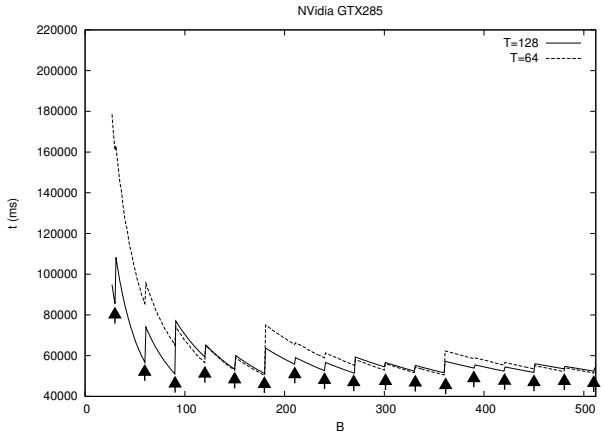
[3]Em outras palavras, o valor de B ideal para a execução deve ser múltiplo do número de blocos residentes em cada multiprocessador multiplicado pelo número de multiprocessadores. Sendo assim, para o kernel sem redução de registradores temos: na GTX 260, $T = 64 \Rightarrow B = 6 \times 27 = 162$ e $T = 128 \Rightarrow B = 3 \times 27 = 81$; na GTX 285, $T = 64 \Rightarrow B = 6 \times 30 = 180$ e $T = 128 \Rightarrow B = 3 \times 30 = 90$. Para o kernel com redução de registradores temos na GTX 260, $T = 64 \Rightarrow B = 8 \times 27 = 216$ e $T = 128 \Rightarrow B = 4 \times 27 = 108$; na GTX 285, $T = 64 \Rightarrow B = 8 \times 30 = 240$ e $T = 128 \Rightarrow B = 4 \times 30 = 120$



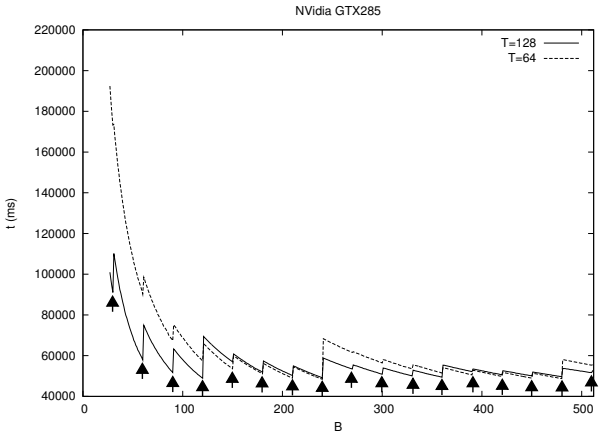
(a) GTX 260 - Sem redução de regs.



(b) GTX 260 - Com redução de regs.



(c) GTX 285 - Sem redução de regs.



(d) GTX 285 - Com redução de regs.

Figura 6.1: Tempo de execução da comparação $1044K \times 1073K$ variando o número de blocos B. Picos de desempenho estão indicados com setas. Os melhores picos ocorrem quando B é múltiplo do número máximo de blocos residentes em todos os multiprocessadores.

comparação $5227K \times 5229K$. Aplicando uma regressão não linear para a fórmula $t(m,n) = c_1 + c_2m + c_3n + c_4mn$, obtemos as constantes c_1, c_2, c_3 e c_4 descritas na Tabela 6.5. As constantes c_2 são menores que as constantes c_3, o que gera a preferência de que n seja menor que m, ou seja, S_1 seja menor que S_0.

A Figura 6.2 ilustra o tempo de execução de cada iteração deste experimento utilizando a placa GTX 285. Também está ilustrada na figura a fórmula $t(m,n) = c_1 + c_2m + c_3n + c_4mn$ com as constantes obtidas para a placa GTX 285 (Tabela 6.5).

GTX 260					GTX 285				
B	Sem Redução		Com Redução		B	Sem Redução		Com Redução	
	T=64	T=128	T=64	T=128		T=64	T=128	T=64	T=128
27	195.7	104.0	0.0	111.6	30	160.9	85.5	173.2	91.0
54	103.6	68.8	111.6	70.5	60	85.6	56.7	90.2	57.9
81	79.5	*__61.9__	82.6	62.6	90	65.1	*__51.0__	67.4	51.7
108	68.7	72.2	70.4	*__60.0__	120	56.5	59.4	57.6	*__48.9__
135	64.0	64.7	65.5	69.9	150	52.8	53.3	53.6	57.0
162	*__61.2__	*__62.1__	62.3	63.4	180	*__50.3__	*__51.2__	51.1	51.9
189	79.8	67.8	59.9	61.3	210	65.5	55.8	49.0	50.3
216	71.6	63.8	*__58.9__	*__60.3__	240	58.9	52.7	*__48.4__	*__49.1__
243	67.3	*__62.3__	75.1	65.5	270	55.2	*__51.4__	61.6	53.4
270	64.3	66.1	68.7	62.3	300	52.9	54.5	56.4	50.9
297	62.8	63.6	64.9	61.1	330	51.7	52.4	53.5	50.1
324	*__61.6__	*__62.5__	62.6	*__60.5__	360	*__50.6__	*__51.6__	51.5	*__49.4__
351	71.2	65.3	61.4	63.8	390	58.4	53.9	50.6	52.2
378	67.4	63.5	60.4	61.8	420	55.4	52.4	49.8	50.6
405	65.1	*__62.7__	59.6	61.2	450	53.5	*__51.8__	49.0	50.0
432	63.4	64.9	*__59.1__	*__60.8__	480	52.2	53.5	*__48.8__	*__49.7__
459	62.5	63.5	67.3	63.2	510	51.5	52.4	55.3	51.7
486	*__61.8__	*__62.9__	64.3	61.8					
B^*	162	81	216	108	B^*	180	90	240	120

Tabela 6.4: Tempo de execução da comparação $1044K \times 1073K$ nas placas GTX 260 e GTX 285, variando o número B de blocos. Apenas os valores B múltiplos do número de multiprocessadores foram listados. Tempos em destaque ('*') são os melhores resultados. A última linha (B^*) apresenta a melhor escolha de B em cada coluna.

GPU	c_1	c_2	c_3	c_4
GTX 260	0.040	6.0542e-07	9.9308e-07	5.07586e-11
GTX 285	0.032	5.8016e-07	9.0299e-07	4.17438e-11

Tabela 6.5: Constantes de previsão do tempo de execução para as placas GTX 260 e GTX 285.

Utilizando a fórmula de previsão de desempenho com as constantes apresentadas na Tabela 6.5, estimamos os tempos de execução das comparações listadas na Tabela 6.2. Os tempos previstos para a GTX 260 e GTX 285 estão na Tabela 6.6. Na Seção 6.6 estes valores serão comparados com os tempos reais.

O desempenho máximo previsto $CUPS_{max} = \frac{1}{c_4}$ (Equação 5.5) foi de 19.701 MCUPS para a GTX 260 e 23.933 MCUPS para a GTX 285.

Comparação	GTX 260		GTX 285	
	Previsão	MCUPS	Previsão	MCUPS
162K×172K	1.7	16171	1.4	19286
543K×536K	15.7	18569	13.0	22421
1044K×1073K	58.6	19117	48.4	23160
3147K×3283K	529.6	19507	436.1	23691
5227K×5229K	1395.7	19583	1148.7	23793
7146K×5227K	1905.5	19602	1568.1	23820
23012K×24544K	28706.0	19675	23611.8	23920
32799K×46944K	78221.2	19684	64335.7	23933

Tabela 6.6: Previsão de tempo (em segundos) e desempenho (em MCUPS) para as sequências listadas na Tabela 6.2, utilizando as constantes da Tabela 6.5.

Comparação	Células	Escore	Posição final
162K×172K	2.79E+10	18	(41058 , 44353)
543K×536K	2.91E+11	48	(308558 , 455134)
1044K×1073K	1.12E+12	88353	(1072950 , 722725)
3147K×3283K	1.03E+13	4226	(2991493 , 2689488)
5227K×5229K	2.73E+13	5220960	(5227292 , 5228663)
7146K×5227K	3.74E+13	172	(4655867 , 5077642)
23012K×24544K	5.65E+14	9063	(14651731 , 11501313)
32799K×46944K	1.54E+15	27206434	(32718231 , 46919080)

Tabela 6.7: Dados das comparações utilizadas nos testes. Estão apresentados o número de células da matriz de programção dinâmica, o melhor escore e a posição final do alinhamento ótimo.

6.5 Dados experimentais

Na Tabela 6.7 estão apresentados o melhor escore e a posição final do alinhamento ótimo das comparações da Tabela 6.2. Com exceção da comparação $32799K \times 46944K$, as sequências foram as mesmas escolhidas em [69]. Os parâmetros do algoritmo Smith-Waterman utilizados no teste foram: *match*: $+1$; *mismatch* -3; *first gap*: -5; *extention gap*: -2. Os parâmetros de execução do CUDAlign foram $T = 64$ e $B = 8 \times SM$, onde SM é o número de multiprocessadores da GPU utilizada. Sendo assim, para a GTX 260 foi escolhido o valor $B = 216$ e para a GTX 285, o valor $B = 240$. O valor de α foi fixado em 4, indicando que cada *thread* calcula 4 linhas da matriz.

O tempo de execução de cada comparação e o desempenho medido em MCUPS (milhões de células processadas por segundo) estão listados na Tabela 6.8.

A Figura 6.3 apresenta o tempo de execução na GTX 260 e na GTX 285, plotados em escala logarítmica. As linhas tracejadas indicam o desempenho máximo teórico ($CUPS_{max}$). A Figura 6.4 apresenta os mesmos resultados, mas em termos do desempenho relativo (MCUPS). As linhas tracejadas indicam o desempenho estimado

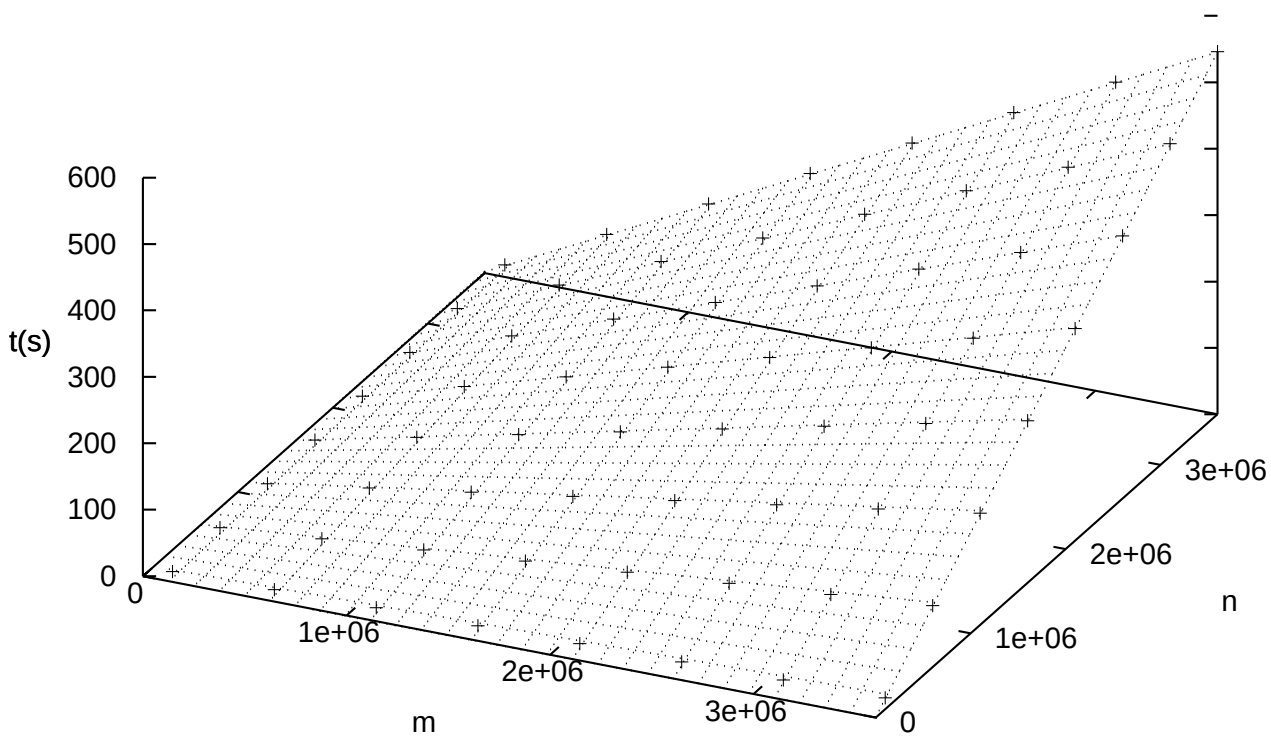


Figura 6.2: Tempo de execução da GTX 285 para vários tamanhos de sequência. Os pontos marcados com o símbolo '+' representam os tempos obtidos com testes reais. Já o grid pontilhado representa a fórmula estimada para a previsão de desempenho.

para cada uma das placas (Equação 5.4), considerando que as duas sequências possuam o mesmo tamanho ($S_1 = S_2 = \sqrt{x}$, sendo x o número de células da matriz de similaridade).

