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

Comparação paralela de múltiplos pares de sequências biológicas em GPU e CPU com work stealing

Thiago Costa Marques
Willian Picinato da Silva

Monografia apresentada como requisito parcial
para conclusão do Curso de Computação — Licenciatura

Orientadora
Prof.ª Dr.ª Alba Cristina Magalhães Alves de Melo

Brasília
2017

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

Comparação paralela de múltiplos pares de sequências biológicas em GPU e CPU com work stealing

Thiago Costa Marques
Willian Picinato da Silva

Monografia apresentada como requisito parcial
para conclusão do Curso de Computação — Licenciatura

Prof.ª Dr.ª Alba Cristina Magalhães Alves de Melo (Orientadora)
CIC/UnB

Prof. Dr. George Luiz Medeiros Teodoro
CIC/UnB

Prof. Dr. Ricardo Pezzuol Jacobi
CIC/UnB

Prof. Dr. Pedro Antonio Dourado Rezende
Coordenador do Curso de Computação — Licenciatura

Brasília, 03 de julho de 2017

Resumo

Bioinformática é uma área da ciência que utiliza abordagens computacionais para responder questões biológicas e a comparação de sequências é a base para análises biológicas na bioinformática. Devido ao grande tempo gasto na comparação de sequências biológicas e ao crescimento das bases genômicas, é preciso desenvolver soluções que utilizem mecanismos paralelos de execução, de modo a reduzir o tempo gasto para processamento. Dentre as soluções paralelas, destaca-se a ferramenta MASA-OpenCL, que permite que comparações de sequências sejam feitas em CPU ou GPU, mas não simultaneamente.

O objetivo do presente trabalho de graduação é propor, implementar e avaliar uma estratégia híbrida capaz de executar o MASA-OpenCL em CPU e GPU paralelamente, visando acelerar o processo de comparação de sequências biológicas. Dentre as características do trabalho, estão a habilidade de executar diferentes comparações ao mesmo tempo, melhor distribuição de tarefas entre a CPU e GPU, adequando ao poder computacional de cada uma. Além disso, a técnica *work stealing* utilizada no projeto permitiu a redução do tempo gasto para o processamento, pois distribuiu melhor as comparações entre CPU e GPU.

Os resultados obtidos com a utilização do ambiente misto (CPU e GPU) e oito pares de sequências mostraram que há uma redução considerável no tempo gasto para o processamento das comparações de sequências biológicas, chegando a até 41%.

Palavras-chave: bioinformática, computação em GPU, escalonamento de comparações biológicas

Abstract

Bioinformatics is an area of science that uses computational approaches to answer biological questions and comparison of sequences is the basis for biological analyses in bioinformatics. Due to the long time spent comparing biological sequences and exponential growth of genomic bases, it is necessary to develop solutions using parallel execution mechanisms, in order to reduce the time spent for processing. Among the parallel solutions, we highlight MASA-OpenCL, which allows sequence comparisons to be made in CPU or GPU, but not simultaneously.

This project aims to propose, implement and evaluate a hybrid strategy capable of executing MASA-OpenCL in CPU and GPU in parallel, in order to accelerate the process of comparison of biological sequences. The main goals of this projetct are: ability to perform different comparisons at the same time, tasks better distributed between the CPU and GPU, adapting to the computational power of each one. In addition, the work stealing technique used in the project allowed time spent for the processing to be reduced, because comparisons are better distributed between the CPU and GPU.

The results obtained using the mixed environment (CPU and GPU) and eight pairs of sequences showed that there is a considerable reduction in time spent for the processing of biological sequence comparisons, reaching up to 41 %.

Keywords: bioinformatics, GPU computing, scheduling of biological comparisons

Sumário

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

A bioinformática é uma área da ciência que utiliza abordagens computacionais para responder questões biológicas [1], considerando, dentre outros, aspectos genômicos evolutivos e funcionais. O avanço dos estudos sobre genomas produziram (e continuam produzindo) quantidades cada vez maiores de dados, daí a necessidade de se utilizar ferramentas computacionais para o gerenciamento e análise destas informações.

Um dos primeiros grandes projetos de bioinformática foi realizado por Margaret Dayhoff e sua equipe, em 1965, ao desenvolver um banco de dados de sequências de proteínas chamado *Atlas of Protein Sequence and Structure* [9]. Desde então, um número muito grande de sequências biológicas tem sido colocado em bancos públicos de sequências, estando disponíveis para análises de pesquisadores. O crescimento desses bancos genômicos apresenta atualmente uma taxa exponencial, dando origem a um fenômeno conhecido como dilúvio de dados (*data deluge*) [15].

A comparação de sequências é uma das base para análises biológicas na Bioinformática. Os algoritmos de Smith-Waterman [16] e Needleman-Wunsh [11] comparam duas sequências biológicas em tempo e espaço quadrático, obtendo o resultado ótimo. Devido ao crescimento das bases genômicas, existe uma quantidade enorme de sequências que podem ser comparadas entre si e a execução de tais algoritmos requer muito tempo. Para acelerar essas comparações é necessário desenvolver soluções que utilizem mecanismos paralelos de execução, de modo a permitir a otimização do desempenho e, consequentemente, reduzir do tempo gasto para processamento. Algoritmos como Smith–Waterman e Needleman–Wunsch foram modificados para serem executados em ambientes de processamento paralelo.

A arquitetura MASA (*Multi-platform Architecture for Sequence Aligners*) foi projetada visando oferecer um *framework* flexível e customizável para a obtenção do alinhamento de sequências biológicas com Smith-Watterman de forma simplificada, independente da plataforma. O MASA-OpenCL é uma das opções fornecidas pelo MASA, entretanto sua

execução está limitada apenas a uma arquitetura durante a execução: GPU ou CPU. Essa limitação gera uma subutilização do poder computacional disponível, pois ao trabalhar apenas com a GPU ou CPU o MASA-OpenCL não utiliza toda capacidade de processamento que tem a disposição.

Por isso, o presente trabalho propôs, implementou e avaliou uma estratégia híbrida capaz de executar o MASA-OpenCL em CPU e GPU paralelamente, visando acelerar o processo de comparação de sequências biológicas. Dentre suas características estão a habilidade de executar diferentes comparações ao mesmo tempo e melhor distribuição de tarefas entre a CPU e GPU, adequando ao poder computacional de cada uma. Além disso, a técnica de *work stealing* [3], utilizada no projeto, permitiu a redução do tempo gasto para o processamento, pois evitou que a GPU permanecesse ociosa durante a execução. Assim, ambas estão sempre trabalhando, com exceção de quando não há mais comparações a serem feitas dentro da capacidade predefinida para cada uma.

O restante desse documento está organizado como se segue. No Capítulo 2 descreve algoritmos básicos para a comparação de sequências biológicas. O Capítulo 3 discute acerca da solução proposta neste trabalho, apresentando detalhamentos sobre sua implementação. O Capítulo 4 aponta os resultados alcançados durante os testes e, finalmente, o Capítulo 5 apresenta as conclusões do trabalho e propõe trabalhos futuros.

Capítulo 2

Comparação de sequências biológicas

2.1 Alinhamento e escore

O alinhamento de sequências biológicas é o resultado da comparação entre duas ou mais sequências de DNA, RNA ou proteínas para identificar regiões de similaridade entre as sequências. O alinhamento permite, por exemplo, mostrar genes em comum que dividem uma história evolucionária entre os organismos [1].

Sequências biológicas são cadeias de caracteres pertencentes aos alfabetos Σ = {A, T, G, C} (DNA), Σ = {A, U, G , C} (RNA) e Σ = {A, C, D, E, F, G, H, I, K, L, M, N, P, Q, R, S, T, V, W, Y} (proteínas) [9]. Caracteres idênticos ou similares, chamados de *match*, são colocados em uma mesma coluna e os não-idênticos podem ser colocados na mesma coluna como *mismatch* ou ainda pode ser inserido um espaço na coluna em uma dessas sequências, conhecido como *gap* (“ - ”) [9].

Existem dois tipos básicos de alinhamento: global e local. O alinhamento global percorre toda a extensão das duas sequências, ou seja, todos os caracteres participam do alinhamento. Sequências que são similares ou com tamanhos aproximadamente iguais são possíveis candidatas a esse tipo de alinhamento. No caso local, o alinhamento para no fim de regiões de igualdade ou forte similaridade, e é dada uma prioridade muito maior para encontrar estas regiões locais do que para estender o alinhamento para incluir mais pares de aminoácidos vizinhos [9].

A similaridade entre as sequências é obtida atribuindo-se para *match*, *mismatch* e *gap* valores numéricos, chamados de escores. Esses valores podem variar, mas é comum optar-se por escores positivos para os casos de *match*, e negativos para as situações de *mismatch* e *gaps* [5]. O sistema de escores atribui um escore final no alinhamento entre sequências, sendo esse valor o somatório dos escores aferidos em cada coluna do alinhamento.

No exemplo abaixo, usaremos os valores -2 para *gap*, -1 para *mismatch* e +1 para *match*. Nesta situação, o escore final do alinhamento é +1.

