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

Comparação Paralela de Sequências Biológicas em Plataformas de Hardware Uniformes e Híbridas

Carlos Antônio Campos Jorge

Tese apresentada como requisito parcial para
conclusão do Doutorado em Informática

Orientadora
Profa. Dra. Alba Cristina Magalhães Alves de Melo

Brasília
2022

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

Coordenador: Prof. Dr. Ricardo Pezzuol Jacobi

Banca examinadora composta por:

Profa. Dra. Alba Cristina Magalhães Alves de Melo (Orientadora) — CIC/UnB
Prof. Dr. Ricardo dos Santos Ferreira — CCE/UFV
Prof. Dr. Edward David Moreno Ordonez — PROCC/UFS
Prof. Dr. Ricardo Pezzuol Jacobi — CIC/UnB

CIP — Catalogação Internacional na Publicação

Jorge, Carlos Antônio Campos.

Comparação Paralela de Sequências Biológicas em Plataformas de Hardware Uniformes e Híbridas / Carlos Antônio Campos Jorge. Brasília : UnB, 2022.
114 p. : il. ; 29,5 cm.

Tese de Doutorado — Universidade de Brasília, Brasília, 2022.

1. comparação de sequências biológicas, 2. FPGA, 3. Longest Common Subsequence

CDU 004 Cutter F475c

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 em Plataformas de Hardware Uniformes e Híbridas

Carlos Antônio Campos Jorge

Tese apresentada como requisito parcial para
conclusão do Doutorado em Informática

Profa. Dra. Alba Cristina Magalhães Alves de Melo (Orientadora)
CIC/UnB

Prof. Dr. Ricardo dos Santos Ferreira
CCE/UFV

Prof. Dr. Edward David Moreno Ordonez
PROCC/UFS

Prof. Dr. Ricardo Pezzuol Jacobi
CIC/UnB

Prof. Dr. Ricardo Pezzuol Jacobi
Coordenador do Programa de Pós-graduação em Informática

Brasília, 19 de abril de 2022

Agradecimentos

Agradeço aos meus pais, pelo apoio em todos os sentidos. Sem eles, eu não teria conseguido chegar até aqui.

À minha orientadora, Prof[a]. Dr[a]. Alba Cristina Magalhães Alves de Melo, por sua paciência e contribuição para o desenvolvimento deste trabalho, por todos os conselhos que me forneceu e por me ajudar na escrita dos artigos, os quais foram importantes para a realização deste trabalho.

Ao Prof. Dr. Alexandre Solon Nery e ao Prof. Dr. Alfredo Goldman, também por sua paciência e contribuição na escrita dos artigos e para o desenvolvimento deste trabalho.

À minha esposa, Wanessa Rodrigues de Sousa, e minha filha, Liz Rodrigues Jorge, pelo carinho, pelo apoio, pela paciência, pela motivação e compreensão em todos os momentos. Sem elas na minha vida, nada faria sentido.

Agradeço à Coordenação de Aperfeiçoamento de Pessoal de Nível Superior (CAPES) pelo fornecimento de minha bolsa de estudo. Sem esse apoio financeiro, dificilmente eu teria conseguido realizar este trabalho e adquirir os equipamentos que foram necessários para a obtenção de resultados.

Enfim, agradeço a todos aqueles que, de alguma forma, contribuíram para a realização deste trabalho, que, na verdade, é um meio para se atingir um fim.

Resumo

O alinhamento de sequências expõe claramente os padrões mais relevantes entre duas sequências, sendo útil para descobrir informação funcional, estrutural e evolucionária em sequências biológicas. Para tanto, é necessário descobrir o alinhamento ótimo, ou seja, o padrão que maximiza a similaridade entre as sequências. O algoritmo *Longest Common Subsequence* (LCS) produz o alinhamento ótimo e, por isso, é muito utilizado ao redor do mundo. Devido à complexidade quadrática de tempo, sua execução pode demorar muito se as sequências comparadas forem longas. Por isso, plataformas de hardware como FPGAs (*Field Programmable Gate Arrays*) tem sido exploradas. Para simplificar a programação de hardware, foram propostas soluções de alto nível, como HLS (*High Level Synthesis*), que transforma automaticamente um programa C, C++ ou SystemC em uma especificação de hardware, simplificando a programação. Para atingir certa simplicidade da programação e, consequentemente, reduzir o tempo necessário para a obtenção do circuito, exploramos nessa Tese a programação em HLS e em C. Além disso, investigamos soluções para a execução do algoritmo LCS tanto em plataformas FPGA *stand-alone* como em plataformas híbridas (CPU+FPGA). Inicialmente, confeccionamos um circuito programado em HLS, que foi executado em uma plataforma FPGA *stand-alone*. O circuito projetado comporta até dois co-processadores, permitindo a comparação paralela de dois pares de sequências. As execuções no ambiente *stand-alone* de um lote de 20 comparações de sequências sintéticas de 10K, 20K e 50K mostraram que a execução em FPGA consome menos energia e que as execuções híbridas (CPU+FPGA) atingem desempenho muito bom. Adicionalmente, foi comparado um lote de 20 sequências reais do vírus SARS-CoV-2 (30K) na plataforma *stand-alone*. Nesse caso, a configuração com 2 co-processadores, executando 2 comparações em CPU e 18 comparações em FPGA obteve bons resultados, tanto em tempo de execução como em consumo de energia. Finalmente, a solução projetada para execução híbrida (CPU+FPGA) em plataformas *stand-alone* foi adaptada para execução na instância f1 da nuvem AWS. Na comparação de lotes de 20 sequências de 50K na nuvem AWS, mostramos que a configuração FPGA-*only* com dois co-processadores executou-se em menor tempo e consumiu menos energia do que as configurações CPU-*only* e híbridas.

Palavras-chave: comparação de sequências biológicas,FPGA,Longest Common Subsequence

Abstract

Sequence alignment clearly exposes the most relevant patterns between two sequences, being useful for discovering functional, structural and evolutionary information in biological sequences. Therefore, it is necessary to find the optimal alignment, that is, the pattern that maximizes the similarity between the sequences. The *Longest Common Subsequence* (LCS) algorithm produces the optimal alignment and is therefore widely used around the world. Due to the quadratic complexity of time, its execution can take a long time if the compared sequences are long. Therefore, hardware platforms such as FPGAs (*Field Programmable Gate Arrays*) have been explored. To simplify hardware programming, high-level solutions have been proposed, such as HLS (*High Level Synthesis*), which automatically transforms a C, C++ or SystemC program into a hardware specification, simplifying programming. In order to achieve a certain simplicity of programming and, consequently, reduce the time needed to obtain the circuit, in this thesis we explore programming in HLS and in C. In addition, we investigate solutions for the execution of the LCS algorithm both on FPGA *stand -alone* as on hybrid platforms (CPU+FPGA). Initially, we made a circuit programmed in HLS, which was executed on an FPGA *stand-alone* platform. The designed circuit supports up to two co-processors, allowing the parallel comparison of two pairs of sequences. Executions in the *stand-alone* environment of a batch of 20 comparisons of 10K, 20K and 50K synthetic sequences showed that the FPGA execution consumes less power and that the hybrid executions (CPU+FPGA) achieve very good performance. Additionally, a batch of 20 real sequences of the SARS-CoV-2 virus (30K) on the *stand-alone* platform was compared. In this case, the configuration with 2 coprocessors, running 2 comparisons on CPU and 18 comparisons on FPGA obtained good results, both in execution time and in power consumption. Finally, the solution designed for hybrid execution (CPU+FPGA) on *stand-alone* platforms was adapted to run on the f1 instance of the AWS cloud. In comparing batches of 20 50K sequences in the AWS cloud, we showed that the FPGA-*only* configuration with two co-processors ran in less time and consumed less power than the CPU-*only* configurations. and hybrids.

Keywords: biology sequence comparison,FPGA,Longest Common Subsequence

Sumário

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

A descoberta de padrões em sequências é um dos problemas mais desafiadores em biologia molecular e ciência da computação. Em sua forma mais simples, o problema pode ser formulado da seguinte forma [51]: dado um conjunto de sequências, deve-se encontrar o padrão que ocorre com maior similaridade. No reconhecimento exato de padrões, a busca por um padrão de m letras pode ser rapidamente resolvida por uma simples enumeração de todos os padrões de m letras que aparecem nas sequências. No entanto, quando se trabalha com sequências biológicas, realiza-se a busca aproximada pois os padrões incluem mutações, inserções ou remoções de nucleotídeos [51].

O alinhamento de sequências expõe claramente os padrões mais relevantes, sendo útil para descobrir informação funcional, estrutural e evolucionária em sequências biológicas. Para tanto, é necessário descobrir o alinhamento ótimo, ou seja, o padrão que maximiza a similaridade entre as sequências. Sequências muito parecidas (similares) provavelmente têm a mesma função e, se forem de organismos diferentes, são definidos como homólogos caso tenha existido uma sequência que seja ancestral de ambas. A similaridade de sequências pode ser um indício de várias possíveis relações de ancestralidade, inclusive a ausência de uma origem comum [16, 51].

1.1 Comparação de Sequências em Hardware

Na comparação de duas sequências biológicas calculam-se, a partir de métodos computacionais, métricas que ajudam a identificar o seu grau de relacionamento. Uma dessas métricas é o *escore*, que é atribuído a um alinhamento. Um alinhamento é definido como um pareamento, caractere a caractere, das sequências. Um par de caracteres das duas sequências pode ser definido como *match*, quando os caracteres são iguais; *mismatch*, quando são distintos; ou *gap*, quando o caractere de uma sequência está alinhado com uma lacuna [31].

Para se calcular o *escore* de similaridade, existem diversas abordagens, porém, geralmente, os métodos de alinhamentos visam minimizar o número de *gaps* e *mismatches* penalizando-os no cálculo do *escore* final [66]. Nesse contexto, o alinhamento ótimo é aquele que possui o escore máximo.

Se duas sequências possuem o mesmo ancestral, espera-se que elas possuam muitos símbolos em comum (*matches*). Assim, o alinhamento busca maximizar o número de símbolos comuns das sequências analisadas. Existem dois tipos básicos de alinhamentos [31]: (a) alinhamento global, que alinha todos os caracteres das sequências e (b) alinhamento local, que busca regiões de similaridade máxima entre as sequências, possivelmente alinhando um subconjunto dos caracteres das mesmas.

O desenvolvimento de algoritmos para obtenção dos alinhamentos foi feito em saltos lentos e os principais resultados vieram com espaçamento de vários anos. Em 1970, Needleman e Wunsch [58] propuseram um algoritmo baseado em programação dinâmica que recupera o alinhamento global ótimo, com complexidade quadrática de tempo e memória. Em 1974, Wagner e Fischer [73] definiram o problema *Longest Common Subsequence* (LCS), que visa encontrar a maior subsequência de carateres que é comum entre duas sequências, também com complexidade quadrática tanto de tempo como de memória. Em 1975, Hirschberg [38] propôs um algoritmo de complexidade linear de memória para o problema LCS. A complexidade de tempo, no entanto, continuou quadrática. Em 1981, Smith e Waterman [69] modificaram o algoritmo de Needleman-Wunsch e propuseram um algoritmo que recupera o alinhamento local ótimo, com complexidade de tempo e memória quadráticos. Gotoh [36], em 1982, modificou a equação de recorrência de Needleman-Wunsch e propôs um algoritmo que recupera o alinhamento ótimo para um modelo de lacunas mais elaborado, chamado *affine gap*. Finalmente, em 1988, Myers e Miller [54], combinaram os algoritmos de Gotoh e Hirschberg de maneira a obter o alinhamento ótimo com *affine gap* em memória linear e tempo quadrático.

Apesar de sua longa história, a pesquisa em alinhamento de sequências continua a florescer. O alinhamento de sequências na biologia computacional moderna é a base de muitos estudos de bioinformática e os avanços na metodologia de alinhamento podem conferir benefícios abrangentes em uma ampla variedade de domínios de aplicação. Embora muitas dessas abordagens dependam dos mesmos princípios básicos, os detalhes das implementações podem ter grandes efeitos sobre o desempenho, tanto em termos de precisão quanto de velocidade.

Os algoritmos que produzem resultados ótimos [58, 73, 69, 36, 55] se executam em tempo quadrático ($O(n^2)$) onde n é o tamanho das sequências. Por essa razão, seu tempo de execução é muito grande caso as sequências comparadas sejam longas ou exista um número muito grande de alinhamentos a serem feitos.

O problema *Longest Common Subsequence* (LCS) é conhecido desde o início da década de 1970 e consiste em encontrar a maior subsequência comum entre duas sequências [73]. O LCS é bastante utilizado atualmente em aplicações de comparação de sequências genéticas [29] e também em processamento de linguagem natural [77], dentre outros.

Enquanto se desenvolviam as pesquisas em comparação de sequências, surgiam também diversas propostas para execução mais rápida de aplicações, com hardware específico, tal como ASIC e FPGA.

Application-Specific Integrated Circuit (ASIC) são dispositivos de hardware criados visando um propósito específico ou usados para um sistema específico. São configurados para executar somente um projeto de circuito e, portanto, são pouco flexíveis. Os ASICs possuem um circuito permanente, pois, uma vez que o circuito específico é gravado em silício, ele não pode ser alterado e esse circuito terá o mesmo comportamento por todo o período operacional [68]. A programação de ASICs é feita através de *Hardware Description Languages* (HDLs) como Verilog ou VHDL e possui alta eficiência de consumo de energia, uma vez que esse consumo pode ser controlado e otimizado minuciosamente. Os custos desses projetos e circuitos são relativamente altos, pois necessitam de um conhecimento aprofundado do hardware, curva de aprendizagem alta, e frequentemente o desenvolvimento de componentes adicionais específicos. Por essas razões, o desenvolvimento de um ASIC a partir do zero pode ter um custo financeiro altíssimo dependendo do circuito. Por isso é geralmente utilizado em produção de larga escala [68].

Field Programmable Gate Arrays (FPGAs) são dispositivos que foram projetados como uma matriz de blocos lógicos interconectados que podem ser reconfigurados de acordo com um projeto. Os FPGAs podem ser usados para implementar quaisquer funções lógicas e podem ser configurados após a fabricação [60, 72, 59]. Os FPGAs também possuem a capacidade de reconfigurar uma parte do chip enquanto as áreas restantes do chip ainda estejam funcionais. Da mesma forma que os ASICs, a programação de FPGAs geralmente se baseia em VHDL ou Verilog. A programação com HDL produz circuitos otimizados porém em tempo extremamente alto, pois é necessário conhecimento aprofundado sobre *Register Transfer Language* (RTL), que é usado para descrever a transferência de instruções ou dados entre registradores, sendo um tipo de representação intermediária (IR) muito próxima do hardware. Para simplificar a programação de hardware, surgiram diversos níveis de abstração, dentre eles o HLS [70, 45].

High Level Synthesis (HLS) é uma ferramenta que transforma, de maneira automática, uma especificação em C em uma implementação *Register Transfer Level* (RTL) que pode ser sintetizada em um FPGA. É possível escrever especificações em C, C++ ou SystemC e através da ferramenta HLS é produzido automaticamente o circuito, simplificando o projeto [75]. O objetivo do HLS é, portanto, aumentar a produtividade em projetar hard-

ware, permitindo um nível de abstração maior para criação desses projetos, com possível melhoria de desempenho via software, diminuindo assim o tempo de implementação ao utilizar abstração de uma linguagem de programação de alto nível [76]. Em outras palavras, a ideia principal do HLS é automatizar o processo de transformação da descrição em alto nível de um circuito digital para a descrição RTL [10].

Um conceito importante que surgiu recentemente é o FPGA as a Service (FaaS) [64]. Nesse caso, é fornecida uma plataforma de hardware unificada e *middleware* na nuvem, o que pode reduzir significativamente os custos de desenvolvimento e implantação de projetos de hardware. Nesse modelo, diversos usuários podem utilizar placas FPGA sem ter necessidade de possuir uma estrutura ou equipamentos próprios. A plataforma FaaS consiste em um ou mais FPGAs conectados à um *host* (servidor genérico em nuvem) por meio de uma conexão PCIe. O usuário final pode se conectar remotamente ao *host* e interagir com o FPGA usando utilitários fornecidos pelo provedor de nuvem. Nessa infraestrutura, os recursos do FPGA podem ser compartilhados entre vários usuários ou alocados para uso dedicado em um projeto específico. Um exemplo de provedor de nuvem que oferece FaaS é o *Amazon Web Services* (AWS) [2].

1.2 Motivação

Na literatura, existem diversas soluções para implementação de algoritmos de comparação de sequências biológicas em FPGA [12, 74, 24, 34, 26, 13, 22, 71, 35], porém a grande maioria dessas soluções utilizam linguagens de descrição de hardware (VHDL ou Verilog). Destes, [12, 74, 24, 34, 26, 71, 35] implementam os algoritmos Needleman-Wunsch ou Smith-Waterman e somente em Bonny et al. [22] é proposta uma solução LCS em FPGA, mas também utiliza HDL como linguagem de programação. A nosso conhecimento, não há na literatura estudo sobre o algoritmo LCS em alto nível para plataformas FPGA.

Portanto, a motivação da presente Tese é desenvolver uma solução de comparação de sequências biológicas de maneira rápida e simples utilizando a abstração de uma linguagem de programação de alto nível como C, C++ ou HLS, visando a flexibilidade, portabilidade, desempenho e maior eficiência de consumo de energia que a plataforma FPGA pode proporcionar. Ainda tendo como referência os trabalhos na literatura que implementam tanto em CPU quanto em FPGA [26, 13], visamos explorar o uso de ambas as plataformas como um ambiente híbrido de execução, a fim de melhorar o desempenho e utilizar recursos oferecidos tanto pela CPU como pelo FPGA.

1.3 Objetivo e Contribuições

O objetivo da presente Tese é investigar estratégias e soluções para execução do algoritmo LCS com baixo consumo de energia e baixo tempo de execução em FPGA e em plataformas híbridas (CPU+FPGA), utilizando linguagens de programação de alto nível, para acelerar o problema de comparação de sequências biológicas.

Nesse sentido, as contribuições da presente Tese são:

- Uma solução LCS em FPGA utilizando HLS para comparação de duas sequências de DNA de tamanho médio (até 50k), sendo um dos primeiros trabalhos com tamanho médio, analisando os recursos utilizados, tempo de execução e consumo de energia do projeto proposto. Na comparação de duas sequências de 50k com o circuito proposto, a energia gasta foi 425,60 Joules em FPGA enquanto a mesma comparação gastou 2907,90 Joules em CPU. O artigo descrevendo a solução LCS em FPGA foi publicado no evento WSCAD 2020 [40];

- Uma solução LCS em HLS para plataformas híbridas (CPU+FPGA), com um escalonador ponderado que divide a execução das comparações para as duas plataformas. Ao comparar 20 pares de sequências de 50k com o circuito proposto e com as configurações CPU-*only* e FPGA-*only*, a energia utilizada foi de 8572,74 Joules em FPGA enquanto as mesmas comparações consumiram 72019,44 Joules em CPU. Porém, na execução com a configuração híbrida CPU+FPGA com dois co-processadores em FPGA, duas comparações em CPU e dezoito comparações no FPGA, constatamos que houve uma redução de cerca de 67,45% no consumo de energia em relação à configuração CPU-*only* e uma redução de 36,65% no tempo de execução em relação à configuração FPGA-*only*. O artigo descrevendo a solução LCS para plataformas híbridas *stand-alone* foi publicado no periódico *Concurrency and Computation: Practice and Experience* 2020 [41]

- Uma solução LCS em HLS para plataformas híbridas (CPU+FPGA), com um escalonador ponderado que divide a execução das comparações para as duas plataformas, em ambientes de nuvem computacional. A confecção dessa solução envolveu desafios referentes à programação de FPGAs na nuvem e sua posterior execução, de maneira que não haja alta degradação de desempenho devido à virtualização. Ao comparar 20 pares de sequências de 50k com o circuito proposto e com as configurações CPU-*only* e FPGA-*only*, a energia utilizada foi de 9383,31 Joules em FPGA enquanto as mesmas comparações consumiram 77921,57 Joules em CPU. Porém, na execução com a configuração híbrida CPU+FPGA com dois co-processadores

em FPGA, duas comparações em CPU e dezoito comparações no FPGA, constatamos que houve uma redução de cerca de 69,12% no consumo de energia em relação à configuração CPU-*only*, entretanto na execução FPGA-*only* houve uma redução de 42,43% no tempo de execução em relação à configuração CPU-*only*. O artigo descrevendo a solução na nuvem está sendo confeccionado.

Este documento está organizado em três partes. A primeira parte apresenta o *Background* e a Contextualização do problema e é composta de 3 capítulos. O Capítulo 2 apresenta o problema de Comparação de Sequências Biológicas e os principais algoritmos que resolvem esse problema. O Capítulo 3 discorre sobre FPGAs, apresentando sua estrutura, arquitetura e fluxo de execução, bem como a programação de hardware em alto nível. Por fim, o Capítulo 4 discute os Trabalhos Relacionados. A segunda parte, composta em 3 capítulos, apresenta as contribuições da presente Tese. O Capítulo 5 apresenta o projeto da solução LCS utilizando HLS e os resultados experimentais obtidos, considerando tanto o tempo como o consumo de energia. O Capítulo 6 apresenta a arquitetura híbrida CPU+FPGA para executar o algoritmo LCS em plataforma *stand-alone* bem como os resultados obtidos para tempo de execução e consumo de energia. O Capítulo 7 apresenta o projeto para execução do algoritmo LCS em ambiente híbrido CPU+FPGA na nuvem e os resultados obtidos. E por fim, terceira parte apresenta a conclusão da presente Tese e sugestões de trabalhos futuros.

Parte I

Background/Contextualização

Capítulo 2

Comparação de Sequências Biológicas

2.1 Visão Geral

As sequências biológicas desempenham um papel importante na vida dos organismos e por isso são muito estudadas no campo da Biologia Molecular. Uma sequência biológica é uma cadeia ordenada de caracteres que representam resíduos, podendo ser nucleotídeos (sequências de DNA / RNA) ou aminoácidos (sequências de proteínas) [51].

As sequências de DNA (Ácido Desoxirribonucleico) são compostas por 4 nucleotídeos: A-Adenina, T-Timina, G-Guanina, C-Citosina e armazenam informação genética. De maneira similar ao DNA, as cadeias de RNA (Ácido Ribonucleico) também são construídas a partir de nucleotídeos. O açúcar de monossacarídeo do RNA é ribose em vez de desoxirribose, podendo ser composto 4 nucleotídeos, com U-Uracila em vez de T-Timina [31]. As proteínas possuem como blocos de construção os aminoácidos, que são cadeias de 3 nucleotídeos (A, T, G ou C). Existem na natureza 20 aminoácidos: Y-Tirosina, W-Triptofano, V-Valina, T-Treonina, S-Serina, R-Arginina, Q-Glutamina, P-Prolina, N-Asparagina, M-Metionina, L-Leucina, K-Lisina, I-Isoleucina, H-Histidina, G-Glicina, FP-Fenilalanina, ácido E-Glutâmico, ácido D-Aspártico, C-Cisteína e A-Alanina [51].

A comparação de sequências biológicas é uma operação muito frequente em Bioinformática. Seu objetivo é determinar o quão similares duas sequências são e essa informação é usada em muitos problemas complexos, como determinação de aspectos evolutivos dos organismos, construção de árvores filogenéticas, entre outros [31]. Como resultado, a comparação de sequências produz um escore, que é uma métrica de similaridade, e um alinhamento, onde os caracteres similares e diferentes entre as sequências são ressaltados [19].

No cálculo, são atribuídos valores para cada par de caracteres analisado, de acordo com a correspondência entre as duas sequências, seja ela um *match* (iguais), um *mismatch* (diferentes) ou quando é inserido um *gap* (lacuna). Esses valores podem variar, mas na literatura são comumente utilizados valores positivos para *match* e valores negativos para *mismatch* e *gaps* [51, 20, 47]. O escore final é obtido com o somatório dos valores atribuídos em cada uma das posições analisadas das sequências.

A Figura 2.1 ilustra tanto o alinhamento como o escore de uma comparação de seqüências de DNA. Nessa figura, para cada par de caracteres iguais (*match*), o valor 1 é atribuído. Caso as bases comparadas sejam diferentes (*mismatch*), então a penalidade -1 é atribuída. Para obter melhores pontuações, é possível adicionarmos lacunas (*gaps*) nas sequências. Tal artifício é compatível com o processo evolutivo das sequências genéticas e corresponde à deleções ou inserções [51]. Para fins de pontuação, tal situação possui uma penalidade maior, no caso ilustrado, -2.

S_0	-	G	A	C	-	T	A	C	A	T	G	A	T	C		
		‖	‖	‖		‖	‖	‖		‖			‖			
S_1	C	G	A	C	C	T	A	C	G	T	T	-	T	-		
	-2	+1	+1	+1	-2	+1	+1	+1	-1	+1	-1	-2	+1	-2	=	-2

Figura 2.1: Exemplo de alinhamento entre duas sequências de DNA, com escore = -2.

Um alinhamento pode ser global, que contem todos os caracteres das duas sequências, ou local, contendo um subconjunto de caracteres das sequências. Existe ainda o alinhamento semi-global, onde o prefixo ou sufixo de uma das sequências é descartado. O alinhamento mostrado na Figura 2.1 é um alinhamento global e pode não ser aquele que possui o melhor escore. Caso o alinhamento tenha o melhor escore possível, dizemos que o alinhamento é ótimo. Nas seções de 2.3 a 2.8 descreveremos algoritmos para a obtenção do alinhamento ótimo.

2.2 Notações e Definições usadas no Capítulo

A Tabela 2.1 apresenta as notações utilizadas ao longo do presente capítulo. Como é tratado o problema de comparação par-a-par, temos que duas sequências (S_0 e S_1) serão comparadas. A notação $S_u[1..v]$ representa o prefixo de tamanho v da sequência S_u e a notação $S_u[i]$ denota o i-ésimo caractere da sequência S_u. A função $sbt(S_0[i], S_1[j])$ compara os caracteres $S_0[i]$e $S_1[j]$, associando valores de *match* e *mismatch*, dependendo do resultado da comparação. A penalidade associada ao *gap* pode ser G, caso o modelo linear de *gap* seja utilizado, ou G_{first} e G_{ext}, caso o modelo *affine gap* (Seção 2.6 seja usado.

Tabela 2.1: Notações e Definições do Capítulo

Símbolo	**Descrição**
S_u	Sequências a serem comparadas – S_0 e S_1
m	Tamanho da sequência S_0
n	Tamanho da sequência S_1
$S_u[1..v]$	Prefixo da sequência S_u, com v caracteres
$S_u[i]$	i-ésimo caracter da sequência S_u
$sbt(S_0[i], S_1[j])$	Função que compara $S_0[i]$e $S_1[j]$, associando um valor à comparação
G	Penalidade associada ao gap (modelo linear)
G_{first}	Penalidade associada ao primeiro gap (modelo affine)
G_{ext}	Penalidade associada aos demais gaps consecutivos (modelo affine)

2.2.1 Definição 1 - Sequência

Define-se como sequência o encadeamento ordenado de objetos representados por símbolos, comumente caracteres alfa-numéricos [51]. No caso da presente Tese, as letras A, C, G e T representam os quatro nucleotídeos de uma cadeia de DNA (i.e. as bases adenina, citosina, guanina e timina), e sequências como ACCCGGTTT representam uma sequência de DNA.

2.2.2 Definição 2 – Subsequência

Uma subsequência da sequência $S_u = S_u[0], S_u[1], \ldots, S_u[m-1]$ é um string na forma $S_u[i], S_u[j], \ldots, S_u[k]$ de tal forma que $i < j < \ldots < k$. Uma subsequência qualquer pode ser obtida extraindo-se zero ou mais caracteres da sequência original, mantendo seu ordenamento. Por exemplo, as sequências ACCCGGTTT, ACC, AGTT e ACGT são todas subsequências de ACCCGGTTT. Por extensão, a sequência vazia é subsequência de todas as sequências da natureza, e toda sequência é subsequência de si própria [73].

2.3 Algoritmo Longest Common Subsequence (LCS)

O problema *Longest Common Subsequence (LCS)* foi proposto em 1974 e visa encontrar a subsequência comum máxima entre 2 sequências [73], retornando o resultado ótimo.

Em sua formulação genérica, o LCS é resolvido com um algoritmo de programação dinâmica, que retorna o resultado ótimo. O algoritmo é executado em duas fases: (1) obtenção do escore LCS e (2) obtenção do alinhamento LCS. Na fase 1, calcula-se a matriz de programação dinâmica LCS com a equação de recorrência mostrada na Equação 2.1 [73].

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

Figura 2.2: Matriz de programação dinâmica H entre as sequências S_0 and S_1.

$$LCS[i,j] = max \begin{cases} LCS[i-1,j] \\ LCS[i,j-1] \\ LCS[i-1,j-1] + sbt(i,j) \end{cases} \tag{2.1}$$

Onde $sbt(i,j) = \ 1$, se $S_0[i] = S_1[j]$; 0 caso contrário.

Ao final dessa fase, o escore LCS estará na posição $LCS[m,n]$. O escore LCS é também conhecido como Distância de Edição ou Distância de Levenshtein [51].

A fase 2 percorre a matriz LCS a partir da posição $LCS[m,n]$ com dois laços (um para cada sequência), do fim para o início das sequências. Se os caracteres $S[i]$ e $S[j]$ são iguais então esse caractere faz parte da LCS e tanto i como j são decrementados. Senão, $LCS[i-1,j]$ e $LCS[i,j-1]$ são comparados. Se $LCS[i-1,j]$ for maior, i é decrementado. Caso contrário, j é decrementado.

A Figura 2.2 apresenta a matriz LCS para as sequências $S_0 = AATCGC$ e $S_1 = ATGAC$, onde o escore LCS é 4 e a maior subsequência comum é ATGC. Os valores destacados representam as posições utilizadas na obtenção do LCS.

O algoritmo LCS possui complexidade quadrática de tempo e memória ($O(mn)$), onde m e n são os tamanhos das sequências.

2.4 Algoritmo Needleman-Wunsch (NW) - Alinhamento Global

O algoritmo de Needleman-Wunsch (NW) [58] foi proposto em 1970 e é utilizado para ser obter o alinhamento global ótimo entre duas sequências S_0 e S_1, permitindo-se uso de *gaps* para melhorar o alinhamento. Da mesma forma que o LCS (Seção 2.3), possui complexidade quadrática de tempo e memória, ($O(mn)$).

O alinhamento global ótimo é obtido a partir de duas fases: (1) construção da matriz de programação dinâmica H; e (2) *traceback* para construir o alinhamento propriamente dito.

Na primeira fase, calcula-se uma matriz H onde o elemento $H_{i,j}$ representa o escore do alinhamento global ótimo entre os prefixos $S_0[1..i]$ e $S_1[1..j]$. 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 inicializadas com os valores $H_{i,0} = -i \cdot G$ e $H_{0,j} = -j \cdot G$, onde G é a penalidade linear de *gap*. A equação de NW baseia-se no fato de que o valor $H_{i,j}$ de um alinhamento de prefixos terminado nos caracteres $S_0[i]$ e $S_1[j]$ deve ser o maior escore (operação max) dentre os seguintes cenários: (1) alinhar $S_0[1..i-1]$ e $S_1[1..j-1]$ acrescido dos caracteres $S_0[i]$ e $S_1[j]$; (2) alinhar $S_0[1..i]$ e $S_1[1..j-1]$ e inserir um *gap* alinhado com $S_1[j]$; (3) alinhar $S_0[1..i-1]$ e $S_1[1..j]$ e inserir $S_0[i]$ alinhado com um *gap*. Desta forma, a recorrência é descrita pela Equação 2.2. Além dos valores, cada célula da matriz H contem uma indicação sobre a célula que foi usada para produzir o valor ($H_{i-1,j-1}, H_{i-1,j}$ ou H_{i-j-1}).

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

A fase de *traceback* tem por objetivo recuperar o alinhamento global ótimo. Assim, percorre-se a sequência de indicações contidas na matriz no sentido reverso, iniciando no canto inferior direito da matriz H até alcançar o canto superior esquerdo da matriz H. Existem casos em que o *traceback* irá percorrer vários caminhos na matriz, significando que existe mais de um alinhamento global ótimo [31]. De maneira prática, geralmente recupera-se um único alinhamento ótimo, dando prioridades às indicações. Por exemplo, indicações $H_{i-1,j-1}$ possuem geralmente precedência sobre as outras e indicações $H_{i,j-1}$ possuem precedência sobre $H_{i-1,j}$.