A Tabela 6.9 apresenta o resultado do BLAST para as mesmas comparações. Inicialmente, a comparação homem-chimpanzé não pode ser obtida utilizando a mesma plataforma das demais comparação, por problemas de falta de memória. Somente foi possível realizar uma execução especial do BLAST utilizando a ferramenta Megablast, em uma máquina com maior poder computacional e maior quantidade de memória.

6.6 Análise dos dados

Observando a Tabela 6.8, o valor de MCUPS para as comparações maiores que $1044K \times 1073K$ apresentam valores quase uniformes, variando de 19118 a 19670 para a GTX 260 e de 23157 a 23909 para a GTX 285. Essa variação é de 2.9% para a GTX 260 e de 3.2% para a GTX 285.

Comparação	GTX 260		GTX 285	
	Tempo	MCUPS	Tempo	MCUPS
162K×172K	1.7	16306	1.4	19420
543K×536K	15.7	18581	13.0	22434
1044K×1073K	58.6	19118	48.4	23157
3147K×3283K	529	19518	436	23690
5227K×5229K	1395	19597	1148	23804
7146K×5227K	1904	19620	1568	23823
23012K×24544K	28739	19652	23622	23909
32799K×46944K	78276	19670	64507	23869

Tabela 6.8: Tempo de execução (em segundos) e desempenho (em MCUPS) das comparações listadas na Tabela 6.2.

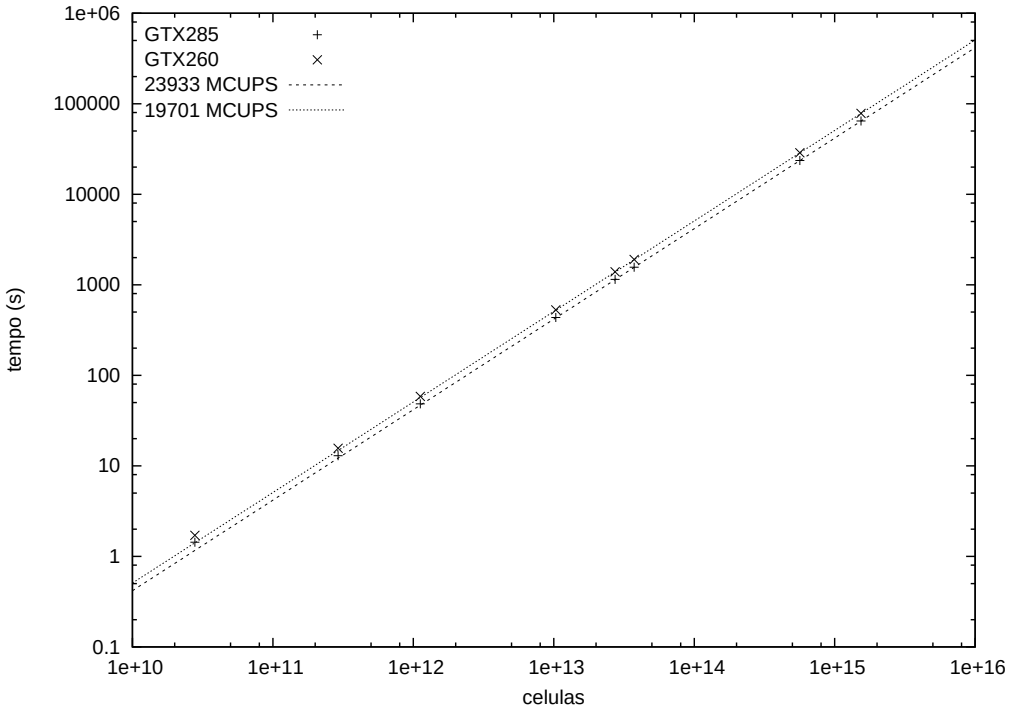


Figura 6.3: Tempo de execução em escala logarítmica.

Quando o tamanho das sequências aumenta, o desempenho em MCUPS também tende a aumentar. Isso ocorre por causa do *overhead* da reinicialização da execução de cada bloco, que é proporcionalmente reduzido quando aumentamos o valor de n. Esta observação comprova o excelente potencial de escalabilidade do projeto do CUDAlign, inclusive para sequências muito grandes.

O desempenho medido em MCUPS do CUDAlign foi superior que a maioria das

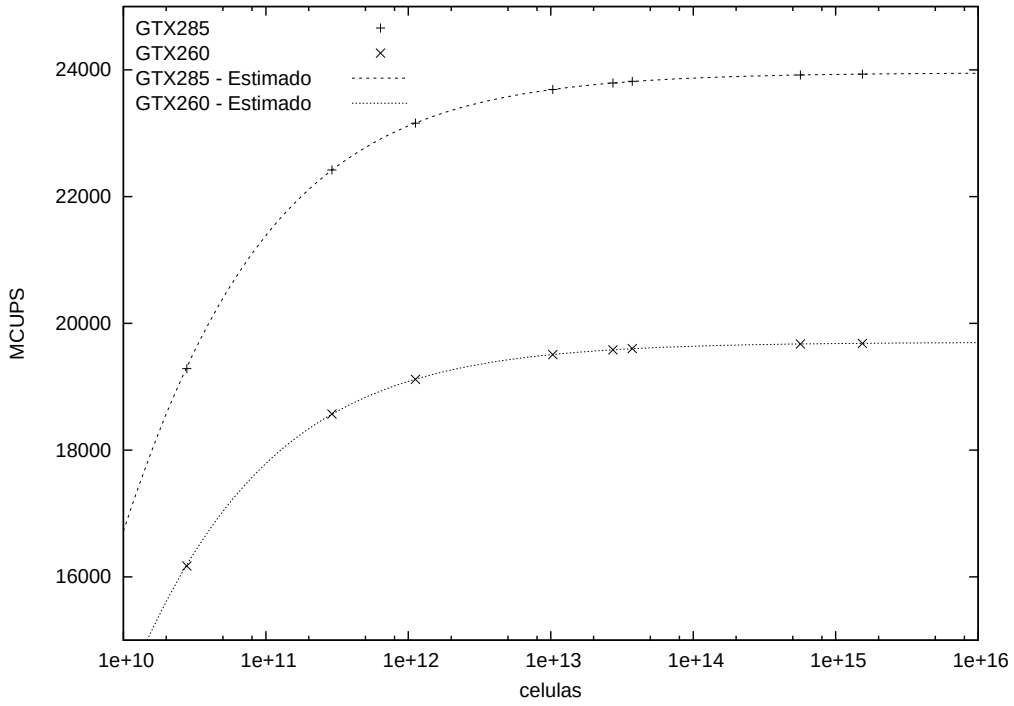


Figura 6.4: Desempenho das comparações em MCUPS. Linhas tracejadas indicam a previsão de desempenho para as duas placas (Equação 5.4).

implementações discutidas na Seção 3. A única ferramenta que obteve desempenho médio superior que o CUDAlign foi o CudaSW++2.0 [13] com a placa GTX 295, mas esta placa é 68% mais rápida que a placa GTX 285 utilizada nos testes do CUDAlign (Tabela 4.1).

Entretanto, a comparação do CUDAlign com os artigos apresentados na Seção 3 não pode ser feita de maneira direta por duas razões. Primeiramente, o CUDAlign foi testado com sequências de DNA e não houve necessidade de acesso à uma matriz de substituição, ao contrário das demais implementações, que realizam um acesso adicional às matrizes de substituição BLOSUM. Em contrapartida, as outras implementações podem explorar melhor as memórias mais rápidas da GPU, o que só é possível devido à limitação no tamanho das sequências de proteínas encontradas na natureza (na ordem de milhares de resíduos). O CUDAlign não pode limitar o tamanho das sequências pois as sequências de DNA podem ser muito maiores (na ordem de milhões de resíduos) que as sequências de proteínas.

A Tabela 6.10 apresenta a diferença entre o tempo de execução real e o tempo previsto pelas fórmulas obtidas na Seção 6.4. O erro absoluto da previsão foi sempre menor que 0.83%, sendo que na maioria das comparações o erro ficou abaixo de 0.10%, o que indica que a previsão foi bastante próxima do tempo real.

O desempenho da GTX 285 é aproximadamente 21% maior que o da GTX 260,

Comparação	BLAST		CUDAlign	
	Tempo	Escore	Tempo	Escore
162K×172K	0.4	18	1.4	18
543K×536K	0.6	48	13.0	48
1044K×1073K	2.4	6973	48.4	88353
3147K×3283K	6.7	3888	436	4226
5227K×5229K	17.4	36159	1148	5220960
7146K×5227K	7.7	157	1568	172
23012K×24544K	110	7085	23622	9063
32799K×46944K	92436	114501	64507	27206434

Tabela 6.9: Comparação entre os tempos (em segundos) e os escores obtidos pelo BLAST e pelo CUDAlign (GTX 285).

Comparação	GTX 260			GTX 285		
	Real	Prev.	Erro	Real	Prev.	Erro
162K×172K	1.71	1.72	0.83%	1.43	1.44	0.69%
543K×536K	15.66	15.68	0.07%	12.97	12.98	0.06%
1044K×1073K	58.62	58.62	<0.01%	48.39	48.39	0.01%
3147K×3283K	529.31	529.59	0.05%	436.09	436.08	<0.01%
5227K×5229K	1394.7	1395.7	0.08%	1148.2	1148.7	0.05%
7146K×5227K	1903.8	1905.5	0.09%	1567.9	1568.1	0.01%
23012K×24544K	28738.8	28706.0	0.11%	23622.3	23611.8	0.04%
32799K×46944K	78276.2	78221.2	0.07%	64507.4	64335.7	0.27%

Tabela 6.10: Diferença entre o tempo de execução real e o tempo previsto.

tanto nos tempos reais como nos tempos previstos. A mesma proporção ocorre ao comparar o desempenho das duas placas medidos em GFLOPS, que é de 1066 na GTX 285 e 875 na GTX 260, indicando que o desempenho teórico em GFLOPS é uma boa métrica comparativa entre as placas.

Conforme descrito em [14], a comparação genômica entre homem-chimpanzé é fundamental para determinar as mudanças genéticas que resultaram nas características individuais do homem. Neste cenário, uma comparação de elevado interesse é a comparação entre o cromossomo 21 do homem (*Homo sapiens*) e o cromossomo 22 do chimpanzé (*Pan troglodytes*). Em [14], esta comparação foi feita com o BLAST, visto que o tamanho enorme desses cromossomos (47MBP e 33MBP) previnem o uso de métodos exatos. Até onde sabemos, não existe nenhuma implementação de Smith-Waterman, executando em *clusters*, FPGAs ou GPUs, que compararam sequências tão grandes quanto esses cromossomos.