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

2.2 Algoritmos de comparação de sequências

Nessa Seção serão apresentados os algoritmos de comparação de sequências de Needleman-Wunsh (NW), Smith-Waterman (SW), Gotoh, Hirschberg e de Myers e Miller (MM).

2.2.1 Needleman-Wunsh (NW)

O algoritmo Needleman-Wunsch (NW) [11] tem por função calcular o alinhamento global ótimo entre duas sequências de entrada, permitindo a utilização de *gaps* com o objetivo de melhorar o alinhamento final obtido.

Esse algoritmo funciona da seguinte forma. É criada uma matriz de similaridade entre as duas sequências, de dimensão $(m+1).(n+1)$, matriz essa que será denominada matriz H, sendo m e n respectivamente os tamanhos das sequências de entrada s_0 e s_1. Na posição inicial da matriz será introduzido o valor 0, por tanto a posição $H_{0,0} = 0$. As outras posições da primeira linha e da primeira coluna, linhas $H_{i,0}$ e colunas $H_{0,j}$ serão preenchidas da seguinte forma: $H_{i,0}$ = -i * G e $H_{0,j}$ = -j * G , sendo G a penalização por *gap* [5].

Para obter o escore de outras posições da matriz serão levadas em consideração três possibilidades, mas sempre escolhendo o que retornar um maior escore para tal posição. São elas:

1. O escore obtido pela diagonal: Quando o valor vem da diagonal, é usado o valor do *match/mismatch* somado com o valor da diagonal para calcular o escore da posição.

2. O escore obtido da esquerda: quando o valor vem da esquerda, é utilizada a penalidade de *gap* somada ao valor da esquerda para obter o escore da posição.

3. O escore obtido da posição superior: quando o valor vem da posição superior é utilizada a penalidade de *gap* somado ao valor da posição de cima para obter o escore da posição.

$$H_{i,j} = \max \begin{cases} H_{i-1,j-1} + \mathrm{p(i,j)} \\ H_{i,j-1} - \mathrm{G} \\ H_{i-1,j} - \mathrm{G} \end{cases} \tag{2.1}$$

Além do valor do escore máximo, cada posição da matriz possui uma referência à posição que gerou esse escore. Assim é possível percorrer a matriz da posição inferior direita até o seu início (*traceback*) recuperando-se o alinhamento global ótimo.

	-	A	A	C	G	T	T	A	C
-	0	-1	-2	-3	-4	-5	-6	-7	-8
C	-1	-1	-2	-1	-2	-3	-4	-5	-4
G	-2	-2	-2	-2	0	-1	-2	-3	-4
A	-3	-1	-1	-2	-1	-1	-2	-1	-2
T	-4	-2	-2	-2	-2	0	0	-1	-2
A	-5	-3	-1	-2	-3	-1	-1	+1	0
A	-6	-4	-2	-2	-3	-2	-2	0	0
C	-7	-5	-3	-1	-2	-3	-3	-1	+1

Figura 2.1: Alinhamento global com NW.

A Figura 2.1 apresenta a matriz NW resultante da comparação entre as sequências $s_0 = CGATAAC$ e $s_1 = AACGTTAC$. Neste exemplo, os *gaps* possuem escore -1, *matches* recebem escore +1 e *mismatches* recebem escore -1.

2.2.2 Smith-Waterman (SW)

O algoritmo Smith-Waterman (SW) [16] também foi criado com o objetivo de obter um alinhamento entre sequências, mas nesse caso o local. Funciona de uma forma parecida com o algoritmo Needleman-Wunsch (Seção 2.2.1) com algumas modificações, pois possui um objetivo diferente.

A primeira modificação é que esse algoritmo não permite a ocorrência de números negativos na matriz H, então na equação de recorrência é utilizado o valor zero para impedir números negativos. Isso acontece porque se um alinhamento for negativo, é mais vantajoso iniciar um novo alinhamento local naquela posição. Portanto, diferente do que acontece no algoritmo Needleman-Wunsch, a primeira linha e a primeira coluna não serão preenchidas com números negativos e sim com o valor zero [5].

$$H_{i,j} = \max \begin{cases} H_{i-1,j-1} + \mathrm{p(i,j)} \\ H_{i-1,j} - \mathrm{G} \\ H_{i,j-1} - \mathrm{G} \\ 0 \end{cases} \tag{2.2}$$

A segunda diferença entre os algoritmos acontece em relação ao *traceback*, que não começa necessariamente a partir da célula na posição inferior direita da matriz, mas sim

partindo da célula que possui o maior escore até encontrar uma posição com valor zero [16]. A Equação 2.2 ilustra o algoritimo SW.

2.2.3 Gotoh

Tanto no algoritmo de Needleman-Wunsch (Seção 2.2.1) quanto no de Smith-Waterman (Seção 2.2.2), a penalidade para a abertura ou extensão de uma sequência de *gaps* é a mesma. Gotoh [7] faz uma modificação na equação usada pelo algoritmo NW, substituindo a constante G (penalidade por *gap*) por uma função genérica, a qual retorna a penalidade de um *gap* com comprimento k, onde k é o comprimento de uma sequência consecutiva de *gaps*. Isso faz com que a complexidade seja $O(n^3)$. A equação original de Gotoh é ilustrada na Equação 2.3 [7].

$$H_{i,j} = \max \begin{cases} H_{i-1,j-1} + p(i,j) & \\ H_{i-k,j} - \gamma(k) & k = 1, ..., i-1 \\ H_{i,j-k} - \gamma(k) & k = 1, ..., j-1 \end{cases} \tag{2.3}$$

Para reduzir a complexidade de tempo para $O(n^2)$, Gotoh utilizou o modelo conhecido como *Affine gap*, onde a abertura de novos *gaps* é mais penalizada do que a extensão de *gaps* já existentes, através da equação y(k) = - G_{first} – (k-1) * G_{ext}, onde k é o número de *gaps* consecutivos. Para implementação, são criadas duas novas matrizes, E e F, que servem para avaliar cada abertura de um novo *gap* ou a extensão em uma das sequências, conforme as Equações 2.4, 2.5 e 2.6 [5].

$$H_{i,j} = \max \begin{cases} 0 \\ E_{i,j} \\ F_{i,j} \\ H_{i-1,j-1} + p(i,j) \end{cases} \tag{2.4}$$

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

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

2.2.4 Hirschberg

O algoritmo proposto por Hirschberg [8] é uma versão do Needleman-Wunsh (Seção 2.2.1), mas que soluciona o problema da Maior Subsequência Comum (*LCS - Longest Common Subsequence*) em espaço linear. Neste solução, ao invés de armazenar a matriz inteira,

conserva-se apenas três linhas da matriz. Numa comparação de sequências pequenas, esse método pode parecer irrelevante, mas para matrizes com centenas de milhões de linhas e colunas, o espaço disponível para armazenar toda a matriz torna-se fator importante ao gerar um alinhamento ótimo.

Partindo do alinhamento global, a ideia é encontrar o ponto médio pelo qual passa um alinhamento ótimo. Para encontrar este ponto, a computação é feita em duas partes. Considerando a matriz H de tamanho (m, n) formada por duas sequências de caracteres s_0 e s_1, inicialmente o processamento da matriz é feito por linhas até a linha central $\frac{m}{2}$. Na segunda parte, o processamento é feito sobre as duas sequências s_0 e s_1 invertidas, até que a mesma linha central $\frac{m}{2}$ seja calculada [8].

A operação é realizada através do cálculo do custo máximo de conversão de vetores que são obtidos percorrendo a matriz nos dois sentidos. Neste estágio, o procedimento é chamado a partir deste ponto recursivamente para as duas matrizes resultantes, até a obtenção de problemas triviais [5].

2.2.5 Myers e Miller (MM)

O algoritmo de Myers e Miller (MM) [10] baseia-se na combinação do algoritmo de *affine gap* utilizado por Gotoh (Seção 2.2.3) com o algoritmo de complexidade linear de espaço proposto por Hirschberg (Seção 2.2.4). Esta abordagem baseia-se na aplicação de sucessivas divisões na matriz de similaridade, determinando-se em cada iteração um ponto médio para a subsequência mais longa (linha ou coluna) [5]. São armazenados quatro vetores durante o processamento:

- CC - Armazena o escore dos alinhamentos que terminam em *match* ou *mismatch*.
- DD - Armazena o escore dos alinhamentos que terminam em *gap*.
- CC' - Análogo ao vetor CC, armazena o processamento das sequências invertidas.
- DD' - Análogo ao vetor DD, armazena o processamento das sequências invertidas.

O escore $K_i = min\{CC_i + CC'_i, DD_i + DD'_i - G_{open}\}$ é atribuído ao alinhamento que atravessa a coluna $\frac{n}{2}$ através da linha i, onde G_{open} é a penalidade de abertura de *gap* ($G_{open} = G_{first} - G_{ext}$). Dessa forma, o escore K_i* é definido como $K_i* = max\{K_i\}$; $\forall\ 0\ \leq i < m$. De forma recursiva, realiza-se o processamento novamente dividindo o tamanho da seção em tamanhos menores [10]. A figura 2.2 ilustra o processamento feito pelo algoritmo de Hirschberg.