Para as sequências S_0=CGACCTACGTTT e S_1=GATTACATGATC, a matriz $H[m,n]$ produzida pelo algoritmo NW é mostrada na Tabela 2.2, com escore ótimo igual a -4 (canto inferior direito da matriz). A Figura 2.3 apresenta o alinhamento global ótimo obtido na fase de *traceback*.

C	G	A	C	C	T	A	C	G	T	T	-	T	-	
-	G	A	T	-	T	A	C	A	T	G	A	T	C	
-2	1	1	-1	-2	1	1	1	-1	1	-1	-2	1	-2	=	-4

Figura 2.3: Alinhamento global ótimo entre as sequências S_0=CGACCTACGTTT e S_1=GATTACATGATC.

Tabela 2.2: Matriz de programação dinâmica gerada pelo alinhamento global ótimo entre as sequências S_0=CGACCTACGTTT e S_1=GATTACATGATC utilizando o algoritmo NW, com ma=+1, mi=-1 e gap=-2.

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

2.5 Algoritmo Smith-Waterman (SW)

Para se obter o alinhamento local ótimo entre as sequências S_0 e S_1, utiliza-se o algoritmo de Smith-Waterman (SW) [69], que foi proposto em 1981. De maneira similar aos algoritmos LCS (Seção 2.3) e NW (Seção 2.4), possui complexidade quadrática de tempo e memória.

O algoritmo SW possui muitas semelhanças com o algoritmo NW (Seção 2.4), com três principais diferenças. A primeira diferença ocorre no preenchimento com zeros na primeira linha e coluna da matriz H. A segunda diferença ocorre na equação de recorrência (Equação 2.3), que utiliza o valor zero para impedir que números negativos apareçam na matriz de programação dinâmica. A ocorrência de um valor zero representa o começo de um novo alinhamento, pois se existem pontos negativos na matriz, então começar um novo alinhamento a partir daquela posição é mais vantajoso para o alinhamento local [69]. A equação de recorrência de SW está descrita na Equação 2.3.

$$H_{i,j} = max \begin{cases} H_{i-1,j-1} + sbt(S_0[i], S_1[j]) \\ H_{i-1,j} - G \\ H_{i,j-1} - G \\ 0 \end{cases} \tag{2.3}$$

A terceira mudança em relação ao algoritmo NW (Seção 2.4) ocorre na forma de se realizar o *traceback*. Em vez de iniciar o *traceback* na posição final $H_{m,n}$, inicia-se na

Tabela 2.3: Matriz de programação dinâmica gerada pelo alinhamento local ótimo entre as sequências S_0=CGACCTACGTTT e S_1=GATTACATGATC com o algoritmo SW com ma=+1,mi=-1 e gap=-2.

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

posição $H_{i,j}$ que possui o maior valor da matriz. O percurso é feito até encontrar uma célula com valor zero. Dessa forma, obtém-se o melhor alinhamento local entre duas sequências.

Ainda utilizando como exemplo as sequências S_0=CGACCTACGTTT e S_1=GATTAC ATGATC, temos a matriz H, calculada com o algoritmo SW, ilustrada na Tabela 2.3. A Figura 2.4 apresenta o alinhamento local ótimo, com escore +3. Os valores em destaque indicam o percurso reverso (*traceback*) capaz de produzir este alinhamento local ótimo entre as duas sequências.

```
T A C G T
‖ ‖ ‖   ‖
T A C A T
```

Figura 2.4: Alinhamento local ótimo entre as sequências S_0=CGACCTACGTTT e S_1=GATTACATGATC.

2.6 Algoritmo de Gotoh (Affine Gap)

Gotoh [36] considerou o modelo *affine gap*, em que iniciar uma nova sequência de *gap* possui maior penalidade do que uma extensão de uma sequência de *gap* já existente[58]. Em seu algoritmo, duas novas matrizes são calculadas para avaliar a abertura de novos *gaps* ou extensão de uma sequência de *gap* existente em cada sequência considerada.

Definem-se então três matrizes H, E e F para cada uma das situações. A equação de recorrência é então modificada conforme as Equações 2.4 a 2.6. Adicionalmente, pode-se incluir uma condição para limitar o valor mínimo da matriz H em zero, adaptando-o assim para a obtenção do alinhamento local. Da mesma forma que LCS (Seção 2.3), NW (Seção 2.4) e SW (Seção 2.5), o algoritmo de Gotoh possui complexidade quadrática de tempo e memória.

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

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

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

Mais uma vez, utilizando como exemplo as duas sequências $S_0 = CGACCTACGTTT$ e $S_1 = GATTACATGATC$ foi gerado o alinhamento affine gap ótimo ilustrado na Figura 2.5.

Os parâmetros utilizados para esse exemplo foram: *Match*: +1; *Mismatch*: -1; *Abertura de Gap*: -2; *Extensão de Gap*: -1 [36].

```
C  G  A  C  C  T  A  C  -  -  G  T  T  T
-  G  A  -  T  T  A  C  A  T  G  A  T  C
```

Figura 2.5: Alinhamento ótimo entre as sequências S_0=CGACCTACGTTT e S_1=GATTAC ATGATC utilizando algoritmo de Gotoh.

2.7 Algoritmo de Hirschberg

Hirschberg [38] propôs em 1975 um algoritmo para solucionar o problema da LCS (Seção 2.3) em memória linear. Esse algoritmo foi facilmente adaptado para os algoritmos de Needleman-Wunsh (Seção 2.4), Smith-Waterman (Seção 2.5) e Gotoh (Seção 2.6). Sua complexidade de tempo é quadrática ($O(mn)$) porém a complexidade de memória é linear ($O(min(n, m))$). Nesse algoritmo, o cálculo da matriz de programação dinâmica é feito armazenando apenas duas linhas: a linha atual e a anterior, diminuindo, assim, a complexidade de memória.

O algoritmo de Hirschberg trabalha com a idéia de partição binária [73]. Dadas duas sequências S_0 e S_1, será processada uma matriz H_{mn}, onde m é o número de linhas e n número de colunas e, utilizando a estratégia de dividir para conquistar, o algoritmo irá dividir a matriz em duas partes: H_A e H_B. O ponto no qual a divisão irá ocorrer será determinado pela linha i na qual passa o alinhamento ótimo na coluna $n/2$. Para descobrir isto, os escores são calculados da mesma forma que nos algoritmos LCS(Seção 2.3), NW(Seção 2.4) ou SW(Seção 2.5), no sentido tradicional (do início da matriz até o meio) e no sentido reverso (do final da matriz até o meio).

Quando o cálculo dos escores chegar na coluna $n/2$, a posição onde a soma dos escores obtidos da duas maneiras é máxima representa a célula por onde passa o alinhamento global ótimo $H(i, n/2)$. A partir deste ponto, a matriz é particionada em duas matrizes $H(1..i, 1..(n/2))$ e $H(i..m, (n/2)..n)$ e o processo é repetido dentro das submatrizes, recuperando dentro delas o elemento pertencente ao alinhamento ótimo até que a recuperação seja trivial.

A recursão de Hirschberg produz a árvore ilustrada na Figura 2.6. As folhas da árvore contêm o alinhamento ótimo.

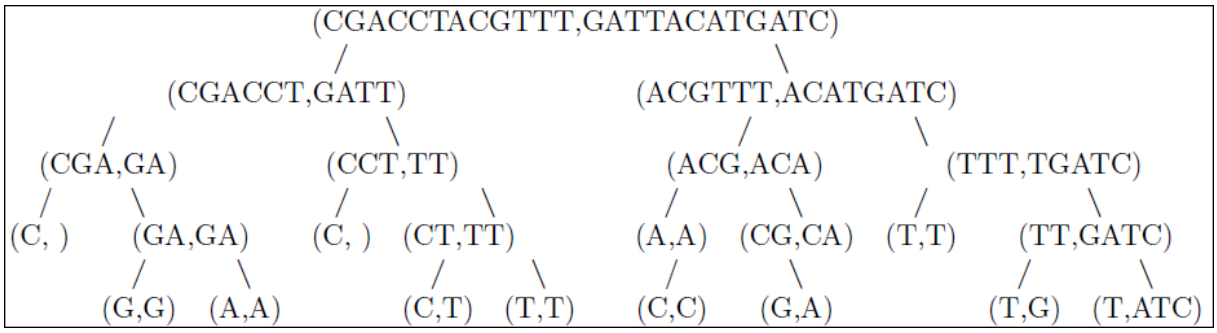

Figura 2.6: Árvore gerada pela recursividade do Algoritmo de Hirschberg.

Graficamente, a execução de Hirschberg é ilustrada pela Figura 2.7 até a segunda recursão, onde os pontos representados por bolas pretas pertencem ao alinhamento ótimo.

2.8 Algoritmo de Myers e Miller (MM)

O algoritmo Myers-Miller (MM) [54] se baseia na combinação do algoritmo de Gotoh (Seção 2.6) com o algoritmo de Hirschberg (Seção 2.7) para se obter uma solução com complexidade de memória linear com alinhamento global utilizando *affine gaps*.

Da mesma maneira que Hirschberg (Seção 2.7), essa abordagem aplica divisões sucessivas na matriz de escores, determinando um ponto médio para cada iteração para a maior subsequência podendo ser tanto para linha quanto para coluna. Através disso, é

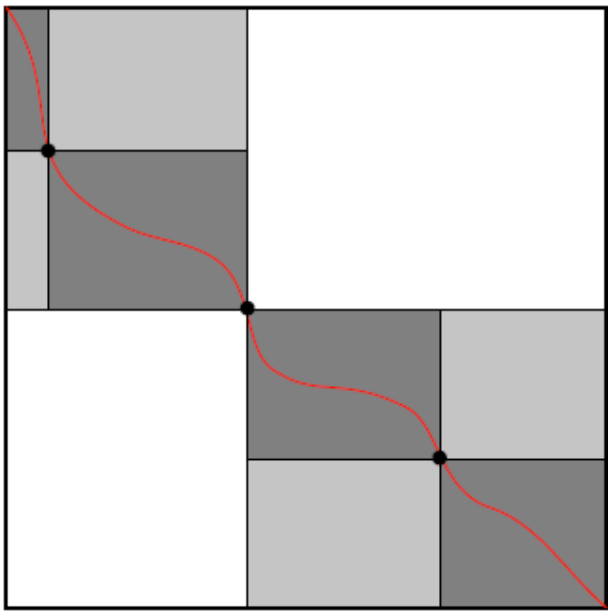

Figura 2.7: Exemplo do algoritmo de Hirschberg percorrendo o caminho pelos pontos médios [30].

necessário uma operação que obtém o ponto médio do alinhamento ótimo ao considerar a linha ou coluna. Essa operação utiliza cálculo do custo mínimo de conversão de vetores que se obtém através da matriz de escores desde o início até a linha média e desde o final até essa linha média. A partir desse ponto, assim como Hirschberg, é utilizado recursivamente o mesmo procedimento para as demais subsequências. Como o modelo *affine gap* é usado, leva-se em consideração o caso do alinhamento ótimo possuir uma sequência de gaps.

Capítulo 3

Hardware Reconfigurável - FPGA

3.1 Introdução

Hardware reconfigurável se destina a preencher a lacuna entre hardware e software, alcançando desempenho potencialmente maior do que o software, enquanto mantém um nível mais alto de flexibilidade do que o hardware tradicional. Dispositivos reconfiguráveis, incluindo *Field Programable Gate Arrays*(FPGAs), contêm uma matriz de elementos computacionais cuja funcionalidade é determinada por meio de vários bits de configuração programáveis. Esses elementos, conhecidos como blocos lógicos, são conectados por meio de um conjunto de recursos de roteamento que também são programáveis. Desta forma, os circuitos digitais personalizados podem ser mapeados para o hardware reconfigurável executando as funções lógicas do circuito dentro dos blocos lógicos e usando o roteamento configurável para conectar os blocos para formar o circuito necessário [27].

Os FPGAs e a computação reconfigurável são capazes de acelerar uma variedade de aplicações. Para obter esses benefícios de desempenho, mas com suporte a uma ampla gama de aplicativos, os sistemas reconfiguráveis podem ser formados com uma combinação de lógica reconfigurável e um microprocessador de uso geral. O processador executa as operações que não podem ser feitas de forma eficiente na lógica reconfigurável, como controle dependente de dados e possivelmente acessos à memória, enquanto os núcleos computacionais são mapeados para o hardware reconfigurável. Essa lógica reconfigurável pode ser composta de FPGAs comerciais ou hardware configurável personalizado [27].

O hardware reconfigurável começou com o uso de dispositivos lógicos programáveis (PLD) na década de 1960, onde esses dispositivos foram capazes de adotar arquiteturas flexíveis e funcionais. Na década de 1970, com o advento de dispositivos baseados em memória somente leitura (ROM), como Programmable Logic Array (PLA), um novo método de implementação de funções lógicas em dispositivos programáveis foi introduzido. Desta forma, mais funções novas puderam ser adicionadas, como lógica sequencial [27, 45].

O conceito de dispositivos reconfiguráveis programáveis está diretamente relacionado à capacidade de programar e reprogramar chaves programáveis e sua arquitetura de roteamento. A computação reconfigurável fornece otimizações para várias implementações, como um alto grau de paralelismo, permitindo que várias tarefas sejam executadas ao mesmo tempo, o que tem um impacto positivo no tempo de execução [27, 45].

Um dos primeiros dispositivos lógicos programáveis foi a memória PROM (*Programmable Read Only Memory*), que é uma memória não volátil que pode ser carregada com informações. Os diferentes tipos de PROM podem ser programados em massa (chips programáveis por máscara) ou programados pelo usuário [60].

A PROM programável em campo pode ser de dois tipos: EPROM (*Erasable PROM*) e EEPROM (*Electronic Erasable PROM*). A EEPROM, mais comumente usada, permite que o usuário apague o conteúdo da memória e reprograme-o várias vezes [23].

Em paralelo, surgiram os *Programmable Logic Devices* (PLDs) que normalmente implementam um conjunto de portas lógicas OR fixas precedidas por uma matriz de portas lógicas programáveis AND e são fabricados como programáveis em fábrica ou programáveis pelo usuário [23].

A seguir, a empresa Xilinx criou um dispositivo que não teria apenas portas programáveis, mas também interconexões programáveis entre as portas. Esse dispositivo ficou conhecido como FPGAs (*Field Programmable Gate Arrays*) e foi, portanto, o próximo passo na evolução dos PLDs [70].

Os FPGAs são dispositivos de silício pré-fabricados que podem ser eletricamente programados no campo para se tornar quase qualquer tipo de circuito digital ou sistema. A Xilinx introduziu os primeiros FPGAs em 1984, embora eles não fossem chamados de FPGAs até que a Actel popularizou o termo por volta de 1988 [70].

Para produções de baixo a médio porte, os FPGAs oferecem uma solução mais barata e um tempo de comercialização mais rápido em comparação aos ASIC (*Application Specific Integrated Circuits*), que normalmente exigem muitos recursos em termos de tempo e dinheiro para obter o primeiro dispositivo.

3.2 Arquitetura do FPGA

A Figura 3.1 mostra uma arquitetura FPGA tradicional. Os *Configurable Logic Blocks* (CLBs) são dispostos em uma grade 2D e são interligados por uma rede de roteamento programável. Os blocos de Entrada/Saída (E/S) localizam-se na periferia do FPGA e também estão conectados à rede de roteamento programável. A rede de roteamento compreende trilhas de canais de roteamento horizontal e vertical e é composta por *Connection Blocks* (CB) e *Switch Blocks* (SB). As SBs conectam trilhos de roteamento horizontais e

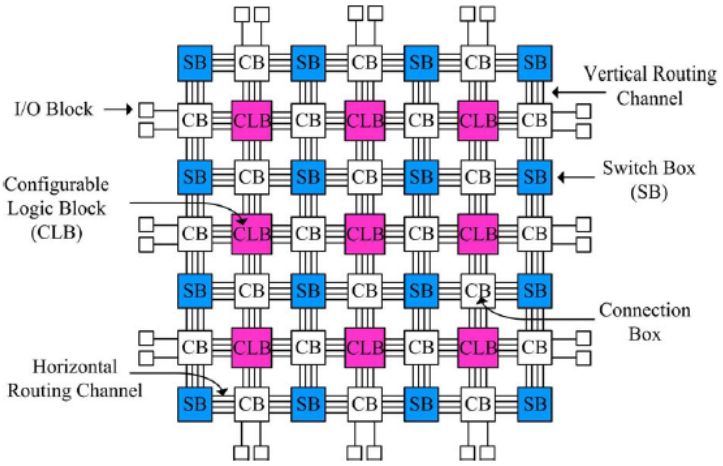

Figura 3.1: Visão geral da Arquitetura FPGA [21].

verticais da rede de roteamento enquanto as CBs conectam os pinos lógicos (*jumpers*) e aos blocos de E/S.

Um fluxo de software converte um circuito de hardware de destino em CLBs e instâncias de E/S interconectadas e, em seguida, mapeia-os no FPGA. O fluxo de software também gera um *bitstream*, que é programado no FPGA para executar o circuito de hardware alvo [60, 72].

A interligação de roteamento de um FPGA consiste de fios e interruptores programáveis que formam a conexão necessária. Uma vez que as arquiteturas FPGA afirmam ser candidatas potenciais para a implementação de qualquer circuito digital, sua interconexão de roteamento deve ser muito flexível para que eles possam acomodar uma grande variedade de circuitos com demandas de roteamento muito variadas. Embora os requisitos de roteamento variem de circuito para circuito, certas características comuns desses circuitos podem ser usadas para o projeto otimizado da interligação de roteamento da arquitetura FPGA [60, 72].

3.2.1 Bloco Lógico Configurável

Um bloco lógico configurável (CLB) é um componente básico de um FPGA que fornece a lógica básica e a funcionalidade de armazenamento. Um CLB pode ser composto de um único *Basic Logic Element* (BLE), ou um conjunto de BLEs interconectados localmente. Um BLE simples consiste em uma *Look-Up Table* (LUT) e um *Flip-Flop*. A Figura 3.2 mostra um BLE composto de uma tabela de consulta de 4 entradas (LUT-4) e um *Flip-Flop* de tipo D. A Figura 3.3 mostra um CLB formado por 4 BLEs [21, 60].

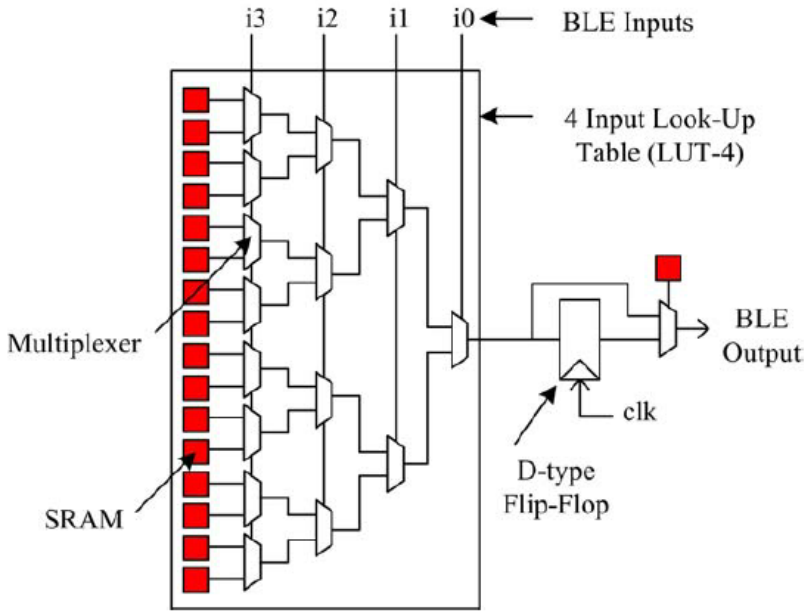

Figura 3.2: *Basic Logic Element*(BLE) [21].

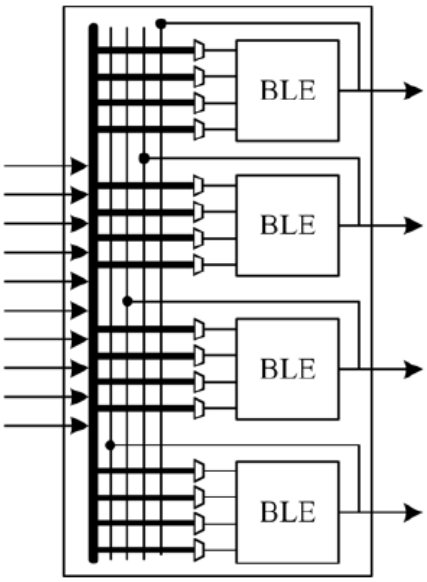

Figura 3.3: Bloco Lógico Configurável(CLB) [21].

Uma LUT com maior número de entradas reduz o número total de LUTs necessários para mapear um circuito de hardware, pois mais funcionalidades lógicas podem ser mapeadas em uma única LUT. Isso, eventualmente, reduz a intercomunicação entre LUTs e, portanto, a velocidade do circuito de hardware melhora [21].

Embora aqui sejam apresentados apenas blocos lógicos básicos, muitos FPGAs modernos contêm uma mistura heterogênea de blocos, alguns dos quais de propósito específico. Os blocos de propósito específico, também chamados blocos rígidos, incluem blocos de memória local pequena e rápida (BRAMs), multiplicadores, somadores e blocos DSP (*Digital Signal Processors*), dentre outros [32, 21].

3.2.2 Rede de roteamento

Tipicamente, a rede de roteamento de um FPGA ocupa $80 - 90\%$ da área de chip do FPGA, enquanto a área lógica ocupa apenas $10 - 20\%$ de área [21]. A flexibilidade de um FPGA depende principalmente de sua rede de roteamento programável. Uma rede de roteamento FPGA consiste em trilhas de roteamento horizontais e verticais que estão interligados através de SBs. Os blocos lógicos são conectados à rede de roteamento através de CB. A flexibilidade de um bloco de conexão (Fc) é dado pelo número de trilhas de roteamento do canal adjacente que estão conectadas ao pino de um bloco.

As trilhas de roteamento conectadas através de uma caixa de comutação podem ser bidirecionais ou unidirecionais [46]. A Figura 3.4 mostra um bloco de comutação bidirecional e unidirecional com Fs igual a 3. As faixas de entrada (ou fios) em ambas as caixas de comutação se conectam a 3 outras faixas da mesma caixa de comutação.

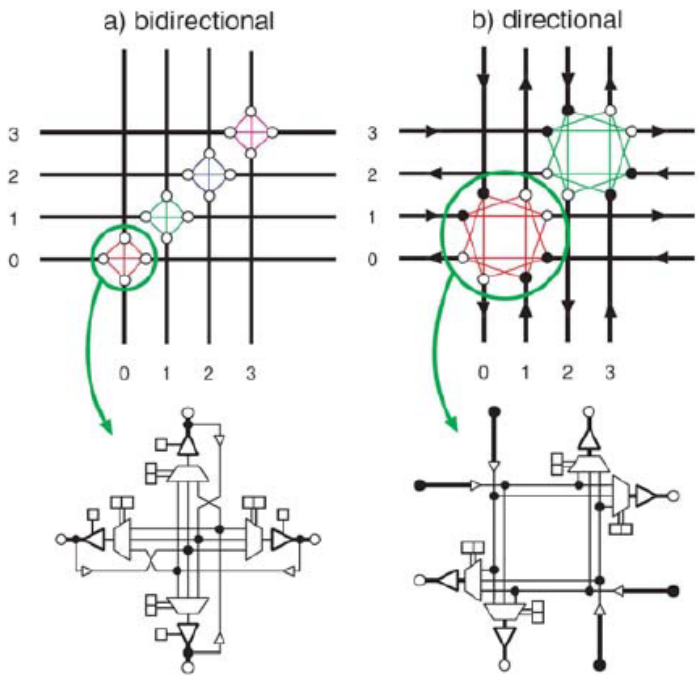

Figura 3.4: Bloco de Comutação (Switch Block) [46].

3.2.3 Fluxo de Projeto

O fluxo de projeto (*Design Flow*) compreende os passos necessários para que uma descrição de operações seja transformada em um arquivo binário executável que descreve um hardware. Em FPGAs, a saída do fluxo de projeto é um *bitstream*.

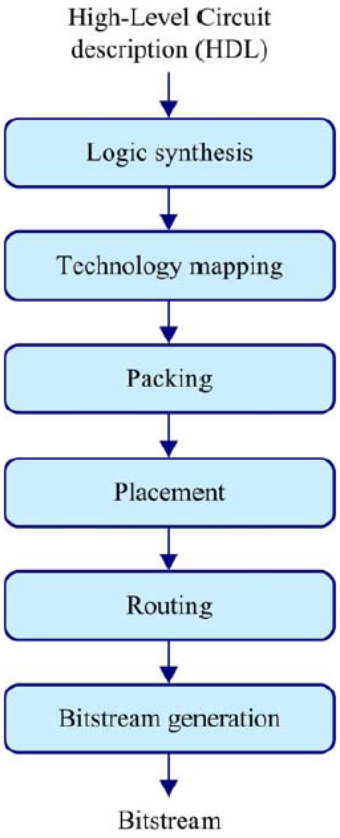

Figura 3.5: Fluxo de Projeto de um FPGA [60].

Portanto, o fluxo de projeto baseado em HDL parte geralmente de uma descrição do projeto em uma Linguagem de Descrição de Hardware (HDL) e converte-o em um *bitstream* que eventualmente é programado no FPGA. Em geral, o processo de converter uma descrição de circuito em um formato que pode ser carregado e executado em um FPGA pode ser dividido em cinco etapas distintas, conforme ilustrado na Figura 3.5 [60].

Inicialmente, a síntese lógica transforma uma descrição de HDL (por exemplo, VHDL ou Verilog) em um conjunto de portas booleanas e *flip-flops*. A seguir, as ferramentas de síntese transformam a descrição do nível de transferência de registro (RTL) de um projeto em uma rede booleana hierárquica [33].

O problema de mapeamento de tecnologia visa encontrar uma rede de células que implementa a rede booleana. No caso do FPGA, as células são compostas por LUTs e

flip-flops. A fase de agrupamento (*packing*) do fluxo do FPGA é o processo de formação de *clusters* de kLBs. Esses *clusters* podem então ser mapeados diretamente para um elemento lógico em um FPGA.

Os algoritmos de posicionamento determinam qual bloco lógico dentro de um FPGA deve implementar o bloco lógico correspondente requerido pelo circuito. Os objetivos de otimização consistem em colocar blocos lógicos conectados juntos para minimizar a fiação necessária (posicionamento acionado pelo comprimento do fio) e, às vezes, posicionar blocos para balancear a densidade de fiação no FPGA (posicionamento direcionado por roteamento) ou maximizar a velocidade do circuito direcionada.

O problema de roteamento do FPGA consiste em atribuir redes aos recursos de roteamento de maneira a conectar os blocos lógicos já dispostos no FPGA pela fase de posicionamento. Um critério muito usado que visa garantir que nenhum recurso de roteamento seja compartilhado por mais de uma rede [33].

Uma vez que o roteamento é definido, as informações de mapeamento, empacotamento e posicionamento do *bitstream* são geradas para a *netlist* (lista de redes ou fios que conectam a saída de uma porta à entrada de outra porta). Durante a programação do *bitstream* no FPGA, as informações contidas no *netlist* são utilizadas para programar corretamente os bits entre os blocos lógicos do FPGA.

O resultado final é um *bitstream* que configura o estado dos *bits* de memória em um FPGA. O estado desses bits determina a função lógica que o FPGA implementa [60]. Um *bitstream* de FPGA é um arquivo contendo dados binários que codifica todas as informações de configuração de um determinado FPGA. Também possui os comandos necessários para controlar a funcionalidade do chip. Todas essas informações são organizadas em quadros que formam os blocos fundamentais para o espaço de memória de configuração do FPGA.

3.3 Exemplos de Arquiteturas que usam FPGA

Na presente seção serão apresentados dois exemplos de arquiteturas que usam FPGA: arquitetura sistólica e arquitetura Splash 2. Esses dois exemplos de arquiteturas foram escolhidos pois o primeiro é amplamente utilizado para implementação de soluções em FPGA que comparam sequências biológicas e o segundo é amplamente utilizado em pesquisa de texto e também em análise de sequências.

3.3.1 Arquitetura Sistólica

A Arquitetura Sistólica foi originalmente proposta por Kung e Leiserson como uma solução prática para a implementação de operações de multiplicação de matrizes em *Very Large-Scale of Integration* (VLSI) [43].

Posteriormente, verificou-se que os algoritmos de programação dinâmica para comparação de sequências, dado o seu alto grau de paralelismo, também poderiam ser mapeados em arquiteturas sistólicas [42].

O conceito de Arquitetura Sistólica é baseado em um fluxo contínuo de dados (*stream*) sincronizado por um sinal de *clock*, vindo de um meio de armazenamento, como por exemplo uma memória, em direção a uma coleção de Elementos de Processamento (PEs). Os PEs realizam o cálculo de maneira síncrona sobre dados diferentes, e passam o resultado para o PE que está imediatamente à sua esquerda (ou direita, em alguns projetos), como ilustrado na Figura 3.6. Por fim, os resultados gerados retornam ao meio de armazenamento.

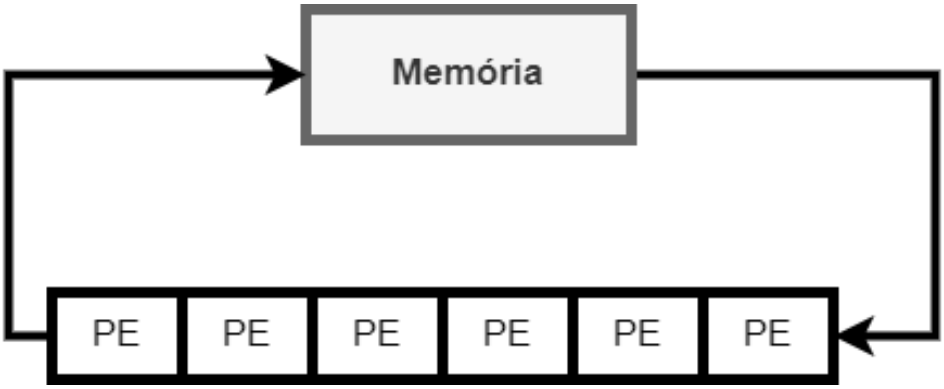

Figura 3.6: Fluxo básico de uma Arquitetura Sistólica [42].

A semelhança com o processo sístole/diástole ventricular que ocorre com o sangue que flui do coração deu nome a esse tipo de arquitetura. Como os PEs são organizados como um vetor (Figura 3.6) ou como uma matriz com duas ou mais dimensões, muitas vezes os termos vetor sistólico ou array sistólico são utilizados.

Nem todas as aplicações podem ser executadas com sucesso em arquiteturas sistólicas. Para que isso aconteça, a dependência de dados deve seguir um padrão regular. Se um determinado sistema possui uma forte dependência de dados e, portanto, impõe operações sequenciais, será difícil mapeá-lo para a Arquitetura Sistólica. Se, por outro lado, houver um alto grau de independência entre as operações, então o sistema pode ser candidato à execução na arquitetura sistólica [43, 42].

Em geral, aplicações que utilizam Arquiteturas Sistólicas têm, como principal característica, altas demandas computacionais. As aplicações mais comuns são: programação dinâmica [17], processamento de imagens [65], operações com matrizes (somas, multiplicações, resolução de sistemas lineares, entre outras) [49], processamento de sinais (FFT (Fast Fourier Transform)) [56, 50], convoluções [42], redes neurais [18], álgebra matricial [44], entre outras.

3.3.2 Splash 2

Splash 2 é uma arquitetura paralela na qual os elementos de computação são FPGA programáveis pelo usuário. O *Splash 2* foi projetado para acelerar aplicações que exibem paralelismo temporal e as aplicações são desenvolvidas em VHDL, que são então refinadas iterativamente e depuradas dentro de um simulador [14].

Uma vez que uma aplicação é determinada como funcionalmente correta na simulação, ela é compilada e otimizada por síntese lógica. A aplicação é então mapeada na arquitetura FPGA por ferramentas de posicionamento e roteamento - para formar um módulo de objeto de programação de FPGA [14].

A Figura 3.7 mostra a arquitetura do Splash 2.