Nas Tabelas 6.7 e 6.8, estão apresentados os resultados obtidos quando comparamos esses dois cromossomos com o CUDAlign. O alto escore obtido (27,206,434) revela a grande similaridade entre essas sequências genômicas e indica o histórico evolucionário entre as duas espécies. O tempo de comparação para esses cromossomos foi de 17h55m07 (64507s) na GTX 285. Este tempo somente foi possível

por causa do projeto cuidadoso do CUDAlign, assim como da correta escolha dos parâmetros de execução.

O resultado do BLAST apresentou, em geral, um tempo de execução muito menor que o do CUDAlign. Isso se deve ao fato de que o BLAST é um método heurístico. Entretanto, o resultado do BLAST costuma divergir bastante quando as sequências forem grandes e o escore ótimo for alto. Por exemplo, para a comparação $5227K \times 5229K$, CUDAlign obteve o escore ótimo de 5.220.960 (Tabela 6.7) enquanto o melhor escore obtido pelo BLAST foi de 36.159 (Tabela 6.9).

Em relação à comparação homem-chimpanzé, o BLAST executou em 92436s (25h40m36s). Este tempo elevado foi devido ao uso intenso de memória para esta comparação, o que fez o sistema operacional entrar em estado de *thrashing*. O melhor escore obtido pelo BLAST foi de 114.501, sendo que o escore ótimo é de 27.206.434. Para esta comparação, o tempo de execução e o resultado do CUDAlign foram muito melhores do que o do BLAST.

Capítulo 7

Conclusão e Trabalhos Futuros

7.1 Conclusão

Nessa dissertação, foi proposta e avaliada uma solução paralela de comparação exata de sequências genômicas em GPU. Esta solução foi chamada de CUDAlign e, diferente das outras estratégias exatas que utilizam GPUs, nossa proposta não impõe fortes restrições no tamanho das sequências. Desta forma, o CUDAlign foi capaz de comparar sequências cromossômicas com mais de 33 milhões de pares de base.

O CUDAlign explora o paralelismo utilizando o método *wavefront*, sendo que esta técnica mostrou-se bastante compatível com a arquitetura paralela da GPU. Por meio da otimização de delegação de células, o CUDAlign mantém o paralelismo máximo durante a maior parte do tempo de processamento. Além disso, os acessos à memória foram cuidadosamente projetados para melhor utilizarem as características hierárquicas da memória das GPUs.

Através da análise de complexidade, foi possível concluir que a solução executa em tempo quadrático (Seção 5.7) e em memória linear (Seção 5.3). Além disso, fórmulas de previsão de desempenho foram obtidas para cada uma das placas utilizadas e foram capazes de estimar o tempo de execução com erro abaixo de 1% (Seção 6.6). Desta forma, é possível constatar que a fórmula de previsão de desempenho é bastante acurada.

Os resultados experimentais obtidos (Capítulo 6) demonstraram que a solução é escalável, permitindo comparações eficientes inclusive para sequências muito grandes. O desempenho máximo das comparações foi mensurado em 19.670 MCUPS na GTX 260 e 23.909 MCUPS na GTX 285. Estes valores de desempenho são melhores que os obtidos nos artigos analisados na Seção 3, exceto para o artigo [13] que utilizou uma GPU de maior desempenho.

O CUDAlign foi capaz de comparar o cromossomo 21 do homem (47 MBP) com o cromossomo 22 do chimpanzé (33 MBP) em 17h55m07 (64507s) na GTX 285. O escore obtido (27.206.434) revela a grande similaridade entre essas sequências. Pelo que temos conhecimento, esta foi a primeira vez que dois cromossomos de tal porte foram comparados utilizando um método exato. Além disso, o tempo de execução foi bastante satisfatório, haja vista que em [69] uma comparação menor

($23012K \times 24544K$) demorou mais que 4 dias executando em um *cluster* de 64 processadores.

7.2 Trabalhos Futuros

O CUDAlign retorna apenas o escore ótimo de similaridade entre duas sequências, não apresentando um alinhamento que indique quais as regiões de similaridade. Como trabalho futuro, deseja-se propor uma solução capaz de retornar o alinhamento completo entre duas sequências, mantendo o algoritmo com complexidade quadrática de tempo e complexidade linear de memória. Para isso, deverão ser utilizadas técnicas baseadas em [22] para obtenção do alinhamento ótimo com memória linear.

Além disso, o CUDAlign será adaptado para executar em plataformas com múltiplas GPUs, inclusive permitindo cenários com placas heterogêneas. Desta forma, poderemos paralelizar ainda mais as tarefas na expectativa de obtermos tempos de execução menores. Outras sequências também serão comparadas, incluindo cromossomos de outros organismos.

Por fim, planeja-se a migração do CUDAlign para outras plataformas, tanto da própria NVidia (ex. Fermi) como de outros fabricantes (ex. AMD). Para isso, consideraremos a migração para a plataforma OpenCL, avaliando a sua capacidade de executar em ambientes heterogêneos.

Referências

[1] D. A. Benson, I. Karsch-Mizrachi, D. J. Lipman, J. Ostell, and E. W. Sayers. Genbank. *Nucleic Acids Research*, 38, 2010. 1, 13

[2] National center for biotechnology information. `http://www.ncbi.nlm.nih.gov/`. 1, 54

[3] Ncbi-genbank flat file release 183.0. `ftp://ftp.ncbi.nih.gov/genbank/gbrel.txt`. 1, 2

[4] D. A. Benson, I. Karsch-Mizrachi, D. J. Lipman, J. Ostell, and E. W. Sayers. Genbank. *Nucleic acids research*, 37(Database issue), January 2009. 1, 13

[5] S. Batzoglou. The many faces of sequence alignment. *Brief Bioinform*, 6(1):6–22, March 2005. 1

[6] Richard Durbin, Sean R. Eddy, Anders Krogh, and Graeme Mitchison. *Biological Sequence Analysis: Probabilistic Models of Proteins and Nucleic Acids*. Cambridge University Press, 1999. 1

[7] T. F. Smith and M. S. Waterman. Identification of common molecular subsequences. *J Mol Biol*, 147(1):195–197, March 1981. 1, 6

[8] Weiguo Liu, Bertil Schmidt, Gerrit Voss, and Wolfgang Muller-Wittig. Streaming algorithms for biological sequence alignment on gpus. *IEEE Trans. Parallel Distrib. Syst.*, 18(9):1270–1281, 2007. 2, 15

[9] Weiguo Liu, Bertil Schmidt, and Wolfgang 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. 2, 16

[10] Svetlin Manavski and Giorgio Valle. Cuda compatible gpu cards as efficient hardware accelerators for smith-waterman sequence alignment. *BMC Bioinformatics*, 9(Suppl 2), 2008. 2, 16, 22

[11] Adrianto Wirawan, Chee K. Kwoh, Tri H. Nim, and Bertil Schmidt. Cbesw: Sequence alignment on the playstation 3. *BMC Bioinformatics*, 9, September 2008. 2, 17

[12] Yongchao Liu, Douglas Maskell, and Bertil Schmidt. Cudasw++: optimizing smith-waterman sequence database searches for cuda-enabled graphics processing units. *BMC Research Notes*, 2(1):73, 2009. 2, 17, 22

[13] Yongchao Liu, Bertil Schmidt, and Douglas 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. 2, 18, 62, 65

[14] The international chimpanzee chromosome 22 consortium. Dna sequence and comparative analysis of chimpanzee chromosome 22. *Nature*, 429(6990):382–388, May 2004. 3, 63

[15] R. Durbin, S. Eddy, A. Krogh, and G. Mitchison. *Biological sequence analysis*. Cambridge University Press, 2002. 4, 5

[16] Steven Henikoff and Jorja G. Henikoff. Amino acid substitution matrices from protein blocks. *Proceedings of The National Academy of Sciences*, 89:10915–10919, 1992. 5

[17] S. B. Needleman and C. D. Wunsch. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *J Mol Biol*, 48(3):443–453, March 1970. 5

[18] E. V. Denardo. *Dynamic Programming: Models and Applications*. 2003. 5

[19] M. Waterman, T. Smith, and W. Beyer. Some biological sequence metrics. *Advances in Mathematics*, 20(3):367–387, June 1976. 8

[20] O. Gotoh. An improved algorithm for matching biological sequences. *J Mol Biol*, 162(3):705–708, December 1982. 8

[21] D. S. Hirschberg. A linear space algorithm for computing maximal common subsequences. *Commun. ACM*, 18(6):341–343, 1975. 9

[22] Eugene W. Myers and Webb Miller. Optimal alignments in linear space. *Comput. Appl. Biosci.*, 4(1):11–17, March 1988. 9, 66

[23] James W. Fickett. Fast optimal alignment. *Nucleic Acids Research*, 12(1):175–179, 1984. 11

[24] Stephen F. Altschul, Thomas L. Madden, Alejandro A. Schäffer, Jinghui Zhang, Zheng Zhang, Webb Miller, and David J. Lipman. Gapped blast and psiblast: a new generation of protein database search programs. *NUCLEIC ACIDS RESEARCH*, 25(17):3389–3402, 1997. 11, 14

[25] 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, August 2000. 14

[26] P. Green. Swat. `http://www.phrap.org/phredphrap/swat.html`. 14

[27] M. Farrar. Striped smith-waterman speeds database searches six times over other simd implementations. *Bioinformatics*, 23(2):156–161, January 2007. 15

[28] Julie D. Thompson, Desmond G. Higgins, and Toby 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*. 15

[29] D. Baldwin J. Kessenich and R. Rost. The opengl shading language, document revision 59, technical report. `http://www.opengl.org/documentation/oglsl.html`, 2005. 16

[30] Randi J. Rost. *OpenGL(R) Shading Language (2nd Edition)*. Addison-Wesley Professional, January 2006. 16

[31] NVIDIA. *NVIDIA CUDA Architecture*. 2010. 16, 30

[32] NVIDIA. *NVIDIA CUDA Programming Guide 3.2*. 2010. 16, 22, 25, 30, 38, 55, 56

[33] Aaron Lefohn, Mike Houston, Johan Andersson, Ulf Assarsson, Cass Everitt, Kayvon Fatahalian, Tim Foley, Justin Hensley, Paul Lalonde, and David Luebke. Beyond programmable shading (parts i and ii). In *ACM SIGGRAPH 2009 Courses*, SIGGRAPH '09, pages 7:1–7:312, New York, NY, USA, 2009. ACM. 21

[34] Nick England. A graphics system architecture for interactive application-specific display functions. *IEEE Comput. Graph. Appl.*, 6:60–70, January 1986. 21

[35] Henry Fuchs, John Poulton, John Eyles, Trey Greer, Jack Goldfeather, David Ellsworth, Steve Molnar, Greg Turk, Brice Tebbs, and Laura 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, New York, NY, USA, 1989. ACM. 21

[36] Robert L. Cook. Shade trees. *SIGGRAPH Comput. Graph.*, 18:223–231, January 1984. 21

[37] Ken Perlin. An image synthesizer. *SIGGRAPH Comput. Graph.*, 19:287–296, July 1985. 21

[38] Pat Hanrahan and Jim Lawson. A language for shading and lighting calculations. *SIGGRAPH Comput. Graph.*, 24:289–298, September 1990. 21

[39] Marc Olano and Anselmo 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, New York, NY, USA, 1998. ACM. 21

[40] Paul Lalonde and Eric Schenk. Shader-driven compilation of rendering assets. *ACM Trans. Graph.*, 21:713–720, July 2002. 21

[41] Steven Molnar, John Eyles, and John Poulton. Pixelflow: High-speed rendering using image composition. In *Computer Graphics*, pages 231–240, 1992. 21

[42] Ian Buck, Tim Foley, Daniel Horn, Jeremy Sugerman, Kayvon Fatahalian, Mike Houston, and Pat Hanrahan. Brook for gpus: Stream computing on graphics hardware. *ACM TRANSACTIONS ON GRAPHICS*, 23:777–786, 2004. 22

