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

Comparação Paralela de Sequências Biológicas: Otimização da Área de Descarte de Blocos com Escore Heurístico

Pedro Vitor V. Mizuno

Monografia apresentada como requisito parcial
para conclusão do Bacharelado em Ciência da Computação

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

Brasília
2022

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

Comparação Paralela de Sequências Biológicas: Otimização da Área de Descarte de Blocos com Escore Heurístico

Pedro Vitor V. Mizuno

Monografia apresentada como requisito parcial
para conclusão do Bacharelado em Ciência da Computação

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

Prof.a Dr.a Maristela Holanda
CIC/UnB

Prof. Dr. Marcelo Mandelli
CIC/UnB

Prof. Dr. Marcelo Grandi Mandelli
Coordenador do Bacharelado em Ciência da Computação

Brasília, 05 de maio de 2022

Dedicatória

Dedico este trabalho para minha família, que fez a pessoa que me orgulho de ser, e aos meus amigos, que sempre estiveram comigo em tantas batalhas e por tantos anos.

Agradecimentos

Agradeço à professora Alba, que me orientou com excelência durante todo o processo de desenvolvimento deste trabalho, ao Marco Figueirêdo, autor do módulo MultiBP e que me auxiliou no entendimento de conceitos e do funcionamento da ferramenta, e aos colegas Filipe Maia e Walisson Sousa, por tanto me ajudarem na configuração do *cluster*. Finalmente, agradeço ao CNPq e AWS que, por meio do projeto 440014/2020-4, forneceram acesso ao AWS *Parallel Cluster* e às instâncias GPU da AWS.

Resumo

Este trabalho de graduação apresenta o módulo de obtenção de escore inicial heurístico adicionado à ferramenta de comparação de sequências biológicas MASA-CUDAlign-MultiBP. A partir deste módulo, busca-se expandir a área de descarte de blocos da matriz de programação dinâmica ao inicializar a execução da ferramenta com o escore heurístico, obtido com o alinhador de sequências heurístico *Lastz*, ao contrário da inicialização original da ferramenta, cujo escore inicial é 0. O módulo foi implementado e integrado à ferramenta e, para testá-lo e coletar resultados, foi configurado um *cluster* de duas GPUs NVIDIA T4 Tensor Core hospedado na nuvem AWS (*Amazon Web Services*). Nos testes, foram feitas comparações entre sequências reais de DNA de tamanho 1M, 3M, 5M e 10M. Os resultados encontrados mostram que, para as sequências utilizadas, há uma melhora de desempenho mais significativa em comparações de sequências compridas com alta similaridade e para as quais foi obtido um escore heurístico inicial grande. Ademais, aliado ao escore inicial, também é atestada a melhora de desempenho que a mudança da distribuição de colunas da matriz entre GPUs (*split*) gera para ferramenta MASA-CUDAlign-MultiBP.

Palavras-chave: comparação de sequências biológicas, descarte (*pruning*), *cluster* de GPUs, MASA-CUDAlign-MultiBP

Abstract

This undergraduate project presents a module that obtains a heuristic initial score, which was integrated to the biological sequence comparison tool MASA-CUDAlign-MultiBP. From this module, we look to expand the pruning area in the dynamic programming matrix through the tool's initialization with the heuristic score, obtained by the heuristic sequence aligner *Lastz*, unlike the original initialization of the tool, which set the initial value as 0. In order to test and collect results with our module, we configured a cluster with two NVIDIA T4 Tensor Core GPUs hosted in the AWS (*Amazon Web Service*) cloud. Comparisons were made between real DNA sequences with of size 1M, 3M, 5M and 10M. The results obtained show that, for these sequences, there is a more significant improvement of performance in comparisons between lengthier and similar sequences for which the *Lastz* tool was able to obtain a larger heuristic initial score. Furthermore, combined with the initial score, it is presented the improvement of performance resulted through the change of the column distribution of the matrix between GPUs (split) to the tool MASA-CUDAlign-MultiBP.

Keywords: biologial sequence comparison, pruning, GPU cluster, MASA-CUDAlign-MultiBP

Sumário

1 Introdução **1**

2 Comparação de Sequências Biológicas **4**
2.1 Definições 4
2.2 Algoritmo Needleman-Wunsch (NW) 5
2.3 Algoritmo Smith-Waterman (SW) 7
2.4 Algoritmo Gotoh 9
2.5 Algoritmo Myers-Miller 10

3 GPU e CUDA **12**
3.1 Evolução das GPUs NVIDIA 12
3.2 Arquitetura da GPU NVIDIA 13
3.3 CUDA 15
3.3.1 Funcionalidades do CUDA 18

4 A Ferramenta MASA **22**
4.1 A Evolução da Ferramenta 22
4.2 MASA-CUDAlign 4.0 27
4.3 MASA-CUDAlign-MultiBP 31
4.4 MASA-OpenMP 33
4.5 A Ferramenta MASA no AWS Cloud 35

5 Projeto da Otimização da Área de Descarte de Blocos com Escore Inicial Heurístico **38**
5.1 Motivação 38
5.2 Visão Geral 39
5.3 Sub-módulos da Otimização da Área de Descarte 41
5.3.1 Sub-Módulo de Obtenção do Escore Heurístico 41
5.3.2 Sub-Módulo de Execução do MASA-CUDAlign-MultiBP 43
5.4 Execução do MASA-CUDAlign-MultiBP no *Cluster* de GPUs da Nuvem AWS 44

6 Resultados Experimentais **47**
6.1 Configuração do AWS *ParallelCluster* 47
6.2 Sequências Utilizadas 49
6.3 Resultados do Módulo de Obtenção de Escore Heurístico 50
6.3.1 Resultados obtidos com Escore igual a 0 50
6.3.2 Resultados obtidos com Escore Inicial igual ao Escore Heurístico 52
6.3.3 Comparação dos Tempos de Execução 52
6.3.4 Área de Poda 57

7 Conclusão e Trabalhos Futuros **61**

Referências **64**

Lista de Figuras

2.1 Alinhamento NW (escore = -6). 5
2.2 Algoritmo NW para calcular o escore de um elemento (Adaptado de [1]). . . 6
2.3 Matriz de programação dinâmica NW entre duas sequências. 7
2.4 Matriz de programação dinâmica SW entre duas sequências. 8
2.5 Alinhamento SW (escore = 3). 9
2.6 Matrizes H, E e F e resultado do alinhamento com algoritmo Gotoh. 10
2.7 Funcionamento do algoritmo Myers-Miller (Adaptado de [2]). 11

3.1 *Streaming Multiprocessor* da GPU GV100 (Fonte: [3]). 15
3.2 Hierarquia de memória da GPU GV100 (Fonte: [4]). 16
3.3 A arquitetura do CUDA (Fonte: [5]). 16
3.4 Hierarquia de memória do CUDA (Fonte: [6]). 18
3.5 Representação de um sistema sem e com o UM (Fonte: [7]). 19
3.6 Exemplos de arquiteturas utilizando o NVLink (Fonte: [8]). 20

4.1 Processamento *wavefront* em blocos paralelogramos (Fonte: [9]). 23
4.2 Dependência entre blocos de uma mesma diagonal (Fonte: [9]). 24
4.3 Representação da obtenção do maior escore (Fonte: [9]). 26
4.4 Processamento da matriz em múltiplas GPUs (Fonte: [9]). 26
4.5 Execução com *traceback* com *pipeline* (Fonte: [9]). 28
4.6 Execução com o *traceback* especulativo (Fonte: [9]). 29
4.7 Arquitetura MASA (Fonte: [9]). 30
4.8 Troca de maiores escores locais (Fonte: [10]). 32
4.9 Arquitetura do MultiBP (Fonte: [10]). 34

5.1 Modelo do projeto. 39
5.2 Integração do Projeto ao AWS Parallel Cluster. 45

6.1 Comandos para a configuração do *cluster* e execução do módulo do TCC. . 48
6.2 Desempenho na comparação 1M-1M. 54
6.3 Desempenho na comparação 3M-3M. 54

6.4 Desempenho na comparação 5M-5M. 55
6.5 Desempenho na comparação 10M-10M. 56
6.6 Área de poda da comparação 1M-1M com o escore 0 e o heurístico. 58
6.7 Área de poda da comparação 3M-3M com o escore 0 e o heurístico. 59
6.8 Área de poda da comparação 5M-5M com o escore 0 e o heurístico. 59
6.9 Área de poda da comparação 10M-10M com o escore 0 e o heurístico. 59

Lista de Tabelas

3.1 Primeira parte da tabela de especificações. (Adaptado de [10]) 13
3.2 Segunda parte da tabela de especificações. (Adaptado de [10]) 13
3.3 Descrição de ferramentas do CUDA. 21

6.1 Sequências e os escores produzidos em suas comparações. 50
6.2 Tempos obtidos a partir do escore inicial 0. 51
6.3 Tempos obtidos a partir do escore inicial heurístico. 53
6.4 *Speedup* ao utilizar o escore heurístico. 56
6.5 *Speedup* ao utilizar o escore ótimo. 57
6.6 Percentual de área podada para cada comparação. 60

Capítulo 1

Introdução

A Bioinformática é uma área de estudo que possui um de seus focos na criação de algoritmos e ferramentas capazes de avaliar e comparar sequências biológicas. Entre as muitas utilidades da comparação de sequências, este estudo permite a predição da funcionalidade de um gene de um organismo similar a um outro organismo já conhecido, assim, facilitando o entendimento acerca das informações genéticas de diversos organismos [11]. Para tal, as sequências são comparadas por meio da técnica de programação dinâmica, a partir da qual é criada uma matriz onde são feitas comparações entre os resíduos (caracteres) das sequências biológicas (DNA, RNA ou proteínas). Ao depender do resultado da comparação dos resíduos, sendo eles iguais (*match*), diferentes (*mismatch*) ou havendo um espaçamento (*gap*), é retornado um valor que é somado ao escore, que é a métrica utilizada para determinar o melhor alinhamento obtido na comparação.

Entre os vários métodos utilizados para realizar a comparação de sequências biológicas, e é possível classificá-los em dois tipos, os heurísticos e os exatos. Os métodos heurísticos, entre eles o FASTA [12] e o BLAST [13], se destacam por apresentarem um melhor desempenho e alinhamentos de boa qualidade, em geral. No entanto, esses métodos não produzem o melhor resultado possível (resultado ótimo) [11]. Quanto aos métodos exatos, entre eles o algoritmo Needleman-Wunsch [14], Smith-Waterman [15] e outros, o alinhamento ótimo é obtido em detrimento de um melhor desempenho.

Devido ao possível grande tamanho das sequências comparadas com os métodos exatos e, consequentemente, à enorme matriz de programação dinâmica resultante, se tornou necessária a utilização de unidades de processamento cada vez mais poderosas. Apesar de terem sido originalmente criadas para realizarem cálculos para renderização gráfica, as GPUs (*Graphics Processing Unit*) se mostraram boas unidades de processamento para os cálculos realizados em comparações de sequência. Aliadas ao poder de processamento dessas unidades de processamento, também foram desenvolvidas ferramentas de comparação de sequências biológicas que paralelizam o cálculo entre GPUs. Entre as ferramentas

de processamento paralelo se destacam o SWCuda [16], o CudaSW++ [17] e o MASA-CUDAlign [9], que representa o estado da arte nesta área.

O MASA-CUDAlign [9] consiste em uma ferramenta de comparação de sequências biológicas que obtém o alinhamento ótimo por meio do algoritmo Smith-Waterman [15] e que foi desenvolvida na arquitetura CUDA (*Compute Unified Device Architecture*). Ao longo do desenvolvimento do MASA-CUDAlign foram lançadas várias versões, cujos principais fatores a se destacar são: o processamento *wavefront* e blocos em formato de paralelogramo na versão 1.0; a estratégia da utilização de linhas e colunas especiais para encontrar o alinhamento ótimo, além da separação em estágios, na versão 2.0; a adoção da técnica de descarte de blocos para uma GPU na versão 2.1, em que se descartam blocos que nunca poderão resultar em um escore melhor que o escore máximo momentâneo, ou seja, são descartados aqueles blocos que não fazem parte do alinhamento ótimo; e a paralelização do primeiro estágio da ferramenta entre múltiplas GPUs na versão 3.0. Por último, foi lançada a versão 4.0, em que objetivou-se a extensão da paralelização para os demais estágios da ferramenta, além da criação da arquitetura MASA (*Multi-platform Architecture for Sequence Aligners*), que permitiu que a ferramenta CUDAlign e outras similares fossem executadas independentemente à plataforma.

De forma a complementar ao MASA-CUDAlign, o módulo MultiBP [10] foi desenvolvido para permitir que a ferramenta pudesse realizar o descarte de blocos em execuções com diversas GPUs. A fim de que todas as GPUs saibam o melhor escore atual da execução, este módulo criou a estratégia de anel, em que cada GPU recebe o escore de sua vizinha à esquerda, compara com o seu, atualiza seu melhor escore para ser o maior entre os dois e envia o seu novo melhor escore a sua vizinha à direita. Além disso, ao iniciar a ferramenta MASA-CUDAlign-MultiBP, todas as GPUs são inicializadas com escore 0, de forma que o melhor escore de cada GPU aumente à medida que a matriz é processada e os escores são compartilhados entre elas. Assim, a ferramenta MASA-CUDAlign-MultiBP se mostrou muito efetiva para sequências similares, devido ao grande escore resultante do alinhamento e, consequentemente, grande área de poda.

Então, o objetivo deste trabalho foi otimizar a área de descarte de blocos por meio da inicialização de todas as GPUs com um escore factível e maior que 0. Isto se deve ao fato de que a ferramenta MASA-CUDAlign-MultiBP inicializa todas as GPUs com o escore 0 e, à medida que a matriz de programação dinâmica é processada, este escore aumenta gradativamente. No entanto, com a inicialização com um valor factível e maior que 0, objetiva-se permitir que o descarte de blocos ocorra mais cedo no processamento da matriz de programação dinâmica, visto que se torna mais comum que os blocos nunca atinjam um escore melhor que este escore inicial, resultando em uma maior área de descarte de blocos. De forma que, para obter este escore inicial, foram utilizadas ferramentas heurísticas, que

podem obter um bom escore e em menor tempo, comparativamente à ferramenta MASA-CUDAlign-MultiBP. Assim, este projeto desenvolveu um módulo adicional ao MASA-CUDAlign-MultiBP que obtêm um escore inicial heurístico e inicializa a primeira GPU com este valor, compartilhando-o com as GPUs vizinhas, objetivando a expansão da área de descarte de blocos da matriz e, consequentemente, a diminuição do tempo de execução da ferramenta.