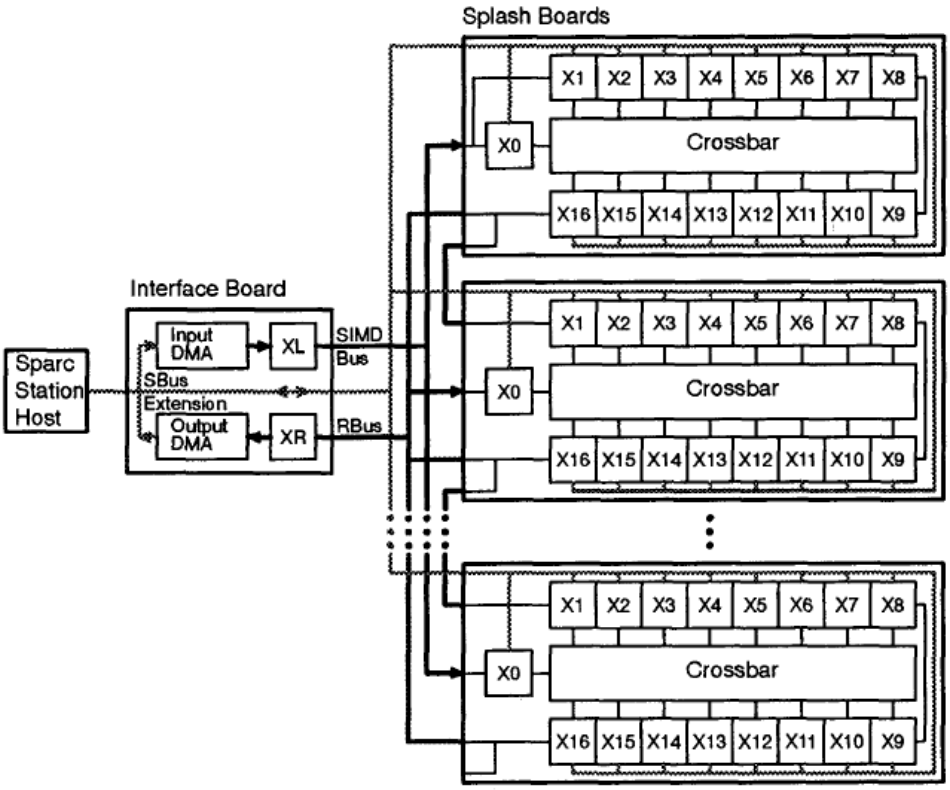

Figura 3.7: Arquitetura Splash 2 [14].

O sistema *Splash* 2 consiste em um *host Sun Sparcstation*, uma placa de interface e de uma a dezesseis placas *Splash array*. A placa de interface contém um relógio de sistema programável e fornece acesso DMA à memória do *host* por meio de dois dispositivos FPGA programáveis pelo usuário, XL e XR, que são usados para processar fluxos de dados de entrada/saída e cada placa de matriz contém 16 elementos de processamento (PES) conforme ilustrado na Figura 3.7 [14, 15].

O *host Sparcstation* executa uma variedade de funções de controle para o *Splash* 2. É responsável por baixar as informações de configuração para os elementos de computação do FPGA, os elementos de interface e o *switch crossbar*. Há uma variedade de mecanismos síncronos e assíncronos com os quais o *host* pode se comunicar com o sistema *Splash*, incluindo acesso direto às memórias do elemento de computação e a capacidade de iniciar e parar o relógio do sistema [14, 15].

O *Splash* 2 foi projetado para suportar uma variedade de modelos de programação, incluindo um modelo de fluxo de dados múltiplos de instrução única (SIMD), um modelo sistólico em *pipeline* unidimensional e vários modelos sistólicos com duas ou mais dimensões (Seção 3.3.1). O sistema *Splash 2* tem aplicações em pesquisa de texto [63], comparação de sequências [39] e processamento de imagem [11].

3.4 Programação de Hardware Reconfigurável

Nessa seção, apresentaremos dois ambientes de programação para FPGAs: OpenCL e HLS.

3.4.1 *Open Computing Language* (OpenCL)

O *Open Computing Language* (OpenCL) é um padrão aberto para programação paralela em diferentes tipos de plataformas e seu objetivo é fornecer uma maneira unificada de desenvolvimento de programas em plataformas de processamento heterogêneas.

O Grupo Khronos lançou a especificação de OpenCL 1.0 em 2009 [3]. Participaram de ssa primeira especificação pesquisadores da AMD, Intel, NVidia, Apple e ARM, entre outros. O OpenCL foi proposto como um framework para programação paralela em ambientes heterogêneos e é composto por uma linguagem, diversas bibliotecas e um sistema runtime [52].

Em junho de 2010, a versão foi atualizada para 1.1 [4] adicionando funcionalidades ao framework, trazendo melhorias em desempenho, flexibilidade e funcionalidades em programação paralela avançadas.

Ao final de 2011 foi lançada a versão 1.2 [5] do OpenCL que trouxe mudanças significativas, dentre elas a capacidade de particionar um dispositivo em sub-dispositivos para que

as atribuições de trabalho possam ser alocadas para mais de uma unidade de computação. Adicionalmente, foi proposta a compilação separada de ligação, característica útil para compilar o OpenCL com bibliotecas externas. Finalmente, os dispositivos personalizados que contêm funcionalidades específicas, tais como processadores de sinal digital (DSP), foram integrados no framework OpenCL.

Em 2013, a ferramenta chegou à versão 2.0 [6], trazendo memória virtual compartilhada, paralelismo aninhado, espaço de endereçamento genérico e extensão para dispositivos móveis com Android. Em 2015, foi implementada a linguagem OpenCL C++ na versão 2.1 [7], baseada em um conjunto de instruções de C++14. Nessa atualização são destacados: melhorias em funcionalidade de subgrupo; cópia de objetos e estados do kernel para o host.

Finalmente em 2017 foi lançada a versão 2.2 [8], contendo *pipe storage*, um novo tipo de dispositivo no OpenCL 2.2 que é útil para implementações em FPGA, tornando o tamanho e o tipo de conectividade conhecidos no tempo de compilação, permitindo uma comunicação eficiente dos *kernels* no dispositivo.

Arquitetura

A arquitetura do OpenCL compreende 4 modelos, que abordam diferentes aspectos da programação paralela. O modelo mais próximo do hardware é o modelo de plataforma, que é composto por um *host* e múltiplos dispositivos OpenCL. O modelo de execução define que um programa *host* executa-se no *host* e *kernels* se executam nos dispositivos OpenCL. O modelo de memória define os tipos de memória suportados por cada componente da arquitetura (*host* ou *kernels*), bem como o tipo de acesso permitido (leitura, escrita, ambos). Finalmente, o modelo de programação suporta dois paradigmas de programação paralela: paralelismo de dados e paralelismo de tarefas.

Modelo de Plataforma

A Figura 3.8 ilustra componentes do modelo de plataforma OpenCL. Um programa OpenCL é executado no *host* e a maior parte da estrutura de controle do programa reside no *host*. Pode haver um ou mais dispositivos de computação conectados ao *host*. Cada dispositivo OpenCL possui uma ou mais Elementos de Computação (CE) e cada CE possui um ou mais Elementos de Processamento (PE). A computação é feita nos PEs. Por exemplo, em uma máquina composta por CPU e FPGA, a CPU é o *host* e o FPGA é o dispositivo OpenCL [6].

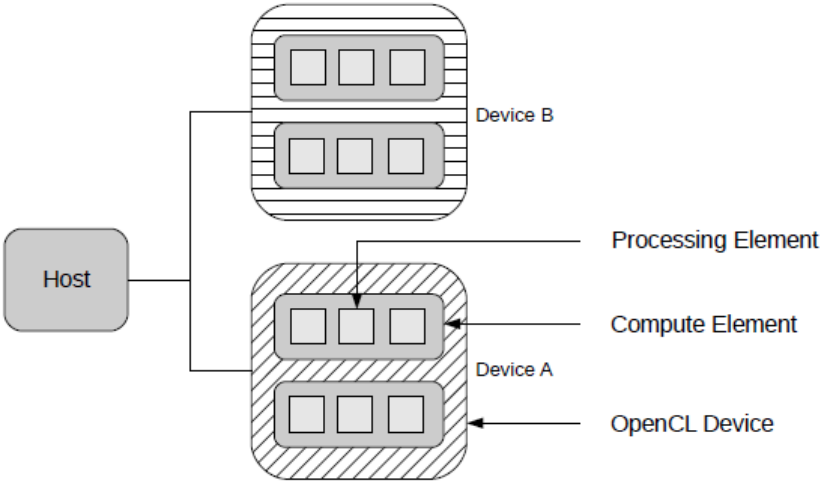

Figura 3.8: Modelo de Plataforma OpenCL [6].

Compilação

No OpenCL, um *kernel* pode ser compilado *online* ou *offline* (Figura 3.9).

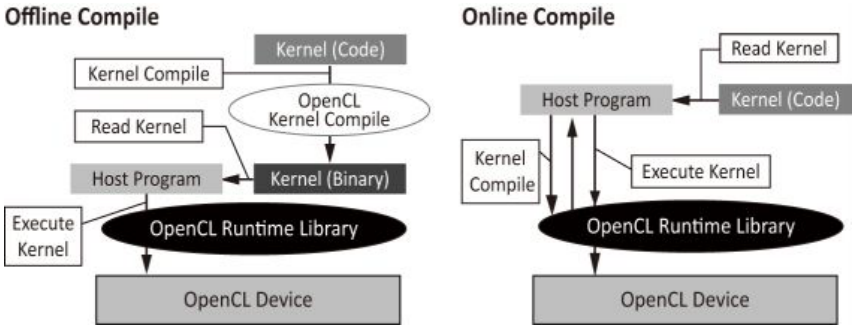

Figura 3.9: Visão geral dos tipos de compilação do OpenCL [6, 52].

A diferença básica entre os dois métodos é que, durante a compilação *online*, o código host deve ler o código-fonte do kernel OpenCL como uma string e durante a compilação *offline* como um binário.

Xilinx Vivado SDAccel

A Xilinx é um membro ativo do Khronos Group, colaborando com a especificação OpenCL, e suporta a compilação de programas C/C++/OpenCL para FPGAs Xilinx. O ambiente de desenvolvimento Xilinx SDAccel é usado para compilar programas C/C++/OpenCL para execução em um FPGA Xilinx [9].

Esse ambiente permite a programação simultânea do processador no sistema e do FPGA sem a necessidade de uma experiência extensiva de projeto de FPGA [9].

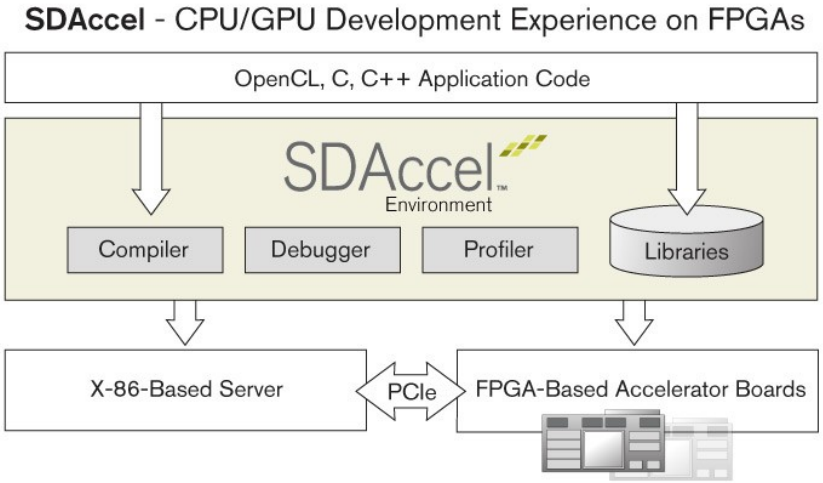

Figura 3.10: Visão Geral do SDAccel [9].

A Figura 3.10 mostra o fluxo completo de operações realizadas pela ferramenta SDAccel da Xilinx para executar o código C/C++/OpenCL e implementá-lo em um FPGA. O primeiro passo no fluxo é obter o código C/C++/OpenCL e verificar sua funcionalidade, executando uma simulação baseada em software (SW) em CPU. O *host* e o código do *kernel* na emulação da CPU são executados no processador baseado em x86. Uma vez que a funcionalidade é verificada, o desempenho de cada *kernel* individual e seu uso de recursos são estimados. Isso dá uma idéia inicial dos ganhos finais de desempenho, levando em conta o dispositivo de *hardware* de destino, bem como as unidades de computação geradas para a execução do programa [9, 53].

Em seguida, a emulação de hardware (HW) é usada para verificar se as unidades de computação criadas para todos os *kernels* estão funcionando corretamente e analisar seu desempenho geral. O ambiente de projeto integrado (IDE) do Vivado é usado posteriormente na etapa do sistema de compilação para conectar as unidades personalizadas geradas aos IPs (*Intelectual Property)* de infraestrutura fornecidos pelo hardware de destino, como a interface do processador usada para transmitir argumentos ao *kernel*, iniciá-lo e aguardar a sua conclusão, bem como a interface DDR DRAM. Por fim, a última etapa é juntar todas as partes geradas para implantação nas placas FPGA suportadas [9, 53].

3.4.2 *High Level Synthesis* (HLS)

Uma vez que a história do HLS é consideravelmente mais longa do que a dos FPGAs, a maioria das primeiras ferramentas HLS visava a projetos ASIC. Na década de 1980 e no

início da década de 1990, uma série de ferramentas de síntese de alto nível semelhantes foram construídas, principalmente para pesquisa. Esses primeiros projetos de pesquisa ajudaram a criar uma base para a síntese algorítmica com muitas inovações, e alguns foram usados até mesmo para produzir chips reais.

No entanto, esses esforços não levaram a uma ampla adoção entre os projetistas. Um dos principais motivos é que a metodologia de uso da síntese de RTL ainda não era amplamente aceita naquela época e as ferramentas de síntese de RTL ainda não estavam bem desenvolvidas. Portanto, a síntese de alto nível, construída sobre a síntese RTL, não tinha uma base sólida na prática. Além disso, muitas vezes eram feitas suposições simplistas nesses primeiros sistemas. Por exemplo, assumia-se que muitos deles eram "independentes da tecnologia" e inevitavelmente levavam a resultados abaixo do ideal.

Com as melhorias nas ferramentas de síntese RTL e a ampla adoção de fluxos de projeto baseados em RTL na década de 1990, a implantação industrial de ferramentas de síntese de alto nível se tornou mais prática. Um dos motivos é devido ao uso de HDLs comportamentais como linguagem de entrada, o que não é popular entre projetistas de algoritmos e sistemas.

Arquitetura

Um sistema típico começa com um modelo de software do sistema. A maior vantagem dos dispositivos programáveis como FPGAs é a capacidade de criar hardware personalizado que é otimizado para qualquer aplicação específica. O compilador Vivado High-Level Synthesis (HLS) fornece um ambiente de programação semelhante aos disponíveis para compiladores para CPU. A principal diferença é que o Vivado HLS compila o código C em uma microarquitetura RTL otimizada, enquanto os compiladores baseados em CPU geram código de montagem a ser executado em uma arquitetura de processador genérico. Arquitetos de sistema, programadores de software ou engenheiros de hardware podem usar Vivado HLS para criar hardware personalizado otimizado para energia e latência. Isso pode permitir a implementação otimizada de sistemas de alto desempenho, baixo consumo de energia ou baixo custo, para diversas aplicações.

Fluxo de Execução

A Figura 3.12 apresenta o fluxo de projeto do Vivado HLS, que se divide em 5 passos: a) Compilação, simulação e depuração do programa C; (b) síntese do programa C em RTL, opcionalmente utilizando diretivas de compilação fornecidas pelo usuário;

Tal como acontece com a compilação tradicional, o HLS primeiro usa um *front-end* para desacoplar os aspectos específicos da linguagem de programação de entrada (por

exemplo, sua sintaxe) de uma representação intermediária (IR). Em seguida, o meio-termo do compilador aplica uma variedade de transformações neutras de arquitetura (por exemplo, dobramento constante, eliminação de subexpressão, eliminação de acessos de memória redundantes) e transformações orientadas à arquitetura (por exemplo, transformações de loop, estreitamento de largura de bits) para expor tipos de dados especializados e operações e oportunidades de paralelismo de granulação fina, bem como de granulação grossa [10].

Enquanto algumas transformações exploram aspectos arquitetônicos de alto nível, como a existência de várias memórias para aumentar a disponibilidade de dados, outras transformações exploram a capacidade de personalizar unidades funcionais específicas em uma RPU (unidade de processamento reconfigurável), implementando diretamente instruções de alto nível em o nível de hardware. O último inclui a personalização de unidades aritméticas para suportar diretamente operações de ponto fixo usando formatos de largura de bits não padrão que são representações ou modos de arredondamento. Finalmente, o fluxo inclui um back-end [10].

Algumas dessas etapas podem ser realizadas integrando ferramentas de síntese comerciais no fluxo, em muitos casos usando módulos de biblioteca de componentes específicos de cada hardware.

A Figura 3.11 apresenta o processo genérico de síntese com HLS.

Ferramentas de Síntese HLS

Usualmente, a ferramenta Vivado HLS é utilizada e os passos até a geração de código com essa ferramenta são descritos a seguir.

A ferramenta Xilinx Vivado HLS sintetiza uma rotina C em um bloco IP (*Intellectual Property*) que pode ser integrado em um sistema de hardware. A Figura 3.12 apresenta o fluxo de projeto do Vivado HLS, que se divide em diversos passos: No primeiro passo, o programa C (C++ ou SystemC)fig-opencl1 é compilado e a ferramenta Vivado oferece um *test bench* para que o usuário simule e depure a sua rotina. Caso a rotina seja implementada de maneira errônea, obviamente a síntese não produzirá o resultado desejado. Por essa razão, esse primeiro passo de simulação e depuração deve ser executado. No próximo passo, Síntese da rotina C para RTL, a rotina C e as diretivas fornecidas pelo usuário são utilizadas para a geração de código VHDL ou Verilog. Ao terminar a síntese, a ferramenta Vivado HLS gera relatórios que podem auxiliar o usuário a analisar o desempenho da implementação. Nesse ponto, o usuário pode tentar diversas diretivas HLS, verificando como a implementação se comporta ao utilizá-las. A Simulação RTL é executada no passo 3, onde é feita verificação funcional do RTL gerado, via simulação, utilizando o simulador Vivado. O passo 4 é chamado Exportação do RTL e Empacotamento em IPs (Intellectual

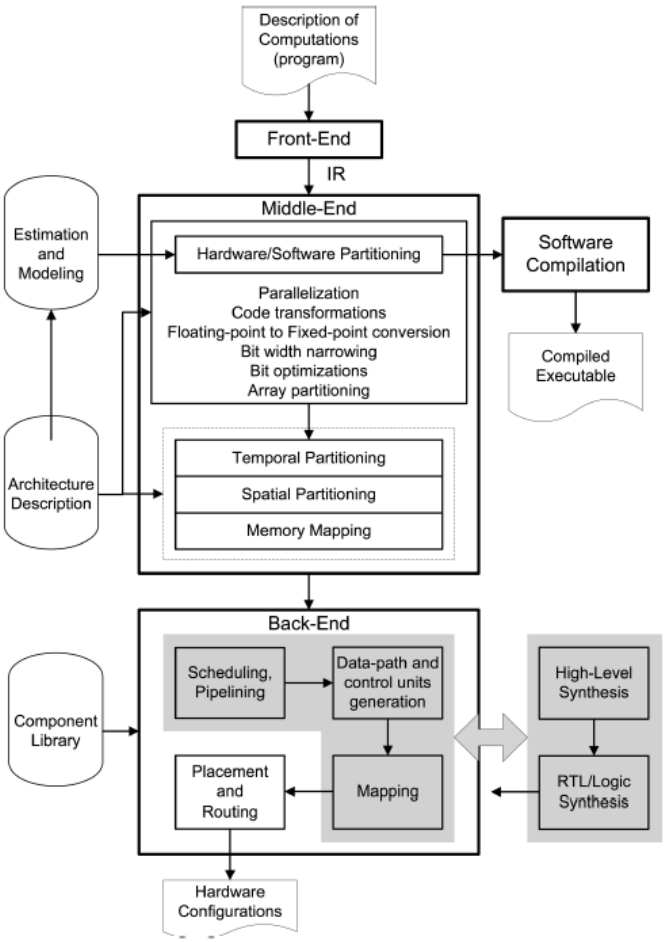

Figura 3.11: Fluxo de compilação genérico para plataformas de computação reconfiguráveis [10].

Properties). Nesse passo, o RTL gerado no passo anterior é exportado e empacotado. Como saída, obtem-se um ou mais IPs, que podem ser executados em FPGA.

Na presente Tese, foi utilizada a ferramenta Vivado HLS. A seguir, apresentaremos sucintamente outras ferramentas de síntese HLS.

GAUT [28] é uma ferramenta HLS de código aberto projetada especificamente para aplicações de processamento digital de sinais (DSP). O potencial paralelismo dos aplicativos é extraído antes das tarefas de agendamento, alocação e ligação. Em seguida, uma arquitetura potencialmente em pipeline é gerada, consistindo de módulos de processamento e memória, juntamente com uma unidade de comunicação com uma interface GALS/LIS. A taxa de transferência e o período de *clock* são obrigatoriamente restringidos durante o processo de síntese, e o diagrama de tempo de E/S e o mapeamento de memória são opcionais.

BAMBU [62] é uma ferramenta HLS de código aberto com estrutura modular e oferece

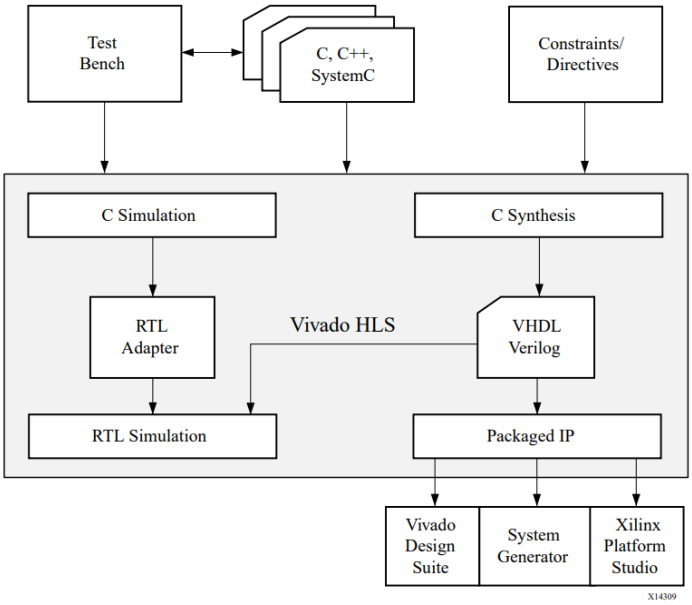

Figura 3.12: Fluxo de Execução do HLS Vivado [10].

suporte às tecnologias ASIC e FPGA. Ele explora arquiteturas de armazenamento para implementar a maioria das construções na linguagem C. As operações de ponto flutuante também são integradas. O fluxo de síntese, como restrições, passagens de transformação, scripts de síntese e opções, é customizado por meio de arquivos XML. Os bancos de teste são gerados automaticamente. Finalmente, o BAMBU propõe uma validação dos resultados contra a descrição de alto nível correspondente.

Catapult-C [1] é uma ferramenta comercial de HLS que é flexível na escolha da tecnologia e bibliotecas, definindo a frequência do ciclo e mapeando os parâmetros da função para interfaces de *streaming*, registradores, RAM ou ROM. Atualmente, o Catapult-C se concentra principalmente em soluções FPGA de baixa potência.

Capítulo 4

Trabalhos Relacionados

Devido à complexidade quadrática do tempo de execução, muitas soluções paralelas foram propostas na literatura para acelerar comparações de sequências biológicas. Neste capítulo, discutimos propostas que usam FPGAs e implementam o algoritmo LCS (Seção 2.3) e algoritmos de comparação de sequência associados, como Smith-Waterman (Seção 2.5), Needleman-Wunsch (Seção 2.4) e Hirschberg (Seção 2.7).

4.1 Junid et al. (2010) [12]

Junid et al. [12] apresentam o projeto de uma técnica de aceleração de computação de alto desempenho e otimização de uso de memória para alinhar sequências de DNA com o algoritmo Smith-Waterman (SW) (Seção 2.5).

O artigo se concentra na otimização de uso de memória e velocidade de processamento, convertendo as sequências em valores de 2 bits e compactando esses valores antes do alinhamento. O objetivo desta técnica é usar a compressão de dados para reduzir a quantidade de dados transmitidos do *desktop* para o FPGA. A técnica proposta foi projetada e implementada em um FPGA Altera Cyclone II 2C70, com clock ajustado para 50 MHz, e o código foi escrito em Verilog HDL.

Os autores fizeram uma análise teórica da redução do espaço de memória, que pode ser até 4x menor do que o método tradicional baseado em caracteres. Entretanto, essa análise e implementação se estendeu somente à simulações e resultados teóricos.

4.2 Weinbrandt (2014) [74]

Weinbrandt [74] apresenta nesse trabalho uma implementação do algoritmo Smith-Waterman (Seção 2.5) no sistema RIVYERA, que originalmente foi desenvolvido para criptoanálise e foi mais tarde introduzido para problemas relacionados à bioinformática.

Neste trabalho, o autor se concentra no modelo mais recente à época (RIVYERA S6-LX150) equipado com 128 FPGAs do tipo Xilinx Spartan6-LX150. A estrutura básica da arquitetura RIVYERA consiste em dois elementos, a máquina FPGA e um servidor com componentes de PC padrão. A máquina FPGA consiste em até 16 módulos FPGA com 8 FPGAs cada. Cada FPGA é conectado a uma memória local de 256 MB DDR3-RAM. As atualizações podem permitir até 16 FPGAs em cada módulo ou mais quantidade de memória. O sistema implementado no computador RIVYERA é organizado como uma cadeia sistólica, ou seja, cada FPGA em um módulo FPGA é conectado diretamente aos seus vizinhos da esquerda e da direita formando um anel. Um controlador de comunicação neste anel fornece a interconexão de cada módulo aos seus módulos vizinhos. O *uplink* para o *host* é realizado por meio do controlador de comunicação do primeiro módulo FPGA.

A implementação do algoritmo SW (Seção 2.5) no FPGA é realizada pelo seguinte esquema de paralelização. Um elemento de processamento chamado *SWcell* é implementado no FPGA e contém um caractere de uma das sequências (sequência de consulta). Ele calcula os valores na linha da matriz de alinhamento correspondentes ao seu caractere, em que i, o índice da linha, é fixo para cada célula. De acordo com a equação de recorrência (Seção 2.5), cada célula requer acesso a três valores vizinhos da matriz para calcular seu valor. Portanto, todos os elementos de processamento são conectados em uma cadeia sistólica de forma que cada elemento tenha acesso ao valor da célula de seu predecessor. Assim, a outra sequência (banco de dados) é transmitida caractere por caractere a cada ciclo de *clock* através da cadeia de elementos de processamento. O processamento é realizado nas anti-diagonais da matriz de programação dinâmica. A Figura 4.1 mostra uma etapa de cálculo da matriz e uma parte da estrutura da cadeia.

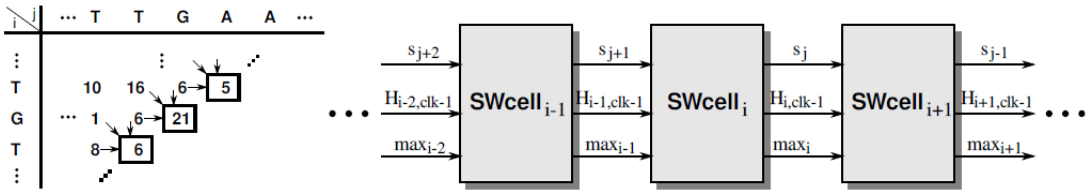

Figura 4.1: Estrutura da cadeia sistólica para o cálculo da matriz SW em um FPGA no sistema RIVYERA [74].

Weinbrandt compara sua solução com duas soluções, dentre elas o BLASTp([48]). O sistema RIVYERA S6-LX150 com 128 FPGAs Xilinx Spartan6-LX150 supera as arquiteturas de computador padrão tanto em termos de tempo de execução como em consumo de energia. A aceleração para alinhamentos Smith-Waterman é mais de 134x em comparação com o BLASTp em uma CPU dual quadcore.

Além disso, o BLASTp é superado em um sistema quad-core por um fator de 19x. Devido à pouca utilização de energia elétrica em FPGAs, a economia de energia em relação à CPU está na mesma faixa do aumento de velocidade, ou seja, cerca de 70% para BLASTp.

Tradicionalmente, a métrica GCUPS (Billions of Cells from the matrix Updated Per Second) é utilizada para comparar o desempenho de aplicações de comparação de sequências que usam programação dinâmica. Com 128 FPGAs, o SW-Ryviera atingiu 6020 GCUPS.

4.3 Buhagiar et al. (2017) [24]

Nesse artigo, Buhagiar et al. [24] apresentam a implementação de hardware do algoritmo Smith-Waterman (Seção 2.5) com codificação diferencial em um FPGA. Um novo algoritmo pra *traceback* é proposto, visando dispensar a necessidade do armazenamento da matriz de programação dinâmica, diminuindo o uso de recursos de hardware.

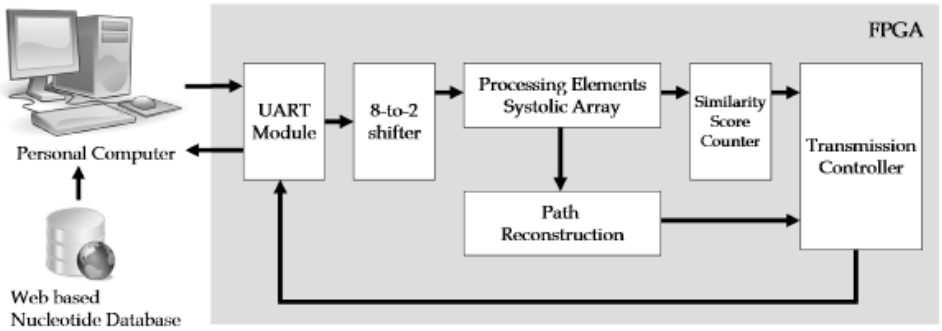

Figura 4.2: Um diagrama de blocos do sistema implementado [24].

A Figura 4.2 mostra o diagrama de blocos simplificado da matriz sistólica SWA com *traceback* implementada em um FPGA com *clock* de 200 MHz. O FPGA é conectado a um computador pessoal (PC) através do protocolo RS-232 implementado por meio de um módulo *Universal Asynchronous Receiver Transmitter* (UART) que permite a comunicação bidirecional. O PC tem acesso à Internet, de forma que as sequências de nucleotídeos podem ser acessadas e baixadas de bancos de dados de código aberto baseados na web, como NCBI (*National Center for Biotechnology Information*) ou EMBL (*European Molecular Biology Laboratory*).

Um programa MATLAB com uma interface gráfica de usuário (GUI) foi implementado no PC para que as sequências de nucleotídeos possam ser recuperadas e transmitidas ao FPGA. Este programa também é usado para exibir os resultados gerados pelo FPGA.

Logo que as sequências são recuperadas do banco de dados online, a sequência de busca é truncada para o comprimento da matriz sistólica (1024 PEs), convertida em um fluxo binário e então dividida em bytes, onde cada byte é composto por quatro nucleotídeos.

A implementação foi feita no Xilinx Spartan-6 XC6LX16-CS324 FPGA usando VHDL, consistindo de 1024 PEs e exibindo uma taxa de 204,8 GCUPS a uma taxa de dados de 600 Mbps, com *clock* de 200MHz.

4.4 Fei et al. (2017) [34]

Neste artigo [34], os autores propõem a paralelização do algoritmo Smith-Waterman (SW) com *affine gap* (Seção 2.6) e implementam estruturas de computação paralelas para acelerar o SW, com *traceback* na plataforma FPGA. Os recursos de computação paralela dinâmica de elementos das anti-diagonais e o problema de expansão de armazenamento resultante do estágio de *traceback* são analisados e são propostas estratégias para eliminar a dependência de dados, reduzir os requisitos de armazenamento e reduzir o tempo de acesso à memória.

Na Figura 4.3, os elementos de processamento (PE) são organizados em um array sistólico que faz o cálculo do escore e gera informações de *traceback* que são repassadas ao módulo intermediário.

A implementação proposta em FPGA obtém um aumento de velocidade de 3,6x–25,2x em comparação com a CPU e 1x–23,5x em comparação com a GPU. O consumo de energia gerado pela carga de trabalho SW das CPUs varia de 22 a 25 W e o das GPUs varia de 170 a 182 W. Em contraste, a plataforma FPGA produz 44W de consumo de energia, economizando mais de 74% em comparação com a GPU.