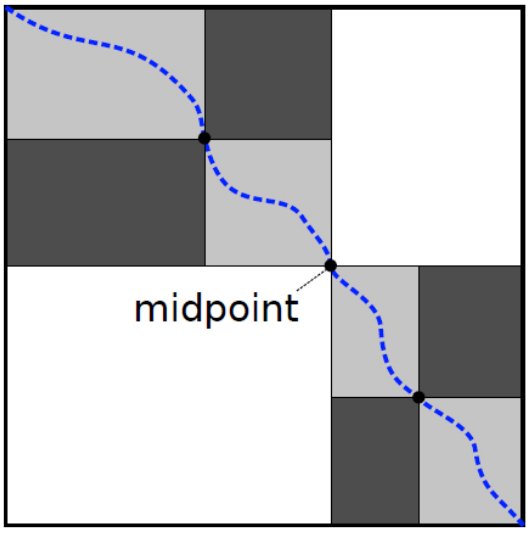

Figura 2.2: Processamento recursivo do algoritmo de Myers e Miller [4]

2.3 Paralelização dos algoritmos

Como as sequências biológicas podem ter milhões de caracteres, a comparação entre elas é um processo que consome muitos recursos computacionais e talvez leve dias, semanas ou meses para encontrar o resultado esperado. Por isso, diversas ferramentas foram propostas a fim de aprimorar essa tarefa, como a utilização de hardwares de alto desempenho e otimizações sofisticadas dos algoritmos.

Nos algoritmos de comparação vistos anteriormente, a maior parte do tempo de processamento é gasto no cálculo da matriz de similaridades. Por isso, é recomendável utilizar estratégias que acelerem este processo, sendo a paralelização uma das mais empregadas.

2.3.1 *Wavefront*

Na matriz H formada pelas sequências s_0 e s_1, o escore da célula H (i,j) é obtido pela posição diagonal, esquerda ou superior. Desse modo, percebe-se que células que estão em uma mesma antidiagonal não possuem dependência entre si e podem ser processadas ao mesmo tempo.

Em um primeiro momento, somente a célula superior esquerda pode ser processada. Depois, as duas células da antidiagonal seguinte podem ser processadas as mesmo tempo, iniciando, então o paralelismo no cálculo matriz. Em seguida, células da próxima antidiagonal são processadas em paralelo e assim sucessivamente. O máximo paralelismo é obtido quando a antidiagonal possui o maior tamanho (antidiagonal principal) e, a partir desse momento, o paralelismo decresce até chegar na última célula do canto inferior direito.

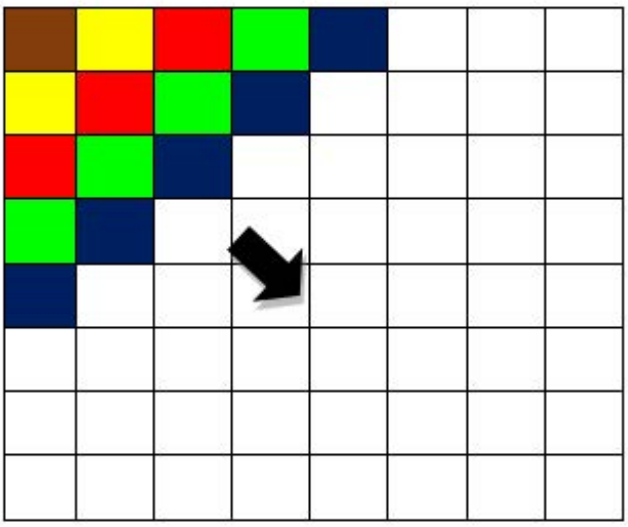

Figura 2.3: Processamento em *wavefront* diagonal na matriz H [5]

2.4 CUDAlign

O CUDAling é uma ferramenta para comparação de sequências de DNA muito longas em GPU, que atualmente está na versão 4.0 [13]. A primeira versão do CUDAlign foi proposta em [14] visando a obtenção do escore local ótimo para sequências longas de DNA em GPU utilizando o algoritmo Smith-Waterman com affine gap e com uso linear de memória. O processamento em GPU do CUDAlign explora o paralelismo externo e interno em todas as suas versões [14].

2.4.1 Paralelismo externo

Baseado na estratégia *wavefront*, o paralelismo externo agrupa a matriz H em diagonais externas de blocos ao invés de dividi -la em células. Cada bloco é processado em paralelo na GPU e, assim como no modelo *wavefront*, apenas o primeiro bloco será processado no início da execução. A segunda diagonal externa será composta de dois blocos que serão processados em paralelo assim que todas as suas dependências forem calculadas. A figura 2.4 exemplifica esta estratégia.

Uma diagonal externa só pode ser processada quando a anterior terminou seu processamento. Como é fácil perceber, é necessário que algumas linhas sejam passadas de um bloco superior para um inferior para que os cálculos sejam realizados e, para isso, utiliza-se uma estrutura denominada barramento horizontal (cujo tamanho é o próprio tamanho das linhas). Outra estrutura necessária é denominada barramento vertical, que

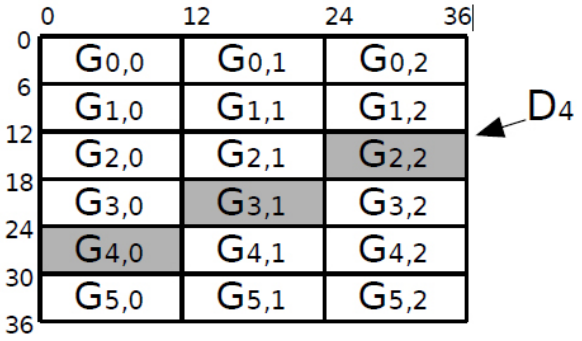

Figura 2.4: Divisão da matriz em blocos. A diagonal externa D4 está com blocos em destaque, observando-se que o número de colunas de blocos é B = 3 [13]

serve para passar os valores das últimas vizinhanças de células do bloco da esquerda para o da direita [2].

A Figura 2.5 ilustra o barramento horizontal, que passa a última linha do bloco superior para o bloco inferior, e barramento vertical, que passa os valores das últimas vizinhanças de células do bloco da esquerda para o bloco da direita [2].

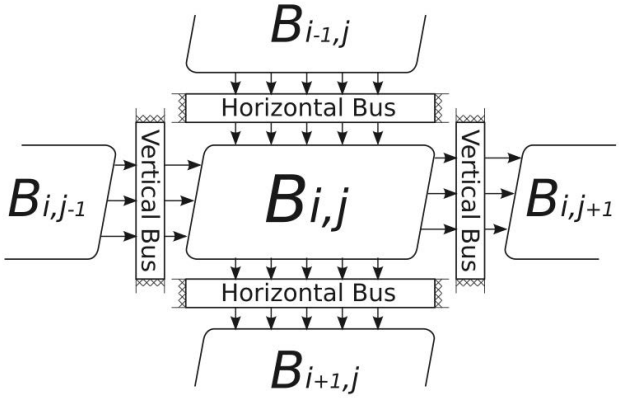

Figura 2.5: Barramento horizontal e vertical [13]

2.4.2 Paralelismo interno

Um outro nível de paralelismo é explorado no interior dos blocos da matriz. De maneira similar ao externo, o paralelismo interno segue o padrão *wavefront* e T *threads* cooperam para processar as células que pertencem ao bloco B. Para permitir a correta execução do wavefront, todas as *threads* de um bloco devem calcular em paralelo a mesma diagonal interna [13]. A Figura 2.6 ilustra esta estratégia.

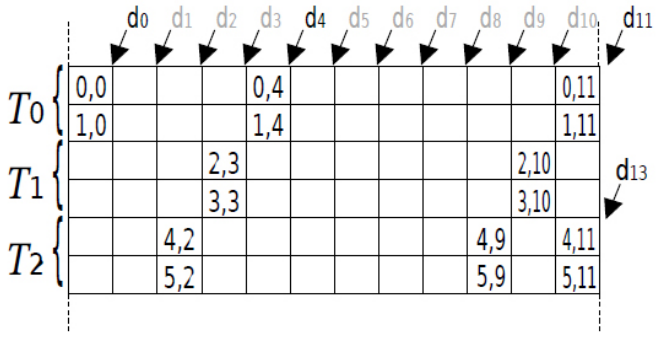

Figura 2.6: Paralelismo interno [13]

2.4.3 Otimização

Esta seção irá apresentar técnicas utilizadas para melhorar o desempenho do CUDAlign 1.0, que foram mantidas em todas as versões desta ferramenta.

Na técnica de *wavefront*, aplicada em ambos os tipos de paralelismo, o processamento se inicia partindo das diagonais internas, tendo o grau de paralelismo de uma diagonal diretamente proporcional com o número de células que ela possui.

Para manter o paralelismo no máximo pelo maior tempo possível, o CUDAlign 1.0 criou uma otimização chamada de delegação de células. Em vez de processar todas as células de um bloco, ele processa todas as células até a ultima diagonal interna com paralelismo máximo. No entanto num bloco com RxC células, C diagonais serão processadas, deixando as ultimas células por conta do próximo bloco [12]. A Figura 2.7 ilustra a delegação de células.