Assim, o Capítulo 2 apresenta conceitos e algoritmos relacionados à comparação de sequências biológicas. O Capítulo 3 explana o histórico e arquitetura das GPUs NVIDIA e do CUDA. O Capítulo 4 apresenta em maiores detalhes a ferramenta MASA-CUDAlign-MultiBP, assim como o ambiente em nuvem onde ela foi executada. No Capítulo 5 é apresentado o projeto desenvolvido neste trabalho. No Capítulo 6 são mostradas a configuração do ambiente de execução, as sequências utilizadas e os resultados obtidos. Por fim, o Capítulo 7 apresenta as conclusões e os trabalhos futuros sugeridos.

Capítulo 2

Comparação de Sequências Biológicas

Neste capítulo são descritos conceitos básicos e algoritmos relevantes para a área de comparação de sequências biológicas. A Seção 2.1 apresenta algumas das definições mais relevantes para o melhor entendimento acerca dos algoritmos de comparação. Ademais, as Seções 2.2, 2.3, 2.4 e 2.5 apresentam o funcionamento de cada um dos algoritmos Needleman-Wunsch, Smith-Waterman, Gotoh e Myers-Miller, respectivamente.

2.1 Definições

O DNA consiste em um conjunto de nucleotídeos os quais, seguindo uma ordem, definem as mais diversas características dos seres vivos e permitem a síntese de proteínas [11]. Devido a essas importantes funções, o DNA é estudado em diversas áreas de conhecimento como na biologia, biotecnologia, bioinformática, dentre outras. O estudo de sequências biológicas possibilita, por exemplo, traçar relações genéticas entre seres vivos, analisar áreas do genoma e o que elas podem causar. Entre as várias técnicas utilizadas para examinar o DNA destaca-se o alinhamento, técnica essa que permite identificar similaridades e diferenças entre duas ou mais sequências biológicas.

O alinhamento é um procedimento que busca caracteres individuais ou padrões em comum entre duas ou mais sequências [11]. Para tal, sequências biológicas são representadas digitalmente por *strings*, sendo que os nucleotídeos (DNA) e aminoácidos que os compõem são caracteres dos alfabetos $\Sigma = \{A, C, G, T, N\}$ e $\Sigma = \{A, C, D, E, F, G, H, I, K, L, M, N, P, Q, R, S, T, W, Y, V\}$, respectivamente. Cada caractere de uma sequência é pareado com um caractere igual, diferente ou um espaço em branco da outra sequência, alinhamentos esses que resultam em *match*, *mismatch* e *gap*, respectivamente. Conforme mostra a Figura 2.1, por meio do algoritmo Needleman-Wunsch (Seção 2.2) caracteres

$$
\begin{array}{ccccccccccccccc}
G & T & A & C & _ & T & T & G & A & G & A & G & T & A & C \\
A & T & C & G & G & A & T & G & A & C & A & C & A & T & T \\
-1 & 1 & -1 & -1 & -2 & -1 & 1 & 1 & 1 & -1 & 1 & -1 & -1 & -1 & -1
\end{array}
$$

Figura 2.1: Alinhamento NW (escore = -6).

das duas sequências são pareados, resultando em *matches* e *mismatches* e, quando necessário haver um espaçamento em alguma das sequências para se obter um melhor escore, é estabelecido um *gap*.

O alinhamento pode ser classificado em três tipos [1]:

- Global: todos os caracteres das sequências são pareados;
- Semi-global: o início ou o fim de uma ou ambas as sequências não são pareados;
- Local: apenas *substrings* das sequências são pareadas.

A qualidade de um alinhamento entre duas sequências é calculada a partir do escore, um sistema de avaliação que pontua o pareamento de símbolos iguais (*match*) e penaliza o pareamento de símbolos diferentes (*mismatch*) e *gaps* [11]. Ao depender do algoritmo utilizado, existem duas formas de calcular o *gap* de um alinhamento. O *linear gap* define que todos os *gaps* geram a mesma penalidade ao escore, proporcionalmente ao seu número de aparições. Ao passo que o *affine gap* utiliza uma penalidade de abertura do *gap* e uma penalidade de extensão do *gap*, ambas com pesos diferentes. A Figura 2.1 apresenta as pontuações utilizadas no exemplo de alinhamento Needleman-Wunsch (Seção 2.2), que utiliza *affine gap*, sendo $match = 1$, $mismatch = -1$ e $gap = -2$.

Os algoritmos de alinhamento buscam o resultado ótimo, ou seja, o alinhamento de maior escore e, para tal, se baseiam na programação dinâmica [1]. A programação dinâmica se fundamenta em que, para se obter a solução ótima de um problema, é necessário dividi-lo em múltiplos sub-problemas e, por meio da recursividade, achar suas soluções ótimas. Ao se otimizar os sub-problemas, é garantido que será obtida a solução ótima do problema original. Dessa forma, nas próximas seções serão apresentados algoritmos que geram alinhamentos ótimos.

2.2 Algoritmo Needleman-Wunsch (NW)

Proposto por Needleman e Wunsch (NW) em 1970 [14], o algoritmo NW produz o alinhamento global de duas sequências. Tanto em complexidade de tempo, quanto em espaço, o algoritmo NW tem desempenho $\mathcal{O}(ij)$, em que i e j correspondem ao comprimento de duas sequências. O NW pode ser dividido em duas fases, sendo elas (1) a fase de cálculo da matriz de programação dinâmica e (2) o *traceback*.

Para construir a matriz de programação dinâmica, dadas as sequências S e R, se obtêm uma matriz H de dimensões $i * j$, em que $R = r_1 r_2 ... r_j$ e $S = s_1 s_2 ... s_i$. Um elemento $H[x, y]$ da matriz possui o escore global até os caracteres r_x e s_y pareados na posição do elemento. A pontuação dos elementos da matriz é o máximo valor obtido a partir de elementos vizinhos cujos escores já foram atribuídos anteriormente, adicionando a pontuação de *match*, *mismatch* ou *gap*, ao depender da situação. A única exceção para essa regra ocorre na primeira coluna e na primeira linha, que são calculados como $H[x, 1] = G * x$ e $H[1, y] = G * y$, respectivamente, em que G corresponde a penalidade do *gap*. Assim, conforme se segue na Equação 2.1 e na Figura 2.2, dado um elemento $H[x, y]$, o seu escore será considerado o valor máximo entre $H[x - 1, y - 1] + p(r_x, s_y)$, em que $p(r_x, s_y)$ corresponde ao valor do *match* ou *mismatch* dos pares r_x e s_y, $H[x - 1, y] + G$ e $H[x, y - 1] + G$.

$$H[x, y] = max \begin{cases} H[x - 1, y - 1] + p(r_x, s_y), \\ H[x - 1, y] + G, \\ H[x, y - 1] + G \end{cases} \tag{2.1}$$

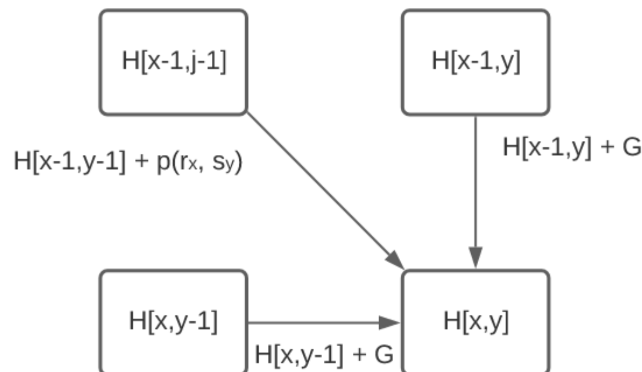


Figura 2.2: Algoritmo NW para calcular o escore de um elemento (Adaptado de [1]).

A medida que são determinados os valores dos elementos da matriz H, são também criados ponteiros que ligam um elemento $H[x, y]$ com os elementos cujos valores ele herdou. Tais ponteiros são essenciais para a segunda fase do algoritmo Needleman-Wunsch, o *traceback*. A partir do elemento no canto inferior direito, onde se localiza o escore global ótimo, é traçado, por meio dos ponteiros, o caminho de volta para a posição superior esquerda, onde se iniciou o alinhamento, listando os *matches* e *mismatches* entre os caracteres das sequências e os *gaps* como um *underscore*. Ao realizar o *traceback*, é possível que haja mais de um caminho que gere o escore do alinhamento global, de forma que possam existir múltiplos alinhamentos globais ótimos resultantes entre duas sequências [9].

A Figura 2.3 apresenta a matriz de programação dinâmica produzida pelo alinhamento de duas sequências $R = GTACTTGAGAGTAC$ e $S = ATCGGATGACACATT$. Para o exemplo, foram adotados os valores $match = +1$, $mismatch = -1$ e $gap = -2$, resultando em um alinhamento de escore global $= -6$. O *traceback* realizado pelo algoritmo é destacado pela sequência de setas que ligam o elemento $H[i, j]$ ao $H[0, 0]$ e o alinhamento produzido é apresentado na Figura 2.1.

	*	A	T	C	G	G	A	T	G	A	C	A	C	A	T	T
*****	**0**	−2	−4	−6	−8	−10	−12	14	−16	−18	−20	−22	−24	−26	−28	−30
G	−2	**-1**	−3	−5	−5	−7	−9	−11	−13	−15	−17	−19	−21	−23	−25	−27
T	−4	−3	**0**	−2	−4	−6	−8	−8	−10	−12	−14	−16	−18	−20	−22	−24
A	−6	−3	−2	**-1**	−3	−5	−5	−7	−9	−9	−11	−13	−15	−17	−19	−21
C	−8	−5	−4	−1	**-2**	← **-4**	−6	−6	−8	−10	−8	−10	−12	−14	−16	−18
T	−10	−7	−4	−3	−2	−3	**-5**	−5	−7	−9	−10	−9	−11	−13	−13	−15
T	−12	−9	−6	−5	−4	−3	−4	**-4**	−6	−8	−10	−11	−10	−12	−12	−12
G	−14	−11	−8	−7	−4	−3	−4	−5	**-3**	−5	−7	−9	−11	−11	−13	−13
A	−16	−13	−10	−9	−6	−5	−2	−4	−5	**-2**	−4	−6	−8	−10	−12	−14
G	−18	−15	−12	−11	−8	−5	4	−3	−3	−4	**-3**	−5	−7	−9	−11	−13
A	−20	−17	−14	−13	−10	−7	−4	−5	−4	−2	−4	**-2**	−4	−6	−8	−10
G	−22	−19	−16	−15	−12	−9	−6	−5	−4	−4	−3	−4	**-3**	−5	−7	−9
T	−24	−21	−18	−17	−14	−11	−8	−5	−6	−5	−5	−4	−5	**-4**	−4	−6
A	−26	−23	−20	−19	−16	−13	−10	−7	−6	−5	−6	−4	−5	−4	**-5**	−5
C	−28	−25	−22	−19	−18	−15	−12	−9	−8	−7	−4	−6	−3	−5	−5	**-6**

Figura 2.3: Matriz de programação dinâmica NW entre duas sequências.

2.3 Algoritmo Smith-Waterman (SW)

Proposto por Smith e Waterman (SW) em 1981 [15], ao contrário do algoritmo Needleman-Wunsch, o algoritmo Smith-Waterman busca o alinhamento local de duas sequências, ou seja, um alinhamento que pode ser obtido *substrings* das sequências. Com complexidade de tempo e espaço $\mathcal{O}(ij)$, sendo i e j o tamanho de duas sequências, o algoritmo SW possui uma estrutura e execução similares ao apresentado pelo NW, diferenciando-se por duas características.

Apesar dos algoritmos SW e NW apresentarem uma ideia similar, existem algumas diferenças quanto às fases de cálculo da matriz de programação dinâmica e do *traceback*. A primeira diferença é que, a fim de se evitar que hajam células da matriz de programação dinâmica com escore negativo e, consequentemente, permitir que sejam encontrados alinhamentos locais, a Equação 2.2 de processamento da matriz de programação dinâmica define que, caso todos os valores herdados das células adjacentes sejam negativos, o escore da célula será 0.

$$H[x,y] = max \begin{cases} H[x-1,y-1] + p(r_x, s_y), \\ H[x-1,y] + G, \\ H[x,y-1] + G, \\ 0 \end{cases} \tag{2.2}$$

A segunda diferença entre o algoritmo SW e NW consiste na posição de início do alinhamento. Ao contrário do algoritmo NW, que sempre inicia o alinhamento global na posição inferior direita, o algoritmo Smith-Waterman inicia o alinhamento na posição $H[x,y]$ que apresenta o maior escore da matriz H, realizando o *traceback* até que seja encontrado um elemento de valor 0. Note que, caso o maior escore da matriz esteja presente em mais de um elemento, é possível obter mais de um alinhamento local ótimo.

Na Figura 2.4 é mostrada a matriz de programação dinâmica produzida pelas sequências $R = GTACTTGAGAGTAC$ e $S = ATCGGATGACACATT$. Os valores adotados são os mesmos utilizados no exemplo do NW, *match* $= +1$, *mismatch* $= -1$ e *gap* $= -2$. Após a matriz estar completa, é determinado que o maior escore é 3 e é realizado o *traceback* até um elemento de valor 0. Note que, no exemplo, existem dois elementos de escore $= 3$, de forma que, para que haja um único alinhamento resultante, é definida uma relação de prioridade entre eles. Com isso dito, a prioridade utilizada para o exemplo foi o maior comprimento, assim, o alinhamento produzido no exemplo segue apresentado na Figura 2.5.

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

Figura 2.4: Matriz de programação dinâmica SW entre duas sequências.

$$\begin{array}{ccccc} T & G & A & G & A \\ T & G & A & C & A \\ 1 & 1 & 1 & -1 & 1 \end{array}$$

Figura 2.5: Alinhamento SW (escore = 3).

2.4 Algoritmo Gotoh

Os algoritmos Needleman-Wunsch (Seção 2.2) e Smith-Waterman (Seção 2.3) utilizam o *linear gap*, ou seja, todos os *gaps* que ocorrem no alinhamento geram a mesma penalidade ao escore, crescendo linearmente de acordo com o número de ocorrências. Ao contrário de seus antecessores, o algoritmo Gotoh [18] calcula o alinhamento global utilizando o *affine gap*, técnica essa que calcula *gap* baseado na penalidade de abertura e penalidade de extensão. Por meio deste método, dado um alinhamento qualquer, a penalidade gerada por uma sequência de k *gaps* pode ser calculada por meio da equação $G(k) = G_{open} + (k-1) * G_{ext}$, em que G_{open} e G_{ext} correspondem a penalidade de abertura e extensão, respectivamente.