[43] Michael D. McCool, Zheng Qin, and Tiberiu S. Popa. Shader metaprogramming. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, HWWS '02, pages 57–68, Aire-la-Ville, Switzerland, Switzerland, 2002. Eurographics Association. 22

[44] Michael McCool, Stefanus Du Toit, Tiberiu Popa, Bryan Chan, and Kevin Moule. Shader algebra. *ACM Trans. Graph.*, 23:787–795, August 2004. 22

[45] M. Flynn. Some computer organizations and their effectiveness. *IEEE Trans. Comput.*, C-21:948+, 1972. 22

[46] Michael Gschwind, H. Peter Hofstee, Brian Flachs, Martin Hopkins, Yukio Watanabe, and Takeshi Yamazaki. Synergistic processing in cell's multicore architecture. *IEEE Micro*, 26(2):10–24, 2006. 22, 24

[47] Larry Seiler, Doug Carmean, Eric Sprangle, Tom Forsyth, Michael Abrash, Pradeep Dubey, Stephen Junkins, Adam Lake, Jeremy Sugerman, Robert Cavin, Roger Espasa, Ed Grochowski, Toni Juan, and Pat Hanrahan. Larrabee: a many-core x86 architecture for visual computing. *ACM Trans. Graph.*, 27(3):1–15, 2008. 22, 23

[48] Shekhar Borkar. Thousand core chips: a technology perspective. In *Proceedings of the 44th annual Design Automation Conference*, DAC '07, pages 746–749, New York, NY, USA, 2007. ACM. 22

[49] Many-core processor. `http://software.intel.com/en-us/articles/many-core-processor/`, 1 2011. 22

[50] Kayvon Fatahalian and Mike Houston. Gpus: A closer look. *Queue*, 6:18–28, March 2008. 22

[51] Mark D. Hill and Michael R. Marty. Amdahl's law in the multicore era. *IEEE COMPUTER*, 2008. 22

[52] Gpgpu.org :: General-purpose computation on graphics processing units. `http://gpgpu.org/`, Jun 2011. 22

[53] Everton Hermann, Bruno Raffin, François Faure, Thierry Gautier, and Jérémie 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, Berlin, Heidelberg, 2010. Springer-Verlag. 22

[54] Jose Ricardo da Silva, Jr., Esteban W. Clua, Paulo A. Pagliosa, and Anselmo 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 20:1–20:1, New York, NY, USA, 2010. ACM. 22

[55] Changxin Li, Hongwei Wu, Shifeng Chen, Xiaochao Li, and Donghui 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, Piscataway, NJ, USA, 2009. IEEE Press. 22

[56] Neil Costigan and Peter 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, Berlin, Heidelberg, 2009. Springer-Verlag. 22

[57] John Johnson Yang Liu, Wayne Huang and Sheila Vaidya. Gpu accelerated smith-waterman. In *Computational Science – ICCS 2006*, volume 3994 of *LNCS*, pages 188–195. Springer, 2006. 22

[58] Weiguo Liu, B. Schmidt, G. Voss, A. Schroder, and W. Muller-Wittig. Bio-sequence database scanning on a gpu. *IPDPS*, page 274, 2006. 22

[59] Lukasz Ligowski and Witold 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 (HiCOMB 2009)*, 2009. 22

[60] CNET. Intel cancels larrabee retail products, larrabee project lives on. `http://www.anandtech.com/show/3592`, 12 2010. 22

[61] Chhugani J. Dubey P. Junkins S. Morrison T. Ragozin D. Smelyanskiy Bader, A. Game physics performance on larrabee architecture. intel whitepaper, available in august, 2008. 2008. 23

[62] M. Abrash. A first look at the larrabee new instructions (lrbni). 23

[63] Top500 supercomputing sites. `http://www.top500.org/`, Nov. 2010. 25

[64] Derek Wilson Anand Lal Shimpi. Nvidia's 1.4 billion transistor gpu: Gt200 arrives as the geforce gtx 280 & 260. `http://www.anandtech.com/show/2549/2`, 06 2008. 26, 27, 28

[65] NVIDIA. *NVIDIA's Next Generation CUDA Compute Architecture: Fermi*. 2009. 26

[66] Nvidia website. `http://www.nvidia.com/`, 12 2010. 29

[67] Video card reviews and specifications. `http://www.gpureview.com`, Jun 2011. 29

[68] NVIDIA. *NVIDIA CUDA C Best Practices Guide*. 2010. 30, 49

[69] Rodolfo Bezerra Batista, Azzedine Boukerche, and Alba Cristina Magalhaes Alves de Melo. A parallel strategy for biological sequence alignment in restricted memory space. *J. Parallel Distrib. Comput.*, 68(4):548–561, 2008. 59, 65

[70] Edans Flavius de Oliveira Sandes and Alba Cristina Magalhaes Alves de Melo. CUDAlign: using GPU to accelerate the comparison of megabase genomic sequences. In R. Govindarajan, David A. Padua, and Mary W. Hall, editors, *PPOPP*, pages 137–146. ACM, 2010. 73

[71] Edans Flavius O. Sandes and Alba Cristina M.A. de Melo. Cudalign: using gpu to accelerate the comparison of megabase genomic sequences. *SIGPLAN Not.*, 45:137–146, January 2010. 73

Anexo I

Artigo Publicado sobre o CUDAlign

CUDAlign: using GPU to accelerate the comparison of megabase genomic sequences. Symposium on Principles and Practice of Parallel Programming (PPoPP), pages 137–146. ACM, 2010. [70, 71]

CUDAlign: Using GPU to Accelerate the Comparison of Megabase Genomic Sequences

Edans Flavius de O. Sandes Alba Cristina M. A. de Melo
University of Brasilia (UnB), Brazil
{edans,albamm}@cic.unb.br

Abstract

Biological sequence comparison is a very important operation in Bioinformatics. Even though there do exist exact methods to compare biological sequences, these methods are often neglected due to their quadratic time and space complexity. In order to accelerate these methods, many GPU algorithms were proposed in the literature. Nevertheless, all of them restrict the size of the smallest sequence in such a way that Megabase genome comparison is prevented. In this paper, we propose and evaluate CUDAlign, a GPU algorithm that is able to compare Megabase biological sequences with an exact Smith-Waterman affine gap variant. CUDAlign was implemented in CUDA and tested in two GPU boards, separately. For real sequences whose size range from 1MBP (Megabase Pairs) to 47MBP, a close to uniform GCUPS (Giga Cells Updates per Second) was obtained, showing the potential scalability of our approach. Also, CUDAlign was able to compare the human chromosome 21 and the chimpanzee chromosome 22. This operation took 21 hours on GeForce GTX 280, resulting in a peak performance of 20.375 GCUPS. As far as we know, this is the first time such huge chromosomes are compared with an exact method.

Categories and Subject Descriptors D.1.3 [*Programming Techniques*]: Concurrent Programming; J.3 [*Life and Medical Sciences*]: Biology and Genetics

General Terms Algorithms

Keywords Biological Sequence Comparison, Smith-Waterman, GPU

1. Introduction

In the last four years, new DNA sequencing technologies have been developed that allow a hundred-fold increase in the throughput over the traditional method. This means that the genomic databases, that have already an exponential growth rate, will experience an unprecedented increase in their sizes. Therefore, a huge amount of new DNA sequences will need to be compared, in order to infer functional/structural characteristics. In this scenario, the time spent in each comparison, as well as the accuracy of the result obtained, will be a fundamental factor to determine the success/failure of the next generation genome projects.

Sequence comparison is, thus, a very basic and important operation in Bioinformatics. As a result of this step, one or more sequence alignments can be produced [1]. A sequence alignment has a similarity score associated to it that is obtained by placing one sequence above the other, making clear the correspondence between the characters and possibly introducing gaps into them [2]. The most common types of sequence alignment are global and local. To solve a global alignment problem is to find the best match between the entire sequences. On the other hand, local alignment algorithms must find the best match between parts of the sequences.

One important issue to be considered is how gaps are treated. A simple solution assigns a constant penalty for gaps. However, it has been observed that keeping gaps together represents better the biological relationships. Hence, the most widely used model among biologists is the affine gap model [3], where the penalty for opening a gap is higher than the penalty for extending it.

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 of size m and n in $O(mn)$ space and time. In this algorithm, a similarity matrix of size $(m+1) \times (n+1)$ is calculated. SW is very accurate but it needs a lot of computational resources.

In order to reduce execution time, heuristic methods such as BLAST [5] were proposed. These methods combine exact pattern matching with dynamic programming in order to produce good solutions faster. BLAST can align sequences in a very short time, still producing good results. Nevertheless, there is no guarantee that the best result will be produced.

Therefore, many efforts were made to develop methods and techniques that execute the SW algorithm in high performance architectures, allowing the production of exact results in a shorter time. One recent trend in high performance architectures is the Graphics Processing Units (GPUs). In addition to the usual graphics functions, recent GPU architectures are able to execute general purpose algorithms (GPGPUs). These GPUs contain elements that execute massive vector operations in a highly parallel way. Because of its TFlops peak performance and its availability in PC desktops, the utilization of GPUs is rapidly increasing in many scientific areas.

PPoPP'10, January 9–14, 2010, Bangalore, India.

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

$\sum$=1

Figure 1. Example of alignment and score between sequences S_0=ACTTCCAGA and S_1=AGTTCCGGAGG

In the Bioinformatics research area, there are some implementations of SW in GPUs [6, 7, 8, 9, 10], that were able to produce very good speedups. Nevertheless, all of them impose restrictions on the maximum size of the smallest sequence, that range from 2K[8] to 4MB [6]. That means that two huge sequences cannot be compared in such implementations.

In this paper, we propose and evaluate CUDAlign, a new GPU algorithm that is able to compare sequences of unrestricted size with the SW algorithm, using the affine gap model. Here, unrestricted size means that the implementation is only bound by the available global memory, which can permit comparison of sequences with approximately 100MBP (Megabase Pairs) depending on the GPU. As output, the similarity score and its coordinates in the similarity matrix are provided. The proposed algorithm was executed in two NVIDIA boards separately: 8600GT and GTX 280. In the second board, we were able to obtain 20.375 GCUPS (Giga Cell Updates per Second) when comparing Homo sapiens chromosome 21 with chimpanzee chromosome 22, with respectively 47MBP and 33MBP.

The rest of this paper is organized as follows. In Section 2, we present the Smith-Waterman algorithm with affine gap. Section 3 summarizes the CUDA architecture. In Section 4, related work is discussed. Section 5 describes our proposed GPU algorithm. Experimental results are shown in Section 6. Finally, Section 7 concludes the paper and presents future work.

2. Sequence Alignment

To compare two sequences, we need to find the best alignment between them, which is to place one sequence above the other making clear the correspondence between similar characters [2]. In an alignment, spaces can be inserted in arbitrary locations along the sequences.

In order to measure the similarity between two sequences, a score is calculated as follows. Given an alignment between sequences S_0 and S_1, the following values are assigned, for instance, for each column: a) $+1$, if both characters are identical (match); b) -1, if the characters are not identical (mismatch); and c) -2, if one of the characters is a space (gap).

The score is the sum of all these values. The similarity between two sequences is the highest score. Figure 1 presents one possible alignment between two DNA sequences and its associated score.

In Figure 1, a constant value is assigned to gaps. However, keeping gaps together generates more significant results, in a biological perspective [2]. For this reason, the opening of a gap must have a greater penalty than its extension (affine gap model). The penalty for the first gap is G_{first} and for each successive gap, the penalty is G_{ext}.

The algorithm SW [4] is an exact method based on dynamic programming to obtain the best local alignment between two sequences in quadratic time and space. The

		A	G	T	T	C	C	G	G	A	G	G
	0	0	0	0	0	0	0	0	0	0	0	0
A	0	1	0	0	0	0	0	0	1	0	0	0
C	0	0	**0**	0	0	1	1	0	0	0	0	0
T	0	0	0	**1**	1	0	0	0	0	0	0	0
T	0	0	0	1	**2**	0	0	0	0	0	0	0
C	0	0	0	0	0	**3**	1	0	0	0	0	0
C	0	0	0	0	0	1	**4**	2	0	0	0	0
A	0	1	0	0	0	0	2	**3**	1	1	0	0
G	0	0	2	0	0	0	0	3	**4**	2	2	1
A	0	1	0	1	0	0	0	1	2	**5**	3	1

Figure 2. Similarity matrix between sequences S_0=AGTTCCGGAGG and S_1=ACTTCCAGA. The arrows indicate the cell from where the value was obtained. Bold arrows indicate the traceback to obtain the optimal alignment.