Essa implementação é capaz de suportar aplicações de alinhamento de sequências biológicas longas (1M ou mais caracteres) e múltiplos tipos (DNA e RNA).

4.5 Cinti et al. (2018) [26]

Cinti et al. (2018) [26] apresentam um algoritmo para correspondência de cadeia aproximada online (OASM) capaz de filtrar *shadow hits* em tempo real, de acordo com regras de prioridade de finalidade geral que atribuem prioridades a ocorrências sobrepostas. Uma implementação em FPGA do OASM é proposta e comparada com uma versão de software serial.

A arquitetura proposta está ilustrada na Figura 4.4. O módulo LEV CORE calcula as múltiplas distâncias Levenshtein 2.3 entre o padrão p e todas as substrings s extraídas de um fluxo de símbolos t e retorna apenas o resultado da lista de ocorrências abaixo de

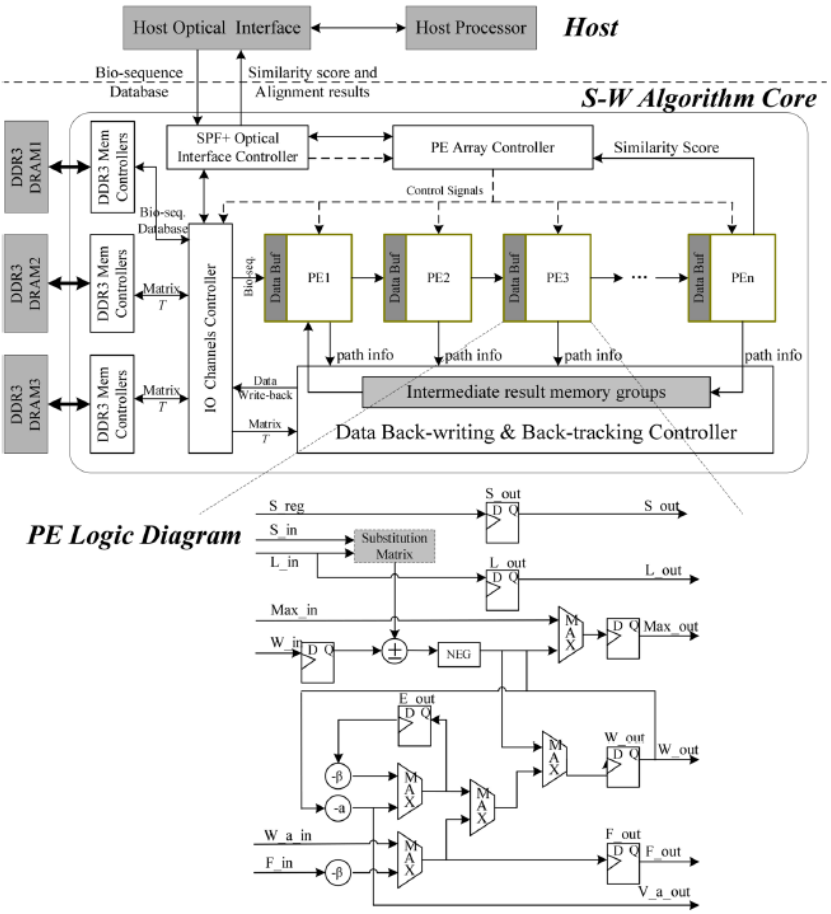

Figura 4.3: Projeto do algoritmo paralelo SW e a estrutura do núcleo PE com modelo de penalidade afim [34].

um limite K. Para o projeto, o comprimento máximo do padrão é $lp = 15$, o tamanho do alfabeto = 3 bits (estendido devido a símbolos especiais) e o limite máximo de respostas é 5 ($K = 5$). O LINK EMULATOR possui memória ROM de 1Mbit, contendo o texto t adicionado ao controle lógico. Um novo símbolo é enviado para o LEV CORE a cada nova requisição. O módulo RAM armazena os resultados, o módulo USB transfere os resultados para um dispositivo externo e por último, o módulo SYS FSM que é uma máquina de estados finita que controla e coordena as operações de entrada e saída. Sinais de controle são definidos como *start_elab* (inicio da elaboração) e *result_return* (comando para iniciar o retorno dos resultados armazenados na RAM através do módulo USB).

A implementação proposta alcançou alto grau de paralelismo e desempenho superior ao da implementação de software, ao mesmo tempo em que manteve baixo o uso de elementos lógicos. Foram utilizadas uma CPU Intel® i7 4700MQ e uma FPGA Altera Cyclone® IV E com programação em C++ para a CPU e VHDL para a FPGA com

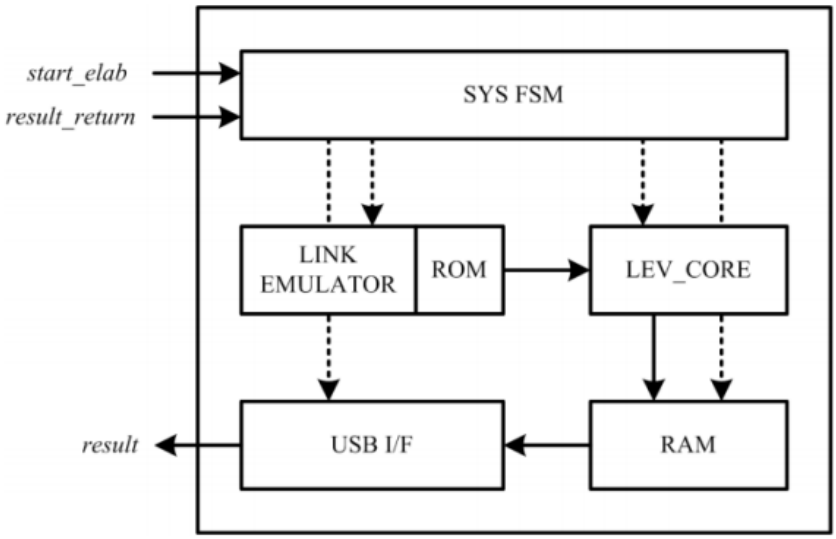

Figura 4.4: Sistema implementado do OASM para FPGA [26].

sequências sintéticas de busca com tamanho de 3104 caracteres.

A arquitetura OASM proposta pelos autores permite realizar um procedimento de recuperação tão complexo contando com um hardware de baixo custo, como o Altera Cyclone IV E FPGA adotado, onde apenas 3% dos elementos lógicos disponíveis são usados. O módulo LEV CORE calcula as múltiplas distâncias de Levenshtein entre todas as substrings extraídas de um fluxo de símbolos e retorna apenas a lista de resultados de ocorrências abaixo de um limite estipulado pelos autores.

4.6 Alser et al. (2019) [13]

Alser et al. [13] apresentam um algoritmo chamado Shouji, que é um filtro heurístico relativamente preciso de alinhamento paralelo, usando uma abordagem de janela de busca deslizante para identificar rapidamente sequências diferentes, sem a necessidade de algoritmos de alinhamento.

O objetivo dos autores é conseguir rejeitar rapidamente sequências diferentes com alta precisão, de modo que seja reduzida a necessidade da etapa de alinhamento de grande custo computacional. O primeiro objetivo do filtro de pré-alinhamento proposto é fornecer alta precisão de filtragem, detectando todas as subsequências comuns compartilhadas entre duas sequências dadas. O segundo objetivo é projetar um acelerador de hardware que adote arquiteturas modernas de FPGA para aumentar ainda mais o desempenho do algoritmo Shouji.

O filtro Shouji se executa em 3 passos: (a) construção do mapa de vizinhança contendo *matches* e *mismatches* entre as duas sequências, dado um *threshold*; (b) obtenção de todas as subsequências comuns não intercaladas com o mapa de vizinhança; e (c) rejeição dos pares de sequências com número de *matches* pequeno. O acelerador de hardware projetado para o Shouji está na Figura 4.5. Como pode ser visto, a arquitetura comporta diversos filtros, que são executados em paralelo.

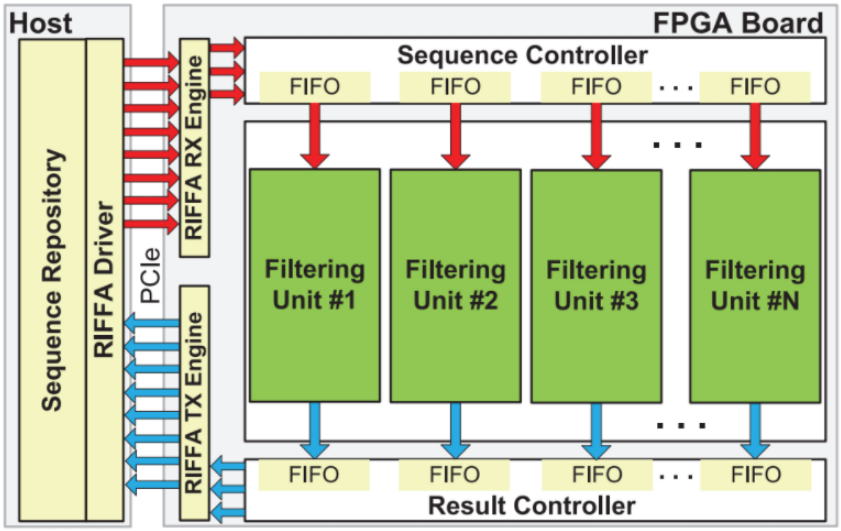

Figura 4.5: Visão geral da arquitetura de acelerador de hardware Shouji [13].

Para a coleta dos resultados, foram utilizados uma CPU Intel® i7-3820 e um FPGA Xilinx Virtex®-7 VC709 com programação em C para CPU e Verilog para a FPGA. O circuito resultante operava com *clock* de 250 MHz, com vazão de I/O de 3.3 GB/s. Foram comparadas 30 milhões de sequências biológicas de até 250 caracteres. A solução Shouji em FPGA com 16 unidades paralelas de filtragem foi cerca de 335x mais rápida do que uma implementação em CPU com operações vetoriais.

4.7 Bonny et al. (2019) [22]

Em Bonny et al. [22], os autores propuseram melhorar o tempo de computação dos algoritmos baseados em programação dinâmica, com uma nova técnica, que é chamada de Programação Dinâmica Segmentada (SDP). O SDP encontra a melhor maneira de dividir as sequências comparadas em segmentos e, em seguida, aplica o algoritmo de programação dinâmica a cada segmento individualmente.

O SDP pode ser aplicado a vários algoritmos baseados em programação dinâmica melhorando seu tempo de computação, porém com perda potencial na acurácia dos resultados. Como estudos de caso, os autores aplicaram a técnica SDP em dois algoritmos:

Needleman-Wunsch (NW)(Seção 2.4) e o algoritmo LCS (Seção 2.3). No primeiro estudo de caso, os autores compararam os resultados da utilização do único NW com os resultados obtidos após a aplicação do SDP no algoritmo NW, “NW + SDP”. Além disso, os autores comparam os resultados com o algoritmo heurístico de alinhamento de sequência FASTA, GGSEARCH [61].

Os autores mostraram que o “NW + SDP” tem melhor desempenho em relação ao NW, com uma leve degradação na qualidade do alinhamento. Ainda mostraram que essa degradação é controlável e é uma compensação com o tempo de computação. No segundo estudo de caso, compararam os resultados do uso de LCS com os resultados obtidos após a aplicação de SDP no algoritmo LCS, "LCS + SDP". A Figura 4.6 ilustra a aplicação do SDP.

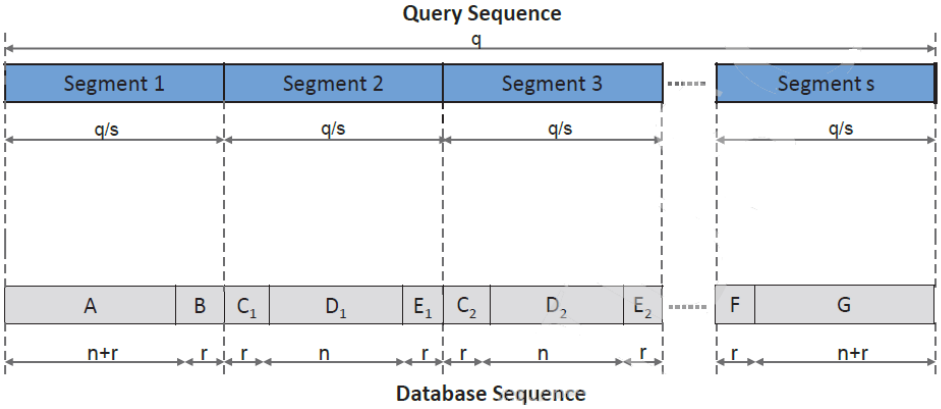

Figura 4.6: Dividindo as consulta e as sequências de banco de dados em S segmentos [22].

O SDP divide a sequência de consulta (que tem um comprimento de q) em um número específico de segmentos S. O número de segmentos é fornecido pelo usuário e é baseado na velocidade/precisão necessária. Cada segmento tem um comprimento igual a q/s. A sequência do banco de dados também é dividida no mesmo número de segmentos que a sequência de consulta e mesmo comprimento de segmento q/s.

Os resultados mostraram que a aplicação da técnica SDP em conjunto com os algoritmos baseados em programação dinâmica melhora o tempo de computação em até 80% com degradação pequena ou ignorável na comparação com os resultados ótimos.

4.8 Tucci et al. (2020) [71]

Neste artigo, os autores apresentam SALSA [71], uma arquitetura específica de domínio para alinhamento de sequências que é configurável e extensível. A *Instruction Set Architecture* (ISA) do SALSA é baseada em RISC-V e a arquitetura foi projetada em Chisel

HDL, sendo extensível, parametrizável e customizável, com a possibilidade de modificar a microarquitetura por ALUs e instruções específicas de projeto.

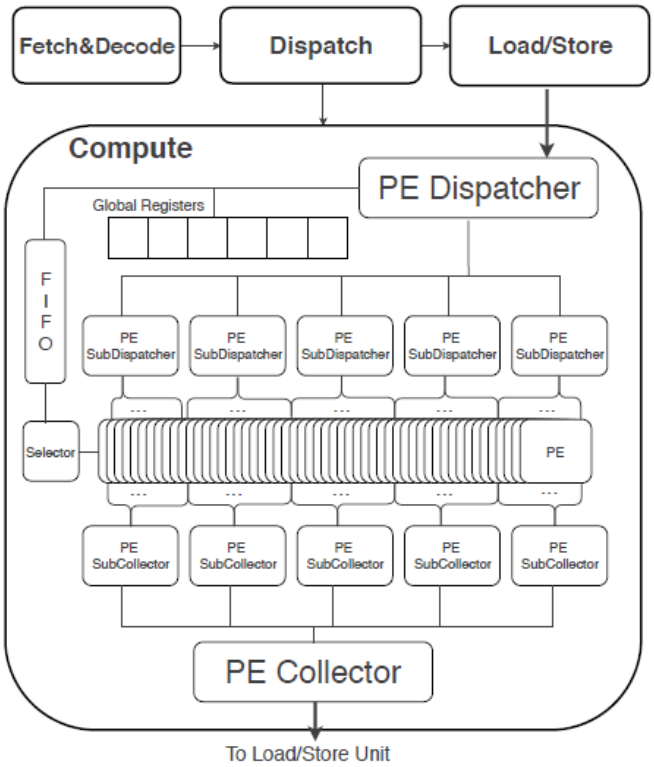

Figura 4.7: Arquitetura de nível superior da SALSA [71].

A arquitetura de nível superior da SALSA é mostrada na Figura 4.7. Ela é composta principalmente por um pipeline com 4 estágios: *Fetch* e *Decode*, *Dispatch*, *Load/Store* e *Compute* para execução da RISC-V. Cada estágio está associado a uma unidade de hardware. O estágio *Fetch* e *Decode* obtém as instruções do processador *host* e as entrega ao *Dispatcher*, que pode então decidir enviar a instrução para *Load/Store* ou diretamente para o estágio *Compute*, dependendo do tipo de instrução. A unidade *Load/Store* tem uma conexão direta com o controlador de memória para realizar acessos únicos ou múltiplos. A unidade *Load/Store* só trata uma solicitação de memória por vez.

A computação e o I/O são completamente separados, portanto, se não houver dependência entre as operações, o *Load/Store* e a *Compute* podem trabalhar em paralelo. Entre todos os estágios, existem FIFOs para armazenar os dados em *buffer*: sempre que um FIFO fica cheio, um mecanismo de paralisação é ativado até que o FIFO seja capaz de armazenar novos dados.

A unidade de computação é responsável pelo cálculo da matriz, com os algoritmos NW (Seção 2.4), SW (Seção 2.5) Gotoh (Seção 2.6). Os PEs são organizados como um

vetor sistólico linear, divididos em grupos (PE subdispatcher e PE subcollector).

O tamanho dos registradores, as interfaces entre os diferentes módulos, bem como a interface entre a unidade Load/Store e o controlador de memória, são ajustáveis alterando parâmetros específicos no código do Chisel. Desta forma, dependendo do controlador de memória, é possível ter transações de 512 ou 32/64 bits. O valor depende da interface da SALSA com uma memória cache, como em um cenário onde a SALSA é usada como um coprocessador compartilhando o mesmo chip, ou diretamente com o DDR onde a SALSA teria acesso direto à memória. As instruções SALSA podem levar mais do que um ciclo de clock por estágio, ao contrário dos pipelines RISC tradicionais que requerem 1 ciclo de clock por estágio.

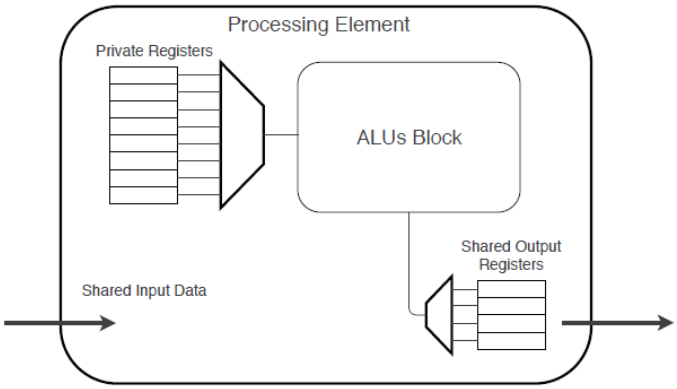

Figura 4.8: Detalhes do PE dentro do SA. [71].

O PE, observável na Figura 4.8, contém toda a lógica para realizar a operação matemática e mover os dados de saída para o PE vizinho. O PE pode ler os valores compartilhados do PE à sua esquerda e pode armazenar o valor que precisa ser passado para o PE à direita nos registradores de Saída Compartilhada.

A SALSA foi projetado usando Chisel HDL. No artigo, o SALSA foi integrado ao Rocket, que é um core RISCV de código aberto. O Rocket foi gravado em diferentes processos comerciais, atingindo frequência acima de 1,65 GHz no IBM 45nm SOI. As transações de memória possuem até 64 bits, sem *memory burst*. Apesar de ser um fator limitante, essa não é uma limitação da SALSA, mas da infraestrutura de teste. A SALSA foi gerada com 160 PEs e 16 registradores globais de 32 bits. O FIFO que insere o valor para o primeiro PE é capaz de hospedar 128 valores de 64 bits.

Cada PE possui 20 registradores privados de 32 bits, sendo os últimos 5 reservados para as saídas e 6 registradores de 32 bits servindo como saídas compartilhadas. A síntese e implementação da SALSA foram feitas com Firesim 1.5 sendo testados em uma única instância FPGA na nuvem AWS EC2 F1. A frequência alvo foi definida para 200

MHz. Comparações em aplicativos provenientes do domínio genômico, com o processador Rocket, executado em um Intel Xeon E3, mostraram como o SALSA integrado ao Rocket é até 790 vezes mais eficiente em termos de energia e até 60 vezes mais rápido.

4.9 Fujiki et al. (2020) [35]

Neste artigo, é proposto o SeedEx [35], que é uma arquitetura de hardware que combina métodos heurísticos de extensão de *seed* baseados em especulação e teste. Os autores perceberam que com carga de trabalho usando genomas humanos reais a maioria das comparações entre sequências requer bandas estreitas da matriz de programação dinâmica. Portanto, poucos PEs poderiam recuperar o melhor alinhamento para a maioria das entradas. No entanto, como o número mínimo de PEs necessário não é conhecido antes da execução, o acelerador precisaria implementar uma grande quantidade de PEs. Isso levaria a uma baixa utilização, resultando em uma redução significativa no desempenho das comparações. O SeedEx resolve esse problema permitindo que as comparações sejam realizadas especulativamente no hardware com aceleradores de banda estreita, com poucos PEs. Para tanto, o SeedEx precisa garantir a otimização, introduzindo um mecanismo de verificação rígido, que determina se o número de PEs foi suficiente.

A arquitetura do acelerador proposta consiste em vários blocos. Cada bloco do acelerador SeedEx consiste em uma hierarquia de máquinas de matriz sistólica Smith-Waterman em bandas. O primeiro nível da hierarquia consiste em máquinas de pontuação de *affine gap* de banda estreita, executando o algoritmo de Gotoh (Seção 2.6). O segundo nível consiste em máquinas de edição leves que calculam a distância de edição (ou Levenshtein), sem penalidade ponderada e que podem ser implementadas com baixa complexidade. A máquina de edição ajuda a melhorar a verificação de otimização para entrada com comprimentos de *string* assimétricos. O SeedEx testa a otimização comparando primeiro a pontuação de uma máquina de intervalo afim de banda estreita com as pontuações do limite superior. Se essas verificações falharem, as sequências são enviadas para uma máquina de edição, seguida por uma comparação de pontuações de banda estreita e pontuações de edição otimistas. O pequeno número de comparações que falham em todos os testes SeedEx são enviadas de volta do acelerador para a CPU do *host*. A Figura 4.9 apresenta essa sequência de testes.

A Figura 4.10 mostra a arquitetura de nível superior do acelerador SeedEx, que consiste em uma interface que recebe as duas sequências e vários SeedEx Cores. As consultas de entrada da interface de memória são armazenadas em *buffer* e analisadas, depois carregadas em um núcleo SeedEx. Cada núcleo SeedEx consiste em lógica de controle de análise de entrada, alguns núcleos Banded Smith-Waterman (BSW) e um núcleo *Edit*

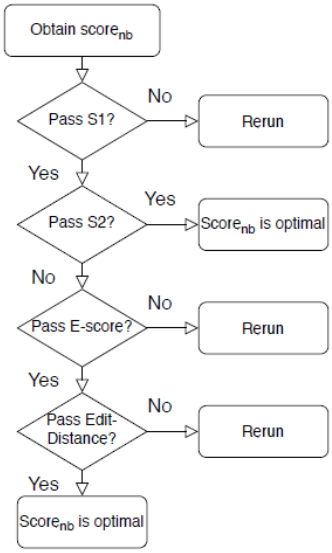

Figura 4.9: Fluxo de trabalho do SeedEx [35].

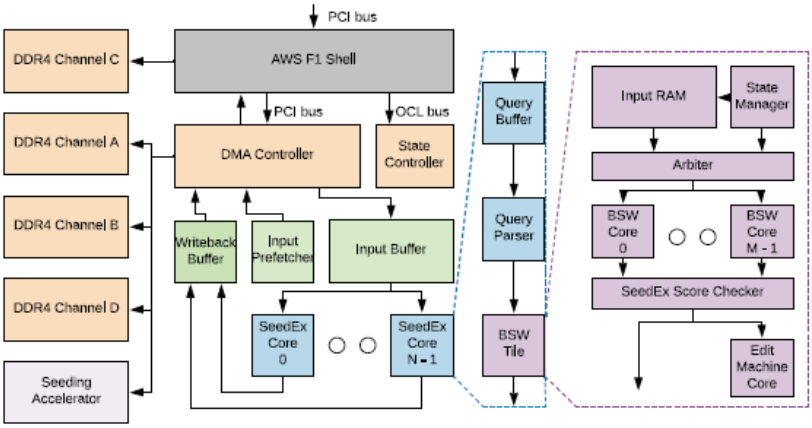

Figura 4.10: Arquitetura do SeedEx [35].

Machine. As sequências de entrada são fragmentadas e alimentadas sequencialmente da RAM de entrada para o BSW Core pelo *Arbiter* e *State Manager*, que são capazes de registrar vários fluxos de entrada. O núcleo BSW executa Smith-Waterman com faixas e gera o escore de pontuação de banda estreita e a pontuação de verificação. Se a pontuação de banda estreita passa na primeira verificação de limite, mas falha na segunda (ou seja, entre S1 e S2, na Figura 4.9), e passa no Verificação de pontuação E, a consulta é enviada ao núcleo da máquina de edição. O núcleo da máquina de edição gera a pontuação de edição otimista pontuada. Como o algoritmo SeedEx é um algoritmo baseado em especulação, ele assume que uma pequena banda é suficiente para problemas de entrada e valida

os resultados realizando verificações de validação de pontuação.

O SeedEX foi sintetizado usando Vivado com *clock* de 8ns na maior parte do circuito, mas com 4ns nos cores SeedEX, e foi executado na instância FPGA f1.2xlarge da nuvem AWS. Foram feitas mais de 700 milhões de comparações de sequências de DNA de 101 caracteres. Na comparação com a execução em CPU, a execução em FPGA conseguiu ser 2.3x mais rápida.

4.10 Quadro Comparativo

Tabela 4.1: Trabalhos relacionados

Artigo	Ano	Algoritmo	Tipo Algoritmo	Plataforma	Programação	Tamanho (caracteres)	Energia
Junid et al. [12]	2010	SW	Ótimo	1x FPGA	Verilog(FPGA)	1024	NA
Wienbrandt [74]	2014	SW	Ótimo	128x FPGA	VHDL(FPGA)	100	Watts 780
Buhagiar et al. [24]	2017	SW	Ótimo	1x FPGA	VHDL(FPGA)	1024	NA
Fei et al. [34]	2017	SW	Ótimo	1x FPGA	Verilog (FPGA)	8192	NA
Cinti et al. [26]	2018	OASM, SW-OASM, e HW-OASM	Heurístico	1x CPU + 1x FPGA	C++ (CPU) + VHDL (FPGA)	3104	NA
Alser et al. [13]	2019	Shouji	Ótimo	1x CPU + 1x FPGA	C(CPU) + Verilog(FPGA)	250	NA
Bonny et al. [22]	2019	LCS Segmentado	Heurístico	1x FPGA	VHDL(FPGA)	100000	NA
Tucci et al. [71]	2020	ALUs Específicas: SW e NW	Ótimo	1x FPGA Amazon F1	Chisel HDL(FPGA)	128	GCUPS/Watts 712.66
Fujiki et al. [35]	2020	Banded SW	Ótimo	1x FPGA Amazon F1	VHDL(FPGA)	101	NA

Na Tabela 4.1, apresentamos os principais aspectos dos trabalhos discutidos neste capítulo. Pode-se notar que a maioria dos trabalhos utilizam o algoritmo Smith-Waterman [12, 74, 24, 34, 71, 35]. Os trabalhos [12, 74, 24, 34] calculam SW com array sistólico e implementações específicas de domínio. O trabalho [71] usa a arquitetura RISC-V para fazer o *Fetch/Decode* e *Dispatch* de instruções, bem como os acessos à memória. O estágio *Compute* é feito por ALUs específicas, que executam tanto o SW como o NW. O trabalho [35] implementa a etapa de extensão de um método de comparação heurístico com a versão *banded* SW. Dentre os trabalhos analisados, somente [22] implementa o LCS, porém sua versão segmentada (heurística). Em [26], é usado o OASM, que calcula a distância de edição (Levenshtein) e em [13] o algoritmo heurístico Shouji é implementado.

Como plataforma de execução, a grande maioria dos trabalhos utiliza uma CPU e um FPGA, onde a CPU gerencia a execução no FPGA, que atua como um acelerador. No trabalho [74], uma máquina composta por 128 FPGAs é utilizada nas comparações.

Dentre os trabalhos analisados, apenas os trabalhos de Weinbrandt [74] e Tucci [71] levam em consideração o consumo de energia. Com exceção do trabalho de Bonny et

al. [22] com tamanho de sequências de até 100k, os demais trabalhos conseguem utilizar sequências de DNA de até 10k.

Ao nosso conhecimento, não existem trabalhos utilizando abstração de alto nível para utilização de FPGAs em comparações de sequências biológicas com o algoritmo LCS, que obtém o resultado ótimo. Como pode ser visto na Tabela 4.1, existe uma lacuna de pesquisas em FPGA utilizando ferramentas e linguagens de programação de alto nível que abordam e implementam algoritmos de comparação de sequências biológicas que retornam o resultado ótimo, com tamanhos acima de 10K em FPGA.

Parte II

Contribuições

Capítulo 5

Projeto do LCS com *High level Synthesis*

Neste capítulo, como primeira contribuição dessa tese, propõe-se uma implementação do algoritmo *Longest Common Subsequence* (LCS) (Seção 2.3) em HLS para comparação de duas sequências de DNA de tamanho médio (até 50000 caracteres), analisando os recursos utilizados, tempo de execução e consumo de energia do *design* proposto. O objetivo da nossa proposta é fornecer uma implementação em FPGA eficiente, onde o consumo de energia seja reduzido. Para tanto, utilizamos as *BlockRAMs* do FPGA para armazenar dados (ao invés da memória DDR presente no FPGA) e preenchemos a matriz diagonal por diagonal. Os resultados experimentais mostram que o consumo de energia da nossa solução é significativamente menor que aquele da solução em CPU.

5.1 Arquitetura Proposta

Neste trabalho, utiliza-se a ferramenta *Vivado® HLS* desenvolvida pela *Xilinx* com o objetivo de fornecer suporte para o desenvolvimento de circuitos em FPGAs Xilinx. Conforme explicado na Seção 3.4.2 a ferramenta permite que a especificação funcional de um sistema em alto nível (C/C++) seja usada para a produção de um circuito em nível *Register Transfer Level (RTL)*, sem a necessidade de fazê-lo manualmente [75]. Além disso, ela fornece algumas diretivas de compilação para otimização da arquitetura RTL produzida, como por exemplo: *loop unrolling*, *pipeline*, *array partition*, etc.

A Figura 5.1 apresenta a arquitetura da solução proposta mostrando interação entre CPU e a FPGA. A CPU transfere para a FPGA (*Block RAMs*)as duas sequências a serem comparadas e o circuito gerado pelo HLS já na FPGA calcula a matriz de programação dinâmica e retorna para a CPU o escore LCS entre as duas sequências.

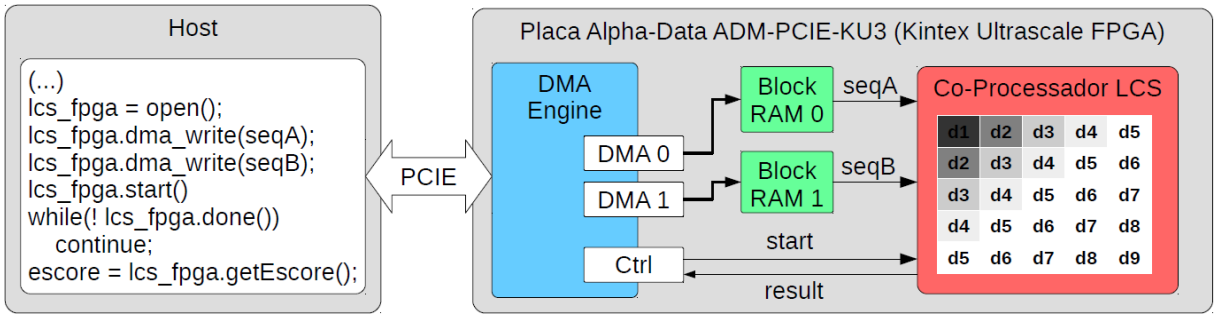

Figura 5.1: Arquitetura da implementação proposta.

As Figuras 5.2 e 5.3 mostram os módulos *BlockRAM* 0 e *BlockRAM* 1 para armazenar em *BlockRAM* as duas sequências a serem comparadas. O programa *host* copia de maneira paralela as duas sequências para as *BlockRAMs* através de dois barramentos *DMA*, o que diminuiu o tempo de transferência dos dados. Os outros blocos de circuitos apresentados nas figuras são interfaces geradas pelo Vivado® para controle e gerenciamento das memórias.