Apesar da complexidade de tempo e espaço se manter $\mathcal{O}(ij)$, em que i e j são o tamanho de duas sequências, o algoritmo Gotoh também se diferencia dos anteriores devido ao fato de utilizar três matrizes para realizar o alinhamento. A matriz H, como usual, é utilizada para calcular os *matches* e *mismatches* que ocorrem no alinhamento, enquanto as matrizes auxiliares E e F calculam os gaps da sequência R e S, respectivamente. Para tal, os elementos das matrizes são calculados por meio das equações (2.3), (2.4) e (2.5).

$$H[x,y] = max \begin{cases} H[x-1,y-1] + p(r_x, s_y), \\ E[x,y], \\ F[x,y] \end{cases} \tag{2.3}$$

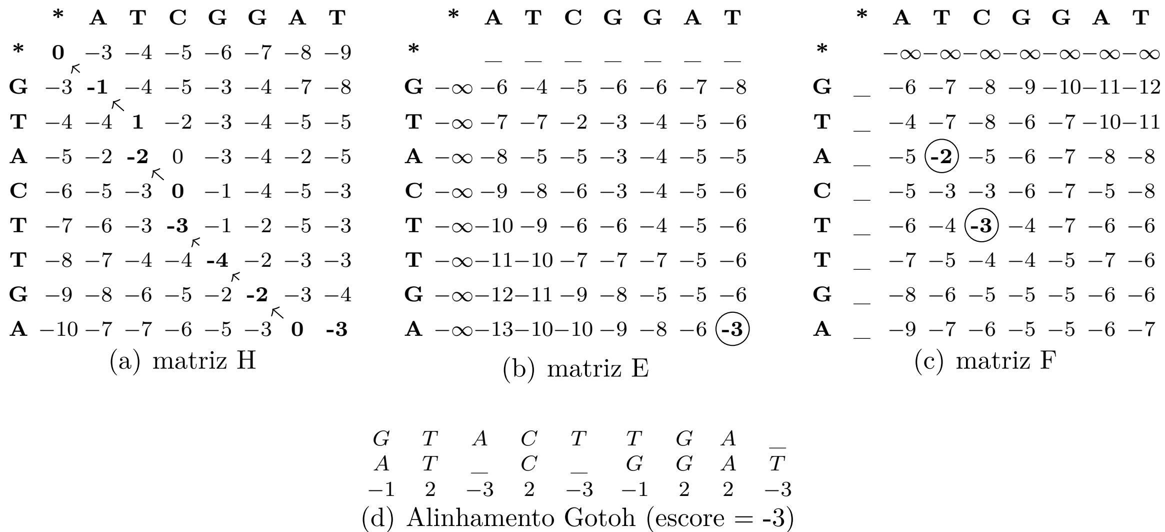
$$E[x,y] = max \begin{cases} E[x,y-1] + G_{ext}, \\ H[x,y-1] + G_{open} \end{cases} \tag{2.4}$$

$$F[x,y] = max \begin{cases} F[x-1,y] + G_{ext}, \\ H[x-1,y] + G_{open} \end{cases} \tag{2.5}$$

A obtenção do alinhamento no algoritmo Gotoh se dá de forma semelhante aos seus antecessores, utilizando ponteiros que ligam os elementos da matriz H com outros elementos a quem eles herdaram o valor. No entanto, caso o escore de um elemento de posição (x, y) seja maior em $E[x, y]$ ou $F[x, y]$, o ponteiro é ligado ao elemento de uma dessas matrizes, de forma que possa ser necessário visitar as duas tabelas auxiliares durante a

fase de *traceback*.

A Figura 2.6 apresenta a execução do algoritmo Gotoh sobre duas sequências $R = GTACTTGA$ e $S = ATCGGAT$. Para o exemplo, foi determinado que *match* $= 2$, *mismatch* $= -1$, $G_{open} = -3$ e $G_{ext} = -1$. A matriz H apresenta o *traceback* realizado para ser obtido o alinhamento, mas é possível notar que foram utilizados escores das matrizes E e F ao longo do caminho, destacados pelos círculos. Assim, o alinhamento global ótimo de escore $= -3$ produzido pelo algoritmo Gotoh é expresso na Figura 2.6(d).



	*	A	T	C	G	G	A	T
*	**0**	−3	−4	−5	−6	−7	−8	−9
G	−3	**-1**	−4	−5	−3	−4	−7	−8
T	−4	−4	**1**	−2	−3	−4	−5	−5
A	−5	−2	**-2**	0	−3	−4	−2	−5
C	−6	−5	−3	**0**	−1	−4	−5	−3
T	−7	−6	−3	**-3**	−1	−2	−5	−3
T	−8	−7	−4	−4	**-4**	−2	−3	−3
G	−9	−8	−6	−5	−2	**-2**	−3	−4
A	−10	−7	−7	−6	−5	−3	**0**	**-3**

(a) matriz H

	*	A	T	C	G	G	A	T
*		_	_	_	_	_	_	_
G	−∞	−6	−4	−5	−6	−6	−7	−8
T	−∞	−7	−7	−2	−3	−4	−5	−6
A	−∞	−8	−5	−5	−3	−4	−5	−5
C	−∞	−9	−8	−6	−3	−4	−5	−6
T	−∞	−10	−9	−6	−6	−4	−5	−6
T	−∞	−11	−10	−7	−7	−7	−5	−6
G	−∞	−12	−11	−9	−8	−5	−5	−6
A	−∞	−13	−10	−10	−9	−8	−6	**-3**

(b) matriz E

	*	A	T	C	G	G	A	T
*		−∞	−∞	−∞	−∞	−∞	−∞	−∞
G	_	−6	−7	−8	−9	−10	−11	−12
T	_	−4	−7	−8	−6	−7	−10	−11
A	_	−5	**-2**	−5	−6	−7	−8	−8
C	_	−5	−3	−3	−6	−7	−5	−8
T	_	−6	−4	**-3**	−4	−7	−6	−6
T	_	−7	−5	−4	−4	−5	−7	−6
G	_	−8	−6	−5	−5	−5	−6	−6
A	_	−9	−7	−6	−5	−5	−6	−7

(c) matriz F

G	*T*	*A*	*C*	*T*	*T*	*G*	*A*	_
A	*T*	_	*C*	_	*G*	*G*	*A*	*T*
−1	2	−3	2	−3	−1	2	2	−3

(d) Alinhamento Gotoh (escore = -3)

Figura 2.6: Matrizes H, E e F e resultado do alinhamento com algoritmo Gotoh.

2.5 Algoritmo Myers-Miller

Com base no trabalho de Hirschberg [19], que trata sobre um algoritmo recursivo com estratégia de divisão e conquista para encontrar a subsequência mais longa em comum em espaço linear (LCS - *Longest Common Subsequence*), e aplicando-o no algoritmo proposto por Gotoh (Seção 2.4), o algoritmo Myers-Miller [2] se mostrou capaz de encontrar um alinhamento ótimo com espaço linear $\mathcal{O}(i + j)$.

O funcionamento do algoritmo MM consiste em calcular a matriz de alinhamento de duas sequências R e S até a linha $\frac{i}{2}$ e armazenar em vetores CC e DD os escores dos alinhamentos que terminam em *match* e *mismatch* e os escores dos alinhamentos que terminam em *gap*, respectivamente. Após isso, é calculado o alinhamento sobre as mesmas sequências, no entanto, invertidas (R' e S') e são armazenados os escores em vetores CC' e DD'. Após serem obtidos os vetores, são calculados os escores dos elementos da linha $\frac{i}{2}$ e coluna y a partir da Equação 2.6, em que G_{open} consiste na penalidade da abertura de um *gap*, em que $G_{open} = G_{first} + G_{ext}$.

$$C_y = max \begin{cases} CC_y + CC'_y, \\ DD_y + DD'_y - G_{open} \end{cases} \tag{2.6}$$

$$C_{y'} = max_{0 \leq y \leq j} C_y \tag{2.7}$$

A partir dos *scores* C_y calculados, é possível encontrar o ponto médio (*crosspoint*), o elemento da matriz por onde passa o alinhamento ótimo. Para tal, basta encontrar o maior valor entre os C_y, i.e., a Equação 2.7. Com o resultado máximo obtido, é correto dizer que o elemento $H[\frac{i}{2}, y']$ é o ponto médio para a matriz utilizada. Então, é utilizada a recursão para executar o algoritmo Myers-Miller sobre as submatrizes H' e H" contidas em H, cujas dimensões consistem em $H[0,0]$ a $H[\frac{i}{2}, y']$ e $H[\frac{i}{2}, y']$ e $H[i,j]$. Dessa forma, são obtidos outros pontos médios para cada submatriz, realizando mais chamadas recursivas subsequentes até que o problema se torne trivial e, por fim, encontrando o alinhamento global ótimo, conforme é apresentado na Figura 2.7.

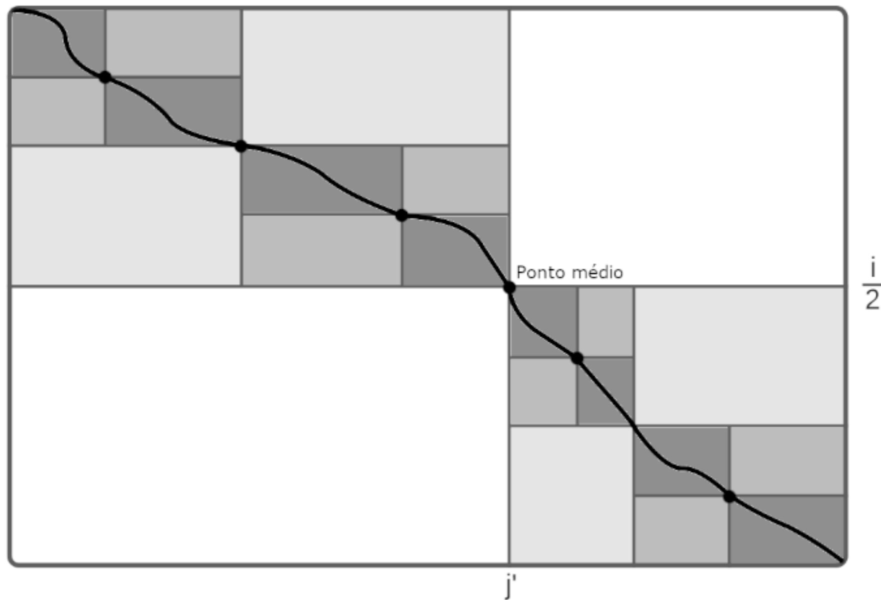


Figura 2.7: Funcionamento do algoritmo Myers-Miller (Adaptado de [2]).

Capítulo 3

GPU e CUDA

Neste capítulo será apresentado o processo de evolução, arquiteturas e componentes das GPUs NVIDIA, e a arquitetura CUDA. A Seção 3.1 apresenta o processo do desenvolvimento das GPUs NVIDIA, da origem à comparação com GPUs atuais. Na Seção 3.2 são destacados os componentes e aspectos mais importantes da GPU NVIDIA GV100. Por fim, a Seção 3.3 apresenta a arquitetura CUDA.

3.1 Evolução das GPUs NVIDIA

As GPUs (*Graphics Processing Unit*) são multiprocessadores otimizados para realizar cálculos voltados para a renderização gráfica [9]. Para aproveitar o grande poder computacional provido por estas placas gráficas, áreas de estudo não relacionadas a renderização gráfica, como a comparação de sequências biológicas, começaram a utilizá-las para resolver problemas genéricos, tornando-as GPGPUs (*General Purpose Graphics Processing Unit*) [10]. Apesar de existirem diversos fabricantes de GPUs, neste capítulo serão apresentadas apenas as GPUs NVIDIA, visto que apenas estas serão utilizadas no trabalho.

Em 1999 foi criada a primeira GPU da NVIDIA, a GeForce 256. Esta GPU foi criada com o foco em processamento de gráficos 3D em tempo real e foi programada por meio das APIs do OpenGL e DirectX 7. As próximas edições das placas gráficas NVIDIA se destacaram por novas funcionalidades relacionadas a renderização gráfica. No entanto, em 2004, juntamente com a placa GeForce 6800 foi introduzida a linguagem Cg, utilizada para a programação de GPUs, o que proveu um modelo de programação escalável e paralelo, e que permitiu a utilização de placas gráficas para problemas genéricos (GPGPU) [20].

No ano de 2006, a GPU GeForce 8800 introduziu primeira arquitetura capaz de unificar a computação e a renderização gráfica. Para isto, foi utilizada a recém-criada arquitetura CUDA (Compute Unified Device Architecture) (Seção 3.3), o DirectX 10 e o OpenGL. Além disso, ao contrário das placas gráficas antecedentes, que utilizavam uma arquitetura

Tabela 3.1: Primeira parte da tabela de especificações. (Adaptado de [10])

Especificações	**Tesla**	**Fermi**	**Kepler**	**Maxwell**	**Pascal**
GPU	GTX 260	GTX 460	GTX 650	GTX 750	GTX 1050
Núcleos	192	336	384	512	640
Clock (MHz)	576	675	1058	1020	1354
TFLOPS	0.477	0.873	0.844	1.076	1.8
Memória (GB)	0.896	1.024	1.024	2.048	2.048

Tabela 3.2: Segunda parte da tabela de especificações. (Adaptado de [10])

Especificações	**Turing**	**Volta**	**Ampere**
GPU	GTX 1650	Tesla V100	Tesla A100
Núcleos	896	5120	6912
Clock (MHz)	1485	1530	1410
TFLOPS	8.08	31.4	78
Memória (GB)	4.096	32	40

baseada em processamento vetorial, a GeForce 8800 foi a primeira GPU a utilizar processadores escalares baseados em *threads* [20]. Desde então, a NVIDIA tem lançado placas gráficas cada vez mais complexas, aprimorando características técnicas como o número de núcleos, *clock*, FLOPS (*Floating-point Operations Per Second*) e memória. Por fim, foram criadas arquiteturas que serviram como base para as diversas placas NVIDIA desenvolvidas ao longo dos anos, sendo elas: Tesla, Fermi, Kepler, Maxwell, Pascal, Turing, Volta e Ampere.

Nas Tabelas 3.1 e 3.2 são apresentadas as especificações de GPUs de cada arquitetura, onde é possível ver a evolução no número de núcleos, memória e desempenho (TFLOPS), indo de 192 núcleos, 0.896 GB de RAM e 0.477 TFLOPS, na arquitetura Tesla, a 6912 núcleos, 40 GB de RAM e 78 TFLOPS na arquitetura mais recente, que é a Ampere (A100).

3.2 Arquitetura da GPU NVIDIA

Com o aumento de complexidade das GPUs produzidas pela NVIDIA, novas gerações de placas gráficas são lançadas para o público. Cada geração de GPU tem sua arquitetura otimizada para melhorar a performance e a eficiência energética, adicionar novas ferramentas e simplificar a programação em GPU [3]. Podendo ser considerada uma causa e consequência deste fato, a utilização de placas gráficas nas mais diversas áreas de estudo é muito impactada pelo aprimoramento de processamento provido por novas arquiteturas. Assim, a fim de apresentar aspectos gerais das GPUs, nesta Seção será utilizada como base a placa gráfica GV100, que utiliza a arquitetura Volta (Tabela 3.2).