SW algorithm was modified by [3] in order to calculate affine gap penalties. It is divided in two phases: calculate the dynamic programming matrices and obtain the best local alignment.

Phase 1 – Calculate the Dynamic Programming (DP) Matrices - The first phase of the algorithm receives as input sequences S_0 and S_1, with sizes $|S_0| = m$ and $|S_1| = n$. For sequences S_0 and S_1, there are $m + 1$ and $n + 1$ possible prefixes, respectively, including the empty sequence. The notation used to represent the n-th character of a sequence seq is $seq[n]$ and, to represent a prefix with n characters, we use $seq[1..n]$. The similarity matrix is denoted H, where $H_{i,j}$ contains the similarity score between prefixes $S_0[1..i]$ and $S_1[1..j]$. At the beginning, the first row and column are filled with zeroes. The remaining elements of H are obtained from equation 1. In equation 1, $p(i, j) = ma$ if $S_0[i] = S_1[j]$ (match) and mi otherwise (mismatch). In order to calculate gaps according to the affine gap model, two additional matrices E and F are needed. Even with this, time complexity remains quadratic [3]. Equations 2 and 3 are used to calculate matrices E and F respectively. The similarity score between sequences S_0 and S_1 is the highest value in H and the position (i, j) of occurence of this value represents the end of alignment. Note that there can be many positions with the highest value, resulting in many optimal alignments.

$$H_{i,j} = \max \begin{cases} 0 \\ E_{i,j} \\ F_{i,j} \\ H_{i-1,j-1} - p(i,j) \end{cases} \quad (1)$$

$$E_{i,j} = \max \begin{cases} E_{i,j-1} - G_{ext} \\ H_{i,j-1} - G_{first} \end{cases} \quad (2)$$

$$F_{i,j} = \max \begin{cases} E_{i-1,j} - G_{ext} \\ H_{i-1,j} - G_{first} \end{cases} \quad (3)$$

Phase 2 – Obtain the best alignment - In order to retrieve the best local alignment, the algorithm starts from the cell that contains the highest score value and follows the arrows until a zero-valued cell is reached. A left arrow in $H_{i,j}$ (Figure 2) indicates the alignment of $S_0[i]$

with a gap in S_1. An up arrow represents the alignment of $S_1[j]$ with a gap in S_0. Finally, an arrow on the diagonal indicates that $S_0[i]$ is aligned with $S_1[j]$.

3. CUDA architecture

CUDA (Compute Unified Device Architecture) [11] is a general purpose parallel computing architecture introduced by NVIDIA. A GPU with CUDA support is able to run computation intensive algorithms with high speedup compared to CPU executions. CUDA is based on a many-core architecture capable of running a high number of threads in parallel, executing arithmetics operations over data stored in a memory hierarchy.

In CUDA, threads are grouped in blocks with 1, 2 or 3 dimensions, and blocks are also grouped in grids of 1, 2 or 3 dimensions. Threads inside the same block can share data through a fast shared memory and execution can be synchronized by barriers. Threads from different blocks exchange data by a slow global memory and are not usually synchronized.

CUDA extends the C language, allowing the programming of functions that can be executed in GPU. These functions are named kernels and are invoked with an execution configuration that defines at least the number of blocks in the grid (B) and the number of threads in each block (T). This execution configuration is described as the pair $\lll B, T \ggg$. After the invocation, the kernel is executed many times in parallel, each thread running the same kernel code. Threads are distinguished by the thread and block identifiers, accessed by the built-in variables `threadIdx` and `blockIdx`, respectively.

When programming with CUDA, special care must be taken with its memory model. The correct placement of data in the hierarchy is crucial to achieve good speedups. The memory model in CUDA has 5 types of memory: shared memory, global memory, local memory, constant memory and texture memory. The shared memory is a read-write per-block memory which is very fast, if there are no bank conflits. The global memory is a slow read-write per-grid memory, where accesses are not cached and many MB are available. The local memory is in fact the global memory, only allocated per thread. For this reason, it is a slow read-write memory that is usually used to store large data structures. The texture and the constant memory are both read-only per-grid memories, with cached accesses. The main difference between them is that, while the texture memory is slow and big (many MB), the constant memory is fast and small (up to 64KB).

4. Related Work

In the Smith-Waterman (SW) algorithm (Section 2), most of the time is spent calculating the similarity matrix H and this is the part which is usually parallelized. The access pattern presented by the matrix calculation is non-uniform and the parallelization strategy that is traditionally used in this kind of problem is the wavefront method [12], since the calculations that can be done in parallel evolve as waves on diagonals. More specifically, in order to calculate each cell $H[i, j]$, we need to access $H[i-1, j]$, $H[i-1, j-1]$ and $H[i, j-1]$.

Figure 3 illustrates the wavefront method. At the beginning of the computation, only the top-left cell can be calculated. After this computation, two cells in the following antidiagonal can be calculated in parallel. The maximum parallelism is attained at the matrix main antidiagonal and the bottom-right cell is the last to be calculated.

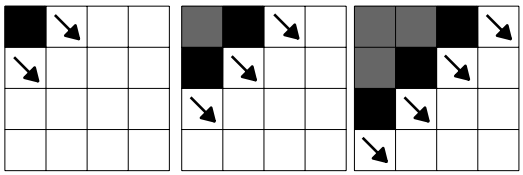


Figure 3. Wavefront execution. Three steps are shown, where each step calculates a diagonal. Black cells represent the cells being processed in the current diagonal, gray cells were already processed and white cells will be processed in the next steps.

Many implementations of the SW algorithm that use Single Instruction Multiple Data (SIMD) optimizations were proposed. Vector instructions, like MMX, SSE, SSE2 and Altivec, were used for reducing the execution time of SW, as shown in [13, 14, 15, 16]. Of those, only [13] uses the wavefront method. The other ones [14, 15, 16] use variations of the SWAT optimization [17]. Nevertheless, in all these SIMD proposals, parallelism is obtained by using SIMD instructions to process several elements simultaneously in each column, row or diagonal. In order to improve the speedup, operations are usually executed with 8-bit operands. However, if the score reaches 255, a new run is made, with larger operands [14, 15].

Several SW implementations were also developed for clusters of computers [18, 19, 20], usually with variants of the wavefront method. These algorithms usually divide the DP matrices into a set of columns or rows, which are assigned in a per node basis. Speedups close to linear, or even superlinear [21, 22], were reported for most implementations. In order to further reduce the execution time, Field Programmable Gate Arrays (FPGAs) have also been used to implement SW [23, 24, 25]. It is also a good alternative, presenting impressive speedups over the software-only implementations, but it is still not commodity hardware. Also, its programming interface is rather complex, since hardware description languages are generally used to describe the circuits.

Recently, it seems to be a trend to use GPUs to implement SW, taking advantage of its massively parallel architecture. With GPUs, impressive speedups can be achieved with commodity hardware, with the advantage of using a programming model that is simpler than the one required to program FPGAs. In the following paragraphs, we discuss some GPU implementations of SW. Unless otherwise stated, all of them execute SW with affine gap, compare a query sequence with a genomic database, and output the score as the result of the computation.

In [6], SW with double affine-gap function is implemented in GPU. In this case, besides the penalty for gap opening, there is a separate penalty for each gap that extends the alignment. Since 16-bit floats are used to store the elements of the matrices, the maximum value for the score is 64K. Also, the size of the query sequence is restricted to 4M. Target sequences are concatenated and between each sequence a special character is inserted. During the DP computation, when a special character is found, the corresponding values are set to zero. This optimization reduces reinitialization overhead of the wavefront. Two modes of operation are available: ASM and ATM. The ASM mode outputs the score and the ATM mode outputs the alignment. In the latter mode, the entire DP matri-

Paper	Algorithm	Output	Max. query	MCUPS	GPU
[6]	SW (double affine gap)	score/alignment	4M	241.12	GeForce 7800 GTX
[7]	SW (affine gap)	score	4K	650	GeForce 7800 GTX
[8]	SW (affine gap)	score	2050	3,612	2×GeForce 8800 GTX
[9]	SW (affine gap)	score	59K	16,087	2×GeForce GTX 295
[10]	SW (affine gap)	score	1K	14,500	GeForce 9800 GX2

Table 1. Comparison of GPU Smith-Waterman papers. All of them compare a query sequence against a database

ces are stored in the texture memory. A total of 241.12 MCUPS and 178.41 MCUPS were achieved for ASM and ATM, respectively.

In [7], a GPU implementation is proposed where, as a first step, the protein sequences and the substitution matrix are stored in the texture memory. Diagonals k, $k-1$ and $k-2$ are stored in separate circular buffers. As an optimization, cells of each diagonal are stored as columns. Query protein sequences of up to 4,095 aminoacids were compared to the SwissProt genomic database.

A Multi GPU-accelerated version of SW is proposed in [8]. In order to optimize the access to the substitution matrix, the authors use a technique called query profile [14], where a specific substitution matrix is pre-computed, replacing the random accesses by sequential ones. Scores are restricted to 16-bit values. Each GPU thread computes the whole alignment between the query sequence and one target sequence (coarse-grained parallelism). Query sequences of up to 2050 aminoacids are supported. Results of 1.889 GCUPS and 3.612 GCUPS were obtained for one and two GPUs, respectively.

In [9], single GPU and multi-GPU versions are provided. Two levels of parallelism exist: inter-task and intra-task parallelization. The first one is coarse-grained and assigns each query × target sequence comparison to a unique thread. The second one assigns several threads to each query × target comparison (fine-grained mode). If the query sequence length is below a threshold, the first mode is used. Otherwise, the second mode is chosen. Subject sequences are pre-ordered by their lengths. Optimized access patterns and block-based accesses are used in order to reduce access times for the texture and global memories. Query sequences of up to 59K are supported. Results of 9.660 GCUPS and 16.087 GCUPS were obtained for one and two GPUs, respectively.

In [10], the proposed implementation compares a query sequence against a database, returning the best score obtained by each alignment. For a single alignment, the DP Matix is divided in groups of 12 rows. Each step processes 12 cells with 2 global memory transactions, giving additional speedup. Results of 7.5 GCUPS and 14.5 GCUPS were obtained for one and two GPUs respectively, when comparing sequences of up to 1000 bases.

Table 1 summarizes the main characteristics of the GPU approaches discussed in the previous paragraphs.

As can be seen in Table 1, all GPU proposals discussed in this paper execute SW with simple affine gap, with the exception of [6], that calculates a double affine gap function. All algorithms compare a query sequence with a protein database. With the exception of [6], that calculates also the alignment on GPU, all proposals provide the similarity score as output. The maximum size allowed for the query sequence varies from 2,050 BP [8] to 4 MBP [6]. The maximum GCUPS obtained was 16.087 [9], with a dual-GPU. As far as we know, there is no proposal using GPU that aligns two Megabase genomic sequences with more than 4MBP.

5. Design of CUDAlign

5.1 General Overview

The GPU implementations of SW discussed in Section 4 do not compare two huge sequences. For these proposals (Table 1), high GCUPS are obtained when comparing small sequences against a large database. Aligning small sequences allows better speedup because the accesses to slower memories can be reduced and also the instructions can manipulate operands with reduced size, like 8-bit. In [10], very high GCUPS were obtained when comparing small sequences, using 16-bit variables. In Bioinformatics, however, there is a need to compare large DNA sequences, to analyze how similar two species are.

The goal of CUDAlign is to align two huge DNA sequences with an exact variant of SW. As output, it provides the similarity score and also the end coordinates of the optimal alignment, allowing future works to backtrace the full alignment.