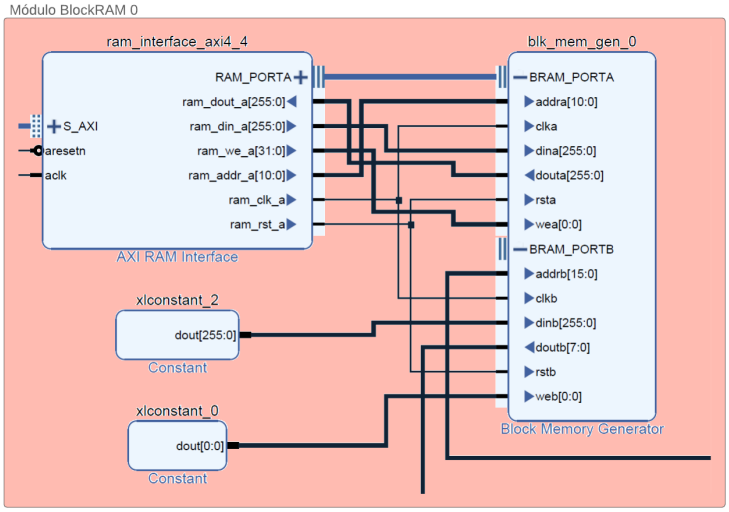

Figura 5.2: Módulo BLOCKRAM 0(circuito gerado pela ferramenta Vivado).

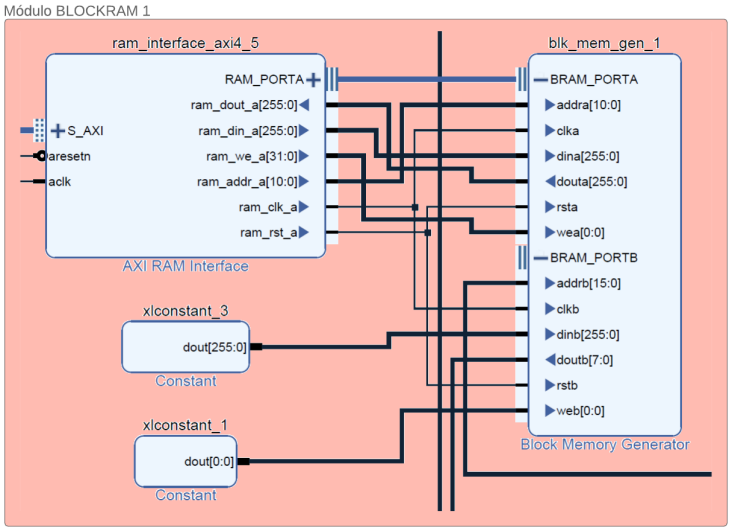

Figura 5.3: Módulo BLOCKRAM 1(circuito gerado pela ferramenta Vivado).

A Figura 5.4 apresenta o módulo de computação do algoritmo proposto também gerado pela ferramenta Vivado® HLS que interliga as portas *addrb* de cada um dos módulos de BlockRAMs às portas *X_address0* e *Y_address0* respectivamente. Então o Módulo de Computação acessa diretamente as duas *BlockRAMs* simultaneamente para cada caractere.

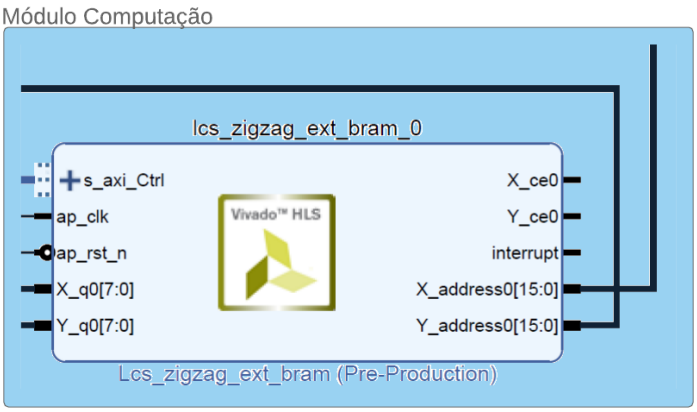

Figura 5.4: Módulo Computação (circuito gerado pela ferramenta Vivado).

A especificação do algoritmo implementado no Co-Processador LCS é uma adaptação do algoritmo LCS (Seção 2.3), calculando a matriz de programação dinâmica entre duas sequências na diagonal, e usando uma matriz de apenas 3 linhas. A Figura 5.5 exemplifica o fluxo de trabalho do algoritmo modificado para duas sequências de tamanho 5 (ACGTA e CCGTT). O cálculo da matriz de programação dinâmica é feito na diagonal utilizando-se de três vetores que armazenam os resultados das três diagonais mais recentes e à medida que o cálculo progride dentro da matriz, o vetor com a diagonal mais antiga é sobreposto pelo cálculo da diagonal corrente.

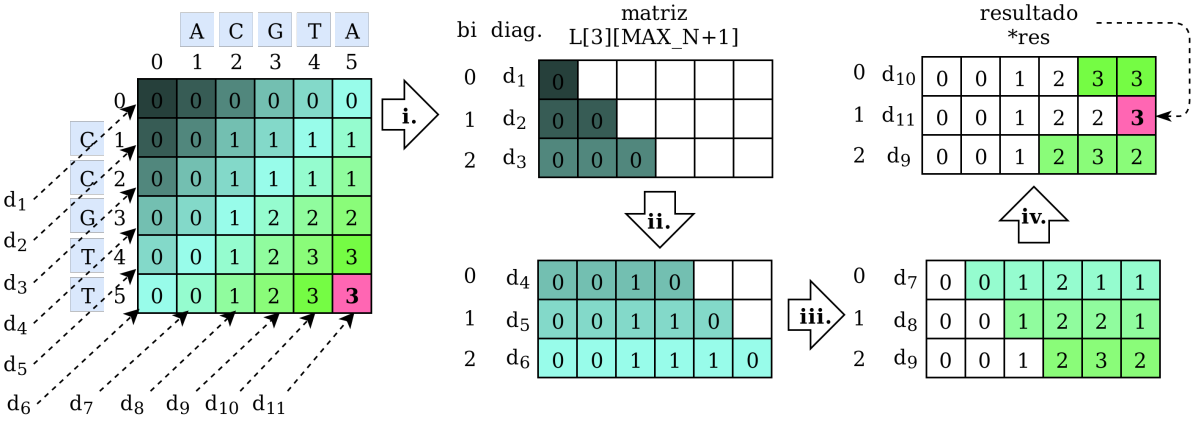

Figura 5.5: Fluxo do cálculo da matriz de programação dinâmica.

O trecho de código principal que especifica o LCS usando Vivado HLS é apresentado no Algoritmo 5.1. Inicialmente é definida uma matriz estática de três linhas $L[3][MAX\_N+1]$, onde MAX_N é o tamanho das sequências. Na matriz L serão armazenados os resultados intermediários a medida que a execução do algoritmo LCS progride. O tipo *u16* representa dados inteiros de 16-bits, sem sinal. Observe que a linha 7 especifica uma otimização de particionamento de array, indicando que a matriz L deve ser particionada em 3 blocos na dimensão 1, ou seja, 3 vetores. Isto é importante para que o sintetizador HLS possa instanciar elementos de memória com endereços de leitura/escrita separados para cada linha da matriz, permitindo o acesso em paralelo a qualquer uma das 3 linhas. Do contrário, a matriz inteira poderia ser mapeada em um conjunto de BlockRAMs com um único endereço de leitura/escrita, o que poderia gerar um gargalo no sistema.

No Algoritmo 5.1 observa-se a síntese de interfaces na linha 6, que especifica que os argumentos *m*,*n* e $*res$ obedecerão ao protocolo *AXI-Lite slave*, e que ainda serão agrupados em uma porta Ctrl (Controle). Logo, cada um destes argumentos será traduzido em registradores que poderão ser acessados por mapeamento de memória através de uma máquina *host*. Em seguida, a variável *bi* indica o índice do vetor (linha da matriz) que representa a diagonal a ser acessada e modificada. O tipo *u2* representa dados inteiros de 2-bits, sem sinal, pois a variável *u2* só pode indexar uma das 3 diagonais.

Algoritmo 5.1: Trecho da especificação funcional do LCS usando Vivado HLS.

```
1  #define MAX_N 50000
2  static u16 L[3][MAX_N + 1]; //matriz de apenas 3 linhas para otimizar uso de BRAM
3
4  void lcs(volatile uchar X[MAX_N],volatile uchar Y[MAX_N],int m,int n,int *res) {
5
6    #pragma HLS INTERFACE s_axilite port=m,n,res,return bundle=Ctrl
7    #pragma HLS ARRAY_PARTITION variable=L block factor=3 dim=1
8
9    u2 bi = 0;
10   for (int line=1; line <= ((m + 1) + (n + 1) - 1); line++) {
11     int start_col = max(0, line - (m + 1));
12     int count = min3(line, ((n + 1) - start_col), (m + 1));
13
14     if (bi > 2) bi = 0; //alterna entre uma das 3 linhas de L
```

```
15
16      for (int k = 0; k < count; k++) { #pragma HLS PIPELINE
17        int i = (min((m + 1), line) - k - 1);
18        int j = (start_col + k);
19
20        if (i == 0 || j == 0) { L[bi][j] = 0; }
21        else if (X[i - 1] == Y[j - 1]) {
22          if (bi == 0)       L[bi][j] = L[1][j - 1] + 1;
23          else if (bi == 1)  L[bi][j] = L[2][j - 1] + 1;
24          else               L[bi][j] = L[0][j - 1] + 1;
25        } else {
26          if (bi == 0)       L[bi][j] = max(L[2][j - 1], L[2][j]);
27          else if (bi == 1)  L[bi][j] = max(L[0][j - 1], L[0][j]);
28          else               L[bi][j] = max(L[1][j - 1], L[1][j]);
29        }
30      }
31      if (start_col < m) bi ++; //incrementa se existirem diagonais a processar
32    }
33    *res = L[bi][n];
34  }
```

A partir da linha 10 temos as iterações no cálculo da matriz de programação dinâmica por diagonal (indicada pela variável *line*) e armazenando os valores na matriz L particionada em três vetores das diagonais. Note que o particionamento é automático, ou seja, a forma de acesso à matriz L não precisa ser modificada para indicar o acesso a cada um dos vetores em particular. Na linha 16 do algoritmo também foi inserida uma otimização do HLS conhecida como PIPELINE. O pragma PIPELINE gera na arquitetura RTL um *pipeline* com um determinado intervalo de iniciação para uma função ou *loop*, permitindo a execução simultânea de operações em diferentes estágios do *pipeline*, diminuindo o tempo de execução entre cada uma das iterações. Durante a síntese, o intervalo de iniciação (*Initiation Interval*, II) é configurado em 1, indicando que a cada ciclo uma nova operação da função ou *loop* pode ser iniciada. Caso este valor inicial seja proibitivo para produção do circuito dentro das especificações de tempo fornecidas pelo desenvolvedor, o valor é incrementado até atingir um intervalo de iniciação aceitável para o circuito proposto operar conforme as especificações desejadas. Por fim, o resultado (escore máximo identificado entre as duas sequências de entrada) estará no último elemento da matriz L de programação dinâmica indicado pelas variáveis *bi* e *n*.

5.2 Experimentos

O ambiente de testes utilizado para os experimentos foi uma máquina portando uma CPU Intel Core i7-3770 e uma placa Alpha-Data ADM-PCIE-KU3 que possui um FPGA Xilinx Kintex® UltraScale XCKU060. O circuito gerado pela implementação possui frequência de 250MHz e utiliza os recursos da placa como mostra a Tabela 5.1.

Os dados utilizados para os testes realizados foram as sequências de DNA com identificadores de acesso NC_024791 e KM224878 ambas obtidas do site público NCBI (National Center for Biotechnology Information) [57] e truncadas em 50k.As mesmas sequências foram utilizadas para os testes de 20K e 10K, com o devido truncamento. Devido à sín-

Tabela 5.1: Tabela de recursos utilizados pelo circuito no FPGA Xilinx XCKU060.

Tipo	Usado	Disponível	Utilizado(%)
LUTS	26129	331680	7,88
Registradores	42208	663360	6,36
Blockram	223	1080	20,65
DSPs	64	2760	2,32
IOB	53	520	10,19
IO	49	104	47,12

tese ter chegado ao limite do caminho crítico, foi definido um único circuito de capacidade para tamanho máximo em 50k, o que permitiu a execução dos testes de 10k e 20k com o mesmo circuito.

No caso da solução proposta o intervalo de iniciação do *pipeline* foi configurado automaticamente em 4 pela ferramenta, ou seja, a cada 4 ciclos inicia-se a computação de um novo elemento da diagonal, e menos que isso não foi possível por conta das dependências de dados nos circuitos que foram gerados com intervalo de iniciação II = 1,2 e 3.

A energia elétrica *power* foi medida usando-se dos sensores existentes no FPGA (que medem amperagem e voltagem dos dois principais *power rails* que alimentam a placa). A ferramenta *sysmon* faz parte das referências de projetos da *Alpha-Data*, montadora da placa, em que é possível visualizar a amperagem e voltagem registrados pelos sensores durante o tempo de execução do algoritmo. Utilizando-se das DMAs para a transferência de dados, o tempo de transferência entre a CPU e FPGA não produziu impacto significativo e, portanto, foi desprezado.

A energia elétrica (*power*) foi a mesma para todas as comparações, pois foi implementado um único circuito com capacidade de armazenar sequências de até 50000 caracteres. O tempo de execução foi medido através do *host*, que contabiliza desde o tempo de transferência dos dados para o FPGA até o retorno do escore para o *host*. Para a CPU foi utilizada a ferramenta *powerstat* [25] que mede a energia elétrica do processador quando o algoritmo está em execução.

A Tabela 5.2 apresenta os resultados obtidos para cada teste realizado no FPGA. Nessa tabela, a energia (*energy*) foi calculada multiplicando-se a potência pelo tempo de execução.

Como pode ser visto na Tabela 5.2, o tempo de execução das três comparações (10K, 20K e 50K) é menor na CPU, sendo em torno de 25% mais rápido quando comparado com o nosso projeto em FPGA. No entanto, quando consideramos a energia gasta na execução nas duas plataformas, a nossa solução em FPGA consome 15% da energia gasta em CPU. Por exemplo, temos 17,02 Joules em FPGA e 108,40 Joules em CPU para a comparação de 10K.

Tabela 5.2: Tempos de execução, energia elétrica (W) e energia (J) para as comparações de 10K, 20K e 50K no FPGA XCKU060 e em CPU.

	Tamanho	**Tempo (s)**	**Escore**	**Potência Elétrica (W)**	**Energia (J)**
FPGA	10K	01,60	8672	10,64	17,02
CPU	10K	01,22	8672	88,86	108,40
FPGA	20K	06,40	17141	10,64	68,09
CPU	20K	04,80	17141	90,80	435,84
FPGA	50K	40,00	41497	10,64	425,60
CPU	50K	30,00	41497	93,96	2907,90

5.3 Considerações Finais

Neste capítulo foi proposta e avaliada uma solução HLS em FPGA para comparação de sequências biológicas com o algoritmo LCS. A nossa solução utilizou *Block RAMs* para armazenar as sequências, que são acessadas através dos barramentos DMA do FPGA, otimizou as iterações do cálculo da matriz de programação dinâmica utilizando pipeline, reduziu o custo de espaço em memória utilizando somente 3 vetores para armazenar os valores calculados e diminuiu o tempo de execução calculando-se na diagonal.

Os resultados experimentais mostraram que a solução proposta em FPGA é capaz de consumir muito menos energia que a solução em CPU. Na comparação de duas sequências de 50000 caracteres, a energia gasta foi 425,60 Joules em FPGA enquanto a mesma comparação gastou 2907,90 Joules em CPU. Com isso, nota-se que a CPU consumiu, nesse caso, 6,8x mais energia para executar a implementação em relação à FPGA. Nota-se também que o tempo de execução em FPGA aumenta cerca de 25% em relação à CPU. Com a nossa solução, o consumo de energia foi significativamente menor no FPGA.

Capítulo 6

Arquitetura Heterogênea de CPU+FPGA para execuções LCS

No Capítulo 5, propusemos e avaliamos um projeto em HLS para execução de comparações LCS de um par de sequências em FPGA. No presente capítulo, consideramos a aplicação como um conjunto de comparações e propomos um módulo de alocação de tarefas para arquiteturas heterogêneas (CPU + FPGA). Com isso, o usuário pode fazer uma comparação de pares ou várias comparações de pares em um único aplicativo. Mostramos que usar tanto a CPU quanto o FPGA para a aplicação LCS composta por várias tarefas de comparação traz benefícios em termos de tempo de execução e consumo de energia, quando comparada às versões que utilizam somente CPU ou somente FPGA.

As crescentes demandas por desempenho de computação têm sido uma realidade independentemente dos requisitos para dispositivos menores e mais eficientes em energia, com o FPGA alcançando uma economia de energia muito maior do que a CPU na maioria das situações. Em nossa aplicação particular, observamos no Capítulo 5 que a CPU tem melhor desempenho enquanto o FPGA consome menos energia. Neste cenário, nossa abordagem heterogênea (CPU + FPGA) visa tirar proveito das melhores características em ambas as plataformas, alcançando um melhor *trade-off* entre tempo de execução e consumo de energia, quando comparada às soluções em plataformas separadas (somente CPU ou somente FPGA).

6.1 Arquitetura CPU+FPGA Proposta

Conforme mencionado no Capítulo 4, a maioria das abordagens FPGA para comparação de sequências na literatura usa linguagens de descrição de hardware (VHDL ou Verilog) em seu projeto. Mesmo que essa estratégia possa levar a um alto desempenho, o tempo de desenvolvimento é muito longo, é altamente sujeito a erros e a portabilidade do código

é baixa. Para lidar com esses problemas, linguagens de alto nível, como Vivado® HLS, foram propostas e são, de fato, uma alternativa muito boa para o design de hardware tradicional.

No presente capítulo, descrevemos o projeto de um *framework* que integra as soluções CPU e FPGA descritas no Capítulo 5 com um escalonador CPU-FPGA, responsável por distribuir as comparações entre as plataformas e gerenciar a execução paralela.

6.1.1 Escalonador CPU + FPGA

Em muitas situações, os biólogos desejam fazer mais do que uma comparação de sequências. Assim, eles definem a aplicação como um conjunto de comparações, onde cada comparação executa o algoritmo LCS para um par de sequências.

A fim de distribuir as comparações entre as plataformas FPGA e CPU, um escalonador *round robin* baseado em peso foi desenvolvido. Dois pesos são atribuídos a cada uma das plataformas, um relacionado ao tempo de execução e outro ao consumo de energia. Esses pesos são obtidos experimentalmente executando algumas comparações de sequências em ambas as plataformas. Em seguida, os pesos obtidos são comparados para calcular a proporção do consumo de energia e o tempo de execução. Caso o usuário assim o prefira, ele pode especificar os pesos manualmente.

Uma vez configurados os pesos, o escalonador gera alocações para a CPU e FPGA, dividindo as comparações entre as plataformas, que ocorrem simultaneamente, após a divisão.

Algoritmo 6.1: Trecho do escalonador usando C ++.

```
1 std::vector<std::string> files;
2 std::string path = PATH;
3 read_files_on(files,PATH);
4 obtain_weights(&cpuWeight, &fpgaWeight);
5 weighted_scheduler(files, cpuQueue, fpgaQueue, cpuWeight, fpgaWeight);
6 // both threads are executed in parallel
7 int timeExecCPU, timeExecFPGA,timeExecTotal;
8 thread(threadCPU,cpuQueue,timeExecCPU);
9 thread(threadFPGA,fpgaQueue,timeExecFPGA);
10 getTime(timeExecTotal);
```

O algoritmo 6.1 apresenta o pseudo-código do escalonador. Na linha 3, são lidos os arquivos contendo as sequências a serem comparadas. Na linha 4, são obtidos os pesos para CPU e FPGA. Então, na linha 5, o escalonador usa os pesos para atribuir comparações aos dispositivos (CPU ou FPGA) escrevendo os identificadores das comparações nas filas locais da CPU ou FPGA. Depois disso, duas threads separadas (*threadCPU* e *threadFPGA*) acessam suas filas locais e executam as comparações em paralelo (linhas 8 e 9). Mostramos também as diretrizes para obtenção do tempo de execução nas linhas

7 e 10. Quando o consumo de energia é considerado, na linha 7 iniciamos a medição da potência e a linha 10 marca o final da medição.

6.1.2 Integração dos módulos escalonador, CPU e FPGA

O *Host* CPU é responsável pelo gerenciamento da execução no FPGA. Como o nosso projeto admite uma arquitetura híbrida (CPU-FPGA), a aplicação LCS será executada tanto no FPGA como no *host CPU*. O escalonamento e controle da execução são feitos integralmente no *Host CPU*.

Dentro do *CPU* é executado o escalonador (Figura 6.1(2)). Esse módulo faz leitura das sequências (Figura 6.1(1)) inseridas pelo usuário e seleciona quais sequências serão comparadas em CPU (Figura 6.1(3-a)) e quais serão comparadas em FPGA (Figura 6.1(3-b)). As sequências que serão comparadas em CPU são enviadas ao LCS CPU (Figura 6.1(4)). As sequências que serão comparadas em FPGA, então são enviadas para o controle de execução do FPGA (Figura 6.1(5)) que, através do DMA, envia para o FPGA utilizando o barramento de PCIe.

As sequências 0 e 1 recebidas pelo DMA (Figura 6.1(6)) do FPGA são armazenadas dentro das *Block RAMs* do FPGA através das suas interfaces de RAM (Figura 6.1(7 e 8)). A seguir, o Co-Processador LCS (Figura 6.1(9)) faz a leitura das sequências nas *Block RAMs* e executa a comparação.

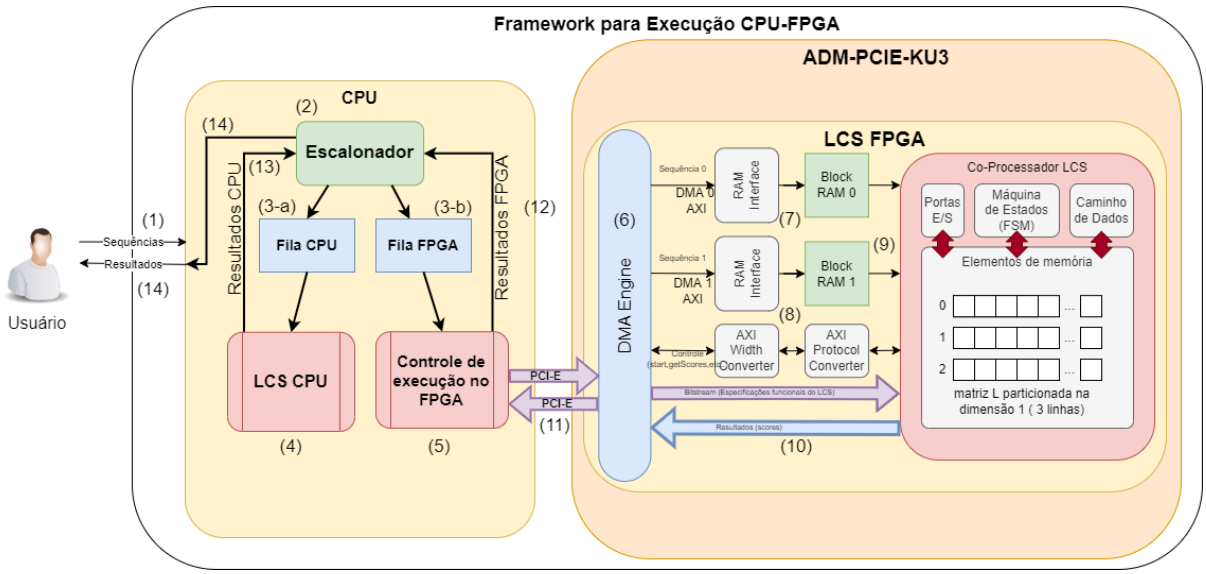

Figura 6.1: Visão geral da arquitetura proposta em ambiente dedicado.

Após o término de todas as comparações, o score obtido é enviado para a memória do *CPU* pela DMA do FPGA (Figura 6.1(10)) e a DMA entrega os resultados para o Controle de execução no FPGA (Figura 6.1(11)). Após receber os resultados de todas

as comparações, estes são colocados em um único arquivo pelo Controle de execução no FPGA (Figura 6.1(5)) e entregues ao Escalonador (Figura 6.1(12)) . Os resultados obtidos pela LCS CPU (Figura 6.1(4)) também são entregues ao Escalonador (Figura 6.1(13)). E por fim, os resultados são agrupados e enviados ao usuário (Figura 6.1(14)).

6.2 Experimentos

O ambiente de teste usado para os experimentos foi uma máquina dedicada com CPU Intel Core i7-3770 de 3,40 GHz e placa Alpha-Data ADM-PCIE-KU3 com Xilinx Kintex® UltraScale XCKU060 FPGA, ou seja, o mesmo ambiente utilizado no Capítulo 5. O circuito gerado pela implementação de nosso componente FPGA proposto tem uma frequência de 250 MHz e utiliza os recursos da placa conforme mostrado na Tabela 5.1.

Para criar o conjunto de comparações, rodamos um *script* automatizado que gerou 40 sequências aleatórias de DNA de cada tamanho (10k, 20k, 50k), com um total de 20 comparações para cada tamanho de sequência.

No caso da solução FPGA, o intervalo de iniciação *pipeline* foi automaticamente definido em 4 pela ferramenta, ou seja, a cada 4 ciclos ela passa a computar um novo elemento diagonal.Tentamos definir o intervalo de inicialização para 1, 2 e 3, mas a ferramenta Vivado não foi capaz de lidar com isso a 250 MHz devido às dependências de dados.

Da mesma maneira descrita no Capítulo 5, a potência (energia elétrica - *power*) foi medida usando os sensores disponíveis na placa *Alpha-Data*. Os sensores medem a amperagem e a tensão dos dois barramentos de alimentação principais que alimentam os componentes da placa, incluindo o chip FPGA.

A ferramenta *sysmon* faz parte do projeto de referência da placa *Alpha-Data*, na qual é possível visualizar a amperagem e a tensão registradas pelos sensores durante o tempo de execução do algoritmo. A energia é a mesma para todos os testes de FPGA, pois um único circuito foi implementado com capacidade para armazenar sequências de até 50.000 caracteres cada. Para a CPU, foi utilizada a ferramenta *powerstat* [25], que mede a energia elétrica do processador quando o algoritmo está rodando.

O tempo de execução foi medido através do *host* conforme mostrado no Algoritmo 6.1 (Linhas 7 e 10) então, começamos a medir o tempo quando os *threads* de CPU e FPGA são criados e paramos de medir quando ambos os *threads* terminam a execução, ou seja, o o tempo de execução é o tempo de execução das 20 comparações. No caso do FPGA, inclui o tempo de transferência dos dados para o FPGA até o retorno da pontuação ao *host*. Como os testes foram feitos em um ambiente dedicado, a variação nas execuções repetidas é insignificante.

6.2.1 Resultados na CPU e no FPGA Separadamente

A Tabela 6.1 mostra os resultados obtidos para cada teste realizado no FPGA e na CPU separadamente. Nesta tabela, a energia foi calculada multiplicando a potência pelo tempo de execução.

Como pode ser visto na Tabela 6.1, o tempo de execução das três comparações (10K, 20K e 50K) é menor na CPU, sendo cerca de 25% mais rápido quando comparado ao nosso projeto FPGA.No entanto, quando consideramos a energia gasta em execução em ambas as plataformas, nossa solução FPGA consumiu 12,21% para tamanho de 10k, 11,87% para tamanhos de 20k e 11,90% para tamanhos de 50k, da energia consumida na CPU.

Tabela 6.1: Tempo de execução, resultados de potência e energia para 20 comparações nas plataformas somente CPU e somente FPGA.

	Tamanho	**Tempo (s)**	**Potência (W)**	**Energia (J)**
CPU-only	10k	28,30	99,24	2808,49
FPGA-only	10k	32,09	10,69	343,04
CPU-only	20k	113,42	101,88	11555,23
FPGA-only	20k	128,35	10,69	1372,06
CPU-only	50k	706.35	101,96	72019,44
FPGA-only	50k	801,94	10,69	8572,74

6.2.2 Resultados Combinados de CPU-FPGA

Nesta seção, usamos o mesmo método para medição de energia explicado no final da Seção 6.2 e adicionamos o escalonador para dividir as comparações nas duas plataformas usando o *round-robin* ponderado conforme descrito na Seção 6.1.1.

Para a CPU e FPGA, respectivamente, foram feitas duas implementações: (1) uma CPU e um co-processador na FPGA e (2) uma CPU e dois co-processadores na FPGA. Dentro de cada implementação, foram feitas variações de pesos para cada plataforma, como descrito na Tabela 6.2. Nessa tabela, a notação 1CO-03C-17F significa um co-processador no FPGA (1CO), 3 comparações em CPU (03C) e 17 comparações em FPGA (17F).

A partir da Tabela 6.2 temos os resultados para cada implementação e para cada configuração.

Uma CPU e um co-processador na FPGA

A Tabela 6.3 apresenta o tempo de execução e energia obtidos para a configuração 1CO-03C-17F. Pode-se notar que essa distribuição é a que prioriza o consumo de energia,

Tabela 6.2: Configurações de execução CPU-FPGA

1 CPU e 1 co-processador na FPGA	**Quantidade de Comparações**		**1 CPU e 2 co-processadores na FPGA**	**Quantidade de Comparações**	
Configuração	**CPU**	**FPGA**	**Configuração**	**CPU**	**FPGA**
1CO-03C-17F	03	17	**2CO-02C-18F**	02	18
1CO-06C-14F	06	14	**2CO-06C-14F**	06	14
1CO-10C-10F	10	10	**2CO-10C-10F**	10	10
1CO-14C-06F	14	06	**2CO-14C-06F**	14	06
1CO-17C-03F	17	03	**2CO-18C-02F**	18	02

entretanto o tempo de execução é maior do que as configurações que usam mais CPUs. Note que, para o lote de comparações de 50K, a energia gasta com 3 comparações em CPU é maior do que a energia gasta com 17 comparações em FPGA.

Tabela 6.3: Tempo e energia obtidos para a configuração 1CO-03C-17F.

03 CPU/ 17 FPGA	**Tamanho**	**Tempo(s)**	**Tempo Total(s)**	**Potência(W)**	**Energia(J)**	**Energia Total(J)**
CPU	10k	04,28	27,28	98,43	421,28	712,9
FPGA		27,28		10,69	291,62	
CPU	20k	17,06	109,08	99,38	1695,42	2861,48
FPGA		109,08		10,69	1166,06	
CPU	50k	106,14	681,65	101,13	10733,94	18020,78
FPGA		681,65		10,69	7286,84	

A Tabela 6.4 apresenta os resultados para 6 comparações em CPU e 14 comparações em FPGA. Nessa distribuição das comparações, pode-se notar que, em relação à configuração 1CO-03C-17F, há uma diminuição do tempo de execução, porém há um aumento no consumo de energia.

Tabela 6.4: Tempo e energia obtidos para a configuração 1CO-06C-14F.

06 CPU/ 14 FPGA	**Tamanho**	**Tempo(s)**	**Tempo Total(s)**	**Potência(W)**	**Energia(J)**	**Energia Total(J)**
CPU	10k	08,56	22,46	108,24	926,53	1166,63
FPGA		22,46		10,69	240,10	
CPU	20k	34,12	89,83	111,64	3809,15	4769,43
FPGA		89,83		10,69	960,29	
CPU	50k	212,28	561,35	107,38	22794,62	28795,45
FPGA		561,35		10,69	6000,83	

Na Tabela 6.5 temos a distribuição igual das comparações para CPU e FPGA, com 10 comparações para cada. Aqui, nota-se que houve uma aproximação do tempo de execução

para cada uma das plataformas, porém o consumo de energia é significativamente maior na CPU, cerca de 8,9x em relação à FPGA.

Tabela 6.5: Tempo e energia obtidos para a configuração 1CO-10C-10F.

10 CPU/ 10 FPGA	**Tamanho**	**Tempo(s)**	**Tempo Total(s)**	**Potência(W)**	**Energia(J)**	**Energia Total(J)**
CPU	10k	14,26	16,04	108,24	1543,50	1714,96
FPGA		16,04		10,69	171,46	
CPU	20k	56,86	64,16	111,64	6347,85	7033,72
FPGA		64,16		10,69	685,87	
CPU	50k	353,8	400,09	107,38	37991,04	42268
FPGA		400,09		10,69	4276,96	

Na Tabela 6.6 temos a distribuição de 14 comparações sendo executadas em CPU e 06 comparações na FPGA. Aqui, nota-se que houve um aumento tanto no tempo de execução como no consumo de energia.