Nos anos de 1960, foi proposto um modelo de categorização para a paralelização de *hardware*, conhecido como taxonomia de Flynn. Neste modelo o *hardware* é classificado segundo o número simultâneo de fluxos de instruções e de fluxos de dados. Entre as classificações de Flynn, a que mais se aproxima das GPUs é a SIMD (*Single Instruction Multiple Data*). Neste modelo de paralelização, uma única instrução é executada por múltiplas unidades de processamento sincronizadas e paralelizadas [21], o que serviu como base para o processamento em GPUs. Porém, a partir da GPU GeForce 8800 foi criada uma abstração que permite a utilização de processamento escalar baseado em *threads* (Seção 3.1), originando o modelo SIMT (*Single Instruction Multiple Threads*). O modelo SIMT é usado atualmente para a programação das GPUs NVIDA, no entanto, o modelo de execução continua SIMD.

A GPU GV100 é composta por 6 GPCs (*Graphics Processing Clusters*), os quais são os blocos de *hardware* que realizam o processamento, a rasterização, dentre outros, atribuídos a GPU. Cada um dos GPCs possui 7 TPCs (*Texture Processing Clusters*), que incluem 2 SMs (*Streaming Multiprocessors*), as quais são processadores que executam *threads* de instruções SIMD independentemente dos demais (modelo SIMT) [21]. Um SM contém 64 *cores* FP32, 64 *cores* FP64, 32 *cores* INT32, que são processadores de *shading* com ponto flutuante e precisão simples, ponto flutuante e precisão dupla, e inteiro, respectivamente. Além disso, um SM possui 8 *Tensor Cores*, utilizados para processamentos relacionados a tensor, e 4 unidades de textura. Por fim, a GPU GV100 também apresenta 8 controladores de memória de 512 bits [3]. A Figura 3.1 apresenta a arquitetura de uma SM da placa gráfica GV100.

Em adição aos componentes de processamento da GPU GV100, também está presente uma hierarquia complexa de memória. Cada SM possui uma memória *cache* L1 privada, separada em partes de dados, constantes e instruções, enquanto a *cache* L2 é compartilhada por todas SMs da GPU e é unificada, ou seja, as informações não estão separadas em partes. Ademais, bancos de registradores, de tamanho de 64 bits, e *caches* L0 de instruções estão presentes e são privados para cada bloco de processamento. Por fim, também estão inclusas TLBs (*Translation Lookaside Buffer*) e uma DRAM (*Dynamic Random Access Memory*) na GPU [4]. A Figura 3.2 apresenta a hierarquia de memória, suas camadas e peças que a compõe.

Por último, foi introduzida na GPU P100, da arquitetura Pascal, a nova memória HBM2 (*High Bandwidth Memory 2*) [22]. Esta memória utiliza uma estrutura de pilha, com múltiplas camadas de DRAM empilhada (3D), atingindo a capacidade de 16 GB de memória e 720 GB por segundo de largura de banda. Além disso, a memória HBM2 possui suporte nativo a ECC (*error correcting code*), compondo uma camada de segurança adicional para aplicações, detectando e corrigindo erros de corrupção de bits.

Figura 3.1: *Streaming Multiprocessor* da GPU GV100 (Fonte: [3]).

3.3 CUDA

Em 2006 foi lançada pela NVIDIA o CUDA (*Compute Unified Device Arquitecture*), o qual é uma arquitetura de *software* e *hardware* para gerenciar a utilização de uma GPU como uma unidade de processamento com paralelização de dados. De forma que o processamento da GPU deixasse de ser focado apenas em aplicações gráficas, expandindo suas utilizações para outras áreas, tornando-as GPGPUs. Para tal, os mecanismos de multitarefa gerenciam o acesso das aplicações CUDA e gráficas na GPU [23].

A arquitetura CUDA tem sua estrutura representada na Figura 3.3. O bloco verde com numeração 1 consiste nas unidades de computação de GPU empregada. O módulo 2 tem como objetivo realizar a comunicação com o *kernel* do sistema operacional, que provê a inicialização, configuração, entre outros, do hardware. Quanto ao módulo 3, este corresponde ao *driver* em modo de usuário do CUDA, que dá suporte a uma API de baixo

Figura 3.2: Hierarquia de memória da GPU GV100 (Fonte: [4]).

nível para desenvolvedores. Além disso, nele está contido o módulo 4, que consiste em uma ISA (*Instruction Set Architecture*) para programação paralela para propósitos gerais, chamada PTX (*Parallel Thread Execution*), que é similar ao *Assembly*. Por último, são apresentadas as APIs de alto nível, como DirectX, OpenCL, C etc [10].

Figura 3.3: A arquitetura do CUDA (Fonte: [5]).

A programação em CUDA foi criada para permitir o uso escalável de um grande número de núcleos, a medida que placas de vídeo com cada vez mais unidades de proces-

Listing 3.1: Exemplo do CUDA C.

```
__global__ void hello() {
    printf("Hello world!");
}

int main() {
    hello<<<4, 2>>>();
}
```

samento são lançadas. Em adição a isto, a fim de simplificar seu aprendizado e utilização, o CUDA estende linguagens básicas como C e C++ [6].

O CUDA C, que estende a linguagem C, permite que sejam criadas funções, chamadas *kernel*, que são executadas em paralelo pelas múltiplas *threads* CUDA que possam existir. Uma função é definida como um *kernel* por meio do especificador __global__. Além disso, outras características que definem um *kernel* são (1) que estas funções possuem tipo de retorno *void*, (2) chamadas a *kernels* são assíncronas, de forma que possa ser retornado antes destas funções serem completadas, e (3) chamadas devem especificar sua configuração de execução, ou seja, a dimensão da *grid*, blocos e *threads* que serão utilizados.

Conforme citado anteriormente, um *kernel* pode ser chamado com configurações de execução diferentes. Tais configurações se baseiam na hierarquia de *threads* existente no CUDA. Uma *thread* é identificada no *kernel* por meio de um identificador *threadIdx*, o qual composto por um vetor de 3 dimensões. Desta forma, todas as *threads* contidas nestas dimensões do vetor formam um bloco de *threads*. Além disso, em um *kernel* também é possível executar blocos de mesmo tamanho, novamente organizados em 3 dimensões, de forma que possam ser agrupados em um *grid*. Assim, uma configuração de execução de *kernel* pode ser composta por um *grid*, que contém diversos blocos de *threads* e que, por sua vez, contém múltiplas *threads* que executam a função em paralelo [6].

No Código 3.1 é apresentado um exemplo básico da utilização do CUDA C. Nele é possível notar a definição de um *kernel* por meio do especificador __global__, o qual apenas imprime a string "Hello world!". Já na função *main*, o *kernel* é chamado com quatro blocos de duas *threads*, resultando em oito impressões da *string*.

A hierarquia de memória empregada pelo CUDA pode ser observada pela Figura 3.4. Cada uma das *threads* possui uma memória local, de forma que apenas a *thread* que a detém tem acesso. Os blocos também possuem uma memória compartilhada entre as *threads* que os compõem. Ademais, todas as *threads* das *grids* podem acessar a memória global. Por fim, existem as memórias *read-only*, constante e de textura, que, assim como a global, são mantidas ao longo da execução de um programa, mesmo com várias chamadas

de *kernel*.

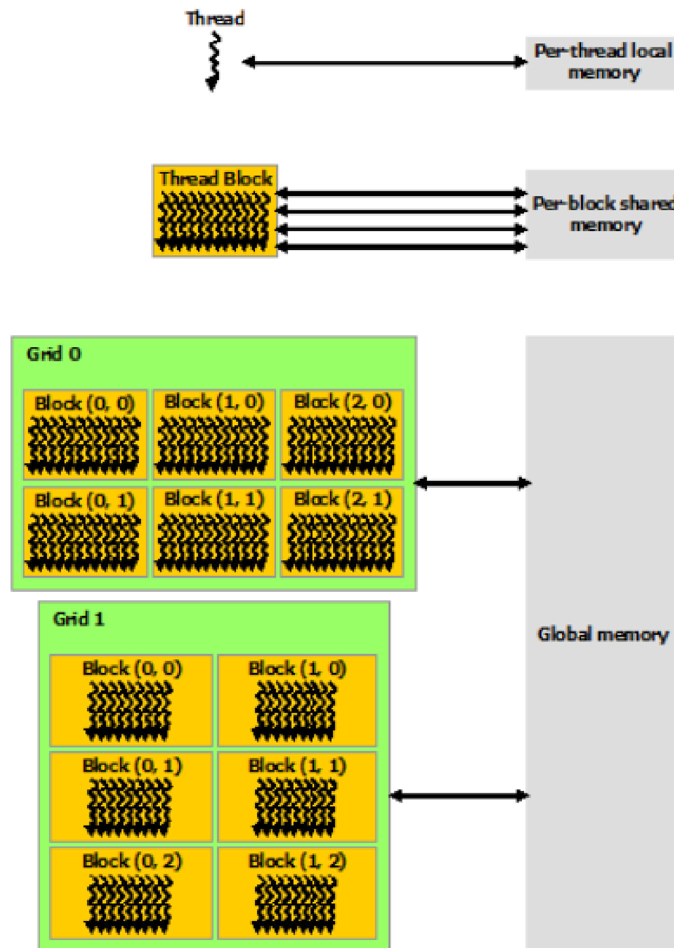


Figura 3.4: Hierarquia de memória do CUDA (Fonte: [6]).

3.3.1 Funcionalidades do CUDA

O CUDA é uma arquitetura em constante mudança e tem sido lançadas novas versões e atualizações com uma alta frequência nos últimos anos. Em cada uma das versões do CUDA, são adicionadas funcionalidades que objetivam melhorar o desempenho da GPGPUs que utilizam esta arquitetura. Assim, nesta subseção serão apresentadas algumas das funcionalidades contidas no CUDA, focando em duas em específico: o *Unified Memory* e o NVLink.

No início da arquitetura CUDA, a comunicação a partir de memória entre a GPU e a CPU era complexa. Isto se deve pelo fato de que quaisquer dados necessários de serem compartilhados entre a CPU e a GPU precisavam ser explicitamente transferidos entre suas memórias. Para simplificar este processo, o UM (*Unified Memory*), introduzido ao

CUDA em sua versão 6.0, consiste em uma ferramenta que permite a criação de uma abstração de memória que pode ser acessada tanto pela GPU quanto pela CPU, sem a necessidade de cópias de dados [24].

O *Unified Memory* consiste em um único endereço de memória ao qual pode ser referenciado por qualquer uma das unidades de processamento do sistema. Para tal, o UM utiliza um mecanismo de migração de páginas capaz de copiar páginas da memória de CPU para a memória da GPU de acordo com a necessidade. As GPUs contém TLBs (*Translation Lookaside Buffer*) que auxiliam na tradução dos endereços, de forma que, ao ocorrer um *page fault*, esta exceção é logo tratada e a página faltante é migrada, de maneira transparente ao programador [25]. Assim, o principal resultado obtido pela adição do *Unified Memory* foi a simplificação do modelo de programação, devido a mudança na gerência de memória [7].

No trabalho de Landaverde et al. [7] foram realizados experimentos para determinar as implicações causadas pela utilização do UM em diversos ambientes. Nele foi constatado que a performance do UM varia ao depender do acesso à memória, o modelo de programação e se a aplicação utiliza muito o *kernel*, que pode gerar uma queda de performance devido à migração de dados. Assim, apesar de proporcionar vantagens quanto a simplificação na manipulação das memórias da GPU e CPU, o *Unified Memory* pode resultar em um alto *overhead* caso haja uma grande quantidade de *page faults*.

A Figura 3.5 apresenta uma representação da memória sem e com o *Unified Memory* do ponto de vista do programador. O sistema sem o UM apresenta duas memórias completamente separadas, necessitando de transferências explicitas de dados para que haja uma comunicação, representada pela seta vermelha. Por outro lado, no modelo com UM é possível considerar a memória como um bloco único, compartilhado por ambos os processadores.

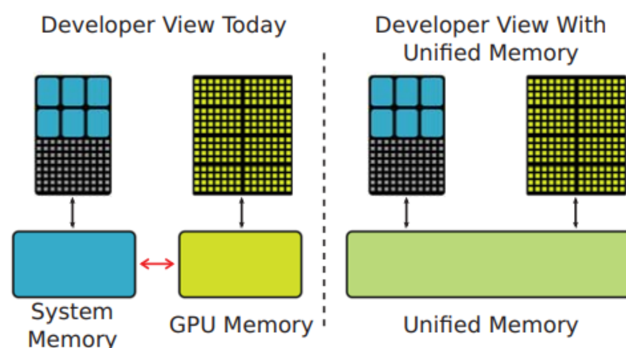


Figura 3.5: Representação de um sistema sem e com o UM (Fonte: [7]).

Além da simplicidade fornecida pelo *Unified Memory*, o CUDA necessitava de uma

interface de comunicação capaz de realizar as trocas de informação de forma eficaz e eficiente. Para tal, em sua versão 8.0, o CUDA adicionou a compatibilidade com a ferramenta NVLink. O NVLink consiste em uma interface de comunicação baseada em fio, com uma alta largura de banda e que intermedeia a comunicação entre GPU e GPU. Esta ferramenta permite que sejam criados *clusters* de GPUs e CPUs, podendo considerá-los um único elemento de processamento [26].

O NVLink é constituído por oito enlaces, os quais são bidirecionais e diferenciais. Os enlaces podem ser organizados de diferentes formas, podendo ser utilizados em conjunto para unir uma GPU e uma CPU diretamente, ou utilizados separadamente para ligar entre múltiplas GPUs e a CPU. Aliada ao fato de que cada enlace é capaz de enviar até 20 Gbits por segundo, a flexibilidade do NVLink permite a comunicação entre as unidades de processamento possa ser arquitetada da melhor forma a maximizar o aproveitamento das GPUs [8]. A Figura 3.6 apresenta exemplos de organização de *clusters*, em que o de 2 GPUs utiliza todos os enlaces do NVLink para a troca de informações entre elas, enquanto a comunicação com a CPU é feita a partir do PCIe (*PCI-Express*). Os demais exemplos mostram outras possíveis arquiteturas ao utilizar mais GPUs.

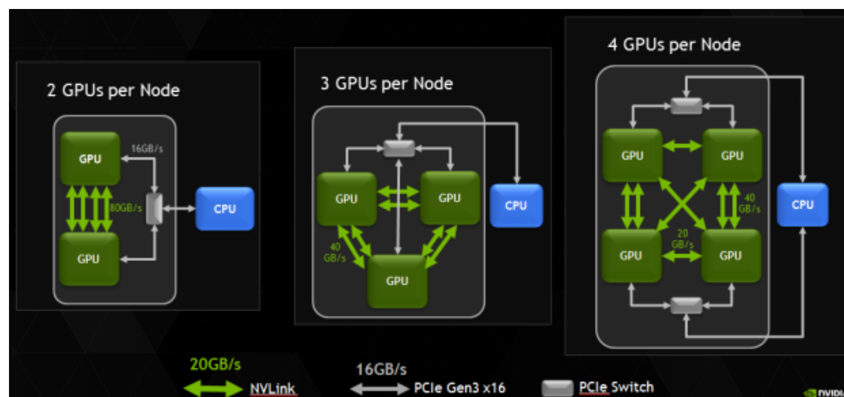


Figura 3.6: Exemplos de arquiteturas utilizando o NVLink (Fonte: [8]).