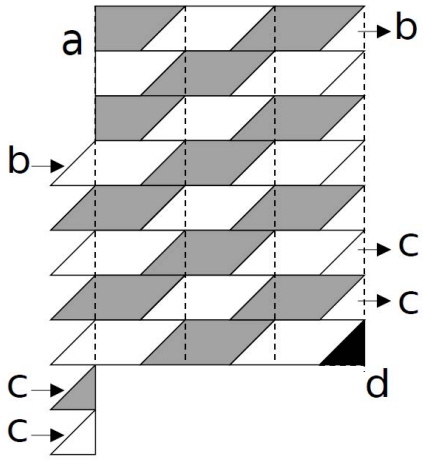

Figura 2.7: Representação da delegação de células do CUDAlign 1.0 [12]

No processo de delegação de células, os blocos da mesma diagonal externa não podem possuir dependência de dados entre si. Isso ocorre porque o escalonador CUDA executa os blocos em qualquer ordem, mas por existir uma área do bloco que foi delegada para

a diagonal externa seguinte, é criada uma dependência entre dois blocos da próxima diagonal.

Para eliminar esta dependência entre blocos da mesma diagonal externa, deve-se calcular todas as células pendentes antes delas serem lidas e os blocos devem ser sincronizados para garantir a correta leitura dos valores das células. Com isso, o processamento de cada diagonal externa foi dividido em duas fases: fase curta e fase longa [12].

- Fase Curta: O objetivo dessa fase é terminar de processar todas as células pendentes. Para isso, processam-se as T - 1 primeiras diagonais internas de todos os blocos e, em seguida, o controle é retornado à CPU para forçar o sincronismo entre os blocos.
- Fase Longa: Essa fase termina de processar as $\frac{n}{b}$ – (T-1) diagonais internas restantes desse bloco. Essa fase foi chamada de longa pois usualmente o número de colunas C de cada bloco é muito maior que o número de *threads* T que esse bloco possui. Sendo assim, o tempo de execução da fase longa será muito maior do que o da fase curta.

2.5 MASA

Como o CUDAlign é uma versão do algoritmo Smith-Waterman (SW) (Seção 2.2.2) desenvolvida apenas para a arquitetura *Compute Unified Device Architecture* (CUDA) da NVIDIA, observou-se que algumas de suas otimizações poderiam ser empregadas em outras plataformas de hardware, como GPUs de outros fabricantes, CPUs, *Field Programmable Gate Array* (FPGAs), e plataformas de software, tais como *Open Computing Language* (OpenCL), *Open Multi-Processing* (OpenMP) e OmpSs.

Com isso, verificou-se que a arquitetura do CUDAlign poderia ser modificada para permitir o desacoplamento entre o código independente da plataforma e o código específico, sendo esta arquitetura chamada de *Multi-platform Architecture for Sequence Aligners* (MASA) [13].

O principal objetivo do MASA é fornecer um *framework* flexível e customizável de modo que o desenvolvimento de ferramentas similares ao CUDAlign seja simplificado, permitindo, assim, o alinhamento de sequências com métodos exatos em outras plataformas.

A arquitetura MASA possui um conjunto de módulos e otimizações que pode ser reutilizado em diferentes ambientes. Além disso, devido a sua característica modular, como mostra a Figura 2.8, esta arquitetura permite que novas otimizações possam ser aplicadas de uma só vez para diversas plataformas de hardware e software [13].

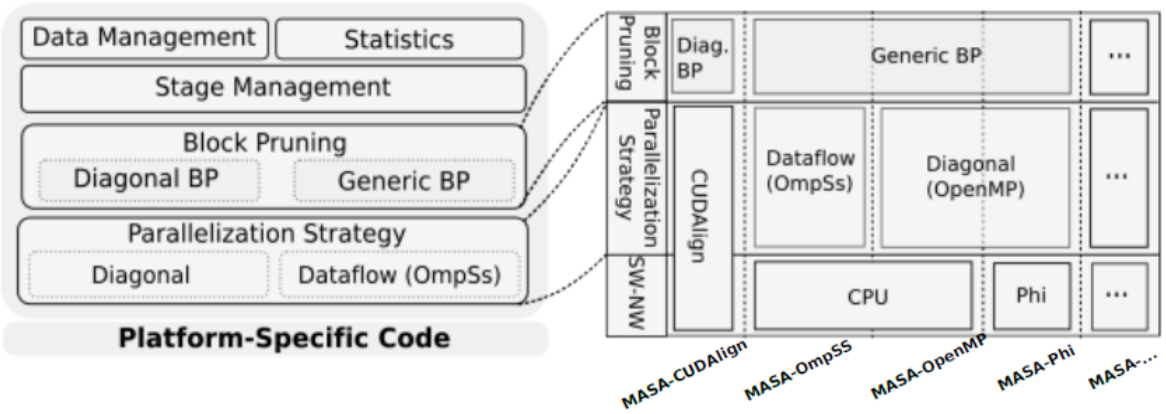

Figura 2.8: Modelagem da arquitetura MASA [5]

A arquitetura do MASA foi implementada utilizando a linguagem de programação C++, baseada no paradigma de orientação a objetos, e projetada em cinco módulos diferentes, como mostra a Figura 2.9, responsáveis por executar a equação de recorrência de SW ou NW (Seção 2.2.1), gerenciar a entrada e saída de dados, prover informações sobre o tempo de execução gasto em cada estágio, dentre outros.

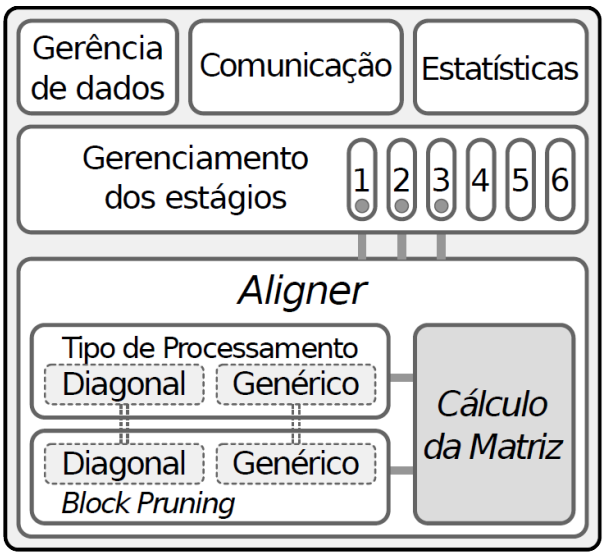

Figura 2.9: Arquitetura MASA [5]

Quatro extensões foram criadas para o MASA afim de avaliar sua capacidade de portabilidade em diferentes plataformas de *hardware* e ambientes de software: MASA-CUDAlign, MASA-OmpSs-CPU, MASAOpenMP-CPU e MASA-OpenMP-Phi. Por meio da arquitetura MASA, pretende-se criar uma infraestrutura comum para alinhamento de sequências, permitindo agregar várias implementações em plataformas distintas para acelerar ainda mais o desempenho de comparações de sequências longas.

O principal benefício dessa arquitetura é que a grande maioria das funcionalidades foram criadas de maneira portável a todas as extensões, permitindo que essas funcionalidades sejam aplicadas a diversas plataformas com pouco ou nenhum esforço adicional e, assim, acelerando o desenvolvimento de novos projetos.

De forma paralela ao desenvolvimento destas extensões, uma outra foi elaborada pelo mesmo grupo de trabalho, chamada de MASA-OpenCL e que será tratada a seguir.

2.5.1 MASA-OpenCL

O MASA-OpenCL é uma ferramenta, inicialmente proposta em [5], com o objetivo de realizar comparações de sequências biológicas longas de DNA em ambiente heterogêneo, isto é, permitindo a execução em GPUs de diversos fabricantes ou em outras arquiteturas, incluindo CPU, necessitando apenas de poucas modificações no seu código-fonte.

A solução foi projetada como uma extensão da arquitetura MASA, utilizando para implementação o *framework* OpenCL, este escolhido para que fosse possível testar e avaliar a portabilidade do código em diferentes arquiteturas. O foco do MASA-OpenCL é o cálculo do escore ótimo entre duas sequências longas de DNA, provendo também suas coordenadas na matriz de programação dinâmica.[5]

Desenvolver uma aplicação a partir do MASA mostrou-se mais apropriado, uma vez que a utilização de uma arquitetura de classes preexistente na arquitetura simplifica o processo, e, com isso, ainda é possível manter o desempenho apresentado pelo MASA-CUDAlign durante a comparação de sequências.

No MASA-OpenCL foi feita uma nova implementação da estratégia de paralelização e do algoritmo de Smith-Waterman (Seção 2.2.2), visando facilitar a portabilidade do código para outras arquiteturas. Comparando as Figuras 2.8 e 2.10, é possível perceber como a extensão foi adicionada à arquitetura.