Given sequences S_0 and S_1 with lengths m and n respectively, the SW algorithm computes a $(m+1) \times (n+1)$ matrix (Section 2). In our proposal, the computation is made with *affine gap*, so we need, in fact, to compute three matrices H, E and F. These three matrices are logically grouped into a single matrix M, where each cell (i, j) contains the three values H_{ij}, E_{ij} and F_{ij}.

In our solution, the cells of matrix M are grouped into blocks with $R \times C$ cells, resulting in a grid G with $\frac{m}{R} \times \frac{n}{C}$ blocks. If the division n/C is not exact, there must be a redistribution of the remaining columns between the blocks, then the blocks may have different number of columns. In contrast, each block must contain exactly R rows. If m is not divisible by R, the last blocks will be padded with rows that will not be processed. Given these observations, we will further consider that n is divisible by C and m is divisible by R.

The values C and R are chosen according to the execution configuration $\lll B, T \ggg$, where B is the number of concurrent blocks running in the grid and T is the number of threads per block. The execution configuration is chosen considering the GPU specifications and empirical results for each board. Given the values B and T, we set $C = \frac{n}{B}$ and $R = \alpha.T$, where α is an integer constant representing the number of rows that each thread is responsible to process.

For example, supose $m = 36$, $n = 36$, $B = 3$, $T = 3$ and $\alpha = 2$. Considering these values, the grid G contains 6×3 blocks, and each block G_{ij} contains 6×12 cells. Grid G is illustrated by Figure 4.

After the cells are grouped into blocks, the blocks are grouped into a set of antidiagonals. Antidiagonal D_k contains the blocks defined by equation $D_k = \{G_{ij} | i + j = k\}$.

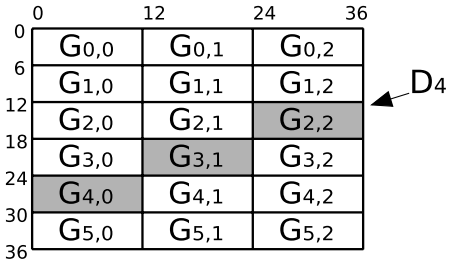


Figure 4. Blocks Grouping when $m = n = 36$, $B = 3$, $T = 3$ and $\alpha = 2$. Blocks in gray represent the external diagonal D_4

These diagonals are called *external diagonals*. The number of external diagonals is $|D| = B + \frac{m}{\alpha T} - 1$ and the number of blocks in the diagonal D_k ranges from 1 to B. Figure 4 shows in gray the blocks that compose external diagonal D_4.

For each diagonal D_k, the CPU invokes a kernel execution over all the blocks of that diagonal in parallel. When the GPU completes the execution of all blocks of that external diagonal, the CPU reinvokes the kernel for the next diagonal, successively until the end of the grid.

Inside the block execution, each block also contains parallelism, that is achieved by many threads processing the diagonals internal to the block. Similar to the grouping of external diagonals, these diagonals are called *internal diagonals*, and the internal diagonal d_k is expressed as the set of cells $d_k = \{(i,j) | \lfloor \frac{i}{\alpha} \rfloor + j = k\}$, considering that values i and j are relative to the top-left cell of the block (i_0, j_0). Each block contains T threads and thread T_k processes rows αk to $\alpha k + \alpha - 1$ of the block, from left to right. All threads calculate in parallel the same internal diagonal. In this way, thread T_i processes cells from column j in the same time as thread T_{i+1} processes cells from column $j - 1$. Figure 5 illustrates the execution of a single block with three threads. Note the diagonal d_4, whose cells can be computed in a parallel way by each thread.

Unlike to the external diagonals, the number of internal diagonals is simply $|d| = \frac{n}{B}$. When the first thread (T_0) hits the last column of the block, all other threads finish their execution, leaving some cells unprocessed. More precisely, thread T_k leaves αk cells unprocessed and the total block execution finishes with $\sum_{k=0}^{T-1} \alpha k = \frac{\alpha}{2} T(T-1)$ pending cells. Figure 5 shows the pending cells as hatched squares.

5.2 Optimizations

5.2.1 Cells Delegation

In order to treat the pending cells, we propose an optimization called *cells delegation*, that is a variation of the sequences aggregation proposed in [6]. With cells delegation, pending cells are processed by other block in the next external diagonal, enabling the first internal diagonal of the next block to start processing with full parallelism, i.e. with all T threads computing values simultaneously. Without this optimization, we would need to restart the wavefront procedure for each block, starting with one thread in the first diagonal, until we achieve the maximum parallelism again, only at diagonal d_{T-1}.

The cells delegation optimization is ilustrated in Figure 6. Delegations are only made between consecutive external diagonals because block $G_{i,j}$ (diagonal $i + j$) only delegates cells to block $G_{i,j+1}$ (diagonal $i + j + 1$). This optimization is also applied to the right-most blocks of the grid. In these boundary situations, the delegation will be made from blocks $G_{i,B-1}$ (last column of the grid) to blocks $G_{i+B,0}$ (first column of the grid) as can be seen in Figure 6 (b). The last external diagonals also delegates cells to the next diagonal (Figure 6 (c)), but the last block does not have next diagonal to delegate. So we need to create one extra diagonal, with only one block (Figure 6 (d)), in order to process the last pending cells. The number of external diagonals is then increased by one, totalizing $|D| = B + \frac{m}{T}$.

Note that in [6], the delegation is made between different target sequences and there is a need to separate sequences by a special character. In the Cells Delegation, the delegation is more complex because it happens between blocks inside the two-dimensional DP matrix, with an extra level of parallelism inside the external diagonal. The boundary test is made with the sizes of the matrix, without the need for a special delimiter character. Observe that

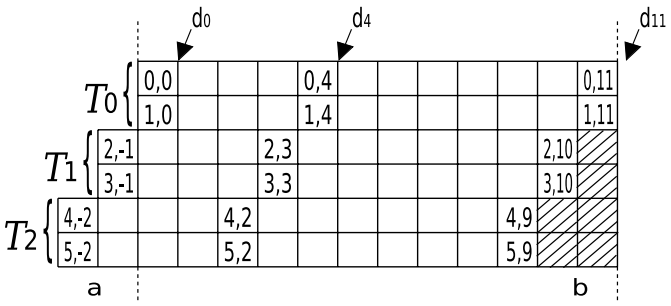


Figure 5. Each thread is responsible for performing two rows ($\alpha = 2$) of the block and threads execute simultaneously in the same internal diagonal. Diagonals range from d_0 to d_{11} in this example. Cell numbers are relative to the first cell of the block, so negative columns (a) represents delegated cells from the left block. Hatched cells (b) are delegated to the right block.

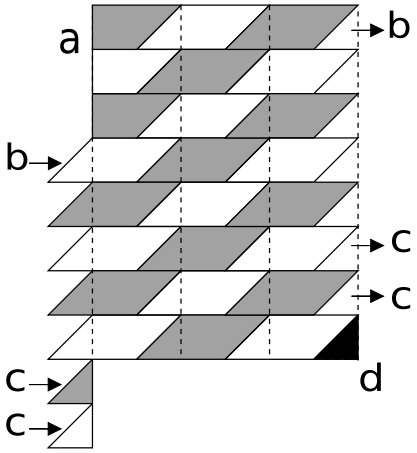


Figure 6. Cells Delegation Details. Gray blocks are from odd external diagonals and white blocks are from even external diagonals. First Blocks (a) does not receive delegated cells. Blocks in the right-most column delegate cells to left-most blocks (b). The last blocks also delegate to the next external diagonal (c), except the very last block, that requires an extra diagonal to process its pending cells. The extra diagonal are ilustrated in black color (d)

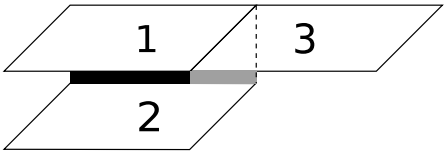


Figure 7. Delegation Hazard

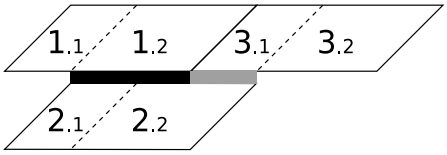


Figure 8. Block Execution divided in two phases. Short and long phases are labeled with .1 and .2 suffixes, respectively

both methods reduce the overhead of reinitialization compared with the conventional wavefront method. This happens because, once the wavefront is filled after processing the first diagonals, the algorithm explores the maximum parallelism, until the very last diagonals.

5.2.2 Phase Division

An implementation detail that must be considered in cells delegation is that, in CUDA, thread blocks can be executed in any order, in parallel or in series [11]. So, the scheduling of blocks of the same external diagonal is unpredictable, which can create a hazard during the delegations cells processing. Figure 7 shows this hazard during the execution of blocks 1, 2 and 3. Block 1 is executed first, because it is in the previous diagonal of blocks 2 and 3. The black bar represents the values calculated when block 1 finishes to be processed and block 2 can read them correctly. Suppose that the second external diagonal is being executed, but the scheduler is not executing the threads of block 3. So, block 2 will start reading all values from top blocks, until it reaches an unspecified area, represented in Figure 7 as a gray bar.

In order to remove this hazard, we must synchronize blocks to ensure that all top cells are ready to be read. In CUDA, there is no primitive to synchronize blocks, so our solution consists on dividing the external diagonal computation in two phases: the short phase and the long phase. The short phase will process T internal diagonals, from all blocks, and then return control to CPU to force block synchronization. After this, the CPU reinvokes the kernel to complete the remaining $\frac{n}{B} - T$ diagonals (long phase). Figure 8 shows the blocks execution split into these two phases. Note that when the short phase terminates, the gray area on Figure 8 was already processed and the long phase will read all values correctly from that area. Although Figure 8 visually presents both phases with almost the same length, note that the short phase is exactly T diagonals wide, but the long phase can contain up to millions diagonals, depending of the size m and the number of blocks B.

The only requirement still needed is that the number of cells that can be read in the short phase is bigger than T. As already explained, the last row $T-1$ of the upper block lefts $T-1$ pending cells. So, the bottom block can read up to $\frac{n}{B} - (T-1)$ cells in the short phase without hazard. Then, the delegation method requires that $n \geq 2BT$ (we round up because we need to compensate when n is not divisable by B). We will call this *minimum size requirement*. As we are dealing with huge sequences, the minimum size requirement will normally hold with the ideal B. Nevertheless, when small sequences are compared, the minimum size requirement is achieved by reducing the number of blocks B per invocation.

Besides removing the hazard, the short/long phase division presents one performance benefit. As shown in Figure 6 (b-d), the last column of blocks delegates cells to the first

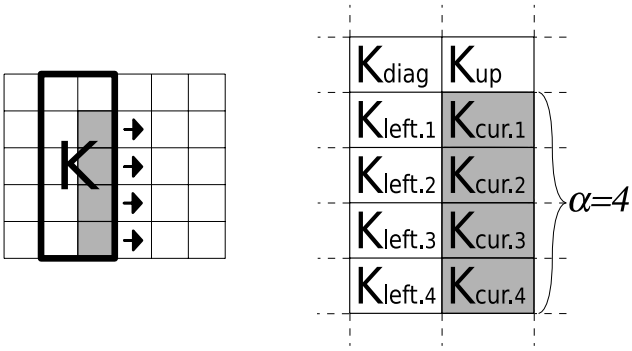


Figure 9. K-neighborhood

column of blocks. This requires a condition to test if the current cell belongs to a previous row or if it belongs to the current row. When the condition test detects that the row has changed, some variables must be reinitialized in order to represent the new row of the matrix. Note that this condition is only relevant in the short phase, so we removed this conditional test in the long phase, saving unnecessary computations.