Citado anteriormente como um fator de diminuição de desempenho do UM, a migração de dados entre memórias se torna muito mais rápida ao utilizar o NVLink. Isto se deve a capacidade de envio que cada enlace possui. Ademais, o *overhead* gerado pelos *page faults* também pode ser severamente diminuído por meio do NVLink. Assim, o *Unified Memory* em conjunto ao NVLink permite que o processamento de aplicações CUDA utilizando GPGPUs seja feito de forma mais simples e eficiente.

Por fim, na Tabela 3.3 são apresentadas outras funcionalidades e ferramentas presentes no CUDA e uma breve descrição da função de cada uma.

Tabela 3.3: Descrição de ferramentas do CUDA.

Ferramenta	**Descrição**
NVIDIA Visual Profiler [27]	Consiste em uma ferramenta de visualização e análise de desempenho para a otimização de aplicações CUDA em C e C++. A partir dela, é possível encontrar possíveis gargalos de performance e receber recomendações de mudanças para melhorar a performance. Além disso, é possível configurar a geração de diversos gráficos e tabelas para melhor apresentar os resultados.
CUDA-GDB [28]	Esta ferramenta é usada para depuração de código CUDA, tanto em partes de código executadas em CPU quanto em GPU. De forma similar ao GDB utilizado em aplicações comuns, o CUDA-GDB também apresenta uma interface a partir de linhas de comando, necessitando apenas o aprendizado de alguns comandos de depuração adicionais, além dos contidos no GDB original.
CUDA-MEMCHECK [29]	Esta ferramenta visa a identificação de erros causados por acesso de memória indevidos. Além disso, o CUDA-MEMCHECK consegue monitorar o acesso de memória por milhares de *threads* executando em paralelo na GPU.
GPUDirect RDMA [30]	Consiste em uma tecnologia que permite a troca de dados diretamente entre uma GPU e outra em sistemas remotos, com o auxílio de um PCIe. Para tal, o RDMA (*Remote Direct Memory Access*) permite que o PCIe periférico tenha acesso direto à memória da GPU.
Cooperative Groups [31]	Com o objetivo de obter uma maior eficiência e sincronização no paralelismo de *threads*, este modelo de programação foi criado para permitir que o *kernel* do CUDA possa organizar dinamicamente grupos de *threads*. Desta forma, todas as *threads* deste grupo podem realizar uma mesma operação coletiva.
Multi-Instance GPU [32]	Anunciada para a versão 11.0 do CUDA, esta ferramenta permite que a GPU A100 seja dividida em múltiplas GPUs isoladas. Desta forma, podem existir múltiplos usuários executando processos e máquinas virtuais simultaneamente, cada qual recebendo erros isoladamente, ou seja, sem afetar os demais.

Capítulo 4

A Ferramenta MASA

Nesse capítulo é apresentada a ferramenta MASA. A Seção 4.1 apresenta o desenvolvimento da ferramenta MASA-CUDAlign, da versão 1.0 à 3.0. Na Seção 4.2 mostra a ferramenta MASA-CUDAlign em sua versão 4.0. A Seção 4.3 apresenta a camada MultiBP adicionada a ferramenta. Na Seção 4.4 é destacado outra extensão da arquitetura MASA, o MASA-OpenMP. Por último, a Seção 4.5 apresenta o conjunto de serviços AWS, utilizados para executar a ferramenta MASA em nuvem.

4.1 A Evolução da Ferramenta

O CUDAlign foi uma ferramenta proposta por Edans Sandes [9] para comparar sequências longas de DNA por meio de GPUs. Esta ferramenta baseou-se no algoritmo Smith-Waterman (Seção 2.3) com *affine gap* e utilizou a arquitetura CUDA como plataforma.

Desenvolvida em 2011, a versão 1.0 do CUDAlign [33] obtinha um paralelismo em seu processamento devido a técnica de *wavefront*. Em um processamento em colunas, as células são agrupadas em blocos de tamanho predefinido e, partir desses blocos, a matriz é processada coluna a coluna. Entretanto, há uma dependência entre blocos, pois um bloco inferior necessita dos valores providos pelo bloco imediatamente superior, de forma que a paralelização seja prejudicada. Para solucionar esta questão, o CUDAlign adotou o *wavefront* em blocos, técnica essa em que a matriz é processada em ondas anti-diagonais pois, dessa forma, não há uma dependência entre blocos de uma mesma anti-diagonal. Além do paralelismo obtido entre os blocos, considerado externo, há também o paralelismo interno, no qual as células que compõem o bloco são processadas por T *threads* e, assim como os blocos, por meio da técnica de *wavefront*. Conforme mostra a Figura 4.1, as células de cada anti-diagonal podem ser processadas de forma paralela pelas *threads*, visto que as células de uma *thread* não dependem do valor de células de outra *thread*.

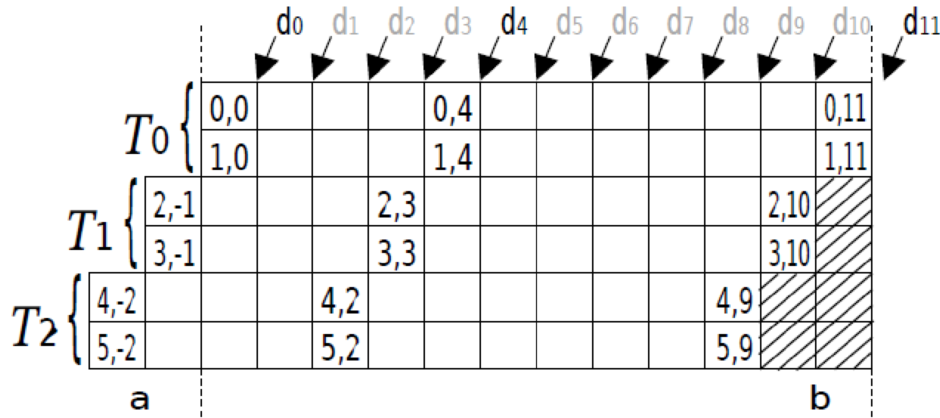


Figura 4.1: Processamento *wavefront* em blocos paralelogramos (Fonte: [9]).

O ponto inovador trazido pelo CUDA-Align 1.0 foi a adoção do formato de bloco em paralelogramos. A utilização deste formato de deve ao fato de que blocos retangulares possuem anti-diagonais mais curtas no seu começo e fim, assim, prejudicando a paralelização, visto que *threads* ficarão ociosas enquanto outras terminam suas computações. Ao passo que o bloco em paralelogramo permite que todos as anti-diagonais possuam o mesmo comprimento, utilizando ao máximo as *threads* alocadas. A Figura 4.1 apresenta um bloco em formato de paralelogramo, em que todas as anti-diagonais apresentam o mesmo comprimento e que as células hachuradas fazem parte do bloco seguinte.

Um problema proveniente da utilização de blocos em paralelogramo é a dependência do processamento de blocos da mesma diagonal, conforme mostra a Figura 4.2. Sejam dois blocos 1 e 2, em que 1 localiza-se imediatamente acima de 2, ao terminar o processamento da diagonal que contém 1, haverá uma área ainda não processada da qual 2 depende, pois essa área será calculada durante a execução diagonal de 2, conforme destacado na área cinza na Figura 4.2. Para evitar que esse problema ocorra, o CUDAlign 1.0 foi dividido em duas fases: curta, em que se calcula as $T-1$ anti-diagonais internas de todos os blocos, processando todas as células que causam dependência, e longa, em que se processa as demais anti-diagonais. Com essa solução, se garante que não haverá problemas quanto a dependência entre os blocos da mesma diagonal.

Como próximo passo do CUDAlign, foi desenvolvida por Edans Sandes e Alba Melo [34] a versão 2.0 da ferramenta. Baseado no algoritmo de Myers-Miller (Seção 2.5), o CUDAlign 2.0 tem como objetivo obter incrementalmente coordenadas por onde passa o alinhamento ótimo, até ser obtido o alinhamento completo a partir de uma quantidade restrita de memória e com um menor número de reprocessamentos realizados na matriz [9]. Para tal, a execução da ferramenta foi dividida em seis estágios:

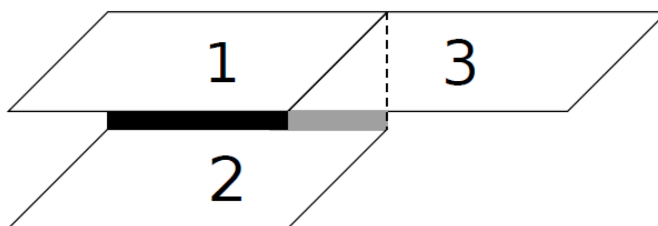


Figura 4.2: Dependência entre blocos de uma mesma diagonal (Fonte: [9]).

1. **Obtenção do Escore Ótimo**: de forma similar ao CUDAlign 1.0, a matriz é processada e tem seu escore ótimo e coordenada encontrados a partir do algoritmo Gotoh (Seção 2.4). Além disso, linhas especiais da matriz são armazenadas em disco, que serão utilizadas no estágio 2.

2. ***Traceback* Parcial**: nessa etapa é realizado o alinhamento semi-global na direção inversa, a partir da coordenada do escore ótimo. Então são obtidas coordenadas (*crosspoints*) por onde o alinhamento ótimo atravessa as linhas especiais encontradas no estágio 1 e, durante esse processo, são salvas colunas especiais. Além disso, foram propostas duas novas otimizações para este estágio:

    - ***Matching* baseado em objetivo**: baseando-se na ideia de buscar a célula da linha central provida pelo algoritmo de Myers-Miller (Seção 2.5), o estágio 2 realiza esse mesmo processo nas linhas especiais. No entanto, como já é conhecido o escore máximo, o procedimento de busca pela coordenada das linhas especiais pode ser encerrado ao encontrá-lo, assim, evitando cálculos posteriores desnecessários.
    - **Execução ortogonal**: para encontrar as coordenadas onde há o cruzamento entre linhas especiais e alinhamento ótimo, as *threads* são executadas verticalmente na matriz, evitando que sejam processadas áreas além da coordenada.

3. **Divisão de Partições**: de forma similar ao estágio 2, este estágio tem como objetivo aumentar o número de coordenadas onde há o cruzamento entre alinhamento ótimo e colunas especiais salvas no passo anterior. Outra característica que difere os dois passos é o fato de que o estágio 3 apresenta partições com dimensões bem definidas, algo que não ocorre no estágio 2, visto que sua coordenada inicial não é sabida a priori.

4. **Myers-Miller otimizado**: a partir das partições obtidas pelo passo 3, o estágio 4 executa o algoritmo Myers-Miller (Seção 2.5) em CPU até que o tamanho das

partições obtidas seja menor que uma constante predefinida, aumentando o número de coordenadas (*crosspoints*). Ademais, nesse estágio também foram propostas duas novas otimizações:

- **Divisão Balanceada**: ao contrário do algoritmo Myers-Miller usual, que utiliza a divisão da matriz na linha central, esta otimização propõe a divisão a partir da linha ou coluna central ao depender da maior dimensão da matriz, o que evita a existência de partições muito desproporcionais.
- **Execução Ortogonal**: por já se saber os escores das coordenadas inicial e final da partição, é possível obter o escore ótimo a partir da diferença entre eles. Dessa forma, é possível dividir em duas metades para serem processadas, de forma que a primeira metade seja completamente calculada, enquanto que a segunda seja calculada até o escore ótimo ser encontrado. Assim, havendo uma melhoria de desempenho devido a área não processada.

5. **Obtenção do alinhamento completo**: por meio do algoritmo Needleman-Wunsch (Seção 2.2), esse estágio realiza o alinhamento em CPU das partições obtidas pelo passo 4. Por fim, os alinhamentos resultantes são concatenados, obtendo o alinhamento ótimo completo e o arquivo binário que o representa.

6. **Visualização**: a partir do arquivo binário resultante do estágio 5, o passo 6 é utilizado para realizar a visualização textual e gráfica do alinhamento ótimo.

O próximo avanço proposto para a ferramenta CUDAlign foi a adoção da técnica de descarte de blocos, também conhecido como *Block Pruning* (BP), o que gerou o CUDAlign 2.1 [35]. O descarte de blocos permite um melhor desempenho no processamento da matriz do alinhamento pois, a partir dele, pode-se ignorar blocos por onde seria impossível obter o alinhamento ótimo. Para tal, essa técnica é aplicada no estágio 1, introduzido no CUDAlign 2.0.

Dada uma matriz H com m linhas e n colunas, e uma célula $H[i,j]$, as distâncias de $H[i,j]$ até a última linha e coluna da matriz são $\Delta_i = m - i$ e $\Delta_j = n - j$. Dessa forma, verifica-se que o maior escore possível derivado a partir de $H[i,j]$ é $H_{máx}[i,j] = H[i,j] + min(\Delta_i, \Delta_j) * M$, em que M é o valor do *match*. Assim, sendo $best(i)$ o maior escore da linha i, é possível descartar a célula $H[i,j]$ caso $H_{máx}[i,j] \leq best(i)$, visto que se provaria impossível um alinhamento ótimo contê-la. A Figura 4.3 representa a questão do cálculo do maior escore possível ser obtido a partir de uma célula $H[i,j]$.

Como última melhoria detalhada nesta Seção, o CUDAlign 3.0 [36] foi criado para permitir que a etapa 1, a mais demorada, possa ser executada em múltiplas GPUs. Foi proposto que a matriz fosse dividida em agrupamentos de colunas para cada GPU, vari-

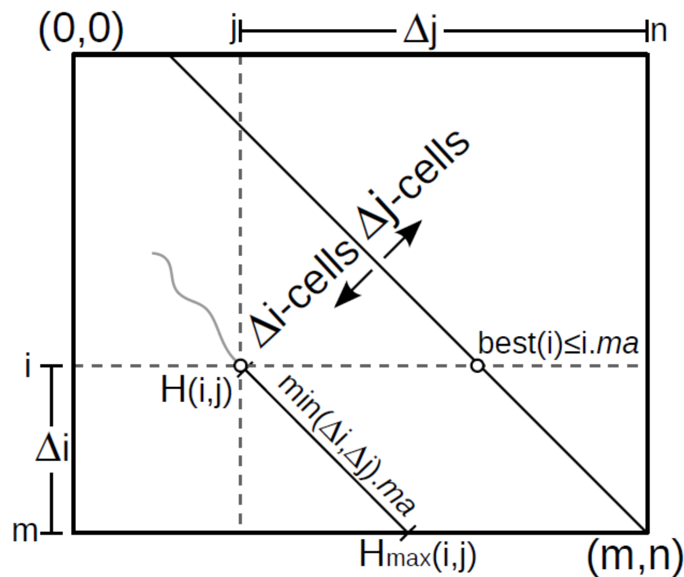


Figura 4.3: Representação da obtenção do maior escore (Fonte: [9]).

ando a quantidade de colunas ao depender da homogeneidade de GPUs e seus desempenhos. Conforme mostra a Figura 4.4, a matriz de programação dinâmica foi dividida, a partir de suas colunas, entre quatro GPUs. Além disso, é possível notar a dependência que existe entre as GPUs, visto que os valores da coluna mais à direita da GPU (coluna de borda) são necessários para a GPU vizinha à direita, resultando em uma dependência entre as GPUs.