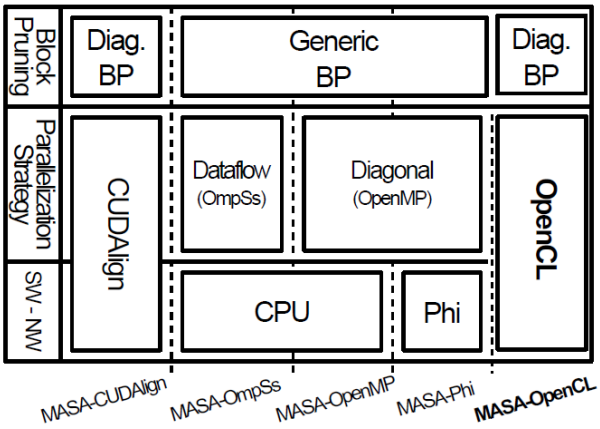

Figura 2.10: Arquitetura MASA já com a extensão OpenCL [5]

A solução arquitetada possui boa flexibilidade, permitindo que sejam escolhidos parâmetros (como plataforma, quantidade de blocos ou quantidade de *threads*) a serem informados na linha de comando de execução do programa, que alteram a forma de processamento da matriz de programação dinâmica. O projeto possui a capacidade de oferecer um código único para ser executado em CPUs e GPUs, descomplicando a tarefa de processar a comparação das sequências em diferentes tipo de ambientes e fabricantes.

Capítulo 3

Projeto da estratégia de execução CPU-GPU

Na solução MASA-OpenCL de comparação de sequências biológicas (Seção 2.5.1) é utilizada apenas a GPU ou a CPU da máquina para realizar comparações de sequências biológicas. Logo, há um desperdício de poder computacional com a não utilização da CPU em conjunto com a GPU. Por essa razão, propomos nesse capítulo um novo projeto de execução que é capaz de utilizar tanto a CPU quanto a GPU.

Inicialmente, a GPU, por ter um poder de processamento muito maior quando comparada com a CPU, foi a única escolhida para realizar o processamento das comparações entre as sequências biológicas. Isso aconteceu porque muitas das sequências e, consequentemente a comparação entre elas, possuem um tamanho muito grande. Logo, o processamento pela CPU demandaria muito tempo, tempo esse que consegue ser reduzido de uma maneira drástica pela GPU.

No entanto, foi constatado que nem todas as sequências são grandes e que durante a execução do processo de comparação entre elas pela GPU, a CPU fica ociosa, representando um poder computacional subutilizado. Então surgiu a ideia de utilizar a CPU para realizar comparações entre as sequências menores, enquanto a GPU trata das maiores.

Junto com essa ideia também surgiram dúvidas quanto à utilidade do esquema proposto. Por exemplo, por diversos fatores a CPU poderia atrasar o processo de comparação e a comparação com a CPU e GPU poderia demorar um tempo maior do que a comparação somente com GPU. Além disso, caso a solução CPU + GPU viesse a agilizar o processo, talvez houvesse um limite para isso, que poderia ser determinado da seguinte forma: o tempo que a GPU leva para fazer comparações de tamanho X, somado do tempo para fazer comparações de tamanho Y, contra o tempo que a GPU leva para fazer apenas as comparações de tamanho X e o tempo que a CPU leva para fazer as comparações de tamanho Y. No presente projeto, CPU e GPU trabalham de forma simultânea, portanto,

se o tempo que a CPU leva para fazer comparações de tamanho Y for menor que o tempo que a GPU leva para fazer as comparações de tamanho X somado as comparações de tamanho Y, a CPU se torna útil, agilizando o processo de comparação das sequências, caso contrário, a utilização da CPU não acelera o processo.

3.1 Visão geral da arquitetura

A Figura 3.1 apresenta o projeto da solução proposta. O usuário indica em um arquivo de texto quais comparações deverão ser executadas e, então, inicia a execução do programa. A *thread* principal é responsável por identificar as comparações, dividi-las, de acordo com seus tamanhos, em duas filas ordenadas e também pela criação e manipulação das *threads* filhas, *ThreadCPU* e *ThreadGPU*. Estas *threads* são encarregadas do tratamento das *FilaCPU* e *FilaGPU*, respectivamente.

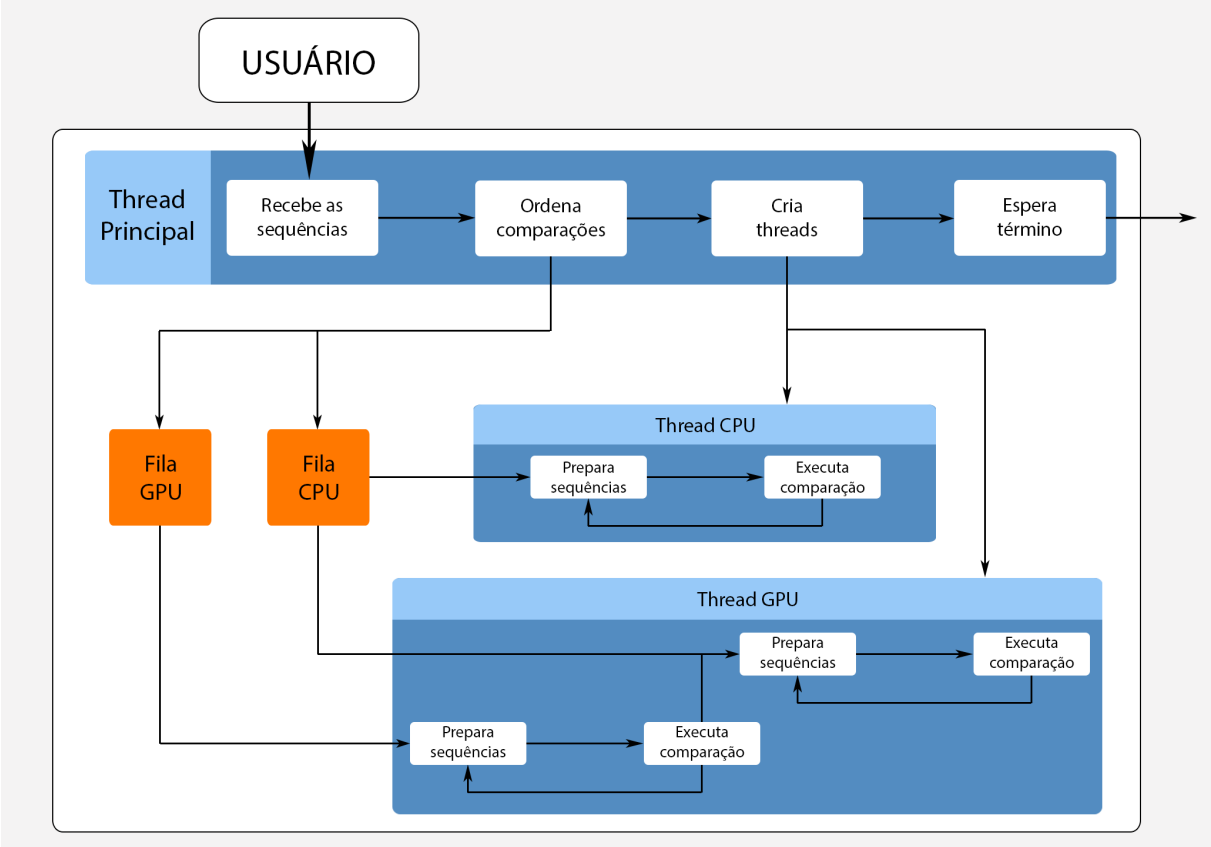

Figura 3.1: Arquitetura da solução CPU-GPU

Primeiramente, a *ThreadGPU* aponta para o primeiro elemento da *FilaGPU*, isto é, a comparação de maior tamanho daquela fila, e executa a ferramenta MASA-OpenCL com essa comparação. Quando a execução do MASA-OpenCL chegar ao fim, os dados da comparação são retirados da fila, o próximo elemento contendo o maior quantidade

de bytes é selecionado e o processo se repete até que a fila esteja vazia. Paralelamente, a *ThreadCPU* realiza o mesmo procedimento, entretanto, *FilaCPU* é ordenada de forma inversa, do menor para o maior, e por isso, as comparações com os menores tamanhos são executadas primeiro.

Depois que a *FilaGPU* estiver vazia, a *ThreadGPU* passa a consumir os dados também da *FilaCPU*, concorrendo com a outra *thread*, porém processando as comparações de maior tamanho, enquanto a *ThreadCPU* executa as menores. Quando uma das *threads* solicitar os dados do último elemento da *FilaCPU* e este já estiver sendo processado pela outra, aquela termina seu processamento e envia um sinal para a *thread* principal indicando sua finalização. Somente quando ambas as filas estiverem vazias e as *threads* encerradas é que o programa é finalizado.

3.2 Organização e ordenação das filas

As comparações são escritas linha a linha em arquivo de texto chamado *comparacoes.txt* no formato "s1 | s2", onde *s1* e *s2* representam os nomes das sequências a serem comparadas. Estipulou-se um limite de 30 comparações a serem executadas de cada vez, para que o tempo não seja tão elevado, portanto, se existirem mais comparações do que o estabelecido, nenhuma delas será calculada e um erro é retornado pela *thread* principal.

Após esta contagem, é verificado o tamanho de cada sequência, a fim de que o tamanho total da comparação seja conhecido. Para isso, as sequências precisam estar salvas no formato ".fasta" em um subdiretório e identificadas de acordo com o *Accession ID* gerado pelo *National Center for Biotechnology Information* (NCBI), disponível em *https://www.ncbi.nlm.nih.gov/*.