Tabela 6.6: Tempo e energia obtidos para a configuração 1CO-14C-06F.

14 CPU/ 6 FPGA	**Tamanho**	**Tempo(s)**	**Tempo Total(s)**	**Potência(W)**	**Energia(J)**	**Energia Total(J)**
CPU	10k	19,96	19,96	98,43	1964,66	2067,50
FPGA		09,62		10,69	102,84	
CPU	20k	79,60	79,60	99,38	7910,65	8322,21
FPGA		38,50		10,69	411,56	
CPU	50k	495,32	495,32	101,13	50091,71	52657,84
FPGA		240,05		10,69	2566,13	

E por último, na Tabela 6.7 temos a distribuição de 17 comparações sendo executadas em CPU e 03 comparações na FPGA, em que o tempo de execução na CPU é cerca de 5x maior que o tempo da FPGA e o consumo de energia total é superior a todas execuções anteriores.

Tabela 6.7: Tempo e energia obtidos para a configuração 1CO-17C-03F.

17 CPU/ 03 FPGA	**Tamanho**	**Tempo (s)**	**Tempo Total(s)**	**Potência (W)**	**Energia (J)**	**Energia Total(J)**
CPU	10k	24,24	24,24	98,43	2385,94	2437,35
FPGA		04,81		10,69	51,41	
CPU	20k	96,66	96,66	99,38	9606,07	9811,85
FPGA		19,25		10,69	205,79	
CPU	50k	601,46	601,46	101,13	60825,65	62108,66
FPGA		120,02		10,69	1283,01	

Gráficos Comparativos por Tamanho para 1 Co-Processador

A Figura 6.2 mostra o comparativo entre todas as configurações de comparações com sequências de tamanho 10k do escalonador juntamente com as execuções somente com CPU e somente FPGA. Pode-se notar que nessa figura os resultados do escalonador que mais se aproximam da configuração somente em FPGA em consumo de energia é a configuração 1CO-03C-17F e se executa mais rápido é a configuração 1CO-10C-10F. Ao se considerar tanto o consumo de energia como o tempo de execução, a configuração 1CO-03C-17F apresentou bons resultados. Em relação somente à consumo de energia, há um aumento de 108% quando comparado à execução somente em FPGA.

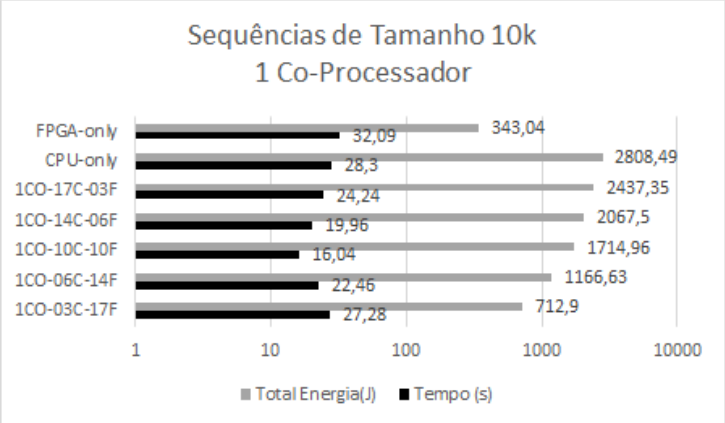

Figura 6.2: Gráfico comparativo em relação à sequências de tamanho 10k para diversas configurações.

A Figura 6.3 mostra o comparativo entre todas as execuções de comparações com sequências de tamanho 20k do escalonador juntamente com as execuções somente com CPU e somente FPGA. Pode-se notar que, ao considerarmos o consumo de energia, os resultados do escalonador que mais se aproximam da execução somente em FPGA é também a configuração 1CO-03C-17F. Da mesma forma, o melhor tempo de execução para as comparações de 20K foi obtido pela configuração 1CO-10C10F, porém essa configuração apresenta um alto consumo de energia.

A Figura 6.4 mostra o comparativo entre todas as execuções de comparações com sequências de tamanho 50k do escalonador juntamente com as execuções somente com CPU e somente FPGA. Também para esse tamanho de sequências, o melhor resultado em consumo de energia foi obtido pela configuração somente FPGA e, em tempo de execução, pela configuração 1CO-10C-10F.

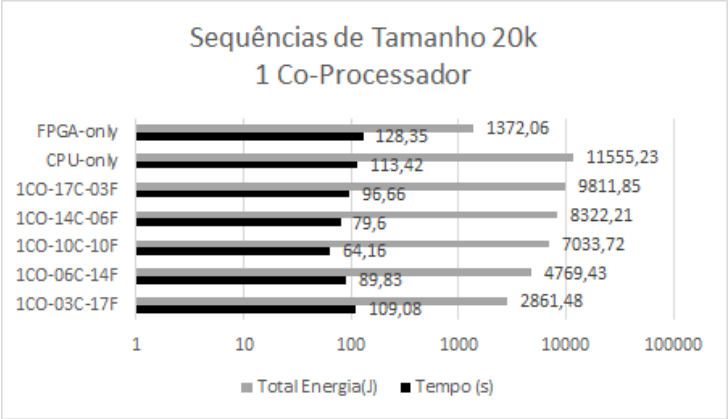

Figura 6.3: Gráfico comparativo em relação à sequências de tamanho 20k para cada execução do escalonador.

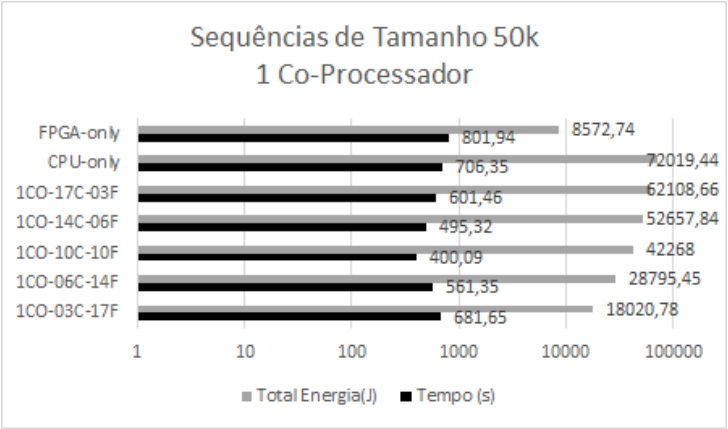

Figura 6.4: Gráfico comparativo em relação à sequências de tamanho 50k para cada execução do escalonador.

Uma CPU e dois co-processadores na FPGA

Na Tabela 6.8 apresenta os resultados para 2 co-processadores, com 2 comparações na CPU e 18 na FPGA. Pode-se notar que essa distribuição é a que prioriza o consumo de energia, entretanto o tempo de execução é maior do que em diversas outras execuções. Para cada plataforma, é possível notar que o tempo de execução é significativamente maior, porém o consumo de energia é relativamente próximo para tamanhos de 10k e 20k, mas para 50k, já percebe-se um aumento significativo no consumo pela CPU.

Na Tabela 6.9 apresentamos os resultados para 2CO-06C-14F. Na comparação com 2CO-02C-18F, nota-se redução no tempo de execução e grande aumento no consumo de energia.

Na Tabela 6.10 temos a distribuição igual das comparações para CPU e FPGA, com 10 comparações para cada. Aqui, nota-se que houve um aumento de 105,36% do tempo de execução na CPU em relação à FPGA e o consumo de energia na FPGA é de 92,86%

Tabela 6.8: Tempo e energia obtidos para a configuração 2CO-02C-18F.

2 CPU/ 18 FPGA	Tamanho	Tempo(s)	Tempo Total(s)	Potência(W)	Energia (J)	Energia Total(J)
CPU	10k	2,7	15,0	100,4	275,1	593,5
FPGA		15,0		21,2	318,4	
CPU	20k	11,1	67,0	111,3	1237,6	2657,8
FPGA		67,0		21,2	1420,2	
CPU	50k	106,1	447,5	131,47	13954,2	23440,8
FPGA		447,5		21,2	9486,6	

Tabela 6.9: Tempo e energia obtidos para a configuração 2CO-06C-14F.

6 CPU/ 14 FPGA	Tamanho	Tempo(s)	Tempo Total(s)	Potência(W)	Energia(J)	Energia Total(J)
CPU	10k	8,3	11,9	100,4	833,3	1085,8
FPGA		11,9		21,2	252,5	
CPU	20k	36,0	53,7	111,3	4010,1	5147,7
FPGA		53,7		21,2	1137,6	
CPU	50k	356,6	417,7	131,5	46886,1	55740,3
FPGA		417,7		21,2	8854,2	

menor que na CPU.

Na Tabela 6.11 temos a distribuição de 14 comparações sendo executadas em CPU e 06 comparações na FPGA. Aqui, nota-se que houve uma disparidade em relação ao tempo de execução e também em consumo de energia na CPU, sendo que o consumo de energia da FPGA é 2,86% do total dessa execução.

E por último, na Tabela 6.12 temos a distribuição de 18 comparações sendo executadas em CPU e 02 comparações na FPGA, em que o tempo de execução na CPU é 19,22x maior que o tempo da FPGA e o consumo de energia total é superior a todas execuções anteriores.

Tabela 6.10: Tempo e energia obtidos para a configuração 2CO-10C-10F.

[h]

10 CPU/ 10 FPGA	Tamanho	Tempo(s)	Tempo Total(s)	Potência(W)	Energia(J)	Energia Total(J)
CPU	10k	13,8	13,8	100,4	1389,5	1584,2
FPGA		9,2		21,2	194,6	
CPU	20k	56,7	56,7	112,4	6374,8	7155,8
FPGA		36,8		21,2	781,0	
CPU	50k	536,0	536,0	144,6	77512,4	83046,2
FPGA		261,0		21,2	5533,8	

Tabela 6.11: Tempo e energia obtidos para a configuração 2CO-14C-06F.

14 CPU/ 06 FPGA	Tamanho	Tempo(s)	Tempo Total(s)	Potência(W)	Energia(J)	Energia Total(J)
CPU	10k	19,2	19,2	119,5	2291,6	2397,8
FPGA		5,0		21,2	106,2	
CPU	20k	11,1	22,3	133,6	1485,2	1958,6
FPGA		22,3		21,2	473,4	
CPU	50k	743,0	743,0	144,6	107451,2	110613,4
FPGA		149,2		21,2	3162,2	

Tabela 6.12: Tempo e energia obtidos para a configuração 2CO-18C-02F.

18 CPU/ 02 FPGA	Tamanho	Tempo(s)	Tempo Total(s)	Potência(W)	Energia(J)	Energia Total(J)
CPU	10k	24,7	24,7	100,4	2477,8	2517,5
FPGA		1,9		21,2	39,6	
CPU	20k	100,1	100,1	111,3	11138,9	11296,6
FPGA		7,4		21,2	157,7	
CPU	50k	955,3	955,3	131,5	125588,0	126642,1
FPGA		49,7		21,2	1054,1	

Gráficos Comparativos por Tamanho para 2 Co-Processadores

A Figura 6.5 mostra o comparativo entre todas as execuções de comparações com sequências de tamanho 10k do escalonador juntamente com as execuções somente com CPU e somente FPGA. Pode-se notar que, em termos de consumo de energia, os resultados do escalonador que mais se aproxima da execução somente em FPGA são obtidos pela configuração 2CO-02C-18F e a apresenta menor tempo de execução é a configuração 2CO-06C-14F. Se considerarmos tanto o tempo de execução como o consumo de energia, a configuração 2CO-02C-18F apresenta um bom compromisso. Em relação somente à consumo de energia, há um aumento de 108% quando comparado à execução somente em FPGA, porém há uma redução de 74,62% quando comparado à execução somente em CPU.

A Figura 6.6 mostra o comparativo entre todas as execuções de comparações com sequências de tamanho 20k do escalonador juntamente com as execuções somente com CPU e somente FPGA. Quanto ao consumo de energia, os melhores resultados são obtidos pela configuração somente FPGA e quanto ao tempo de execução os melhores resultados são também obtidos pela configuração 2CO-06C-14F. Ao se considerar tanto o consumo de energia como o tempo de execução, a configuração 2CO-02C-18F apresenta-se como uma boa escolha.

A Figura 6.7 mostra o comparativo entre diversas configurações de comparações com sequências de tamanho 50k com 2 co-processadores. Para esse tamanho de sequências, os

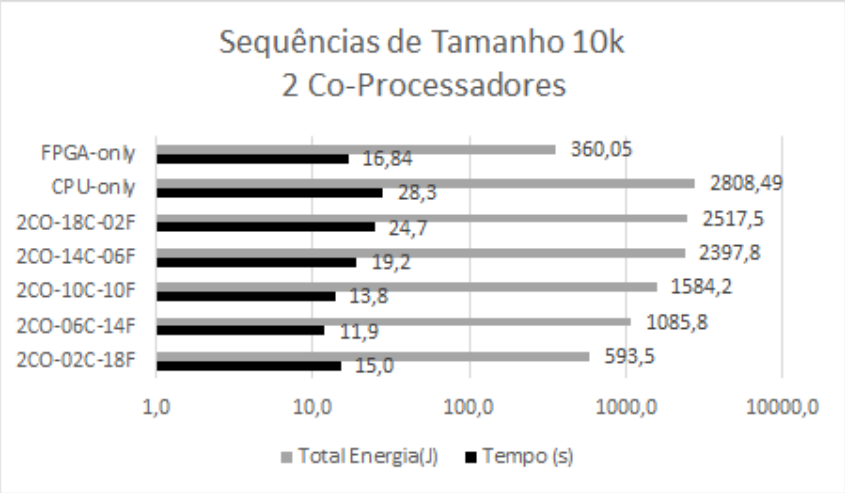

Figura 6.5: Gráfico comparativo em relação à sequências de tamanho 10k para cada execução do escalonador.

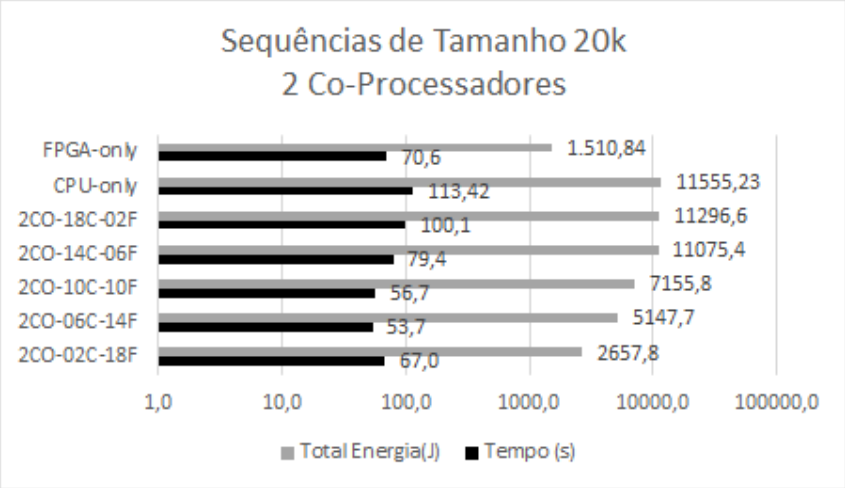

Figura 6.6: Gráfico comparativo em relação à sequências de tamanho 20k para cada execução do escalonador.

dois melhores resultados são também somente FPGA (consumo de energia) e 2CO-06C-14F (tempo de execução).

6.2.3 Comparação com o estado da arte da CPU

Nesta seção, comparamos nossos resultados com a ferramenta *multi-thread* de CPU recentemente proposta por Shikder et al. [67], chamado CPU-OpenMP. No CPU-OpenMP, há uma fase de pré-processamento para determinar o deslocamento máximo em cada linha, com complexidade de tempo $O(n)$. Em seguida, uma relação de recorrência LCS modificada [67] que não possui dependências na mesma linha é executada. Nesta fase, todos os

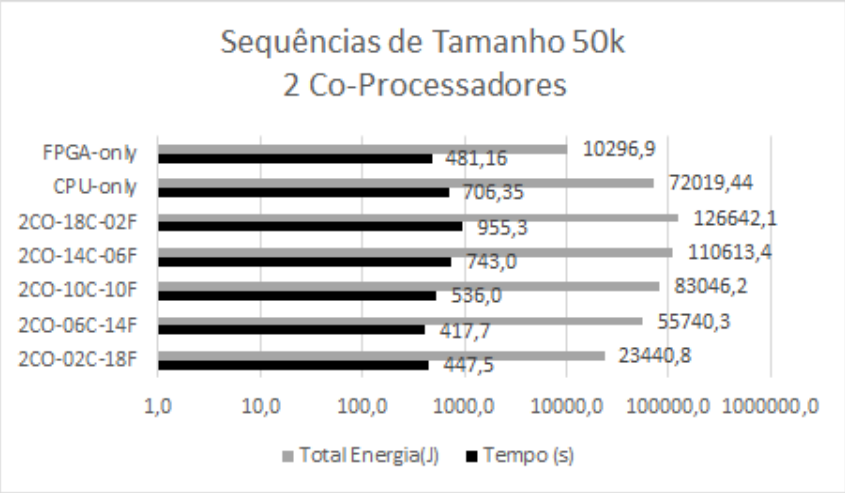

Figura 6.7: Gráfico comparativo em relação à sequências de tamanho 50k para cada execução do escalonador.

Tabela 6.13: Comparação com a ferramenta CPU-OpenMP (tempo de execução e resultados de energia).

	Tamanho	**Tempo (s)**	**Energia (J)**
2CO-02C-18F	10k	15,00	593,50
FPGA	10k	16,84	**360,05**
CPU-OpenMP (4 cores)[67]	10k	**7,21**	890,07
2CO-02C-18F	20k	67,01	2657,80
FPGA	20k	70,60	**1510,84**
CPU-OpenMP (4 cores)[67]	20k	**29,33**	3478,92
2CO-02C-18F	50k	447,50	23440,80
FPGA	50k	481,16	**10296,90**
CPU-OpenMP (4 cores)[67]	50k	**171,32**	21569,19

elementos de uma mesma linha podem ser computados em paralelo e, assim, o processamento é feito linha por linha, ao invés do tradicional anti-diagonal por anti-diagonal.

A Tabela 6.13 mostra os resultados obtidos com CPU+FPGA, FPGA e CPU-OpenMP (4 núcleos) com as mesmas sequências de DNA (10k,20k,50k) descritas na Seção 6.2. Apresentamos na Tabela 6.13 os resultados de FPGA com 2 co-processadores e os resultados da configuração 2CO-02C-18F, que apresentaram bom compromisso entre o tempo de execução e o consumo de energia.

Como o CPU-OpenMP é uma versão altamente otimizada do LCS que roda em múltiplos núcleos, ele obtém bons *speedups* (entre 2.08x a 2.81x), quando comparado com nossas versões FPGA (CPU+FPGA e FPGA). Isso ocorre às custas do alto consumo de energia. O consumo de energia da ferramenta CPU-OpenMP é muito maior do que o consumo de energia da nossa versão FPGA (cerca de 2x maior). Portanto, se o consumo

de energia for uma preocupação, a versão FPGA deve ser usada. Para as comparações de 10K e 20K, a versão CPU+FPGA oferece uma opção intermediária, considerando tanto o tempo de execução quanto o consumo de energia.

Para as sequências de 50K, a configuração 2CO-02C-18F apresentou um aumento de consumo de energia de 8% em relação ao CPU-OpenMP. Nas comparações de 10K e 20K, respectivamente,houve uma redução de 33% e 24% no consumo de energia da configuração 2CO-02C-18F, quando comparado ao CPU-OpenMP.

6.3 Estudo de Caso: Covid-19

Em 12 de dezembro de 2019, diversos casos de síndrome respiratória aguda grave causada por um coronavírus recém-identificado foram anunciados em Wuhan, China. Este coronavírus foi inicialmente nomeado como o novo coronavírus de 2019 (2019-nCoV) em 12 de janeiro de 2020 pela Organização Mundial da Saúde (OMS). A epidemia se espalhou rapidamente e a doença foi nomeada oficialmente como doença pela OMS como Covid-19, sendo o vírus 2019-nCoV renomeado para SARS-CoV-2. Como uma doença infecciosa respiratória aguda, o SARS-CoV-2 se transmite principalmente pelo trato respiratório, gotículas, secreções respiratórias e contato direto [37].

Nesta seção, realizamos um estudo de caso sobre a doença Covid-19, comparando sequências de SARS-CoV-2 provenientes de diversos países.

Para fazer este estudo de caso, comparamos 20 sequências reais de SARS-CoV-2 de vários locais recuperados do NCBI (National Center for Biotechnology Information) em *www.ncbi.nlm.nih.gov/sars-cov-2* com a sequência de referência SARS-CoV-2 de Wuhan, China, recuperada do mesmo local. As sequências selecionadas foram disponibilizadas no primeiro semestre de 2020, possuem de igual tamanho (29903 caracteres) e foram selecionadas aleatoriamente de todos os cinco continentes - pelo menos uma sequência por continente. A configuração CPU-FPGA utilizada foi a 1CO-03C-17F.

Os números de acesso, nome, comprimento e localização das sequências usadas neste estudo estão listados na Tabela 6.14. A primeira sequência da tabela é a sequência de referência e todas as sequências restantes foram comparadas a esta.

A Tabela 6.15 apresenta os tempos de execução e os resultados de energia para o estudo covid-19. Esses resultados mostram que os melhores tempos de execução são obtidos pela versão somente CPU e a solução somente FPGA tem a menor energia consumo. A solução CPU-FPGA oferece um bom equilíbrio entre tempo de execução e energia.

A Tabela 6.16 mostra, para cada sequência comparada com a sequência de referência *NC_045512.2*, o número de acesso, a pontuação LCS obtida por nossa ferramenta, o país da sequência e a data em que ela foi atualizada pela última vez no banco de dados público

Tabela 6.14: Sequências de SARS-CoV-2. Todas as sequências têm 29903 caracteres.

Accesso	Descrição	País
Ásia		
NC_045512.2	*Síndrome respiratória aguda grave coronavírus 2 isolado Wuhan-Hu-1*	China
MT135044.1	*Síndrome respiratória aguda grave coronavírus 2 isolado SARS-CoV-2/humano/CHN/235/2020*	China
MT374104.1	*Síndrome respiratória aguda grave coronavírus 2 isolado SARS-CoV-2/humano/TWN/CGMH-CGU-08/2020*	Taiwan
MT415321.1	*Síndrome respiratória aguda grave coronavírus 2 isolado SARS-CoV-2/humano/IND/GMCKN318/2020*	India
MT762396.1	*Síndrome respiratória aguda grave coronavírus 2 isolado SARS-CoV-2/humano/BGD/BCSIR-NILMRC-254/2020*	Bangladesh
MT740381.1	*Síndrome respiratória aguda grave coronavírus 2 isolado SARS-CoV-2/humano/BGD/BCSIR-NILMRC-145/2020*	Bangladesh
MT428552.1	*Síndrome respiratória aguda grave coronavírus 2 isolado SARS-CoV-2/humano/KAZ/NCB-2/2020*	Cazaquistão
MT371047.1	*Síndrome respiratória aguda grave coronavírus 2 isolado SARS-CoV-2/humano/LKA/COV38/2020*	Sri Lanka
MT755883.1	*Síndrome respiratória aguda grave coronavírus 2 isolado SARS-CoV-2/human/SAU/477/2020*	Arábia Saudita
Oceania		
MT459985.1	*Síndrome respiratória aguda grave coronavírus 2 isolado SARS-CoV-2/humano/GUM/GU-NHG-01/2020*	Guam
Europa		
MT358638.1	*Síndrome respiratória aguda grave coronavírus 2 isolado SARS-CoV-2/humano/DEU/FFM1/2020*	Alemanha
MT328032.1	*Síndrome respiratória aguda grave coronavírus 2 isolado SARS-CoV-2/humano/GRC/10/2020*	Grécia
MT511066.1	*Síndrome respiratória aguda grave coronavírus 2 isolado SARS-CoV-2/humano/POL/PL-P10/2020*	Polônia
America		
MT350282.1	*Síndrome respiratória aguda grave coronavírus 2 isolado SARS-CoV-2/humano/BRA/SP02cc/2020*	Brasil
MT738101.1	*Síndrome respiratória aguda grave coronavírus 2 isolado SARS-CoV-2/humano/BRA/HIAE-SP03/2020*	Brasil
MT466071.1	*Síndrome respiratória aguda grave coronavírus 2 isolado SARS-CoV-2/humano/URY/Mdeo-1/2020*	Uruguai
MT470219.1	*Síndrome respiratória aguda grave coronavírus 2 isolado SARS-CoV-2/humano/COL/Cali-01/2020*	Colômbia
MT679205.1	*Síndrome respiratória aguda grave coronavírus 2 isolado SARS-CoV-2/humano/EUA/NYU-VC-017/2020*	EUA
MT750353.1	*Síndrome respiratória aguda grave coronavírus 2 isolado SARS-CoV-2/humano/EUA/CA-CZB-1829/2020*	EUA
Africa		
MT731285.1	*Síndrome respiratória aguda grave coronavírus 2 isolado SARS-CoV-2/humano/MAR/RMPS01/2020*	Marrocos
MT324062.1	*Síndrome respiratória aguda grave coronavírus 2 isolado SARS-CoV-2/humano/ZAF/R03006/2020*	África do Sul

Tabela 6.15: Resultados das comparações de sequência de SARS-CoV-2.

	Tempo (s)	**Tempo Total (s)**	**Potência (W)**	**Energia (J)**	**Energia Total (J)**
CPU-only	230	230	100,52	23119,60	23119,60
FPGA-only	287	287	10,69	3068,03	3068,03
CPU	35	244	99,86	3495,10	6103,46
FPGA	244		10,69	2608,36	

Tabela 6.16: Pontuações de sequências SARS-CoV-2 em comparação com a sequência de referência NC_045512.2 da China carregada no banco de dados público NCBI em 31-DEZ-2019 com data da última modificação 30-MAR-2020. Pontuações mais altas significam mais similaridade.

Accesso	**Escore LCS**	**País**	**Data de Atualização**
MT135044.1	29899	China	06-ABR-2020
MT415321.1	29899	Índia	30-ABR-2020
MT466071.1	29899	Uruguai	16-MAI-2020
MT324062.1	29897	África do Sul	13-ABR-2020
MT350282.1	29897	Brasil	17-ABR-2020
MT358638.1	29897	Alemanha	14-MAI-2020
MT428552.1	29897	Cazaquistão	05-MAI-2020
MT374104.1	29896	Taiwan	24-ABR-2020
MT470219.1	29896	Colômbia	14-MAI-2020
MT679205.1	29896	EUA	02-JUL-2020
MT731285.1	29896	Marrocos	08-JUL-2020
MT459985.1	29895	Guam	13-MAI-2020
MT328032.1	29894	Grécia	13-ABR-2020
MT371047.1	29893	Sri Lanka	23-ABR-2020
MT511066.1	29892	Polônia	26-MAI-2020
MT738101.1	29891	Brasil	09-JUL-2020
MT740381.1	29891	Bangladesh	10-JUL-2020
MT755883.1	29891	Arábia Saudita	14-JUL-2020
MT750353.1	29889	USA	14-JUL-2020
MT762396.1	29886	Bangladesh	15-JUL-2020

do NCBI. Na tabela, os resultados são ordenados da pontuação LCS mais alta para a mais baixa.

A primeira observação é que, como esperado, todas as sequências são muito semelhantes à sequência de referência. Considerando que todas as sequências possuem 29903 caracteres e que o valor atribuído para correspondências é 1, a pontuação máxima possível do LCS é 29903. Isso significa que a similaridade das sequências na Tabela 6.16 estão na faixa de 99,98%-99,94%.

Nesse cenário, nossa ferramenta LCS pode ser usada como filtro, ajudando a determinar quais sequências analisar com mais detalhes. Particularmente, os biólogos teriam como alvo as sequências que possuem a menor similaridade com a sequência de referência, em busca de possíveis mutações. Nosso estudo indica que as sequências MT750353.1 e MT762396.1 são de especial interesse, pois possuem a pontuação mais baixa e são bastante recentes (última atualização em 14 de julho e 15 de julho de 2020, respectivamente).

O estudo apresentado nesta seção foi reconhecido como relevante para a pesquisa em Covid-19 e o artigo onde o estudo foi publicado [41] foi incluído pela OMS na litera-

tura global sobre a doença do coronavírus (*https://search.bvsalud.org/global-literature-on-novel-coronavirus-2019-ncov/resource/en/covidwho-812770*).

6.4 Considerações Finais

Os resultados experimentais mostraram que a solução proposta em FPGA é capaz de consumir muito menos energia do que a solução em CPU. Ao comparar 20 pares de sequências de 50k, a energia usada foi 8572,74 Joules no FPGA, enquanto as mesmas comparações usaram 72019,44 Joules na CPU. Com isso, observe que a CPU consumiu 8,4x mais energia para executar a aplicação em comparação com o FPGA. Portanto, com a solução somente FPGA, o consumo de energia foi significativamente menor no FPGA. Nota-se também que o tempo de execução em FPGA aumentou cerca de 1,13x em relação ao CPU.

Também mostramos que nossa solução FPGA + CPU heterogênea é capaz de ter o melhor tempo de execução para os tamanhos de sequência considerados (10k, 20k e 50k), quando comparada às soluções somente FPGA e CPU. Se compararmos a solução FPGA + CPU com a solução somente FPGA, observamos uma redução de cerca de 4x no tempo de execução e um aumento de cerca de 2x no consumo de energia. Se a comparação for feita com a solução apenas de CPU, a diminuição no tempo de execução é de cerca de 1,13x e a redução no consumo de energia é de cerca de 4x.

Ainda há melhorias a serem feitas na implementação FPGA do algoritmo LCS tanto na parte do *host* quanto no *design* do HLS, como adicionar uma estratégia de transferência assíncrona entre o *host* e o FPGA. Ainda há espaço para otimizações no código HLS, principalmente em relação ao uso da *AXI-Stream Interface* para aumentar o *throughput* das portas de entrada/saída do LCS.

Capítulo 7

Projeto da Arquitetura de CPU+FPGA para Execução LCS na Nuvem

Neste capítulo, discutiremos o projeto da arquitetura para execução da solução proposta no Capítulo 6 na Nuvem AWS. Nessa abordagem foi necessário implementar o código em linguagem de programação C/C++, uma vez que ainda não havia suporte à linguagem HLS compatível com o sistema atual para instâncias de FPGA disponível na Nuvem AWS. Inicialmente, são fornecidas informações sobre a instância FPGA da Nuvem AWS (Seção 7.1). A seguir, será fornecida uma visão geral do projeto (Seção 7.2), que descreve a arquitetura proposta adaptada à plataforma de nuvem e na seção subsequente serão descritos os módulos que compõem essa arquitetura. Nas seções 7.4 e 7.5 serão discutidos o ambiente de hardware e as sequências utilizadas, respectivamente. A Seção 8.6 discute os resultados experimentais e, por fim, a seção 7.7 apresenta considerações finais.

7.1 Instância FPGA da Nuvem AWS

A nuvem AWS oferece máquinas virtuais (chamadas instâncias) com a plataforma FPGA como um serviço. Esse serviço permite que usuários façam uso dessa plataforma para programar os FPGAs para executar alguma tarefa ou utilizá-las como infraestrutura de outras aplicações em nuvem. Em outras palavras, na AWS os FPGAs são oferecidos como um serviço aos usuários finais para acelerar seus aplicativos e programas.

Através do *AWS Market Place* é possível obter as imagens de sistema e gerenciar as instâncias FPGA com os IPs já inclusos na imagem, o que permite uma maior facilidade em obter as configurações de cada placa FPGA sem necessidade de ter que buscar em ferramentas de terceiros pelas especificações e *datasheets* de hardware.

Em fevereiro de 2021, as instâncias da família F1 incluíam: (a) um a oito FPGAs Virtex Ultrascale+ VU19P, onde cada FPGA contém 8938000 elementos lógicos e até 64GB RAM, (b) um *host* CPU Intel Xeon E5-2686 v4 de 8 a 64 vCPUs. Na instância F1, sendo f1.2xlarge a utilizada na Tese, o módulo *Xilinx Vivado* é fornecido, bem como o AWS CloudShell, que permite a execução de programas e *scripts*.