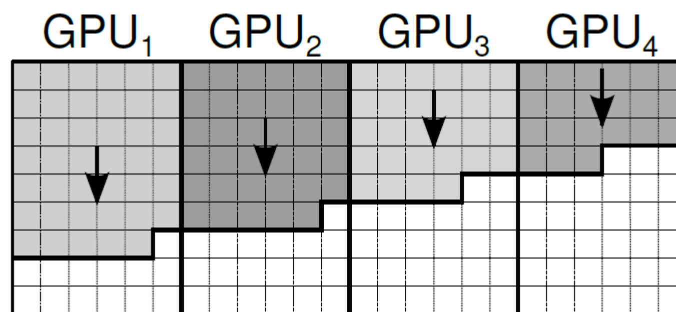


Figura 4.4: Processamento da matriz em múltiplas GPUs (Fonte: [9]).

A arquitetura proposta é composta por GPUs que são associadas a um processo, o qual contêm três *threads*, sendo uma *thread* gerente, que gerencia a execução na GPU e transfere as células entre GPU e *threads* de comunicação, e duas *threads* de comunicação, que transferem as células entre os processos, por meio de *sockets*. Ademais, cada *thread* de comunicação é associada a um *buffer*, o qual é utilizado para entrada e saída. A utilização

de tais *buffers* permite que se esconda a latência de comunicação que há entre as GPUs, assim, possíveis variações de desempenho que possam ocorrer não afetarão o desempenho do *wavefront*.

4.2 MASA-CUDAlign 4.0

Até a versão 3.0 a execução do CUDAlign em múltiplas GPUs se limitava apenas ao primeiro estágio. Então, com o intuito de paralelizar a recuperação de alinhamentos ótimos, foi criado por Edans Sandes e co-autores o CUDAlign 4.0 [9]. Esta versão do CUDAlign trouxe mudanças significativas para a ferramenta, sobretudo nos estágios 2 a 4 criados em sua versão 2.0 (Seção 4.1). A primeira modificação está presente no estágio 1, onde foi adotado o subparticionamento, técnica essa que define um tamanho máximo para subdivisões da matriz de programação dinâmica, as quais são calculadas individualmente e em paralelo, seguindo os mecanismos já utilizados nas versões anteriores do CUDAlign. Além disso, cada uma das GPUs agora possui um diretório específico para armazenar as linhas especiais obtidas na etapa 1 e, a fim de diminuir o *overhead* causado por operações de leitura e escrita em disco, as linhas especiais também podem ser salvas em memória RAM.

A principal funcionalidade proposta no CUDAlign 4.0 foi a adoção de duas novas estratégias de *traceback*:

- ***Pipelined Traceback* (PT)**: nesta técnica, o estágio 1 modificado é executado em múltiplas GPUs de forma semelhante ao que foi apresentado no CUDAlign 3.0 (Seção 4.1). Após a última GPU (n) terminar a execução do estágio 1 na última partição, nela será iniciada a computação do estágio 2. Neste estágio, à medida que são encontrados os *crosspoints* desta partição, eles são enviados a GPU vizinha ($n-1$), a qual realizará o mesmo processo e enviará seus *crosspoints* a GPU $n-2$, assim, criando uma cadeia de dependência entre as GPUs. O mesmo será feito nas etapas 3 e 4, de forma que, resolvidas suas dependências, os estágios podem ser executados em *pipeline*, exceto nos estágios 5 e 6, onde o *pipeline* não é necessário. A Figura 4.5 representa a execução com PT, em que o eixo x do gráfico lista as GPUs, de 1 a n, e o eixo y representa o tempo. Os valores t_a e t_1 simbolizam o tempo antes e durante o cálculo do estágio 1 na GPU i, enquanto t_b representa o tempo ocioso causado pela dependência do estágio 2. A anti-diagonal iniciada em GPU n representa a execução dos estágios 2, 3 e 4, onde é possível ver a cadeia de dependência gerada entre as GPUs. Por fim, a GPU 1 executa a etapa 5 e 6 sem paralelização.

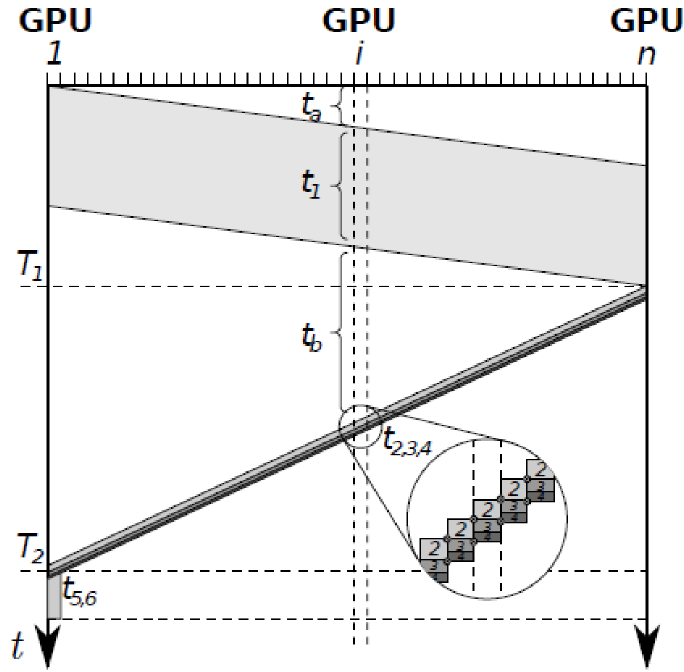


Figura 4.5: Execução com *traceback* com *pipeline* (Fonte: [9]).

- ***Incremental Speculative Traceback*** **(IST)**: esta estratégia se baseia na especulação da localização dos *crosspoints* que serão obtidos no estágio 2. Cada GPU, enquanto espera para executar o estágio 2, realiza estimativas sobre o *crosspoint* que receberá de sua vizinha e computa a partição a partir do valor estimado. À medida que as GPUs recebem os *crosspoints*, é verificado se o especulado é igual ao real. Caso a estimativa esteja correta a GPU i envia imediatamente à GPU $i-1$ os *crosspoints* calculados na fase de especulação e executa os demais estágios em *pipeline*. Caso não, será necessário recalcular a partição com o crosspoint verdadeiro. A estimativa se baseia no fato de *crosspoints* usualmente coincidirem com os escores máximos das colunas especiais, e na técnica de cada GPU redirecionar o *crosspoint* especulado para outras GPUs, assim, fornecendo novas informações para elas. A partir destas informações, a GPU que as recebeu pode especular com mais detalhes acerca do *crosspoint*, de forma que a probabilidade de acerto seja aumentada ao longo do tempo. Esta ideia consiste na especulação incremental.

  A Figura 4.6 apresenta a execução com IST que, diferentemente da Figura 4.5, é possível notar que, devido a especulação de *crosspoints*, o cálculo da etapa 2 é realizado em seguida da etapa 1, representado pela diagonal paralela a diagonal da etapa 1. Após a execução da etapa 1 na GPU n, é possível verificar se os *crosspoints* estimados são corretos. Caso esteja certo, as etapas 3 e 4 podem ser executadas em

pipeline. Caso contrário, é necessário recalcular a etapa 2 da partição. As múltiplas barras anti-diagonais simbolizam as recalculações e as tantas especulações realizadas pelas partições, compondo a especulação incremental.

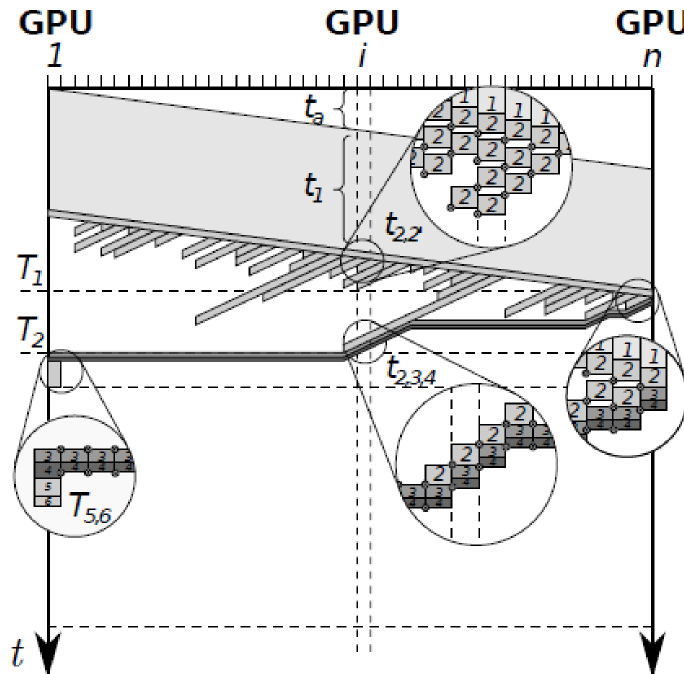


Figura 4.6: Execução com o *traceback* especulativo (Fonte: [9]).

O código do CUDAlign apresentou, em sua grande maioria, uma independência de plataforma, ou seja, podendo ser executado nas mais diversas plataformas e necessitando que o programador codifique apenas a menor parcela do código, que é dependente de plataforma. Então, para simplificar a utilização da ferramenta CUDAlign e similares nas várias plataformas, foi desenvolvida a arquitetura *Multi-platform Architecture for Sequence Aligners* (MASA) [9]. A Figura 4.7 apresenta os módulos que compõem o MASA, em que as gerências de dados e de estágios, comunicação e estatísticas são independentes, enquanto que o *aligner* é dependente da plataforma. Assim, seguem mais detalhes acerca de cada módulo da arquitetura:

- ***Aligner***: este módulo é responsável pelo cálculo da matriz de programação dinâmica a partir do alinhamento SW (Seção 2.3) ou NW (Seção 2.2), os quais são um dos poucos códigos que necessitam ser adaptados de acordo com a plataforma, ou seja, são dependentes. Além disso, neste módulo é possível escolher o tipo de processamento e a técnica de descarte de blocos (*Block Pruning*) a serem utilizados.

- ***Gerência de Dados***: módulo responsável pela gerência de entrada e saída de dados da ferramenta, como leitura de sequências, armazenamento de linhas e colunas especiais, geração de arquivos binários etc.

- ***Estatísticas***: a partir deste módulo é possível obter informações acerca do desempenho da ferramenta, como tempo de execução, gasto de memória e disco etc.

- ***Gerenciamento dos Estágios***: módulo responsável por coordenar a execução dos estágios da ferramenta, em que os estágios 1 a 3 são executados no módulo *Aligner* e os demais em CPU. Ao longo do processamento da matriz de recorrência no *Aligner*, a matriz é dividida em partições a partir das linhas e das colunas especiais salvas nos estágios 2 e 3. Ao fim do cálculo de uma partição, é encontrado o *crosspoint* e uma nova partição é enviada ao *Aligner*, repetindo este processo até que as extremidades do alinhamento sejam encontradas em cada um dos estágios.

- ***Comunicação***: módulo responsável pela comunicação entre os nós de processamento, seguindo a ideia do CUDAlign 3.0 (Seção 4.1).

Figura 4.7: Arquitetura MASA (Fonte: [9]).

Conforme o objetivo de permitir que a ferramenta CUDAlign e similares pudessem ser utilizadas nas mais diversas plataformas, a arquitetura MASA possibilita a adição de extensões. Para se criar uma extensão MASA é necessário criar uma classe *Aligner* e instanciá-la para o ponto de entrada da arquitetura, de forma que o código independente

gerencie como e quando esta classe será utilizada. Além disto, foram propostas em [9] quatro extensões previamente implementadas, sendo elas: MASA-OpenMP em CPU e em Intel Phi, que utilizam a implementação de *multithreading* OpenMP (Seção 4.4), MASA-OmpSs em CPU e o MASA-CUDAlign, que executa em GPU. Além disso, em [37], foi proposta a extensão MASA-OpenCL, que se executa em GPU e CPU. Por fim, em [38], foi proposta a extensão MASA-StarPU, que se executa em CPU utilizando as múltiplas políticas de escalonamento do StarPU [39].

4.3 MASA-CUDAlign-MultiBP

Em sua versão 2.1, o CUDAlign adicionou a possibilidade de realizar *Block Prunning* na matriz de programação dinâmica por meio da técnica de *wavefront*. Porém, com o foco em realizar a comparação de sequências em múltiplas GPUs, as versões seguintes, 3.0 e 4.0, não expandiram o descarte de blocos para ser executado em paralelo entre GPUs. Para tal, Marco Figueiredo e co-autores desenvolveram o Static-MultiBP [10], uma nova camada adicionada ao MASA-CUDAlign capaz de fazer o descarte de blocos em um ambiente multi-GPU com carga estática, ou seja, cada GPU processando uma mesma carga do início ao fim do processamento.

A técnica de descarte de blocos introduzida no CUDAlign 2.1 foi proposta para ambientes com uma única GPU, assim, houve a necessidade de realizar alterações em seu funcionamento, visto que não seria mais executado na matriz por completo, mas sim nas colunas processadas por cada GPUs. No MASA-CUDAlign, cada GPU contém um grupo de colunas ao qual ela é responsável por processar, de forma que uma GPU_i só recebe informações acerca da última coluna de sua vizinha GPU_{i-1} e envia sua última coluna para a GPU_{i+1}. Assim, foi necessário fazer com que o índice da primeira coluna de uma GPU considerasse a posição relativa que ela se localiza na matriz inteira. Ademais, para calcular se o bloco deve ser descartado, o cálculo do maior escore possível a partir de um bloco (Seção 4.1) passou a considerar a distância na matriz por completo, ao invés das dimensões contidas apenas na GPU.