Inicialmente iríamos empregar o utilitário de linha de comando linux *"wc -c"*, que exibe a contagem de bytes em um arquivo, porém percebemos que o cabeçalho *<sys/stat.h>* da biblioteca C POSIX contém a função *stat()*, que realiza a mesma tarefa, facilitando a obtenção de informações sobre as sequências. Com o tamanho total verificado, a comparação é inserida na *FilaCPU* ou na *FilaGPU*, dependendo da quantidade de bytes que possuir e, para isso, definimos que as comparações cujo tamanho sejam menores do que 10 milhões de bytes seriam processados pela CPU e as demais pela GPU, pois comparações com um tamanho maior que esse levariam muito tempo para serem executadas pela CPU. Além disso, as comparações são ordenadas conforme seu tamanho em ordem crescente na *FilaCPU*, isto é, do menor para o maior, e em ordem decrescente na *FilaGPU*, isto é, do maior para o menor.

O projeto CPU-GPU utiliza duas listas em sua estrutura para armazenar as comparações de sequências que serão feitas. Uma das listas, chamada de *Fila CPU*, armazena

as comparações que possuem tamanho suficiente para serem processadas pela CPU. Essa lista é ordenada de forma crescente em relação ao tamanho das comparações. Ela é ordenada dessa forma pois assim a CPU processará as menores comparações primeiro e deixará as maiores por último.

A segunda lista, chamada de *Fila GPU*, armazena comparações muito grandes para serem executadas na CPU, que serão processadas pela GPU. Essa lista é ordenada de forma decrescente, fazendo com que as maiores comparações sejam processadas primeiro.

Na lista *Fila GPU*, a ordenação não influencia a execução do programa, pois todas as comparações presentes nesta lista serão executadas exclusivamente pela GPU. No caso da lista *Fila CPU*, a ordenação é determinante na ordem de execução das comparações, pois da forma como a mesma está ordenada, permite-se que, quando a GPU terminar de processar as comparações de sua fila, execute as maiores comparações presentes na lista *Fila CPU*, que ainda não foram executadas.

Depois que as todas as comparações forem distribuídas, entre as filas, a *thread* principal cria duas novas *threads*, responsáveis pela administração dessas e também pela invocação da ferramenta Masa-OpenCL para o cálculo de comparação entre as sequências.

3.3 Execução na CPU

A *ThreadCPU* é criada pela *thread* principal. É responsável pelas comparações realizadas em CPU. Após ser criada, a *ThreadCPU* recebe a lista com as sequências a serem calculadas e a gerencia conforme demanda.

A *thread* lê qual a primeira comparação disponível na fila e prepara a chamada do Masa-OpenCL para realizar este cálculo, indicando quais as sequências devem ser comparadas e alterando a pasta de trabalho que o MASA-OpenCL utilizará, como ilustrado na Figura 3.2. Quando estiver pronta, invoca a ferramenta para ser executada e retira da fila as sequências que foram comparadas, partindo para a próxima comparação.

```
``./masa-opencl --no-flush --stage-1 --work-dir=./s1_s2 s1.fasta s2.fasta``
```

Figura 3.2: Comando do MASA-OpenCL, onde *s1* e *s2* são os nomes das sequências a serem comparadas e *work-dir* o diretório onde serão salvas as estatísticas geradas pela ferramenta MASA-OpenCL

Estes procedimentos são repetidos até o momento em que a lista administrada pela *ThreadCPU* esteja vazia e, então, um sinal de retorno é enviado à *thread* principal indicando o fim do processamento.

3.4 Execução na GPU-CPU

A *ThreadGPU* é responsável pelas comparações realizadas em GPU e possui acesso a ambas as filas de comparação, porém controla somente uma de cada vez.

Primeiramente, esta *thread* manipula a fila de comparações a serem executadas em GPU, nos mesmos moldes em que a *ThreadCPU* consome a *FilaCPU*. A primeira comparação disponível na lista é preparada e executada pelo MASA-OpenCL e, quando o cálculo terminar, as sequências biológicas são retiradas da fila, a próxima comparação executada e assim sucessivamente.

Quando a *FilaGPU* estiver completamente vazia, ou seja, todas as comparações que deveriam ser executadas na GPU foram concluídas, a *ThreadGPU* passa a consumir elementos também da *Fila CPU*, se ainda existirem comparações nesta fila para serem calculadas. Caso haja, a GPU processa estas comparações, e assim ambas plataformas de computação são mantidas em funcionamento. Essa técnica é conhecida como *work stealing*, onde os processadores subutilizados tomam a seguinte iniciativa: tentam roubar tarefas de outros processadores [3]. Colocando em outras palavras, quando um processador está sem trabalho a executar, ele verifica se há tarefas disponíveis em outras unidades de processamento, as rouba e processa.

Entretanto, cada *thread* manipulará a *FilaCPU* de maneira diferente. Enquanto a *ThreadCPU* administra a lista na ordem crescente, isto é, a menor comparação é executada primeiro e a maior por último, na *ThreadGPU* as comparações são controladas de maneira inversa, ou seja, a maior comparação é calculada primeiro, pois são mais eficientes ao serem executadas em GPU do que as menores. Além disso, reduz-se as chances de uma comparação ser executada em ambas plataformas ao mesmo tempo. Para acelerar potencialmente o processo de comparação das sequencias não foi empregado semáforo ou qualquer outro tipo de controle de acesso sobre as filas de sequências. Caso haja duplo acesso à última uma comparação, ela será calculada tanto pela CPU quanto pela GPU, com a finalidade de retornar o cálculo o mais rápido possível. Nesse casso, assim que a primeira computação terminar (CPU ou GPU), a execução entra na etapa de finalização.

Assim que a *FilaCPU* for totalmente consumida por ambas as *threads*, um sinal é enviado para a função principal para comunicar o fim do processamento, esta realiza os últimos procedimentos e finaliza a execução do programa.

Capítulo 4

Resultados experimentais

Neste capítulo estão descritos os testes utilizados para avaliar o desempenho da solução proposta no Capítulo 3 em ambiente GPU ou misto (CPU e GPU). O principal objetivo desses testes é determinar situações onde o emprego da CPU em conjunto com a GPU pode reduzir o tempo gasto para realizar comparações entre sequências biológicas.

4.1 Ambiente de teste

Os resultados foram coletados no Laboratório de Sistemas Integrados e Concorrentes da Universidade de Brasília (LAICO). Foi utilizado um *desktop* com CPU Intel I7 3770, quatro núcleos, 3.40 GHz, memória RAM de 8 GB, 1333 MHz, disco rígido de 1 TB e GPU NVidia Geforce GTX 680, com 2048 MB de memória, 1006 MHz e 1536 núcleos de processamento.

4.2 Sequências biológicas utilizadas

Diversas sequências foram utilizadas durante os testes. Ao utilizar o MASA-OpenCL, a comparação é formada por duas sequências de quaisquer tamanhos. Estão expressas na Tabela 4.1 as comparações empregadas ao longo do experimento, cujas sequências foram obtidas no NCBI, mantendo o *Accession ID* como identificador destas sequências.

Foram utilizados quatro conjuntos de comparações durante os testes, com a intenção de medir o tempo gasto para a execução de cada um eles nos ambientes especificados:

- o primeiro, onde somente a GPU processa as comparações;
- o segundo onde GPU e CPU executem simultaneamente o cálculo da comparação.

Nome	Tamanho	Sequência 1		Sequência 2	
		Accession ID	Tamanho	Accession ID	Tamanho
C1	115,9 kB	AF494279.1	57,5 kB	NC_001715.1	58,4 kB
C2	338,9 kB	NC_000898.1	164,5 kB	NC_007605.1	174,4 kB
C3	1,1 MB	NC_000914.1	543,9 kB	NC_003064.2	550,7 kB
C4	2,2 MB	CP000051.1	1,1 MB	AE002160.2	1,1 MB
C5	6,5 MB	BA000035.2	3,2 MB	BX927147.1	3,3 MB
C6	10,4 MB	BA000035.2	3,2 MB	NC_005027.1	7,2 MB
C7	10,6 MB	NC_003997.3	5,3 MB	AE017225.1	5,3 MB
C8	20,8 MB	NC_014318.1	10,4 MB	NC_017186.1	10,4 MB

Tabela 4.1: Comparações entre sequências biológicas selecionadas para teste

As comparações referidas na Tabela 4.1 foram divididas em quatro grupos diferentes. Cada grupo é escrito no arquivo de texto (Seção 3.1) para entrada de dados no programa, como ilustra a Figura 3.1. A Tabela 4.2 descreve estess quatro agrupamentos de comparações.

Como descrito na seção 3.1, apenas um grupo é executado por vez, sendo as comparações divididas entre as filas *FilaGPU* e *FilaCPU*, dependendo do tamanho destas comparações.

Grupos	Comparações utilizadas
G1	C1, C2, C3, C4, C7
G2	C1, C2, C3, C4, C8
G3	C1, C2, C7,
G4	C1, C2, C3, C4, C5

Tabela 4.2: Conjunto de comparações

4.3 Resultados obtidos

Para a realização dos testes, os quatro grupos foram executados em todos os ambientes (GPU e misto) por três vezes, apresentando desvio-padrão desprezível em cada um deles. Desta forma, os valores médios obtidos foram os adotados nas análises. A seguir, descrevemos os resultados alcançados durante os testes.