5.2.3 Memory Accesses Design

CUDAlign executes one thread for each group of α rows of the block. The computation in the thread is made in the horizontal direction, iterating for each vertical group of α cells. Consider that K^{α}_{cur} represents the cells $(K_{cur.1}, K_{cur.2}, \ldots, K_{cur.\alpha})$ that are being processing in the current iteration and K^{α}_{left} represents the α cells processed in the last iteration. The cells K_{up} and K_{diag} represents the cell above $K_{cur.1}$ and above $K_{left.1}$, respectively. The aggregation of all these cells is represented as $K = (K^{\alpha}_{cur}, K^{\alpha}_{left}, K_{diag}, K_{up})$ and is called K-neighborhood. Figure 9 shows the positions of each element in the K-neighborhood.

During the thread execution, successive groups of α cells are calculated in each iteration, and each iteration presents a new K-neighborhood. For each new iteration, the new K^{α}_{left} is set from the previous K^{α}_{cur} and the new K_{diag} is set from the previous K_{up}.

The values of K-neighborhood are intensively used and, thus, are stored in registers. Nevertheless, the values of K_{up} must be read from the upper thread. If the upper thread resides in the same block, values are loaded from shared memory. If the upper thread resides in a different block, the values of K_{up} are loaded from the global memory, because blocks cannot share data in shared memory.

Similarly, the bottom values of the K-neighborhood (i.e. $K_{cur.\alpha}$) are stored in memory in order to be read by the bottom thread. If the bottom thread resides in the same block, values are stored in shared memory. Otherwise, global memory is used.

The global area used for exchange values of K_{up} and $K_{cur.\alpha}$ is called *horizontal bus* and it has a size that is proportional to the size $|S_1| = n$. In fact, the only values that must be stored in this bus are the 32-bit values of matrices H and F. So, the size of horizontal bus is equal to $8n$ bytes. Because the top row of the matrix must be set to zero, the horizontal bus is firstly initialized with zeroes.

The load from the horizontal bus is made through texture memory because of its cache capability. We are storing and loading elements over the same vector without any hazard, because reads and writes to/from the same position are always made in different moments.

At the begining of the block execution, the values of K^{α}_{left} and of K_{diag} must also be initialized. These values are obtained from the last cells K^{α}_{cur} and K_{up} of the previous block. This is a direct consequence of cells delegation, because the left block delegates cells to the right block. The delegation happens in global memory. The area used to exchange these values is called *vertical bus* and it has a size that is proportional to the total number of threads from all blocks, i.e. $B.T$. In fact, the only values that must be delegated through the vertical bus are the 32-bit values of matrices H and E for each K^{α}_{cur} cells. Additionally, each thread must also save the H component of the K_{up} cell and the F component of the $K_{cur.\alpha}$ cell, so two extra 32-bit words must be included for each thread. So, the size of vertical bus is equals to $(8.\alpha + 2)BT$ bytes.

Because we are dealing with huge sequences, the size of the horizontal bus is the main restriction of CUDAlign. Considering that the memory used by each sequence is 1 byte per base, we can roughly estimate that the total memory usage in CUDAlign is $9n + m$ bytes. So, with a GPU with 1GB of memory, we would be capable to compare two sequences of approximately 100 Megabases.

5.3 Pseudocode

In Algorithm 1, a pseudocode is presented for the kernel explained in Section 5.1 and 5.2. This code is executed by T threads in parallel for each of the B blocks.

The procedure Kernel is invoked with the parameter D_k (line 1), which represents the external diagonals being processed ($0 \leq D_k \leq (\frac{m}{\alpha T} + B - 1)$).

In line 2, the coordinates (i, j) of the top-left cell to be processed by this thread are obtained. If j has a negative value (line 3), values must be adjusted so that the pending cells from αBT rows above can be processed (lines 4-5). In line 7, the first cell delegated by the previous block is obtained from the vertical bus. The α bases associated with the α rows processed by this thread are loaded in variable b^{α} in line 8.

The internal diagonals are calculated in the main loop (lines 9-26). The condition in line 10 ensures that cells out of the matrix will not be processed. In line 11, K_{up} is loaded from the upper thread, that stored its $K_{cur.\alpha}$ value in the previous iteration (line 13). As already explained, K_{up} is read from the global memory (horizontal bus) if the thread is the first of the block (i.e. `threadIdx.x` is zero), or from the shared memory, otherwise.

The Smith-Waterman (SW) computation is made in line 12 for all K^{α}_{cur} values. After the K^{α}_{cur} values are calculated, the maximum local score is verified in line 14. If any K^{α}_{cur} values contains the best score, the maximum value and the best position are both updated in global memory. In lines 15-16 the initialization of the next iteration is made. In line 18, the column index j is incremented and,

Algorithm 1 Process Block (GPU)

```
 1: procedure KERNEL(D_k)
 2:     (i, j) := GETCOORD(D_k, blockIdx.x, threadIdx.x)
 3:     if j < 0 then
 4:         i := i − αBT
 5:         j := |S_1| − j
 6:     end if
 7:     K := GETDELEGATEDCELL(i)
 8:     b^α := S_0[i..i + α − 1]
 9:     for d_k = 0..|d| − 1 do
10:         if i ≥ 0 or i < m then
11:             K_up := LOADUP(threadIdx.x)
12:             K^α_cur := SW(b^α, S_1[j], K^α_left, K_diag, K_up)
13:             STOREDOWN(threadIdx.x + 1, K_cur.α)
14:             UPDATEMAXSCORE(K^α_cur, i, j)
15:             K^α_left := K^α_cur
16:             K_diag := K_up
17:         end if
18:         j := j + 1
19:         if j = |S_1| then
20:             j := 0
21:             i := i + B.αT
22:             K := STARTNEWLINE
23:             b^α := S_0[i..i + α − 1]
24:         end if
25:         __syncthreads()
26:     end for
27:     DELEGATECELL(K, i)
28: end procedure
```

if it exceeds the boundary of the matrix, the thread moves to the first column (line 20) and to αBT rows below (line 21). Also, the algorithm reinitializes the K values (line 22) and the loaded bases b^{α} (lines 23).

We synchronize all threads of the block in line 25, in order to mantain shared memory consistency.

After all internal diagonals be processed, we delegate, in line 27, the K values in order to be processed by the next block. These values are stored in global memory, so that the execution of the next block can continue the pending cells computation.

In Algorithm 2, the pseudocode of the CPU implementation is shown. In lines 2-3, the grid and thread dimensions are initialized. Because we are using one-dimension organization, we set to 1 the other dimensions. The loop in lines 4-7 iterates through each external diagonal $0 \leq D_k \leq |D| + B - 1$. Each iteration executes the short (line 5) and long (line 6) phases of kernel. Although Algorithm 1 does not show the division in short and long phases, in the implementation this division is made, considering that d_k iterates from 0 to $T - 1$ in the short phase and from T to $|d| - 1$ in the long phase. Moreover, in Algorithm 1 the two conditional statements in lines 3-6 and 19-24 are removed from the long phase. As a result, Algorithm 2 returns the best score and its corresponding position (lines 8-9), previously stored in line 14 of Algorithm 1.

6. Results

CUDAlign was implemented in CUDA 2.2 and tested in two NVIDIA GPUs separately: GeForce 8600GT and GTX 280. These boards were connected to the following desktops: AMD Athlon 64 X2 Dual Core Processor 6000+, with 2GB RAM (GeForce 8600GT) and Intel Pentium D CPU

Algorithm 2 Process Grid (CPU)

```
1: function ProcessGrid
2:     grid := (B, 1, 1)
3:     threads := (T, 1, 1)
4:     for D_k := 0..|D| + B − 1 do
5:         KERNEL.1<<< grid, threads >>>(D_k);
6:         KERNEL.2<<< grid, threads >>>(D_k);
7:     end for
8:     (max, max_pos) := GetMaxScore
9:     return (max, max_pos)
10: end function
```

Geforce Name:	8600 GT	GTX 280
Revision	1.1	1.3
Memory	512MB	1GB
# of multiprocessors	4	30
# of cores	32	240
Theoretical GFlops:	113	1008
Clock Rate:	1.19GHz	1.40GHz*

Table 2. Used GPUs. *GTX 280 is 8% overclocked

Comparison	GeForce 8600GT		GeForce GTX280	
	Time	MCUPS	Time	MCUPS
*128K×128K	8.6s	1895	1.1s	15277
162K×172K	14.5s	1915	1.7s	16421
*256K×256K	34.1s	1923	3.7s	17837
*512K×512K	135s	1939	13.7s	19147
543K×536K	150s	1941	15.2s	19212
*1000K×1000K	514s	1947	50.6s	19773
1044K×1073K	569s	1968	56.6s	19813
*2000K×2000K	2050s	1951	199s	20106
*3000K×3000K	4611s	1952	446s	20189
3147K×3283K	5291s	1953	512s	20196
5227K×5229K	13982s	1955	1348s	20278
7146K×5227K	19120s	1954	1841s	20289
23012K×24544K	-	-	27730s	20367
32799K×46944K	-	-	75571s	20375

Table 4. Runtimes of CUDAlign. Random sequences are marked with '*'

3.40GHz, with 4GB RAM (GTX280). Table 2 presents detailed information about both GPUs.

In our tests, we used real DNA sequences retrieved from the NCBI site (www.ncbi.nlm.nih.gov). The names of the sequences compared, as well as their sizes, are shown in Table 3. As can be seen in Table 3, the sizes of the real sequences range from 162 KBP to 47 MBP. In some tests, random generated DNA sequences were also used.

The best scores and their ending positions obtained for the comparison of real sequences are listed on Table 5, including the longest comparison, between *Homo sapiens* chromosome 21 and *Pan troglodytes* chromosome 22. With the exception of this comparison, the sequences were the same chosen in [19]. The Smith-Waterman score parameters used in the tests were: match: $+1$; mismatch -3; first gap: -5; extention gap: -2. The execution configurations used for 8600GT was $B = 2^4$ and $T = 2^6$ and for GTX280 was $B = 2^9$ and $T = 2^6$. For both boards, we set $\alpha = 4$ rows per thread. Execution times and MCUPS on each board are presented on Table 4.

An usual performance metric for SmithWaterman implementation is the number of cell updates per second (CUPS) and recent GPU implementations have shown performances up to 16 GCUPS (billions of cell updates per second) with dual GPUs. The CUPS metrics is calculated with the formula $\frac{mn}{t \times 10^9}$ where m and n are the sizes of both sequences S_0 and S_1 respectively. When the implementations compare a query sequence Q against a database D, the m and n values in the CUPS formula are replaced by the size of the query sequence $|Q|$ and the size of the database $|D|$.

In Table 4, it can be noted that the GCUPS rates of both GPUs for sequences longer than 1MBP range from 1.946 to 1.968 GCUPS (8600GT) and 19.773 to 20.375 GCUPS (GTX 280). As we increase the sizes of the megabase sequences, the GCUPS values tend to increase. This happens because the overhead of block reinitialization is proportionally reduced as n grows. Note that the difference from the peak and the lower GCUPS is 3% in the GTX280. This shows the potential for scalability of our design. The GCUPS obtained by our algorithm when running in GeForce GTX 280 are higher than all the GCUPS obtained by the other works discussed in Section 4 (Table 1). Nevertheless, we cannot make a direct comparison basically for two reasons. First, CUDAlign was tested against DNA sequences and no substitution matrix was fetched from memory, as other works do. Second, the proposals discussed in Section 4 impose a restriction on the size of the smallest sequence and, therefore, can use some memory access optimizations that are not possible in CUDAlign.

As stated in [26], human-chimpanzee comparative genomics is fundamental to determine the genetic changes that led to the acquisition of unique human features. In this scenario, one comparison of extraordinary interest is the comparison between the human chromosome 21 and the chimpanzee (*Pan troglodytes*) chromosome 22. In [26], this comparison was done with BLAST, since the huge sizes of the chromosomes (47MBP and 33MBP, respectively) prevented the use of exact methods. As far as we know, there is no Smith-Waterman based exact method, implemented either in clusters, FPGAs or GPUs, that compared sequences as huge as those chromosomes.