A imagem da máquina Amazon ou *Amazon Machine Image* (AMI) é o servidor onde a instância é iniciada. A conexão na instância é feita através do SSH. Seguindo a filosofia *pay-per-use* da nuvem Amazon, é possível desligar e ligar esta instância quando necessário para que a instância somente esteja ativa quando for utilizada.

A imagem Amazon FPGA ou *Amazon FPGA Image* (AFI) é como o design completo é definido. Essa imagem é utilizada na instância F1. O AFI pode ser reutilizado quantas vezes for necessário.

7.2 Visão geral do Projeto

Conforme mencionado na Seção 7.1, na nuvem AWS, a máquina virtual que contém o FPGA possui também um *host CPU*, responsável pelo gerenciamento da execução no FPGA. Como o nosso projeto admite uma arquitetura híbrida (CPU-FPGA), a aplicação LCS será executada tanto no FPGA como no *host CPU*. Da mesma maneira que o exposto no Capítulo 6 e controle da execução é feito integralmente no *Host CPU*.

A arquitetura da solução proposta com a instância AWS F1 é composta pelos componentes ilustrados na Figura 7.1. São eles o *Host* CPU e o FPGA. A instância AWS F1 interage com os seguintes módulos AWS: *AWS Market Place*; *Amazon FPGA Image* (*AWS AFI*); *Amazon Machine Image*(*AWS AMI*); e o usuário. O módulo FPGA contem o(s) co-processadores(es) projetado(s) nessa Tese, bem como a DMA Engine e as BlockRAMS. Dentro do *host* CPU estão os módulos: *Host Memory*, CPU, *Xilinx Vivado* e APP. Os módulos CPU e Xilinx Vivado são, respectivamente, utilizados para compilação e geração dos programas para execução na CPU e no FPGA. Por fim o *APP* gerencia e executa o escalonador determinando para qual plataforma cada par de sequências será comparado com o algoritmo LCS.

A Figura 7.1 mostra a sequência de passos e os módulos que compõem a nossa solução. Para que seja instaurada uma instância AWS F1 são necessários alguns requisitos básicos para que ela funcione corretamente e seja inicializada com os recursos necessários para compilação e execução do código, tanto para a CPU quanto para o FPGA. Esses requisitos são a AWS AFI (Figura 7.1(b)) e AWS AMI (Figura 7.1(c)). A AWS AFI é uma imagem de sistema que implementa os *drivers IPs* compatíveis com a placa FPGA que será utilizada dentro da instância e a AWS AMI é uma imagem de sistema que ga-

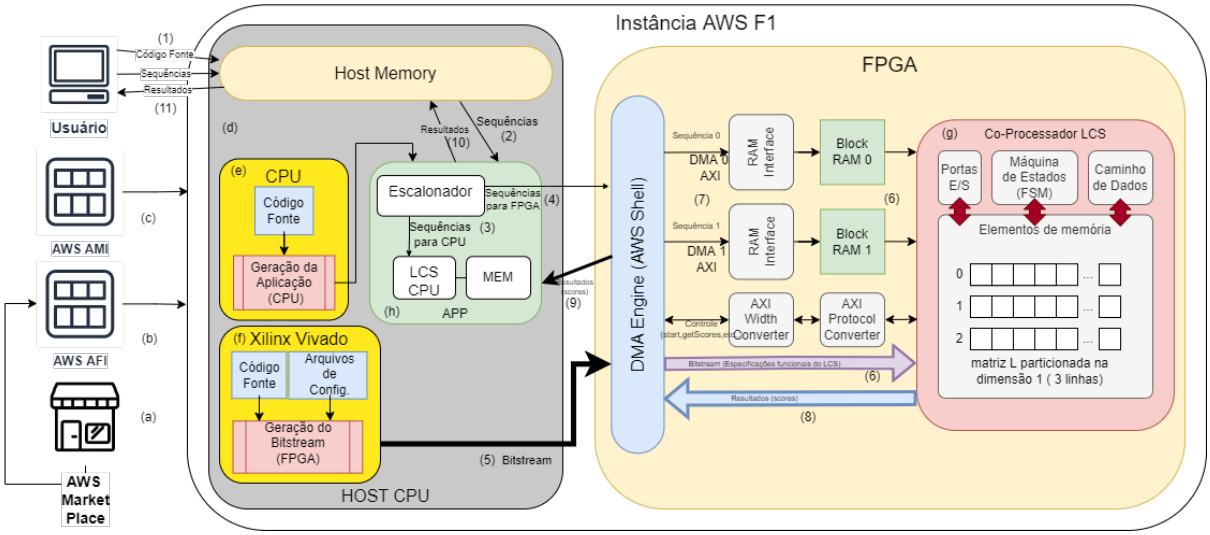

Figura 7.1: Visão geral da arquitetura proposta na AWS.

rante o funcionamento de um sistema operacional utilizando as vCPUs e que gerencia a comunicação com o FPGA.

Uma vez feito isso é necessário fazer a configuração da instância F1 com chaves de segurança que permitem a conexão segura à mesma através de SSH. Uma vez conectado à instância via SSH, é necessário que seja feita outra configuração, agora do ambiente de desenvolvimento para executar a ferramenta *Xilinx Vivado* (Figura 7.1(f)), que será utilizada para compilação do código fonte em *bitstream* que executará no FPGA. Em seguida, é feita a configuração do ambiente de desenvolvimento do código fonte em CPU (Figura 7.1(e)) que será executado no *Host CPU* (Figura 7.1(d)).

Uma vez esse passo pronto, o Usuário envia o código-fonte (Figura 7.1(1)) e as sequências a serem comparadas para o *Host memory* e executa-se a compilação do código fonte para ambas as plataformas (FPGA e CPU). É necessário que essa compilação seja feita dentro da instância, pois as imagens de sistema AMI e AFI são proprietárias e não permitem uso fora da instância, pois o acesso aos arquivos de configuração da placa FPGA da nuvem AWS, bem como os *drivers*, não estão disponíveis abertamente.

Após a compilação e geração do *bitstream* do programa a ser executado no FPGA (Figura 7.1(5)), faz-se o uso da própria ferramenta Xilinx Vivado para programar o FPGA, que enviará o *bitstream* gerado para o FPGA (Figura 7.1(6)), executando a programação do módulo Co-Processador LCS (Figura 7.1(g)) que ficará aguardando as sequências que serão enviadas pelo APP (Figura 7.1(h)) que será executado em CPU.

Dentro do *Host CPU* é executado o APP (Figura 7.1(h)). Esse módulo faz leitura das sequências (Figura 7.1(1)) armazenadas na *Host Memory* e utiliza o escalonador para selecionar quais sequências serão comparadas em CPU (Figura 7.1(3)) e quais serão comparadas em FPGA (Figura 7.1(4)). As sequências que serão comparadas em

FPGA, então são enviadas via DMA para o FPGA utilizando o barramento de PCIe. As sequências 0 e 1 recebidas pelo DMA do FPGA são armazenadas dentro das *Block RAMs* (Figura 7.1(7)) do FPGA através das suas interfaces de RAM (Figura 7.1(6)). A seguir, o Co-Processador LCS (Figura 7.1(g)) faz a leitura das sequências nas *Block RAMs* e executa a comparação.

Após cada comparação, o score obtido é enviado para a memória do *Host CPU* pela DMA do FPGA e a DMA entrega os resultados para o APP (Figura 7.1(7)). Após o término de todas as comparações, os resultados são colocados em um único arquivo e entregues ao usuário (Figura 7.1(11)).

7.3 Descrição dos Módulos

7.3.1 Módulo FPGA

Através do módulo APP (Figura 7.1(h)), as sequências são enviadas para o FPGA através do barramento DMA (Figura 7.1(5)) e armazenadas nas *Block RAMs* (Figura 7.1(6)). Uma vez que se tenha as duas sequências, um sinal de controle dentro do FPGA indica que já é possível iniciar a comparação.

O principal componente do módulo FPGA é o módulo Co-Processador LCS (Figura 7.1(g)) que segue a mesma lógica descrita na Seção 5.1. Apenas o que difere aqui é a linguagem utilizada para implementação. Como não havia suporte a HLS, foi feita uma implementação em C.

O trecho de código principal que especifica o LCS usando Vivado C é apresentado no Algoritmo 7.1. Inicialmente é definida uma matriz estática de três linhas $L[3][MAX\_N]$, onde serão armazenados os resultados intermediários a medida que a execução do algoritmo LCS progride. O tipo *u16* representa dados inteiros de 16-bits, sem sinal. A variável *bi* indica o índice do vetor (linha da matriz) que representa a diagonal a ser acessada e modificada. O tipo *u2* representa dados inteiros de 2-bits, sem sinal, pois a variável *u2* só pode indexar uma das 3 diagonais. O cálculo da matriz de programação dinâmica é feito por dois *loops* (linhas 7 e 13), que percorrem a matriz diagonal a diagonal.

A partir da linha 7 temos as iterações no cálculo da matriz de programação dinâmica por diagonal (indicada pela variável *row*) e armazenando os valores na matriz L particionada em três vetores das diagonais. Por fim, na linha 30, o resultado (escore máximo identificado entre as duas sequências de entrada) estará no último elemento da matriz L de programação dinâmica indicado pelas variáveis *bi* e *n*.

O FPGA utiliza o sinal de controle para iniciar a execução do algoritmo LCS com as duas sequências armazenadas nas *Block RAMs*. Quando finalizada a comparação das

Algoritmo 7.1: Trecho da especificação funcional do LCS usando Vivado.

```
#define MAX_N 50000
static u16 L[3][MAX_N + 1]; //matriz de apenas 3 linhas para otimizar uso de BRAM

void lcs(volatile uchar X[MAX_N],volatile uchar Y[MAX_N],int m,int n,int *res) {

  u2 bi = 0;
  for (int row=1; row <= ((m + 1) + (n + 1) - 1); row++) {
    int start_col = max(0, row - (m + 1));
    int count = min3(row, ((n + 1) - start_col), (m + 1));

    if (bi > 2) bi = 0; //alterna entre uma das 3 linhas de L

    for (int k = 0; k < count; k++) {
      int i = (min((m + 1), row) - k - 1);
      int j = (start_col + k);

      if (i == 0 || j == 0) { L[bi][j] = 0; }
      else if (X[i - 1] == Y[j - 1]) {
        if (bi == 0)       L[bi][j] = L[1][j - 1] + 1;
        else if (bi == 1)  L[bi][j] = L[2][j - 1] + 1;
        else               L[bi][j] = L[0][j - 1] + 1;
      } else {
        if (bi == 0)       L[bi][j] = max(L[2][j - 1], L[2][j]);
        else if (bi == 1)  L[bi][j] = max(L[0][j - 1], L[0][j]);
        else               L[bi][j] = max(L[1][j - 1], L[1][j]);
      }
    }
    if (start_col < m) bi ++; //incrementa se existirem diagonais a processar
  }
  *res = L[bi][n];
}
```

sequências, o sinal de controle é colocado em *idle*. O escore da comparação é então enviado pelo FPGA via DMA (Figura 7.1(7)) que entrega para o APP (Figura 7.1(h)).

7.3.2 Módulo APP

O módulo APP (Figura 7.1(h)) executa o código que foi compilado pela CPU (Figura 7.1(e)), que utilizará o escalonador descrito no pseudo-código 6.1 da Seção 6.1.1, fazendo a escolha de quais comparações serão feitas em CPU e quais serão em FPGA.

Após a separação das sequências pelo escalonador, as mesmas são enviadas para cada plataforma e dá-se início às comparações. Ao final de cada comparação, os resultados são armazenados em um arquivo de texto informando os resultados de cada comparação feita e isso ocorre tanto na CPU como no FPGA. O pseudo-código do escalonador executado no APP é o apresentado no algoritmo 6.1.

7.4 Ambiente de Hardware

O ambiente de hardware utilizado para os experimentos na AWS consiste em uma máquina dedicada com 8 vCPUs (8 *threads* executando em um Intel Xeon E5-2686 v4) e placa FPGA Xilinx Virtex UltraScale VU19P. Em CPU foi desenvolvida uma implementação em C e que utiliza 1 core. O circuito com 2 co-processadores gerado pela implementação

de nosso componente FPGA proposto tem uma freqüência de 250 MHz e utiliza os recursos da placa conforme mostrado na Tabela 7.1.

Tabela 7.1: Tabela de recursos usados pelo circuito no FPGA na instância da Amazon F1.

Recurso	Usado	Disponível	Usado(%)
LUTS	29259	4085760	0,72
Registers	45328	2148480	2,11
Blockram	225	2160	10,42
DSPs	71	3840	1,85
IOB	64	1036	6,18
IO	56	2072	2,56

Como pode ser visto, o circuito proposto ocupa relativamente poucos recursos na FPGA Xilinx Virtex UltraScale VU19P. Note que, apesar de permitir a comparação entre duas sequências de 10K, 20K ou 50K, projetou-se um circuito único e com 2 co-processadores, com capacidade máxima de comparação de 50K.

7.5 Geração de Sequências e Medidas

Para criar o conjunto de comparações, rodamos um *script* automatizado que gerou 40 sequências aleatórias de DNA de cada tamanho (10k, 20k, 50k), com um total de 20 comparações para cada tamanho de sequência.

A potência foi medida usando a ferramenta *Xilinx Vivado*, que utiliza uma métrica especulativa baseada nos recursos utilizados. A energia é a mesma para todos os testes de FPGA, pois um único circuito foi implementado com capacidade para armazenar sequências de até 50.000 caracteres cada, conforme explicado na Seção 7.4. Para a CPU, foi utilizada a ferramenta *powerstat* [25], que mede a energia elétrica do processador quando o algoritmo está rodando.

O tempo de execução foi medido através do *host* conforme mostrado no Algoritmo 6.1 (Linhas 7 e 10) então, a medição do tempo começa quando os *threads* de CPU e FPGA são criados e termina quando ambos os *threads* terminam a execução, ou seja, o tempo de execução é o tempo de execução de todas as 20 comparações. No caso do FPGA, esse tempo inclui o tempo de transferência dos dados para o FPGA até o retorno do escore ao *host*. Apesar de ser em nuvem AWS, a instância F1 permite apenas um usuário por vez, o que torna insignificante a variação nas execuções repetidas.

7.6 Resultados Experimentais

Para a CPU e FPGA, respectivamente, foram feitas duas implementações: (1) uma CPU e um co-processador no FPGA e (2) uma CPU e dois co-processadores no FPGA. Na Seção 7.6.1 serão apresentados os resultados para uma CPU e um co-Processador e na Seção 7.6.2 serão apresentados os resultados para uma CPU e dois co-processadores. Por fim, a Seção 7.7 apresenta as considerações finais.

7.6.1 Uma CPU e um co-processador no FPGA

Tabela 7.2: Configurações do escalonador para 1 co-processador no FPGA

1 CPU e 1 co-processador no FPGA	**Quantidade de Comparações**	
Configuração	**CPU**	**FPGA**
1CO-03C-17F	03	17
1CO-06C-14F	06	14
1CO-10C-10F	10	10
1CO-14C-06F	14	06
1CO-17C-03F	17	03

Para a execução com uma CPU e 1 co-processador no FPGA foram escolhidas 5 configurações, conforme mostrado na Tabela 7.2. A configuração 1CO-10C-10F distribui de maneira uniforme as sequências entre CPU e FPGA. As outras configurações executam mais comparações no FPGA (1CO-03C-17F, 1CO-06C-14F) ou na CPU (1CO-14C-06F, 1CO-17C-03F).

A Tabela 7.3 apresenta os resultados obtidos para a configuração 1CO-03C-17F , nas comparações de 10k, 20k e 50k. Para a nossa proposta CPU-FPGA, essa configuração prioriza o consumo de energia. Portanto, utilizando o FPGA para mais comparações, essa configuração favorece a redução do consumo de energia. No entanto, em FPGA, a comparação de 20k demorou 11,34 vezes mais do que a comparação em CPU (Tabela 7.3). Isso fez com que a energia consumida pelos dois dispositivos (CPU e FPGA) ficasse muito próxima. O mesmo não aconteceu com as comparações de 10k e 50k.

Na Tabela 7.4 temos os resultados de tempo e energia obtidos para a configuração 1CO-06C-14F. Se compararmos a comparação de sequencias de 50k com a configuração 1CO-03C-17F, nota-se que há uma redução de 18,03% no tempo total de execução, para um aumento de 36,55% no consumo de energia.

Na Tabela 7.5 temos a distribuição igual das comparações para CPU e FPGA, com 10 comparações para cada. Aqui, nota-se que houve uma aproximação do tempo de

Tabela 7.3: Tempo e energia obtidos para a configuração 1CO-03C-17F na nuvem AWS.

03 CPU/ 17 FPGA	Tamanho	Tempo(s)	Tempo Total(s)	Potência(W)	Energia(J)	Energia Total(J)
CPU	10k	3,96	29,80	101,12	400,44	743,14
FPGA		29,80		11,5	342,70	
CPU	20k	16,97	192,48	119,32	2024,86	4238,38
FPGA		192,48		11,5	2213,52	
CPU	50k	107,10	683,14	126,82	13582,42	21419,56
FPGA		683,14		11,5	7837,14	

Tabela 7.4: Tempo e energia obtidos para a configuração 1CO-06C-14F na nuvem AWS.

06 CPU/ 14 FPGA	Tamanho	Tempo(s)	Tempo Total(s)	Potência(W)	Energia(J)	Energia Total(J)
CPU	10k	8,56	22,46	108,24	926,53	1184,74
FPGA		22,46		11,5	240,10	
CPU	20k	34,12	89,83	111,64	3809,15	4841,09
FPGA		89,83		11,5	960,29	
CPU	50k	212,28	559,96	107,38	22794,62	29248,00
FPGA		559,96		11,5	6455,53	

execução para cada uma das plataformas, porém o consumo de energia é significativamente maior na CPU em relação à FPGA. Ao se comparar o tempo e energia total com a configuração 1CO-06C-14F, na comparação de 50k, nota-se uma redução de 28,09% no tempo de execução e um aumento de 45,62% na energia total.

Tabela 7.5: Tempo e energia obtidos para a configuração 1CO-10C-10F na nuvem AWS.

10 CPU/ 10 FPGA	Tamanho	Tempo(s)	Tempo Total(s)	Potência(W)	Energia(J)	Energia Total(J)
CPU	10k	14,26	16,04	108,24	1543,50	1727,96
FPGA		16,04		11,50	184,46	
CPU	20k	56,86	64,16	111,64	6347,85	7085,69
FPGA		64,16		11,50	737,84	
CPU	50k	353,80	402,68	107,38	37991,04	42592,08
FPGA		402,68		11,50	4601,04	

Na Tabela 7.6 temos a distribuição de 14 comparações sendo executadas em CPU e 06 comparações no FPGA. Essa configuração foi a que obteve resultados ruins, tanto em tempo de execução total como em energia total. Se compararmos seus resultados para com os da comparação de 50k da configuração 1CO-10C-10F, notamos um aumento considerável tanto no tempo de execução como no consumo de energia.

E por último, a Tabela 7.7 apresenta a distribuição de 17 comparações sendo executadas em CPU e 03 comparações no FPGA, em que o consumo de energia total é e o tempo de execução são superiores aos das outras configurações.

Tabela 7.6: Tempo e energia obtidos para a configuração 1CO-14C-06F na nuvem AWS.

14 CPU/ 6 FPGA	Tamanho	Tempo(s)	Tempo Total(s)	Potência(W)	Energia(J)	Energia Total(J)
CPU	10k	19,96	19,96	98.43	1964,66	2075,29
FPGA		09,62		11,50	110,63	
CPU	20k	79,60	79,60	99.38	7910,65	8353,40
FPGA		38,50		11,50	442,75	
CPU	50k	492,68	492,68	101.13	50091,71	52852,29
FPGA		240,05		11,50	2760,58	

Tabela 7.7: Tempo e energia obtidos para a configuração 1CO-17C-03F na nuvem AWS.

17 CPU/ 3 FPGA	Tamanho	Tempo(s)	Tempo Total(s)	Potência(W)	Energia(J)	Energia Total(J)
CPU	10k	25,02	25,02	102,38	2561,55	2616,86
FPGA		4,81		11,5	55,32	
CPU	20k	95,87	95,87	101,17	9779,09	9920,54
FPGA		19,25		11,5	221,38	
CPU	50k	603,10	603,10	109,95	13895,14	67588,82
FPGA		120,02		11,5	10165,54	

7.6.2 Uma CPU e dois co-processadores no FPGA

Para execução com 1 CPU e 2 co-processadores no FPGA, substituímos as configurações 1CO-03-17F e 1CO-17C-03F por 2CO-02C-18F e 2CO-18C-02F, respectivamente, para que houvesse número par atribuído aos 2 co-processadores em FPGA (Tabela 7.8). As demais configurações são as mesmas da Seção 7.6.1.

Tabela 7.8: Configurações do escalonador para 2 co-processadores no FPGA

1 CPU e 2 co-processadores no FPGA	Quantidade de Comparações	
Configuração	CPU	FPGA
2CO-02C-18F	02	18
2CO-06C-14F	06	14
2CO-10C-10F	10	10
2CO-14C-06F	14	06
2CO-18C-02F	18	02

A Tabela 7.9 apresenta os tempos e consumo de energia para a configuração 2CO-02C-18F. Pode-se notar que essa distribuição é a que prioriza o consumo de e o tempo de execução são superiores aos das outras configurações.

A Tabela 7.10 apresenta os tempos e consumo de energia para a configuração 2CO-06C-14F. Quando comparamos com a configuração 2CO-02C-18F para tamanho 50k, nota-se

Tabela 7.9: Tempo e energia obtidas para a configuração 2CO-02C-18F na nuvem AWS.

02 CPU/ 18 FPGA	Tamanho	Tempo(s)	Tempo Total(s)	Potência(W)	Energia(J)	Energia Total(J)
CPU	10k	2,56	14,58	99,73	255,31	590,65
FPGA		14,58		23	335,34	
CPU	20k	11,10	67,22	111,43	1236,87	2782,93
FPGA		67,22		23	1546,06	
CPU	50k	105,53	441,98	131,67	13895,14	24060,68
FPGA		441,98		23	10165,54	

Tabela 7.10: Tempo e energia obtidas para a configuração 2CO-06C-14F na nuvem AWS.

06 CPU/ 14 FPGA	Tamanho	Tempo(s)	Tempo Total(s)	Potência(W)	Energia(J)	Energia Total(J)
CPU	10k	7,86	11,87	101,78	799,99	1073,00
FPGA		11,87		23	273,01	
CPU	20k	36,00	53,1	116,7	4201,20	5422,50
FPGA		53,10		23	1221,30	
CPU	50k	351,14	415,89	130,7	45894,00	55459,47
FPGA		415,89		23	9565,47	

que o tempo de execução é 5,9% menor, porém o consumo de energia aumenta cerca de 2,3 vezes.

A Tabela 7.11 apresenta os tempos e consumo de energia para a configuração 2CO-10C-10F. Aqui, ao compararmos os resultados de tamanho 50k, nota-se que houve um aumento de 107,13% do tempo de execução na CPU em relação à FPGA e o consumo de energia no FPGA é de 91,90% menor que na CPU. Considerando o consumo de energia total (CPU-FPGA), há um aumento de 43% em relação à comparação de 50k da configuração 2CO-06C-14F.

A Tabela 7.12 apresenta os tempos e consumo de energia para a configuração 2CO-14C-06F. Nessa configuração, é possível notar que o consumo total de energia é 30,83% maior e o tempo de execução é 38,26% maior quando comparado à configuração 2CO-10C-10F para sequências de tamanho 50k.

Tabela 7.11: Tempo e energia obtidas para a configuração 2CO-10C-10F na nuvem AWS.

10 CPU/ 10 FPGA	Tamanho	Tempo(s)	Tempo Total(s)	Potência(W)	Energia(J)	Energia Total(J)
CPU	10k	13,90	13,90	102,06	1418,63	1623,10
FPGA		8,89		23	204,47	
CPU	20k	55,16	55,16	112,40	6199,98	7025,45
FPGA		35,89		23	825,47	
CPU	50k	536,39	536,39	137,04	73506,89	79462,97
FPGA		258,96		23	5956,08	

Tabela 7.12: Tempo e energia obtidas para a configuração 2CO-14C-06F na nuvem AWS.

14 CPU/ 06 FPGA	Tamanho	Tempo(s)	Tempo Total(s)	Potência(W)	Energia(J)	Energia Total(J)
CPU	10k	18,79	18,79	119,50	2245,41	2360,18
FPGA		4,99		23	114,77	
CPU	20k	77,96	77,96	130,40	10165,98	10608,73
FPGA		19,25		23	442,75	
CPU	50k	741,63	741,63	135,60	100565,03	103964,89
FPGA		149,20		23	3399,86	

Tabela 7.13: Tempo e energia obtidas para a configuração 2CO-18C-02F na nuvem AWS.

18 CPU/ 02 FPGA	Tamanho	Tempo(s)	Tempo Total(s)	Potência(W)	Energia(J)	Energia Total(J)
CPU	10k	23,19	23,19	100,4	2328,23	2374,00
FPGA		1,99		23	45,77	
CPU	20k	96,67	96,67	123,18	11907,81	12078,24
FPGA		7,41		23	170,43	
CPU	50k	954,93	954,93	131,50	125573,30	126708,81
FPGA		49,37		23	1135,51	

Por último, a Tabela 7.13 apresenta os tempos e consumo de energia para a configuração 2CO-18C-02F. Nessa configuração, a CPU leva cerca de 1,66 vezes o tempo de execução em relação à FPGA quando comparado à configuração 2CO-02C-18F, porém nota-se que houve um aumento significativo no consumo de energia na CPU , sendo que o consumo de energia do FPGA, com a mesma quantidade de comparações, em relação à CPU foi de 1,28% em relação à configuração 2CO-02C-18F para sequências de tamanho 50k.

7.6.3 Gráficos Comparativos por Tamanho

A Figura 7.2(a) mostra o comparativo de tempo e energia entre diversas configurações de comparações com sequências de tamanho 10k com 1 co-processador no FPGA. Nessa figura, nota-se que o melhor resultado em energia foi obtido pela FPGA-*only* e que o melhor resultado em tempo de execução foi obtido pela configuração 1CO-10C-10F. Se considerarmos tanto o tempo de execução como a energia, a configuração 1CO-03C-17F apresenta um bom compromisso entre esses dois fatores. Em relação somente à consumo de energia, há um aumento de 94,94% quando comparado à execução somente em FPGA, porém há uma redução de 74,84% quando comparado à execução somente em CPU. Pode-se notar também que o comportamento tanto na nuvem AWS quanto no ambiente *stand-alone* são similares.

A Figura 7.2(b) mostra o comparativo entre todas as execuções de comparações com sequências de tamanho 10k com 2 co-processadores no FPGA. Da mesma forma que na Figura 7.2(a), os melhores resultados em energia são obtidos pela FPGA-*only*. Diferentemente de 1 co-processador, os melhores resultados em tempo de execução são obtidos para 2CO-06C-14F. Para esse caso, uma configuração com bons resultados tanto para energia como para tempo de execução seria a 2CO-02C-18F. Em relação somente à consumo de energia, há um aumento de 49,48% quando comparado à execução somente em FPGA, porém há uma redução de 44,95% quando comparado à execução 2CO-06C-14F.

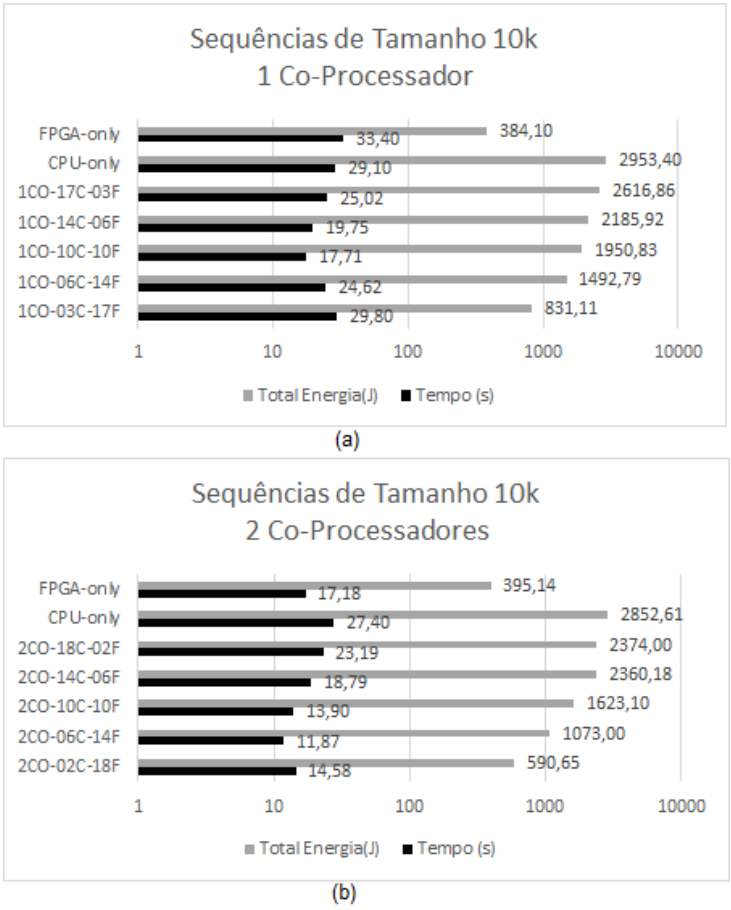

Figura 7.2: Gráfico comparativo em relação à sequências de tamanho 10k para cada execução do escalonador.

A Figura 7.3(a) mostra o comparativo entre diversas configurações de comparações com sequências de tamanho 20k com 1 co-processador no FPGA. Também nesse caso, o menor consumo de energia foi obtido pelo FPGA-*only*. Em relação ao tempo de execução, o menor tempo foi obtido pela configuração 1CO-10C-10F. Nota-se que a uma boa configuração é a 1CO-06C-14F em relação a tempo de execução e consumo de energia. Pode-se notar também que o comportamento tanto na nuvem AWS quanto no ambiente *stand-alone* são similares.

A Figura 7.3(b) mostra o comparativo entre diversas configurações de comparações com sequências de tamanho 20k com 2 co-processadores no FPGA. Para esse caso, o melhor consumo de energia foi obtido pelo FPGA-*only* e o melhor tempo de execução foi obtido pela configuração 2CO-06C-14F. Tanto a FPGA-*only* como a 2CO-02C-18F apresentaram bons compromissos entre o tempo de execução e o consumo de energia, nesse caso.

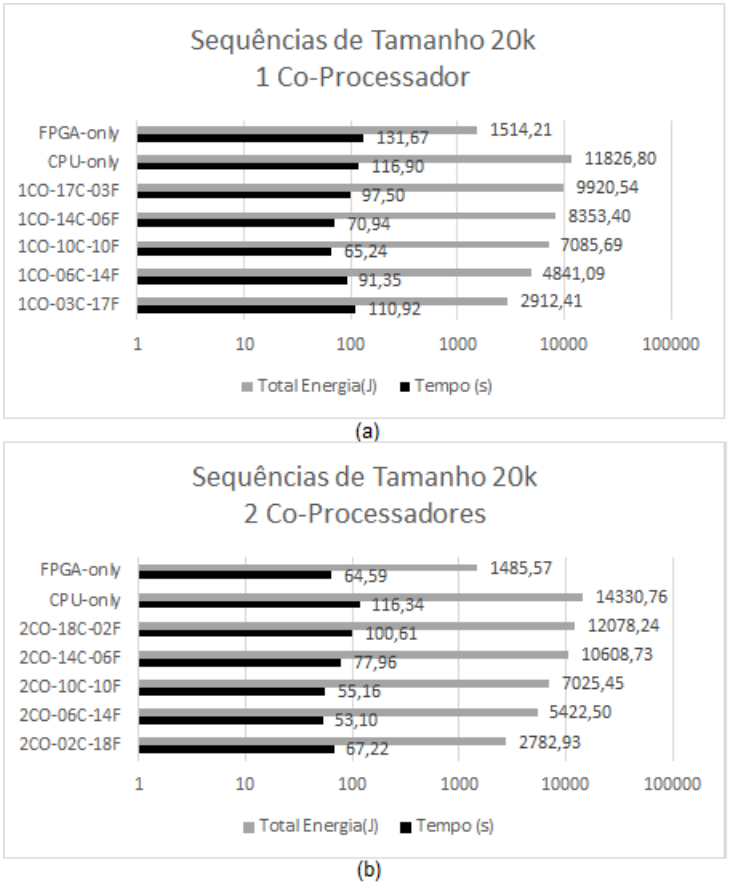

Figura 7.3: Gráfico comparativo em relação à sequências de tamanho 20k para cada execução do escalonador.