4.3.1 Tempo gasto de processamento

Para o primeiro grupo, *G1*, quando as comparações são executadas somente em GPU, o tempo de processamento gasto pela solução proposta é de 4 minutos, 49 segundos. Já no ambiente misto CPU-GPU, a execução foi cerca de 5 segundos mais rápida, como ilustra

o gráfico da Figura 4.1. Essa pequena diferença ocorreu pois a maioria dos arquivos são muito pequenos (apenas um deles possui mais de 10 MB), sendo processados em poucos segundos.

Figura 4.1: Tempo de execução do grupo *G1* em cada ambiente (GPU e misto)

Grande parte do tempo é utilizado para o processamento da maior comparação, sendo assim, este projeto permite que enquanto a GPU está ocupada processando a comparação de maior tamanho a CPU antecipe o processamento de outras comparações. Por essa razão, houve redução do tempo de execução com a solução mista (CPU + GPU).

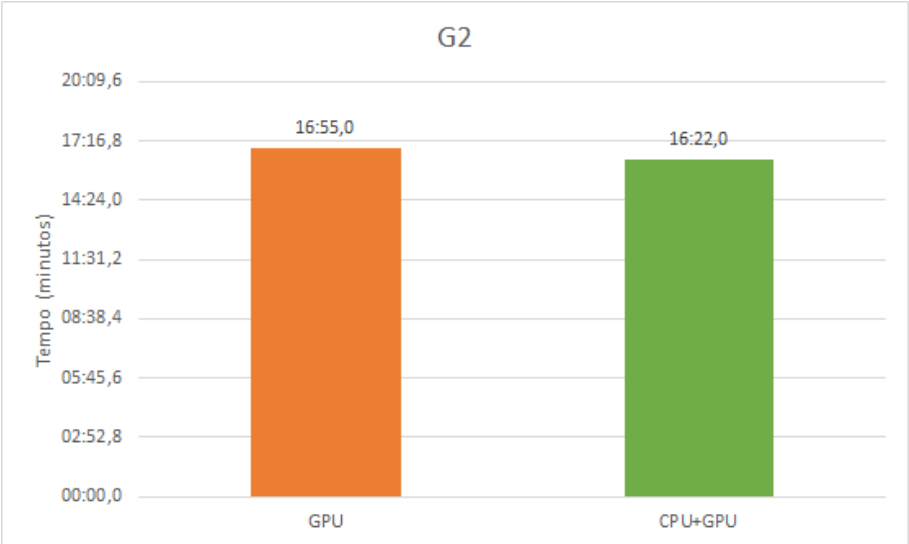

Figura 4.2: Tempo de execução do grupo *G2* em cada ambiente (GPU e misto)

Para o grupo *G2*, o tempo total de execução das comparações em ambiente GPU é mais extenso do que o do grupo *G1*, aproximadamente 17 minutos. Entretanto, no

ambiente misto CPU-GPU, o tempo gasto para a execução das comparações deste grupo foi reduzido para cerca de cerca de 16 minutos e 22 segundos, como mostra a Figura 4.2.

A impressão das mensagens geradas durante do processamento inicial do programa está descrito a seguir (adaptado):

```
>Arquivo 'comparacoes.txt' existe
>Existem 5 linhas no arquivo 'comparacoes.txt'
>Quantidade de comparacoes est'a dentro do limite
>Iniciando leitura do arquivo 'comparacoes.txt'

------------------- FILAS ------------------

    ==== FILA GPU
        Sequencia 1    |     Sequencia2    | Tamanho MB da comparacao
    NC_014318.1.fasta | NC_017186.1.fasta | 20766156

    ====

    ==== FILA CPU
        Sequencia 1    |     Sequencia2    | Tamanho MB da comparacao
    AF494279.1.fasta  | NC_001715.1.fasta | 115852
    NC_000898.1.fasta | NC_007605.1.fasta | 338853
    NC_000914.1.fasta | NC_003064.2.fasta | 1094629
    CP000051.1.fasta  | AE002160.2.fasta  | 2147806
    ====


>Filas prontas!
```

Após as comparações do *G2* serem divididas, ambas filas são consumidas de forma paralela. A impressão de mensagens durante o processamento continua (adaptado):

```
>Comecando Thread GPU
>Criada thread '1517860608'
>Comecando Thread CPU
>Criada thread '1509467904'
>>>>> Executando comparacao entre AF494279.1.fasta e NC_001715.1.fasta'
>>>>> Executando comparacao entre NC_014318.1.fasta e NC_017186.1.fasta'
>Removido da fila a comparacao entre AF494279.1.fasta e NC_001715.1.fasta'
>>>>> Executando comparacao entre NC_000898.1.fasta e NC_007605.1.fasta'
>Removido da fila a comparacao entre NC_000898.1.fasta e NC_007605.1.fasta'
>>>>> Executando comparacao entre NC_000914.1.fasta e NC_003064.2.fasta'
>Removido da fila a comparacao entre NC_000914.1.fasta
```

```
>>>>>>>>>>>>> Thread CPU acabou filaCPU
>>>>>>>>>>>>> Thread CPU esta aguardando o fim da Thread GPU


>Removido da fila a comparacao entre NC_014318.1.fasta e NC_017186.1.fasta'


>>>>>>>>>>>>> Thread GPU acabou filaGPU
>>>>>>>>>>>>> Thread GPU acabou filaCPU


>A thread '1517860608' já terminou
>Terminei Thread GPU
>A thread '1509467904' já terminou
>Terminei Thread CPU

>Acabou execucao
```

Como percebe-se nas mensagens acima, a CPU executou o processamento de diversas comparações e finalizou antes que o GPU consumisse toda a *FilaGPU*. Isso fez com que o tempo total de processamento do grupo fosse menor do que se as comparações fossem processadas apenas na GPU.

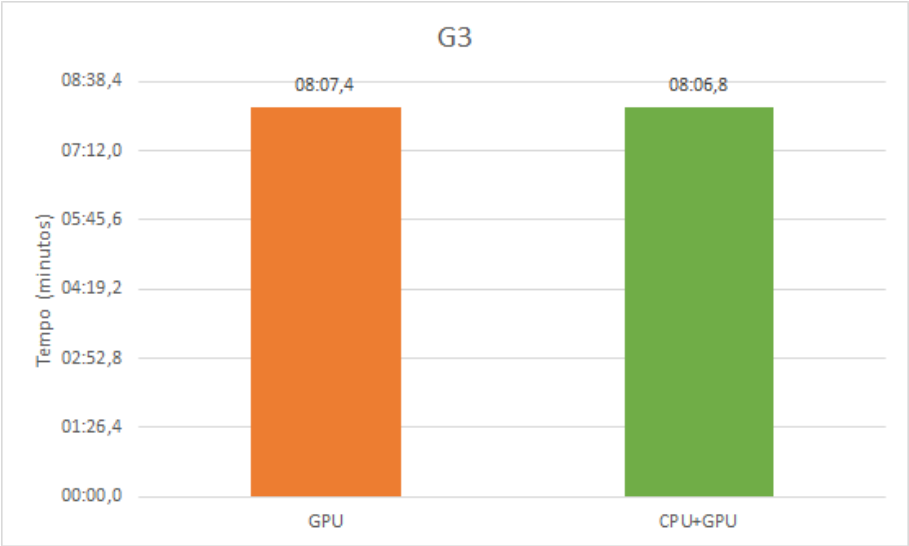

Figura 4.3: Tempo de execução do grupo *G3* em cada ambiente (GPU e misto)

Já o grupo *G3*, em ambiente misto, as comparações do grupo *G3* foram executadas em 8 minutos, 6 segundos e 800 milésimos e 8 minutos, 7 segundos e 400 milésimos em ambiente GPU. A figura 4.3 mostra o tempo de execução do grupo *G3* em cada ambiente.

O grupo *G4* é composto apenas por sequências pequenas, isto é, menores que 10MB, e, em razão disso, o tempo necessário de processamento das comparações ficou similar entre os ambientes, 3 minutos

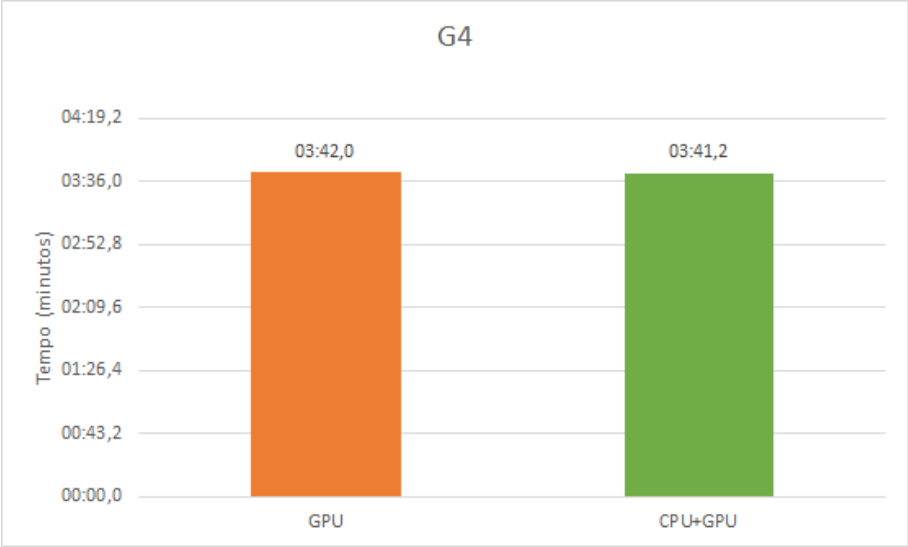

Figura 4.4: Tempo de execução do grupo *G4* em cada ambiente (GPU e misto)

e 42 segundos e 3 minutos e 31 segundos para o processamento em GPU e misto (CPU e GPU), respectivamente. Entretanto, verifica-se que mesmo as comparações pequenas são executadas de maneira mais rápida na GPU do que a CPU. A Figura 4.4 revela o tempo gasto para o processamento em CPU, GPU e CPU+CPU.