Segundo o algoritmo de descarte de blocos do CUDAlign 2.1, o maior escore possível a partir de um bloco é comparado com o melhor escore atual da matriz. Pelo fato da matriz ser dividida grupos de coluna para cada GPU, é possível obter apenas o melhor escore local, ou seja, apenas em relação a GPU que o contém. No entanto, para que ocorra um descarte de blocos eficaz, é necessário que todas as GPUs tenham conhecimento acerca do maior escore obtido por todas as GPUs até o momento. Para tal, o Static-MultiBP adicionou uma nova *thread* assíncrona ao modelo introduzido no CUDAlign 3.0 (Seção 4.1). Dada uma GPU_i, esta *thread* tem como função receber periodicamente o escore da

GPU_{i-1} vizinha, comparar com seu escore local, substituir pelo valor recebido caso este seja maior que o local e enviar o escore local para a GPU_{i+1}, sendo assim considerada uma abordagem de anel. A Figura 4.8 apresenta a troca de escores locais entre GPUs do mesmo *host* e *hosts* diferentes, em que T_S, T_M e T_C são as novas *threads*, as *threads* de gerenciamento e as *threads* de comunicação, respectivamente.

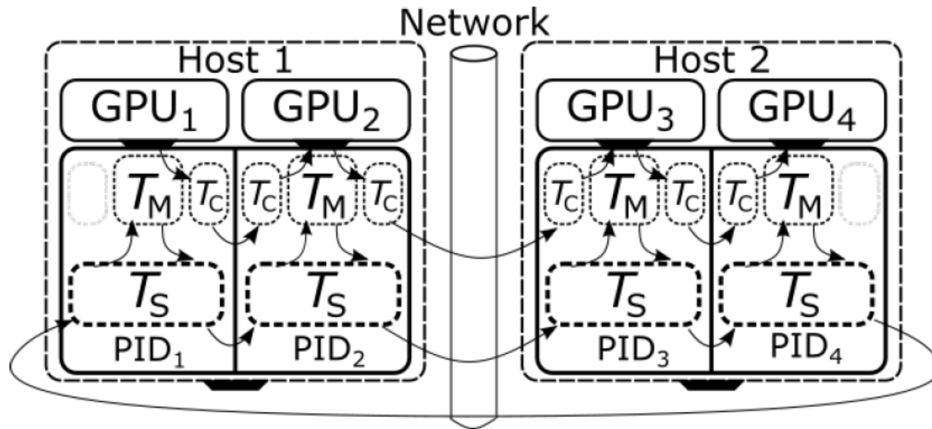


Figura 4.8: Troca de maiores escores locais (Fonte: [10]).

Com o desenvolvimento do Static-MultiBP, notou-se que o descarte de blocos pode gerar um grande desbalanceamento de carga entre GPUs, visto que algumas áreas da matriz podem ser processadas por inteiro, enquanto outras podem ter grandes parcelas descartadas. Assim, foi proposta a nova *framework* MultiBP, que engloba o Static-MultiBP e adiciona uma nova camada capaz de tratar o desbalanceamento entre GPUs, o Dynamic-MultiBP.

A estratégia utilizada pelo Dynamic-MultiBP se difere do Static-MultiBP pela forma como é definida a carga das GPUs. No Static-MultiBP, cada GPU é responsável por uma parcela predefinida de colunas da matriz de programação dinâmica, de forma que esta carga seja mantida até o fim da execução. Quanto ao Dynamic-MultiBP, inicialmente a carga das GPUs também definida estaticamente, porém, limitada até uma certa parcela da matriz, por meio dos pontos de parada (*breakpoints*). Os pontos de parada definem em quantos ciclos uma matriz deve ser computada, ou seja, as matrizes são particionadas a partir de colunas e executadas em separado. No momento em que um *breakpoint* é atingido, os *buffers* de entrada e saída das GPUs são avaliados e, então, a carga é redistribuída entre as GPUs a depender da situação. Assim, o ciclo seguinte se inicia e a próxima parte da matriz é calculada com as novas cargas entre as GPUs.

A *framework* MultiBP apresenta uma arquitetura composta por quatro módulos:

- ***Executor***: módulo onde a matriz é calculada e tem seus blocos descartados por meio do Static-MultiBP ou Dynamic-MultiBP;
- ***Decision-Maker***: responsável por selecionar o modo *Static* ou *Dynamic* a ser utilizado de acordo com os parâmetros recebidos;
- ***Controller***: responsável por ler o arquivo de configuração de execução, dividir e coordenar o processamento entre os múltiplos executores e escrever os resultados em um arquivo de saída;
- ***Monitor***: módulo que intermedeia a comunicação entre seu *Executor* e o *Controller.*

O MultiBP inicia seu funcionamento por meio do *Controller*, que lê o arquivo de configuração e envia as informações obtidas para o *Decision-Maker*, o qual define se será utilizado o Static-MultiBP ou o Dynamic-MultiBP. Com a estratégia escolhida, o *Controller* define os comandos de execução e os envia aos *Monitors* de cada GPU. Então, o módulo *Monitor* repassa o comando ao seu *Executor*, o qual inicia a computação de suas colunas da matriz. Durante o processamento de uma GPU_i, informações sobre a última coluna e escore local da vizinha GPU_{i-1} são recebidas e utilizadas conforme a descrição do Static-MultiBP, assim como são enviadas as mesmas informações ao *Executor* da GPU_{i+1}. Além disso, o *Monitor* da última GPU é responsável por verificar em cada ciclo se existem mais processamentos a serem feitos. Caso não existam, uma mensagem é enviada ao *Controller*, que verifica o número de *breakpoints* e envia novos comandos de execução caso existam mais ciclos, ou o módulo escreve os resultados em um arquivo de saída. A Figura 4.9 apresenta a arquitetura do MultiBP descrita.

4.4 MASA-OpenMP

O OpenMP, é uma implementação de *multithreading* capaz de tirar proveito das características presentes na arquitetura de memória compartilhada. Ao longo do processo de desenvolvimento do OpenMP, foram priorizadas as filosofias de (1) prover uma estrutura de controle simples, visto que é possível criar a maioria das aplicações com poucas estruturas, (2) cada processo possuir um único ambiente de dados associado a ele durante a execução, (3) seguir dois tipos de sincronização, a implícita, utilizada ao fim de construtos, e a explícita, definida pelo usuário, e (4) prover uma biblioteca de tempo de execução e variáveis de ambiente [40].

O OpenMP consiste em um conjunto de diretivas do compilador e rotinas para programas de linguagem C, C++ e Fortran com o objetivo de permitir uma paralelização por memória compartilhada em diversas plataformas. No início da execução de um programa

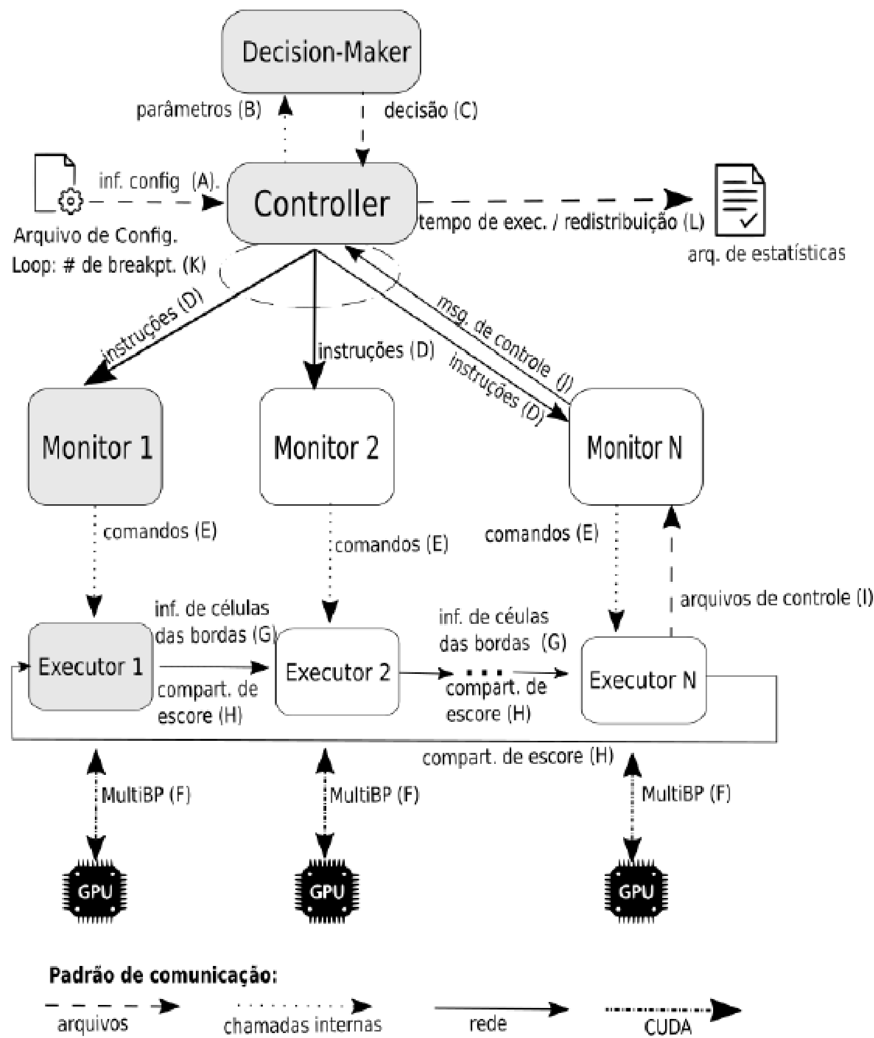


Figura 4.9: Arquitetura do MultiBP (Fonte: [10]).

existe apenas um processo, a *thread* mestra. Porém, ao atingir um construto que define onde deve ocorrer a paralelização, múltiplas *threads* são criadas a partir de um *fork* e todas processam a área de código paralelizada. Após todas as *threads* executarem a área paralelizada, elas são sincronizadas e finalizadas, restando apenas a *thread* original, que continua sua execução. Este modelo utilizado pelo OpenMP é chamado de *fork/join*.

O modelo *fork/join* pode ser aplicado inúmeras vezes ao longo de um código e o modelo de memória compartilhada não necessita decompor um programa para realizar paralelizações simples [40], características diferenciais as quais o OpenMP possui. Além disso, entre as várias ferramentas que compõem o OpenMP, a execução de laços de repetição paralelizada é uma ferramenta do OpenMP a se destacar. O *parallel worksharing-loop construct* é um construto contido no OpenMP capaz de paralelizar um laço de repetição e atribuir um número de iterações para cada uma das *threads* criadas [41]. Assim, o traba-

lho que seria realizado por apenas uma única *thread* é facilmente dividido entre múltiplas *threads*.

Conforme citado na Seção 4.2, o OpenMP foi utilizado na criação das extensões para a arquitetura MASA: MASA-OpenMP/CPU e MASA-OpenMP/Phi. Ambas as extensões utilizam a estratégia de paralelização de *parallel worksharing-loop construct* para calcular as diagonais das matrizes de programação dinâmica. Assim, cada uma das *threads* OpenMP processa as diagonais paralelamente [9]. A Figura **??** apresenta o funcionamento do processamento em diagonais, em que os blocos de anti-diagonais de mesma cor podem ser processados paralelamente.

4.5 A Ferramenta MASA no AWS Cloud

A fim de executar as ferramentas MASA em ambientes com múltiplas GPUs remotamente, foi adotado o *Cloud Amazon Web Services* (AWS). O AWS consiste em um conjunto de serviços providos pela Amazon a partir da tecnologia em nuvem, que tem se tornado cada vez mais popular devido a sua infraestrutura ser completamente transparente ao usuário, facilitando sua utilização, além de apresentar uma boa escalabilidade para as mais diversas aplicações [42]. Entre as funcionalidades das ferramentas que compõem o AWS, destacam-se computação, armazenamento de dados, gerenciamento de banco de dados e aprendizagem de máquina em nuvem.

Provida pela AWS, o serviço *Amazon Elastic Compute Cloud* (EC2) foi utilizado para a computação em nuvem das ferramentas MASA. O serviço EC2 se baseia no modelo de Infraestrutura-como-Serviço (IaaS), em que o usuário delimita aspectos do sistema computacional que será utilizado, como as unidades de processamento, o sistema operacional, a comunicação, entre outros, pagando pelos recursos utilizados [42]. Os principais elementos que compõem a ferramenta Amazon EC2 são [43]:

- **Instâncias**: são ambientes virtuais nos quais são feitas as computações. Durante sua configuração, é possível decidir o tipo da instância, que determina sua capacidade computacional e de memória. Após ser lançada, a instância se comporta como um computador comum para o usuário, de forma que seja possível interagir por meio de linhas de comando.
- **Par de chaves**: é um conjunto de chaves composto por uma chave publica, armazenada pelo Amazon EC2, e uma chave privada, armazenada localmente pelo usuário. Ambas as chaves são utilizadas como credenciais para certificar a identidade do usuário ao se conectar a uma instância EC2.

- **Armazenamento de dados**: os dados enviados às instâncias podem ser armazenados temporariamente, mantendo-os até o fim das instâncias, seja por interrupção, hibernação ou término. Além disso, o serviço Amazon S3 provê ao EC2 a possibilidade de armazenar permanentemente quaisquer dados presentes nas instâncias.

- **Região**: para estar presente em diversas áreas do mundo, o EC2 é hospedado e dividido em regiões. Cada região é criada para ser isolada das outras, permitindo uma maior tolerância a falhas e instabilidade. Durante a configuração para ser lançada uma instância, a região que a hospedará deve ser definida pelo usuário.

- **Endereço IP Elástico**: consiste em um endereço IPv4 estático para computação em nuvem. A partir desse endereço, é possível associá-lo com uma instância para que esta possa se comunicar com a internet. Além disso, os endereços IPs elásticos permitem mascarar erros de instância e software ao remapear o endereço para outra instância.

- ***Virtual Private Clouds* (VPC)**: são redes virtuais logicamente isoladas do restante do AWS Cloud. Dessa forma, é possível lançar instâncias EC2 em uma VPC com faixa de endereços IP, sub-redes, etc, especificados pelo usuário.

Outra importante ferramenta utilizada na utilização de serviços AWS é a AWS *Command Line Interface* (CLI) [44]. Esta ferramenta tem como objetivo permitir que o usuário interaja com os serviços AWS a partir de linhas de comando no *shell*. Para tal, é necessário que o usuário configure a AWS CLI com as seguintes informações: ID da chave de acesso e chave secreta, que são o par de chave utilizado durante a assinatura de pedidos ao AWS; região AWS, a qual define a região *default* para onde são feitos os pedidos; e o formato de saída, que determina o formato das saídas geradas pelo AWS.

Para criar efetivamente o *cluster* a partir dos elementos da Amazon EC2, é utilizada a ferramenta AWS *ParallelCluster*. Esta ferramenta *open source* tem como função criar e gerenciar *clusters* hospedados na nuvem AWS [45]. A partir dela, são configurados internamente e automaticamente os recursos computacionais, assim como os sistemas de arquivos compartilhados. Durante a configuração do *cluster* a ser criado, é permitido ao usuário definir muitos dos elementos da Amazon EC2, como a região, o par de chaves, o tipo e número de instâncias, entre outros.