A Figura 7.4(a) mostra o comparativo entre diversas configurações de comparações com sequências de tamanho 50k com 1 co-processador no FPGA. Como nos casos anteriores, o melhor consumo de energia foi obtido pelo FPGA-*only*. O melhor tempo de execução foi obtido pelo 1CO-10C-10F. Se considerarmos consumo de energia e tempo de execução, tanto a configuração 1CO-06C-14F como a configuração 1CO-03C-17F apresentam bons resultados. Pode-se notar também que o comportamento tanto na nuvem AWS quanto no ambiente *stand-alone* são similares.

A Figura 7.4(b) mostra o comparativo entre diversas configurações de comparações com sequências de tamanho 50k com 2 co-processadores no FPGA. Ainda nesse caso, o

melhor consumo de energia foi obtido pela configuração FPGA-*only*. O melhor tempo de execução foi obtido pela configuração 2CO-06C-14F. Nota-se que tanto a FPGA-*only* como a configuração 2CO-02C-18F apresentam bons resultados tanto em tempo de execução como em consumo de energia.

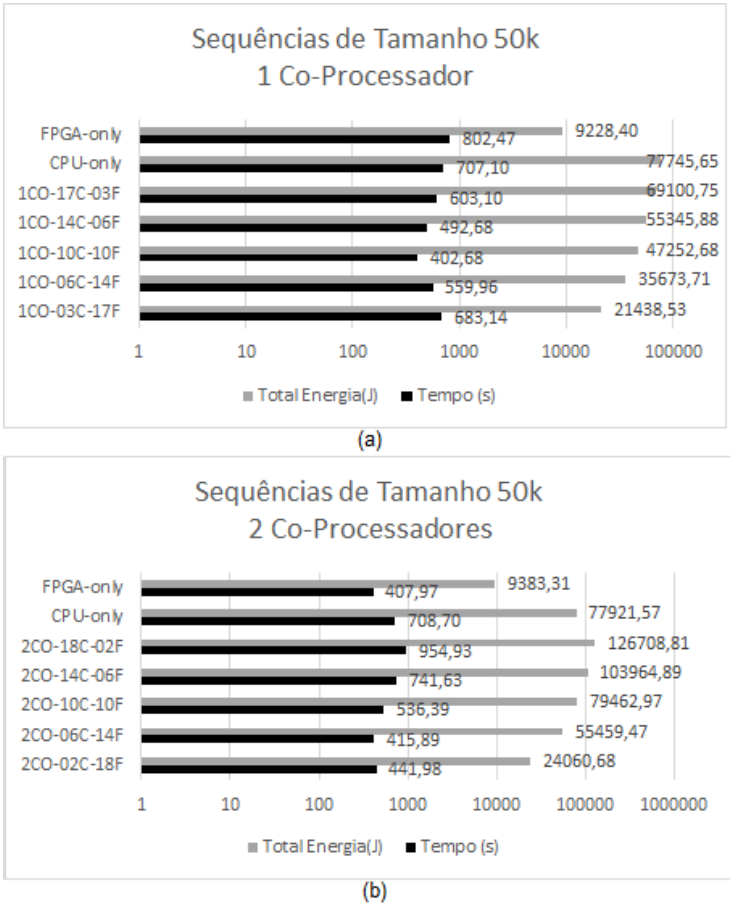

Figura 7.4: Gráfico comparativo em relação à sequências de tamanho 50k para cada execução do escalonador.

7.7 Considerações Finais

Os resultados experimentais mostraram que, como esperado, a solução proposta em FPGA é capaz de consumir muito menos energia do que a solução em CPU. Ao compararmos as configurações que atribuíam maiores comparações à FPGA, nota-se que o consumo de energia, em média, é cerca de 60% menor quando comparamos à solução em CPU-*only*. As configurações 1CO-06C-14F e 2CO-02C-18F para as sequências de tamanho de 50k se apresentam como uma boa solução considerando tanto o tempo de execução como consumo de energia total.

As configurações híbridas (CPU-FPGA) que atribuíram maiores quantidades de comparações para a CPU, se mostraram pouco eficazes em relação ao consumo de energia, quando comparados à FPGA-*only*, porém obtiveram valores menores de consumo de energia total quando comparados à CPU-*only*.

Entretanto, quando comparamos as configurações em que se atribuem as mesmas quantidades de comparações, como por exemplo as configurações 2CO-18C-02F e 2CO-02C-18F, em que são atribuídas 18 comparações executando em cada plataforma, é possível notar que o FPGA consome apenas 8,10% da energia consumida pela CPU para a mesma quantidade de comparações.

Com base nos resultados obtidos, mostramos que a nossa solução híbrida CPU-FPGA apresenta bons resultados se considerarmos somente o tempo de execução ou se considerarmos tanto o tempo de execução como o consumo de energia. Mostramos também que, dependendo da comparação e do número de co-processadores, a configuração que apresenta melhores resultados combinados muda. Isso mostra a importância de uma solução híbrida e flexível, onde o usuário pode definir a carga de trabalho relativa entre os dois tipos de dispositivos (CPU e FPGA).

Parte III

Conclusão

Capítulo 8

Conclusão e Trabalhos Futuros

8.1 Conclusão

A presente Tese propôs e avaliou uma solução do algoritmo LCS para comparação de sequências biológicas em HLS para FPGAs e uma estratégia híbrida CPU+FPGA que usa um escalonador ponderado, tanto para plataformas *stand-alone* como em nuvem.

Nossa solução FPGA (a) utilizou *BlockRAMs* para armazenar as sequências, que são acessadas através dos barramentos DMA da FPGA; (b) otimizou as iterações do cálculo da matriz de programação dinâmica; e (c) reduziu o custo do espaço de memória usando apenas 3 vetores para armazenar os valores com o cálculo na diagonal. Também propusemos uma estratégia CPU+FPGA que utiliza um escalonador *round-robin* ponderado para distribuir as comparações de sequências entre os dispositivos (CPU ou FPGA).

Os resultados mostraram que, conforme esperado, a solução proposta em FPGA é capaz de consumir muito menos energia do que a solução em CPU. Portanto, na maioria das execuções o consumo de energia foi significativamente menor no FPGA. Mostramos também que nossa solução híbrida CPU+FPGA é capaz de obter tempos de execução menores para os tamanhos de sequência considerados (10k, 20k e 50k), quando comparada às configurações FPGA-*only*.

Ao se comparar um lote de 20 pares de sequências de 50K no nosso ambiente *stand-alone*, verificamos que a configuração 2CO-6C-14F (2 co-processadores em FPGA, 6 comparações em CPU e 14 comparações em FPGA) obteve o menor tempo de execução. Isso mostra que o paralelismo entre CPU e FPGA contribui para o desempenho da aplicação. Conforme esperado, a configuração FPGA-*only* obteve menor consumo de energia, que foi mais do que 2 vezes menor do que o segundo melhor resultado.

Na comparação dos mesmos pares de sequências na nuvem AWS, a configuração FPGA-*only* com dois co-processadores executou-se no menor tempo e consumiu menos energia. Em termos de tempo de execução, o desempenho da FPGA-*only* ficou próximo

do desempenho da configuração 2CO-6C-14F porém, para esta configuração, o consumo de energia cresceu mais de 5 vezes. Isso mostra que o comportamento na nuvem é diferente do comportamento na plataforma *stand-alone*, pois uma maior disponibilidade de recursos, configurações e opções tanto da ferramenta quanto da placa, altera a maneira como a ferramenta de síntese calcula o caminho crítico do circuito, roteamento e mapeamento dos blocos lógicos e gera o arquivo *bitstream* diferente na nuvem AWS. Com isso, fica claro que oferecer alternativas de execução somente em CPU, somente em FPGA ou em plataforma híbrida CPU+FPGA é importante para o bom desempenho/baixo consumo de energia da nossa aplicação, tanto na execução *stand-alone* como na execução na nuvem.

8.2 Trabalhos Futuros

Como trabalhos futuros notamos que ainda há espaço para otimizações no código HLS, principalmente no que diz respeito ao uso de *AXI-Stream Interface* para aumentar o *throughput* das portas de entrada/saída do LCS e apontar e verificar os pragmas do HLS que possam ainda ser utilizados na solução.

Com a presente Tese, mostramos que o uso de uma solução híbrida CPU+FPGA pode melhorar tanto o tempo de execução quanto o consumo de energia. No entanto, precisamos entender melhor quais fatores dos ambientes e das sequências comparadas possuem maior impacto no tempo de execução e/ou consumo de energia, e visamos explorar esse aspecto em trabalho futuro. Com o entendimento desses fatores, será possível escolher a configuração mais adequada a determinado conjunto de comparações e a uma plataforma particular.

Também sugerimos realizar estudos complementares para determinar o impacto do consumo de recursos da placa, bem como do circuito gerado pela ferramenta, aumentando o tamanho das sequências a serem comparadas no FPGA. Os resultados preliminares indicam que é possível configurar mais de dois co-processadores LCS no FPGA, de forma que mais de dois pares de sequências de 50K possam ser processados em paralelo, quase sem impacto no consumo total de energia.

Ainda sugerimos, como um trabalho futuro de maior porte, uma modelagem matemática de otimização multiobjetivo com objetivos conflitantes (minimizar o tempo de execução e minimizar o consumo de energia). Com essa modelagem, poderemos determinar a fronteira de pareto considerando esses objetivos. Assim, seria possível construir uma *framework* baseada na modelagem matemática e em execuções passadas que escolheria a melhor distribuição caso a caso.

Como um outro trabalho futuro, sugerimos investigar a aplicação de reconfiguração parcial de forma dinâmica, que adaptaria o circuito do algoritmo de acordo com o tamanho das sequências comparadas ou ainda alocar mais co-processadores de acordo com a disponibilidade de recursos à medida que fosse necessário pela *framework*.

Por fim, deseja-se integrar outras estratégias ao escalonador, como roubo de trabalho e distribuição dinâmica de carga de trabalho, e compará-las ao nosso escalonador ponderado.

Referências

[1] High-level synthesis & verification platform | siemens digital industries software. https://eda.sw.siemens.com/en-US/ic/ic-design/high-level-synthesis-and-verification-platform/. (Accessed on 05/16/2021). 34

[2] Instâncias f1 do amazon ec2. https://aws.amazon.com/pt/ec2/instance-types/f1/. (Accessed on 03/04/2022). 4

[3] Apple previews mac os x snow leopard to developers. *Phys.org*, 2009. 27

[4] Khronos drives momentum of parallel computing standard with release of opencl 1.1 specification. *The Khronos Group*, 2010. 27

[5] Khronos releases opencl 1.2 specification. *The Khronos Group*, 2011. 27

[6] Opencl - the open standard for parallel programming of heterogeneous systems. *The Khronos Group*, Jul 2013. x, 28, 29

[7] Khronos releases opencl 2.1 and spir-v 1.0 specifications for heterogeneous parallel programming. *The Khronos Group*, 2015. 28

[8] Khronos finalizes opencl 2.0 specification for heterogeneous computing. *The Khronos Group*, 2018. 28

[9] Sdaccel environment user guide. *Xilinx*, 2018. x, 29, 30

[10] High-level synthesis user guide. *Xilinx*, 2020. x, 4, 32, 33, 34

[11] A. Abbott, P. Athanas, L. Chen, and R. Elliott. Finding lines and building pyramids with SPLASH 2. In *Proceedings of IEEE Workshop on FPGA's for Custom Computing Machines*. IEEE Comput. Soc. Press. 27

[12] S. Al Junid, A. Haron, Z. Abd Majid, F. Osman, H. Hashim, M. Idros, and M. Dohad. Optimization of dna sequences data for accelerate dna sequences alignment on fpga. *AMS2010: Asia Modelling Symposium 2010 - 4th International Conference on Mathematical Modelling and Computer Simulation*, pages 231 – 236, 06 2010. viii, 4, 35, 47

[13] M. Alser, H. Hassan, A. Kumar, O. Mutlu, and C. Alkan. Shouji: a fast and efficient pre-alignment filter for sequence alignment. *Bioinformatics*, mar 2019. viii, xi, 4, 40, 41, 47

[14] J. Arnold, D. Buell, D. Hoang, D. Pryor, N. Shirazi, and M. Thistle. The splash 2 processor and applications. In *Proceedings of 1993 IEEE International Conference on Computer Design ICCD'93*. IEEE Comput. Soc. Press. x, 26, 27

[15] J. M. Arnold. The splash 2 software environment. *The Journal of Supercomputing*, 9(3):277–290, sep 1995. 27

[16] A. Arslan. *Sequence Alignment*. Biyoinformatik-II, 2004. 1

[17] N. A.S.Alwan, I. K. Ibraheem, and S. M. Shukr. Fast computation of the shortest path problem through simultaneous forward and backward systolic dynamic programming. *International Journal of Computer Applications*, 54(1):21–26, sep 2012. 26

[18] H. Asgari, Y. S. Kavian, and A. Mahani. A systolic architecture for hopfield neural networks. *Procedia Technology*, 17:736–741, 2014. 26

[19] P. Baldi. *Bioinformatics : the machine learning approach*. MIT Press, Cambridge, Mass, 2001. 8

[20] A. Baxevanis. *Bioinformatics : a practical guide to the analysis of genes and proteins*. Wiley-Interscience, New York, 2001. 9

[21] V. Betz and J. Rose. Fpga routing architecture. In *Proceedings of the 1999 ACMSIGDA seventh international symposium on Field programmable gate arrays - FPGA*. ACM Press, 1999. x, 20, 21, 22

[22] T. Bonny, R. A. Debsi, and M. B. Almourad. Time efficient segmented technique for dynamic programming based algorithms with FPGA implementation. *Journal of Circuits, Systems and Computers*, 28(13):1950227, Feb. 2019. viii, xi, 4, 41, 42, 47, 48

[23] S. Brown, , S. Brown, and J. Rose. Architecture of fpgas and cplds: A tutorial. *IEEE Design and Test of Computers*, 13:42–57, 1996. 19

[24] K. Buhagiar, O. Casha, I. Grech, E. Gatt, and J. Micallef. Hardware implementation of efficient path reconstruction for the smith-waterman algorithm. In *2017 4th International Conference on Control, Decision and Information Technologies (CoDIT)*. IEEE, Apr. 2017. viii, x, 4, 37, 47

[25] Canonical. Ubuntu manpage: powerstat - a tool to measure power consumption. *Ubuntu*, 2015. http://manpages.ubuntu.com/manpages/xenial/man8/powerstat.8.html. 55, 60, 79

[26] A. Cinti, F. M. Bianchi, A. Martino, and A. Rizzi. A novel algorithm for online inexact string matching and its fpga implementation. *Cognitive Computation*, May 2018. viii, xi, 4, 38, 40, 47

[27] K. Compton and S. Hauck. Reconfigurable computing. *ACM Computing Surveys*, 34(2):171–210, June 2002. 18, 19

[28] P. Coussy. *High-level synthesis : from algorithm to digital circuit*. Springer, New York, 2008. 33

[29] G. D'Angelo and F. Palmieri. Discovering genomic patterns in SARS-CoV-2 variants. *International Journal of Intelligent Systems*, 35(11):1680–1698, July 2020. 3

[30] E. F. de O. Sandes and A. C. M. de Melo. Retrieving smith-waterman alignments with optimizations for megabase biological sequences using GPU. *IEEE Transactions on Parallel and Distributed Systems*, 24(5):1009–1021, may 2013. x, 17

[31] R. Durbin, S. R. Eddy, A. Krogh, and G. Mitchison. *Biological Sequence Analysis: Probabilistic Models of Proteins and Nucleic Acids*. Cambridge University Press, 1998. 1, 2, 8, 12

[32] J. Fang, Y. T. B. Mulder, J. Hidders, J. Lee, and H. P. Hofstee. In-memory database acceleration on FPGAs: a survey. *The VLDB Journal*, 29(1):33–59, Oct. 2019. 22

[33] U. Farooq, Z. Marrakchi, and H. Mehrez. *Tree-based Heterogeneous FPGA Architectures*. Springer New York, 2012. 23, 24

[34] X. Fei, Z. Dan, L. Lina, M. Xin, and Z. Chunlei. FPGASW: Accelerating large-scale smith–waterman sequence alignment application with backtracking on FPGA linear systolic array. *Interdisciplinary Sciences: Computational Life Sciences*, 10(1):176–188, Apr. 2017. viii, xi, 4, 38, 39, 47

[35] D. Fujiki, S. Wu, N. Ozog, K. Goliya, D. Blaauw, S. Narayanasamy, and R. Das. SeedEx: A genome sequencing accelerator for optimal alignments in subminimal space. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, Oct. 2020. viii, xi, 4, 45, 46, 47

[36] O. Gotoh. An improved algorithm for matching biological sequences. *Journal of Molecular Biology*, 162(3):705 – 708, 1982. 2, 14, 15

[37] Y.-R. Guo, Q.-D. Cao, Z.-S. Hong, Y.-Y. Tan, S.-D. Chen, H.-J. Jin, K.-S. Tan, D.-Y. Wang, and Y. Yan. The origin, transmission and clinical therapies on coronavirus disease 2019 (COVID-19) outbreak – an update on the status. *Military Medical Research*, 7(1), Mar. 2020. 70

[38] D. S. Hirschberg. A linear space algorithm for computing maximal common subsequences. *Commun. ACM*, 18(6):341–343, June 1975. 2, 15

[39] D. Hoang. Searching genetic databases on splash 2. In *[1993] Proceedings IEEE Workshop on FPGAs for Custom Computing Machines*. IEEE Comput. Soc. Press. 27

[40] C. A. C. Jorge, A. Nery, and A. C. M. A. Melo. Uma implementacao do algoritmo lcs em fpga usando high-level synthesis. *Simposio de Sistemas Computacionais de Alto Desempenho (WSCAD)*, pages 1–8, 10 2019. 5, 99

[41] C. A. C. Jorge, A. S. Nery, A. C. M. A. Melo, and A. Goldman. A CPU-FPGA heterogeneous approach for biological sequence comparison using high-level synthesis. *Concurrency and Computation: Practice and Experience*, 33(4), Oct. 2020. 5, 72, 99

[42] Kung. Why systolic architectures? *Computer*, 15(1):37–46, jan 1982. x, 25, 26

[43] H. Kung, C. Leiserson, C.-M. U. P. P. D. of COMPUTER SCIENCE., and C. M. U. C. S. Department. *Systolic Arrays for (VLSI)*. CMU-CS. Carnegie-Mellon University, Department of Computer Science, 1978. 25

[44] H. Kung and W. Lin. AN ALGEBRA FOR SYSTOLIC COMPUTATION. In *Elliptic Problem Solvers*, pages 141–160. Elsevier, 1984. 26

[45] I. Kuon, R. Tessier, and J. Rose. FPGA architecture: Survey and challenges. *Foundations and Trends® in Electronic Design Automation*, 2(2):135–253, 2007. 3, 18, 19

[46] G. Lemieux, E. Lee, M. Tom, and A. Yu. Directional and single-driver wires in FPGA interconnect. In *Proceedings. 2004 IEEE International Conference on Field-Programmable Technology (IEEE Cat. No.04EX921)*. IEEE, 2004. x, 22

[47] A. M. Lesk. *Introduction to Bioinformatics*. Oxford University Press, Oxford New York, 2008. 9

[48] A. Mahram and M. C. Herbordt. NCBI BLASTP on high-performance reconfigurable computing systems. *ACM Transactions on Reconfigurable Technology and Systems*, 7(4):1–20, Jan. 2015. 36

[49] R. Melham. A systolic accelerator for the iterative solution of sparse linear systems. *IEEE Transactions on Computers*, 38(11):1591–1595, 1989. 26

[50] T. Moeller and D. Martinez. Field programmable gate array based radar front-end digital signal processing. In *Seventh Annual IEEE Symposium on Field-Programmable Custom Computing Machines (Cat. No.PR00375)*. IEEE Comput. Soc. 26

[51] D. W. Mount. *Bioinformatics: Sequence and Genome Analysis*. Cold Spring Harbor Laboratory Press, 2001. 1, 8, 9, 10, 11

[52] A. Munshi, B. Gaster, T. G. Mattson, and D. Ginsburg. *OpenCL Programming Guide (OpenCL)*. Addison-Wesley Professional, 2011. x, 27, 29

[53] F. B. Muslim, L. Ma, M. Roozmeh, and L. Lavagno. Efficient FPGA implementation of OpenCL high-performance computing applications via high-level synthesis. *IEEE Access*, 5:2747–2762, 2017. 30

[54] E. W. Myers and W. Miller. Optimal alignments in linear space. *Bioinformatics*, 4(1):11–17, 1988. 2, 16

[55] W. Najjar, W. Bohm, B. Draper, J. Hammes, R. Rinker, M. Chawathe, and C. Ross. High-level language abstraction for reconfigurable computing. *Computer*, 36(8):63–69, Aug. 2003. 2

[56] J. G. Nash. High-throughput programmable systolic array FFT architecture and FPGA implementations. In *2014 International Conference on Computing, Networking and Communications (ICNC)*. IEEE, feb 2014. 26

[57] NCBI. National center for biotechnology information. *National Center for Biotechnology Information*, 2019. Available at https://www.ncbi.nlm.nih.gov/. 54

[58] S. B. Needleman and C. D. Wunsch. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *Journal of Molecular Biology*, 48(3):443–453, mar 1970. 2, 11, 14

[59] J. S. Parab, R. S. Gad, and G. Naik. *Hands-on Experience with Altera FPGA Development Boards*. Springer India, 2018. 3

[60] H. Parvez and H. Mehrez. *Application-Specific Mesh-based Heterogeneous FPGA Architectures*. Springer Publishing Company, Incorporated, 2014. x, 3, 19, 20, 23, 24

[61] W. R. Pearson. Flexible sequence similarity searching with the FASTA3 program package. In *Bioinformatics Methods and Protocols*, pages 185–219. Humana Press, 2000. 42

[62] C. Pilato and F. Ferrandi. Bambu: A modular framework for the high level synthesis of memory-intensive applications. In *2013 23rd International Conference on Field programmable Logic and Applications*. IEEE, Sept. 2013. 33

[63] D. Pryor, M. Thistle, and N. Shirazi. Text searching on splash 2. In *[1993] Proceedings IEEE Workshop on FPGAs for Custom Computing Machines*. IEEE Comput. Soc. Press. 27

[64] D. Rankin, J. Krupa, P. Harris, M. A. Flechas, B. Holzman, T. Klijnsma, K. Pedro, N. Tran, S. Hauck, S.-C. Hsu, M. Trahms, K. Lin, Y. Lou, T.-W. Ho, J. Duarte, and M. Liu. FPGAs-as-a-service toolkit (FaaST). In *2020 IEEE/ACM International Workshop on Heterogeneous High-performance Reconfigurable Computing (H2RC)*. IEEE, Nov. 2020. 4

[65] R. Salvador, A. Otero, J. Mora, E. de la Torre, T. Riesgo, and L. Sekanina. Self-reconfigurable evolvable hardware system for adaptive image processing. *IEEE Transactions on Computers*, 62(8):1481–1493, aug 2013. 26

[66] J. C. Setubal and J. Meidanis. *Introduction to Computational Molecular Biology*. PWS, 1997. 2

[67] R. Shikder, P. Thulasiraman, P. Irani, and P. Hu. An OpenMP-based tool for finding longest common subsequence in bioinformatics. *BMC Research Notes*, 12(1), Apr. 2019. 68, 69

[68] M. J. Smith. *Application Specific Integrated Circuits*. Addison-Wesley Professional, 1997. 3

[69] T. Smith and M. Waterman. Identification of common molecular subsequences. *Journal of Molecular Biology*, 147(1):195–197, 1981. 2, 13

[70] S. M. Trimberger. Three ages of fpgas: A retrospective on the first thirty years of fpga technology. *Proceedings of the IEEE*, 103(3):318–331, March 2015. 3, 19

[71] L. D. Tucci, R. Baghdadi, S. Amarasinghe, and M. D. Santambrogio. SALSA: A domain specific architecture for sequence alignment. In *2020 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, May 2020. viii, xi, 4, 42, 43, 44, 47

[72] B. S. C. Varma, K. Paul, and M. Balakrishnan. *Architecture Exploration of FPGA Based Accelerators for BioInformatics Applications (Springer Series in Advanced Microelectronics)*. Springer, 2016. 3, 20

[73] R. A. Wagner and M. J. Fischer. The string-to-string correction problem. *Journal of the ACM*, 21(1):168–173, jan 1974. 2, 3, 10, 16

[74] L. Wienbrandt. The FPGA-based high-performance computer RIVYERA for applications in bioinformatics. In *Language, Life, Limits*, pages 383–392. Springer International Publishing, 2014. viii, x, 4, 35, 36, 47

[75] Xilinx. Vivado high-level synthesis. *Xilinx*, 2016. Available at https://www.xilinx.com/products/design-tools/vivado/integration/esl-design.html. 3, 50

[76] Xilinx Corporation. Ultrafast high-level productivity design methodology guide, 2019. 4

[77] K. Yalcin, I. Cicekli, and G. Ercan. An external plagiarism detection system based on part-of-speech (POS) tag n-grams and word embedding. *Expert Systems with Applications*, 197:116677, July 2022. 3

Anexo I

Artigos Decorrentes desta Tese

I.1 Artigo completo publicado em conferência nacional

C. A. C. Jorge, A. Nery, and A. C. M. A. Melo. Uma implementacao do algoritmo lcs em fpga usando *high-level synthesis*. Simposio de Sistemas Computacionais de Alto Desempenho (WSCAD), pages 1–8, 10 2019. 5, 96 [40]

I.2 Artigo completo publicado em periódico internacional

C. A. C. Jorge, A. S. Nery, A. C. M. A. Melo, and A. Goldman. *A CPU-FPGA heterogeneous approach for biological sequence comparison using high-level synthesis. Concurrency and Computation: Practice and Experience*, 33(4), Oct. 2020. 5, 96 [41]

I.3 Primeira página dos artigos

(em ordem de publicação)

Uma implementação do algoritmo LCS em FPGA usando High-Level Synthesis

Carlos A. C. Jorge[1], **Alexandre S. Nery**[2], **Alba C. M. A. de Melo**[1]

[1]Universidade de Brasilia - Departamento de Ciência da Computação - Campus Darcy Ribeiro – CIC/EST

[2]Universidade de Brasilia - Departamento de Engenharia Elétrica - Campus Darcy Ribeiro – Faculdade de Tecnologia - ENE

Abstract. *This paper presents an implementation of the Longest Common Subsequence (LCS) algorithm for comparing two biological sequences using High Level Synthesis (HLS) for FPGAs. Results were obtained with a CPU Intel Core® i7-3770 CPU and a FPGA Xilinx® ADM-PCIE-KU3 that has a Xilinx Kintex® UltraScale XCKU060-2. Our experiments showed that the CPU implementation consumed 6.8x more energy to execute compared to FPGA.*

Resumo. *Este trabalho apresenta uma implementação do algoritmo* Longest Common Subsequence (LCS) *para comparação de duas sequências biológicas utilizando linguagem de alto nível* High Level Synthesis (HLS) *para FPGAs. Foram comparados resultados entre a execução em uma CPU Intel Core i7-3770 e uma FPGA Xilinx® ADM-PCIE-KU3 que possui uma Xilinx Kintex® UltraScale XCKU060-2. Os resultados mostraram que a implementação em CPU consumiu 6,8x mais energia em relação à FPGA.*

1. Introdução

A descoberta de padrões em sequências é um dos problemas mais desafiadores em biologia molecular e ciência da computação. Dado um conjunto de sequências, deve-se encontrar o padrão que ocorre com maior frequência. Em um processo de busca exata, a busca por um padrão de *m* letras pode ser resolvida por uma simples enumeração de todos os padrões de m letras que aparecem nas sequências. No entanto, quando se trabalha com sequências, realiza-se a busca aproximada pois os padrões incluem mutações, inserções ou remoções de nucleotídeos [Bucak and Uslan 2011].

O alinhamento de sequências expõe claramente os padrões que ocorrem com maior frequência, sendo útil para descobrir informação funcional, estrutural e evolucionária em sequências biológicas. Para tanto, é necessário descobrir o alinhamento ótimo, que maximiza a similaridade entre as sequências. Sequências muito parecidas (similares) provavelmente têm a mesma função e, se forem de organismos diferentes, são definidas como homólogas caso tenha existido uma sequência que seja ancestral de ambas [Mount 2001]. A similaridade de sequências pode ser um indício de várias possíveis relações de ancestralidade, inclusive a ausência de uma origem comum [Arslan 2004].

Na comparação de duas sequências biológicas calculam-se, a partir de métodos computacionais, métricas que ajudam a identificar o seu grau de relacionamento. Uma dessas métricas é o *escore*, que é atribuído a um alinhamento. Um alinhamento é definido como um pareamento, resíduo a resíduo, das sequências. Um par de resíduos das

Received: 2 March 2020 | Revised: 29 July 2020 | Accepted: 10 August 2020

DOI: 10.1002/cpe.6007

WILEY

SPECIAL ISSUE PAPER

A CPU-FPGA heterogeneous approach for biological sequence comparison using high-level synthesis

Carlos A. C. Jorge[1] | Alexandre S. Nery[2] | Alba C. M. A. Melo[1] | Alfredo Goldman[3]

[1]Department of Computer Science, University of Brasilia, Brasilia, Brazil

[2]Department of Electrical Engineering, University of Brasilia, Brasilia, Brazil

[3]Institute of Mathematics and Statistics, University of Sao Paulo, Sao Paulo, Brazil

Correspondence
Carlos A. C. Jorge, Department of Computer Science, Predio CIC-EST, Campus Asa Norte, University of Brasilia, CEP-70900-100, Brasilia, Brazil.
Email: cacjorge@aluno.unb.br

Funding information
CNPq 426729/2018-8; FAPDF 00193-00002139/2018-79; Capes/PROCAD 183794

Summary

This article presents a high-level synthesis implementation of the longest common subsequence (LCS) algorithm combined with a weighted-based scheduler for comparing biological sequences prioritizing energy consumption or execution time. The LCS algorithm has been thoroughly tailored using Vivado High-Level Synthesis tool, which is able to synthesize register transfer level (RTL) from high-level language descriptions, such as C/C++. Performance and energy consumption results were obtained with a CPU Intel Core i7-3770 CPU and an Alpha-Data ADM-PCIE-KU3 board that has a Xilinx Kintex UltraScale XCKU060 FPGA chip. We executed a batch of 20 comparisons of sequences on 10k, 20k, and 50k sizes. Our experiments showed that the energy consumption on the combined approach was significantly lower when compared to the CPU, achieving 75% energy reduction on 50k comparisons. We also used the tool proposed in this article to do a case study on Covid-19, with real SARS-CoV-2 sequences, comparing their LCS scores.

KEYWORDS

biological sequence comparison, dynamic programming, field-programmable gate array, high-level synthesis, longest common subsequence

1 | INTRODUCTION

Sequence pattern recognition is one of the most challenging problems in molecular biology and computer science. Given a set of sequences, one must find the pattern that occurs most frequently. In exact pattern matching, the search for a pattern of *m* letters can be solved by a simple enumeration of all the patterns of *m* letters that appear in the strings. However, when working with biological sequences, the approximate pattern matching is usually performed because the patterns include nucleotide mutations, insertions or removals.[1] The focus of this article is on approximate pattern matching.

A sequence is considered as an ordered set of residues, that is, characters, and the sequence alignments clearly expose the patterns that occur most frequently, being useful to discover functional, structural, and evolutionary information in biological sequences. Therefore, it is important to find the optimal alignment, which maximizes the similarity between the sequences. Very similar sequences probably have the same function and, if the sequences are from different organisms, they may be defined as homologous, if there is a sequence which is ancestral to both.[2] Sequence similarity can be an indication of several possible ancestral relationships, including the absence of a common origin.[3]

When comparing two biological sequences, metrics are calculated using computational methods to help identify their degree of relationship. One of these metrics is the *score*, which is assigned to an alignment. An alignment is defined as a pairing, residue by residue, of the sequences. A pair of residues (characters) from the two sequences can be defined as *match*, when the characters are equal; *mismatch*, when the characters are distinct; or a *gap*, when the residue of a sequence is aligned with a gap.[2] Among the various biological sequence comparison algorithms in the literature, the *longest common subsequence (LCS)*[4] algorithm is one of the most used. Other algorithms that are also used for sequence comparison are Smith-Waterman,[5] Needleman-Wunsh,[6] and Hirschberg.[7]