In Tables 5 and 4, we present the results obtained when comparing these two chromosomes with CUDAlign. The huge score obtained (27,206,434) reveals the great similarity between these genomic sequences and can be an important indication to establish our evolutionary history. Also, as can be seen in Table 4, this comparison took less than 21 hours (75,571s) on the GTX 280 GPU. This achievement was only possible because of the careful design of the CUDAlign and the optimized placement over the memory hierarchy.

The graphic in Figure 10 presents the results in a logarithmic scale. The MCUPS dotted lines in Figure 10 show the peak performance with 1,968 MCUPS and 20,375 MCUPS, for Geforce 8600GT and Geforce GTX 280 respectively.

Table 6 shows the BLAST results and execution times for the real sequences, executed on an Athlon 64 X2 Dual Core Processor 6000+, with 2GB DDR2 RAM. As BLAST is a heuristic method, its runtime is extremely low when compared to exact methods such as CUDAlign. Nevertheless, BLAST results usually diverge significantly when the sequences are long and the optimal score is high. For instance, for the 5227K $\times$ 5229K comparison, CUDAlign obtained a score of 5,220,960 (Table 5), while the best score obtained by BLAST was 36,159 (Table 6). In the human-

Aprox. Size	Real size	Accession Number	Name
162KBP	162,114 BP	NC_000898.1	*Human Herpesvirus* 6B
172KBP	171,823 BP	NC_007605.1	*Human Herpesvirus* 4
543KBP	542,868 BP	NC_003064.2	*Agrobacterium tumefaciens*
536KBP	536,165 BP	NC_000914.1	*Rhizobium sp.*
1MBP	1,044,459 BP	CP000051.1	*Chlamydia trachomatis*
1MBP	1,072,950 BP	AE002160.2	*Chlamydia muridarum*
3MBP	3,147,090 BP	BA000035.2	*Corynebacterium efficiens*
3MBP	3,282,708 BP	BX927147.1	*Corynebacterium glutamicum*
5MBP	5,227,293 BP	AE016879.1	*Bacillus anthracis* str. Ames
5MBP	5,227,293 BP	NC_003997.3	*Bacillus anthracis* str. Ames
5MBP	5,228,663 BP	AE017225.1	*Bacillus anthracis* str. Sterne
7MBP	7,145,576 BP	NC_005027.1	*Rhodopirellula baltica* SH 1
23MBP	23,011,544 BP	NT_033779.4	*Drosophila melanog.* chromosome 2L
25MBP	24,543,557 BP	NT_037436.3	*Drosophila melanog.* chromosome 3L
33MBP	32,799,110 BP	BA000046.3	*Pan troglodytes DNA*, chromosome 22
47MBP	46,944,323 BP	NC_000021.7	*Homo sapiens* chromosome 21

Table 3. Real sequences details. Sizes range from 162KBP to 47MBP.

Comparison	Cells	S_0	S_1	Score	Position
162K×172K	2.79E+10	NC_000898.1	NC_007605	18	(41058 , 44353)
543K×536K	2.91E+11	NC_003064.2	NC_000914.1	48	(308558 , 455134)
1044K×1073K	1.12E+12	CP000051.1	AE002160.2	88353	(1072950 , 722725)
3147K×3283K	1.03E+13	BA000035.2	BX927147.1	4226	(2991493 , 2689488)
5227K×5229K	2.73E+13	AE016879.1	AE017225.1	5220960	(5227292 , 5228663)
7146K×5227K	3.74E+13	NC_005027.1	NC_003997.3	172	(4655867 , 5077642)
23012K×24544K	5.65E+14	NT_033779.4	NT_037436.3	9063	(14651731 , 11501313)
32799K×46944K	1.54E+15	BA000046.3	NC_000021.7	27206434	(32718231 , 46919080)

Table 5. Comparison for the real sequences used in tests. The best local score and the end position are presented.

Comparison	BLAST	
	Time	Score
162K×172K	0.4s	18
543K×536K	0.6s	48
1044K×1073K	2.4s	6973
3147K×3283K	6.7s	3888
5227K×5229K	17.4s	36159
7146K×5227K	7.7s	157
23012K×24544K	110s	7085
32799K×46944K	-	-

Table 6. BLAST Results.

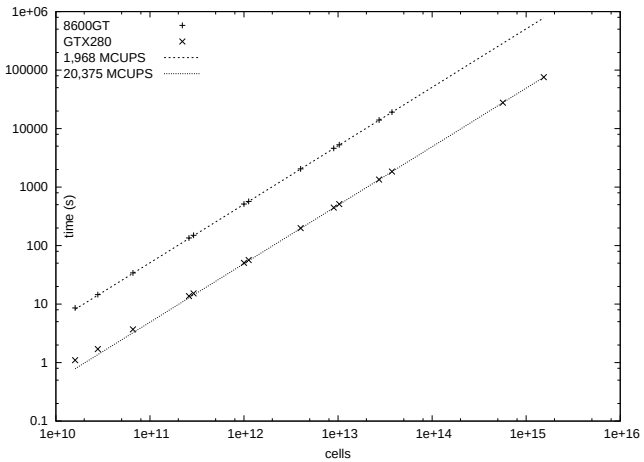


Figure 10. Runtimes (seconds) × DP matrix size (cells) in logarithm scale. Results show scalability and almost constant MCUPS ratio for Megabase sequences (cells $\geq 1e + 12$).

chimpanzee chromosome comparison, BLAST finished its execution with a segmentation fault, due to an out-of-memory error.

7. Conclusion and Future Work

In this paper, we proposed and evaluated CUDAlign, a GPU-accelerated version of Smith-Waterman (SW) that compares two Megabase genomic sequences. Differently from the previous GPU Smith-Waterman (SW) proposals in the literature, our proposal does not impose severe restrictions on the size of the smallest sequence and that allows, for instance, the comparison of two entire chromosomes. CUDAlign is able to heavily exploit the parallelism present in the wavefront SW computations in such a way that the only moment that there are idle threads is at the very beginning and at the very end of the matrix calculations. Also, memory accesses were carefully designed in order to exploit the characteristics of the GPU memory hierarchy.

The experimental results obtained with real and synthetic DNA sequences in two GPU boards show that an almost uniform GCUPS is obtained when the sizes of the sequences increase. This indicates that our algorithm is scalable for Megabase sequence comparisons. In order to compare DNAsequences of size 47MBP and 33MBP, CU-

DAlign took less than 21 hours, with a sustained 20.375 GCUPS. This is an impressive result, since a shorter comparison of 23012K $\times$ 24544K in [19] took more than four days in a 64-processor cluster.

By now, our algorithm outputs the similarity score and the coordinates that correspond to the end of the optimal alignment.

As future work, we intend to extend CUDAlign to return the full alignment. The alignment retrieval for Megabase sequences is a very challenging problem, since we obviously cannot store the whole DP matrices in this case. For instance, the human-chimpanzee chromosome comparison would require more than 16PB for the whole DP matrices. Therefore, several memory usage reduction techniques must be combined to obtain the full alignment.

Also, we intend to align other relevant sequences and extend the tests to more powerful GPUs. Finally, we plan to migrate our work to the OpenCL framework, taking advantage on its heterogeneity capability for testing the algorithm in other platforms.

Acknowledgments

We would like to thank Prof. Rafael Morgado Silva (UnB) for letting us use the NVIDIA GTX 280 board. This work is partially supported by FAPDF/Brazil, CNPq/Brazil and FINEP/Brazil.

References

[1] S. Batzoglou. The many faces of sequence alignment. *Brief Bioinform*, 6(1):6–22, March 2005.

[2] Richard Durbin, Sean R. Eddy, Anders Krogh, and Graeme Mitchison. *Biological Sequence Analysis: Probabilistic Models of Proteins and Nucleic Acids*. Cambridge University Press, 1999.

[3] O. Gotoh. An improved algorithm for matching biological sequences. *J Mol Biol*, 162(3):705–708, December 1982.

[4] T. F. Smith and M. S. Waterman. Identification of common molecular subsequences. *J Mol Biol*, 147(1):195–197, March 1981.

[5] S. F. Altschul, W. Gish, W. Miller, E. W. Myers, and D. J. Lipman. Basic local alignment search tool. *J Mol Biol*, 215(3):403–410, October 1990.

[6] John Johnson Yang Liu, Wayne Huang and Sheila Vaidya. Gpu accelerated smith-waterman. In *Computational Science – ICCS 2006*, volume 3994 of *LNCS*, pages 188–195. Springer, 2006.

[7] Weiguo Liu, B. Schmidt, G. Voss, A. Schroder, and W. Muller-Wittig. Bio-sequence database scanning on a gpu. *IPDPS*, page 274, 2006.

[8] Svetlin Manavski and Giorgio Valle. Cuda compatible gpu cards as efficient hardware accelerators for smith-waterman sequence alignment. *BMC Bioinformatics*, 9(Suppl 2), 2008.

[9] Yongchao Liu, Douglas Maskell, and Bertil Schmidt. Cudasw++: optimizing smith-waterman sequence database searches for cuda-enabled graphics processing units. *BMC Research Notes*, 2(1):73, 2009.

[10] Lukasz Ligowski and Witold 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 (HiCOMB 2009)*, 2009.

[11] NVIDIA. *NVIDIA CUDA Programming Guide 2.1*. 2008.

[12] Gregory F. Pfister. *In search of clusters: the coming battle in lowly parallel computing*. Prentice-Hall, Inc., 1995.

[13] A. Wozniak. Using video-oriented instructions to speed up sequence comparison. *Computer Applications in the Biosciences*, 13(2):145–150, 1997.

[14] 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, August 2000.

[15] M. Farrar. Striped smith-waterman speeds database searches six times over other simd implementations. *Bioinformatics*, 23(2):156–161, January 2007.

[16] Adrianto Wirawan, Chee K. Kwoh, Tri H. Nim, and Bertil Schmidt. Cbesw: Sequence alignment on the playstation 3. *BMC Bioinformatics*, 9, September 2008.

[17] P. Green. Url: http://www.phrap.org/phredphrap/swat.html.

[18] Stjepan Rajko and Srinivas Aluru. Space and time optimal parallel sequence alignments. *IEEE Trans. Parallel Distrib. Syst.*, 15(12):1070–1081, 2004.

[19] Rodolfo Bezerra Batista, Azzedine Boukerche, and Alba Cristina Magalhaes Alves de Melo. A parallel strategy for biological sequence alignment in restricted memory space. *J. Parallel Distrib. Comput.*, 68(4):548–561, 2008.

[20] Chunxi Chen and Bertil Schmidt. Computing large-scale alignments on a multi-cluster. *Cluster Computing, IEEE International Conference on*, page 38, 2003.

[21] Azzedine Boukerche, Alba Cristina Melo, Edans Flavius Sandes, and Mauricio Ayala-Rincon. An exact parallel algorithm to compare very long biological sequences in clusters of workstations. *Cluster Computing*, 10(2):187–202, 2007.

[22] Xiandong Meng and Vipin Chaudhary. Optimised fine and coarse parallelism for sequence homology search. *Int. J. Bioinformatics Res. Appl.*, 2(4):430–441, 2006.

[23] Peiheng Zhang, Guangming Tan, and Guang R. Gao. Implementation of the smith-waterman algorithm on a reconfigurable supercomputing platform. In *HPRCTA '07: Proc. of the 1st int. workshop on High-performance reconfigurable computing technology and applications*, pages 39–48, 2007.

[24] L. Xu P. Zhang N. Sun X. Jiang, X. Liu. "a reconfigurable accelerator for smith-waterman algorithm". *"IEEE Transactions on Circuits and Systems II"*, 54(12):1077–1081, December 2007.

[25] Azzedine Boukerche, Jan Mendonca Correa, Alba Cristina Magalhaes Alves de Melo, Ricardo P. Jacobi, and Adson Ferreira Rocha. Reconfigurable architecture for biological sequence comparison in reduced memory space. In *IPDPS*, pages 1–8, 2007.

[26] The international chimpanzee chromosome 22 consortium. Dna sequence and comparative analysis of chimpanzee chromosome 22. *Nature*, 429(6990):382–388, May 2004.