A seguir é demonstrado o fluxo de processamento durante um dos testes dos grupo *G4*:

1. o arquivo com as comparações do grupo foi lido e processado;
2. as comparações foram distribuídas apenas na *FilaCPU*, pois não possuíam tamanho suficiente para serem inseridas na *FilaGPU* e as filas foram geradas da seguinte maneira:

```
==== FILAGPU
    Sequencia 1    |     Sequencia2    | Tamanho MB da comparacao
(vazia)
====

==== FILA CPU
    Sequencia 1    |     Sequencia2    | Tamanho MB da comparacao
AF494279.1.fasta   | NC_001715.1.fasta | 115852
NC_000898.1.fasta | NC_007605.1.fasta | 338853
NC_000914.1.fasta | NC_003064.2.fasta | 1094629
CP000051.1.fasta   | AE002160.2.fasta  | 2147806
BA000035.2.fasta   | BX927147.1.fasta  | 6521847
====
```

3. *ThreadCPU* e *ThreadGPU* foram iniciadas;
4. *ThreadGPU* constatou que a *FilaGPU* estava vazia e, por isso, iniciou o consumo da *FilaCPU*, da maior comparação para a menor, conforme descrito na seção 3.1;
5. *ThreadCPU* iniciou comparação do primeiro item da *FilaCPU* enquanto a *ThreadGPU* iniciou do último;

6. *ThreadCPU* retirou o primeiro item da fila e processou a segunda comparação;
7. *ThreadCPU* processou a terceira comparação;
8. *ThreadCPU* processou a quarta comparação;
9. *ThreadCPU* finalizou processamento da *FilaCPU*, pois percebeu que o último elemento está sendo realizado pela outra *thread*. Sendo assim, ela apenas aguarda o fim do processamento da *ThreadGPU*;
10. *ThreadGPU* finaliza o processamento da quinta comparação e informa seu encerramento à *thread* principal ;
11. *Thread* principal finaliza execução do programa.

Pudemos observar pelo fluxo de processamento que durante o processamento das comparações do grupo *G4* houve *work stealing*, pois ao perceber que estava sem trabalho, a *ThreadGPU* passou a comparar sequências da *FilaCPU*.

4.3.2 *Speedup*

Speedup é o fator de aceleração de um sistema paralelo e pode ser definido como a relação entre o tempo gasto por um único processador para resolver uma instância de problema dada e o tempo tomado por um sistema paralelo consistindo de n processadores para resolver a mesma instância de problema [6].

Grupos	**Speedup**
G1	1,021x
G2	1,031x
G3	1,002x
G4	1,005x

Tabela 4.3: Valores de *speedup*

Calculamos o *speedup* com relação entre o tempo gasto para a execução somente em ambiente GPU e ambiente misto (em CPU juntamente com a GPU) em segundos. Com isso, alcançamos os dados exibidos na Tabela 4.3.

Como podemos observar, em todos os casos houve diminuição do tempo gasto de processamento ao utilizar o mecanismo de comparação paralela em GPU e CPU, destacando-se o grupo *G2*, onde o *speedup* favoreceu a aceleração do processamento em 3,1%.

Capítulo 5

Conclusão

O presente trabalho de graduação propôs, implementou e avaliou uma estratégia híbrida CPU-GPU para o MASA-OpenCL. Essa estratégia consistiu em fazer com que o MASA-OpenCL passasse a trabalhar paralelamente tanto com a GPU quanto com a CPU da máquina, visando uma melhora no desempenho da comparação de sequências biológicas longas. Essa melhora aconteceu pois o MASA-OpenCL, diferente do que acontecia anteriormente, passou a utilizar também a CPU para realizar as comparações de sequência de menor tamanho. Assim, o processo de comparação foi acelerado, tornando-se mais eficiente e utilizando de uma melhor forma o poder computacional da máquina, tendo em vista que quando o MASA-OpenCL utiliza apenas a GPU ou CPU para fazer comparações, a outra unidade fica ociosa.

Os resultados obtidos com a utilização de uma CPU e uma GPU mostraram que, com a CPU incumbindo-se de realizar as comparações de sequências que possuem um tamanho considerado pequeno, a GPU fica livre para realizar apenas as comparações grandes em um primeiro momento, havendo uma melhoria expressiva no tempo das comparações de sequências. Em alguns casos, a utilização apenas da GPU tornou o processo até 5,42% mais lento em comparação com a utilização do alocador CPU-GPU. Também foi percebido que em todas as situações testadas, o alocador CPU-GPU reduziu o tempo necessário para realizar uma comparação em relação ao tempo obtido com uso exclusivo da GPU, mesmo em casos com um grande número de comparações grandes e poucas comparações pequenas, embora essa redução no tempo não tenha sido tão grande. Portanto, conclui-se que dentro de um limite preestabelecido, limite esse definido a partir da capacidade de processamento da CPU em relação a GPU, a utilização da CPU em conjunto com a GPU agiliza o processo de comparações de sequências biológicas longas.

Como trabalhos futuros, sugerimos:

- Criar mais *threads* para a CPU, a fim de que vários *cores* executem as comparações;

- Desenvolvimento de um sistema que defina um tamanho máximo onde se tornaria vantajoso a utilização da CPU para executar comparações dentro de um conjunto de comparações de cadeias biológicas;
- Incorporação de outros aceleradores, tais como o Intel Phi e FPGA; e
- Execução de uma única comparação de pares de sequências usando CPU e GPU.

Referências

[1] A.D. BAXEVANIS e B.F.F. OULLETTE. *Bioinformatics: a practical guide to the analysis of genes and proteins*. John Wiley, 2001. 1, 3

[2] Jacopo BELLATI. Interface web para execução do algoritmo cudalign para comparação de sequências biológicas em gpu, 2014. 10

[3] Robert D. BLUMOFE e Charles E. LEISERSON. Scheduling multithreaded computations by work stealing. *J. ACM*, 46(5):720–748, September 1999. 2, 20

[4] Marcelo Ramos COLLETTI. Comparação de sequências biológicas utilizando a plataforma xd2000i de hardware reconfigurável, 2016. vii, 8

[5] Marco Antônio C. DE FIGUERÊDO JÚNIOR. Masa-opencl: Comparação paralela de sequências biológicas longas em gpu. Dissertação (Mestrado), UnB, 2015. vii, 3, 4, 5, 6, 7, 9, 13, 14

[6] Hesham EL-REWINI e Mostafa ABD-EL-BARR. *Advanced Computer Architecture and Parallel Processing*. John Wiley and Sons, Inc, 2005. 27

[7] O. GOTOH. An improved algorithm for matching biological sequences. *Journal of Molecular Biology*, page 705–708, Mar 1982. 6

[8] Daniel. S. HIRSCHBERG. A linear space algorithm for computing maximal common subsequences. *Commun. ACM*, 18, 1975. 6, 7

[9] David W. MOUNT. *Bioinformatics: sequence and genome analysis*. Cold Spring Harbor Laboratory Press, 2001. 1, 3

[10] Eugene W. MYERS e Webb MILLER. Optimal alignments in linear space. *Computer applications in the biosciences : CABIOS*, 4(1):11–17, 1988. 7

[11] Saul B. NEEDLEMAN e Christian 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. 1, 4

[12] Edans Flavius de Oliveira SANDES. Comparação paralela de sequências biológicas longas utilizando unidades de processamento gráfico(gpus), 2011. vii, 11, 12

[13] Edans Flavius de Oliveira SANDES. Algoritmos paralelos exatos e otimizações para alinhamento de sequências biológicas longas em plataformas de alto desempenho. Dissertação (Mestrado), UnB, 2015. vii, 9, 10, 11, 12

[14] Edans Flavius de Oliveira SANDES e Alba Cristina M. A. DE MELO. CUDAlign: using GPU to accelerate the comparison of megabase genomic sequences. In *15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP*. ACM, 2010. 9

[15] M.C. SCHATZ e B. LANGMEAD. The dna data deluge: Fast, efficient genome sequencing machines are spewing out more data than geneticists can analyze. *IEEE spectrum*, 50(7):26–33, 2013. 1

[16] T. F. SMITH e M. S. WATERMAN. Identification of common molecular subsequences. *Journal of Molecular Biology*, page 195–197, Mar 1981. 1, 5, 6