Durante a configuração do *ParallelCluster* também é possível definir o escalonador da execução entre as instâncias computacionais. Assim, devido à necessidade da comunicação entre as instâncias do *cluster*, Marco Figueiredo e co-autores desenvolveram *scripts* de execução para o *cluster ParallelCluster* [10]. Os *scripts* foram desenvolvidos no sistema *Slurm*, o qual consiste em um sistema de gerenciamento de *clusters* em ambiente Linux e escalonamento de *jobs*.

Por último, à partir do *ParallelCluster* é possível se conectar, por meio do comando *ssh*, a uma das instâncias criadas, de forma que se possa interagir com a instância de forma análoga à utilização de um terminal Linux local. Ademais, ao utilizar um cliente *secure copy protocol* (SCP) se torna possível a transferência de arquivos locais para a instância na nuvem AWS e vice-versa.

Capítulo 5

Projeto da Otimização da Área de Descarte de Blocos com Escore Inicial Heurístico

Neste capítulo serão apresentados detalhes acerca do desenvolvimento do projeto da otimização que permite o aumento da área de poda do MASA-CUDAlign-MultiBP através da utilização de um escore inicial heurístico. Na Seção 5.1 é apresentada a motivação para ser realizado este projeto. A Seção 5.2 apresenta o aspecto geral do projeto, mostrando o módulo criado de otimização da área de descarte e seus submódulos, enquanto que a Seção 5.3 apresenta em maiores detalhes o funcionamento de cada ferramenta criada neste trabalho. Por fim, a Seção 5.4 mostra o funcionamento da ferramenta MASA-CUDAlign-MultiBP na nuvem AWS.

5.1 Motivação

Conforme apresentado na Seção 4.3, o MultiBP consiste em um módulo acrescentado ao MASA-CUDAlign. Este módulo deu à ferramenta MASA-CUDAlign a capacidade de realizar o descarte de blocos em ambientes multi-GPU com carga estática, por meio do Static-MultiBP, ou com carga dinâmica, a partir do Dynamic-MultiBP. Como base para o descarte de blocos, foi desenvolvida a abordagem em anel para o compartilhamento de melhor escore entre as GPUs, periodicamente recebendo o melhor escore da GPU vizinha da esquerda, comparando com o seu escore atual e enviando o possível novo melhor à vizinha da direita. Assim, cada GPU consegue ter o conhecimento geral acerca do melhor escore entre todas as GPUs, otimizando o processo de poda de blocos.

Ao ser inicializada a execução do estágio 1 do MASA-CUDAlign-MultiBP, cada uma das GPUs é iniciada com melhor escore local 0, aumentando gradativamente este valor à

medida que processa ou recebe de uma GPU vizinha um novo melhor escore. Dessa forma, questionou-se como seria afetado o processo de descarte de blocos e, consequentemente, o tempo de execução da ferramenta caso a primeira GPU fosse inicializada com um escore factível e mais próximo do escore ótimo obtido pelo MASA-CUDAlign-MultiBP. Isto se deve ao fato de que, com um escore prévio enviado à primeira GPU, o mesmo será repassado às GPUs vizinhas conforme a abordagem de anel, antecipando a poda e aumentando a área, pois este valor só poderia ser obtido mais à frente durante a execução do algoritmo de descarte de blocos original do MASA-CUDAlign MultiBP.

Entre as formas para se obter tal escore factível e que fosse relevante para a diminuição do tempo de execução, destacaram-se as ferramenta de obtenção de alinhamentos heurísticos, visto que elas podem retornar escores mais próximos dos escores ótimos obtidos pelo MASA-CUDAlign-MultiBP.

Assim, este projeto objetivou o desenvolvimento de um módulo adicional responsável por obter o melhor escore retornado por algoritmos heurísticos de comparação de sequências biológicas, enviando o escore encontrado para a primeira GPU executando o algoritmo MASA-CUDAlign Static-MultiBP.

5.2 Visão Geral

A Figura 5.1 apresenta a visão geral do funcionamento do módulo desenvolvido.

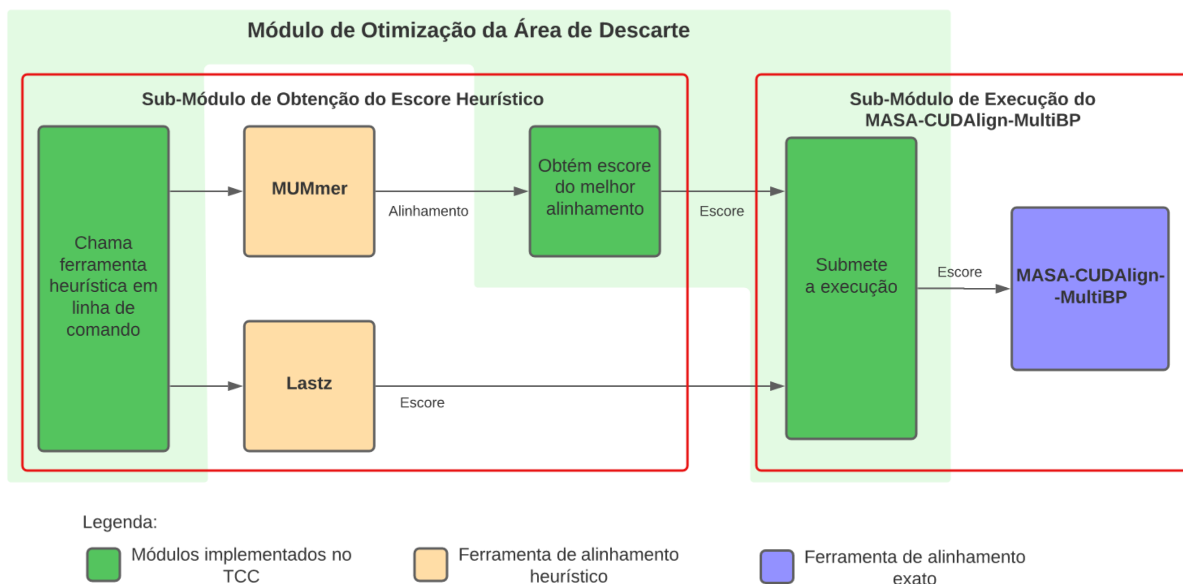


Figura 5.1: Modelo do projeto.

O módulo de obtenção do escore heurístico, representado pela cor verde clara na Figura 5.1, consiste no módulo desenvolvido neste trabalho. Escrito na linguagem *Python* na versão 3.9.5, este módulo realiza as seguintes funções:

- **Chamar as ferramentas de obtenção dos escores heurísticos**: ao depender dos parâmetros de linha de comando inseridos ao iniciar a interpretação do *script Python*, o módulo realizará a chamada em linha de comando a uma das ferramentas de alinhamento heurístico, *MUMmer* ou *Lastz*, cada qual com seus parâmetros também definidos na linha de comando do *script Python*;

- **Obter o escore do *MUMmer***: ao contrário da ferramenta *Lastz*, que pode ser parametrizada, o *MUMmer* não utiliza o mesmo esquema de pontuação de *match*, *mismatch* e abertura e extensão de *gap* utilizado no MASA-CUDAlign. Dessa forma, foi necessário criar uma função capaz de varrer o arquivo resultante contendo os alinhamentos gerados pelo *MUMmer* e, então, encontrar aquele com o melhor escore seguindo os padrões do MASA-CUDAlign.

- **Alterar e submeter o *script***: para ser executado em um *cluster* de GPUs, Marco Figueiredo e co-autores [10] desenvolveram um *script Slurm* que realiza o intermédio para a execução em paralelo de um *job* entre as GPUs. Dessa forma, após obter o melhor escore dos algoritmos heurísticos, os parâmetros de escore e de *split* do *job* são modificados. Então, o *job* é submetido a ser executado a partir de uma linha de comando.

Em adição ao módulo desenvolvido neste projeto, também foram integradas duas ferramentas de alinhamento heurístico, sendo elas o *MUMmer* e o *Lastz*. A ferramenta *MUMmer*, em sua versão 3 [46], não apresenta muitas opções durante sua parametrização, permitindo como parâmetros somente os arquivos *fasta* contendo as sequências que serão alinhadas e o prefixo utilizado para nomear os arquivos de saída. Após sua execução, o *MUMmer* retorna quatro arquivos: *out*, o qual apresenta as posições nas sequências e comprimento dos alinhamentos encontrados; *gaps* e *errosgaps*, que incluem informações adicionais acerca dos *gaps* encontrados; e *align*, o qual contém os alinhamentos obtidos e que foi utilizado para obter o escore baseado no esquema de pontuação do MASA-CUDAlign.

Ao contrario do *MUMmer*, a ferramenta *Lastz* [47] se mostrou altamente parametrizável. Entre os principais parâmetros utilizados, é possível destacar: *match*, que define a pontuação do *match* e *mismatch*; *gap*, que define a penalidade de abertura e extensão de *gap*; *strand*, que define qual o sentido em que é feito o alinhamento (original ou reverso); *gapped*, o qual permite a ocorrência de *gaps* no alinhamento; *hspthresh*, que define um limite mínimo de escore para os alinhamentos encontrados; e *allocate:traceback*, que

define a quantidade de memória alocada durante o estágio de *traceback* em sua execução. Então, a ferramenta *Lastz* gera como resultado um arquivo *maf* contendo os alinhamentos obtidos e seus respectivos escores.

A última ferramenta que compõe o projeto desenvolvido neste trabalho é o MASA-CUDAlign-MultiBP (Seção 4.3). Após o melhor escore heurístico obtido pelo módulo implementado no TCC ser copiado para o arquivo do *job* MASA-CUDAlign-MultiBP, este *job* é submetido pelo *script Slurm* para ser executado no *cluster* de GPUs. Dentro do *script* existe a linha de comando que inicia a execução da ferramenta, de forma que o melhor escore é passado como um parâmetro dela. Assim, a ferramenta é inicializada com o melhor escore heurístico na primeira GPU, resultando em sua transmissão devido à abordagem de anel e, posteriormente, em uma execução mais rápida devido ao aumento na área de poda.

Por fim, a fim de criar uma melhor organização acerca de seu funcionamento, foram classificados dois sub-módulos dentro do projeto, demarcados pelos quadrados vermelhos na Figura 5.1. De forma que o primeiro sub-módulo contém as ferramentas responsáveis para a obtenção do escore heurístico, enquanto que o segundo sub-módulo contém as ferramentas com foco na execução do MASA-CUDAlign-MultiBP.

5.3 Sub-módulos da Otimização da Área de Descarte

Nesta seção são apresentados maiores detalhes acerca dos sub-módulos que compõem o projeto, o que cada um contém e como foram feitas algumas das ferramentas neles presentes.

5.3.1 Sub-Módulo de Obtenção do Escore Heurístico

O projeto realizado neste trabalho de TCC consistiu principalmente na criação do módulo de obtenção do escore heurístico. Esse módulo foi criado na linguagem *Python* versão 3.9.5, a qual foi utilizada devido a operação com *strings* e a chamada em linha de comando à outros programas serem simplificadas, em comparação a outras linguagens.

Inicialmente planejava-se criar o módulo apenas para obter o melhor escore heurístico pela ferramenta *MUMmer* [46]. No entanto, notou-se que a versão 3 da ferramenta diminuiu consideravelmente a parametrização do alinhamento a ser executado e, dessa forma, não retornava o melhor escore seguindo os padrões de pontuação do MASA-CUDAlign-MultiBP, mas, ao invés disso, apresentando os alinhamentos encontrados, no sentido original e reverso. Assim, a primeira ferramenta desenvolvida no módulo teve como objetivo a obtenção do melhor escore MASA-CUDAlign-MultiBP entre os alinhamentos encontrados pelo *MUMmer*.

Listing 5.1: Calcula o melhor escore do *MUMmer*.

```
1  function findMummer() {
2      lê o arquivo contendo os alinhamentos
3      for linha no arquivo
4          if "Reverse" em linha:
5              break
6          if "Errors" em linha:
7              if melhor_escore < escore_atual:
8                  melhor_escore = escore
9              escore = 0
10         if sequência A em linha:
11             seqA = sequência A
12         else if sequência B em linha:
13             seqB = sequência B
14             for comprimento da sequência A:
15                 if match:
16                     escore += 1
17                 else if gap em seqA ou seqB:
18                     if gap de abertura:
19                         escore -= 5
20                     else:
21                         escore -= 2
22                 else if mismatch:
23                     escore -= 3
24     return melhor_escore
25 }
```

O Pseudocódigo 5.1 representa o código utilizado para calcular o escore do melhor alinhamento encontrado pelo *MUMmer*. Primeiramente, é lido o arquivo *align* que contém os alinhamentos encontrados e é feita uma iteração pelas linhas do arquivo. Em seguida, é feito uma sequência de condicionais para identificar o que a linha iterada contém, sendo que como o *MUMmer* realiza o alinhamento no sentido original e, posteriormente, no reverso, na linha 4 é verificado se foi atingido o início do alinhamento reverso, finalizando a iteração caso tenha, visto que não há interesse nesse sentido. Em seguida, a linha 6 confere se chegou ao fim do alinhamento, representado pela *string* "*Errors*", e compara o escore do alinhamento atual com o melhor, substituindo-o caso seja maior. Já as linhas 10 e 12 identificam as sequências A e B que foram alinhadas. Por fim, é feita a iteração sobre o comprimento da sequência A, que é o mesmo da B, comparando os caracteres das sequências e somando ao escore atual 1, -3, -5 ou -2, ao depender se ocorrer um *match*, *mismatch*, abertura ou extensão de *gap*, respectivamente.

Apesar de fornecer os escores dos alinhamentos obtidos pelo *MUMmer* conforme as pontuações do MASA-CUDAlign-MultiBP, notou-se que, em algumas sequências, os ali-

Listing 5.2: Encontra o melhor escore do *Lastz*.

```
1  function findLastz() {
2      lê o arquivo contendo os alinhamentos
3      for linha no arquivo:
4          if "a score=" em linha e melhor_escore < escore_atual:
5              melhor_escore = escore_atual
6      return melhor_escore
7
8  }
```

nhamentos não geravam escores grandes o suficiente para serem relevantes durante a execução do MASA-CUDAlign-MultiBP. Assim, surgiu a necessidade de encontrar outra ferramenta de alinhamento de sequências que obtivesse escores maiores, o que resultou na adoção da ferramenta *Lastz* [47].

Conforme apresentado na Seção 5.2, o *Lastz* apresenta muito mais parâmetros em comparação ao *MUMmer*, o que permitiu um melhor controle sobre o padrão de pontuação, o limite mínimo para o valor do melhor escore e a quantidade de memória alocada durante durante a execução da fase de *traceback* da ferramenta. Assim, o *Lastz* mostrou, em geral, escores muito mais interessantes que o *MUMmer*, se tornando a principal ferramenta utilizada para a obtenção de melhor escore.

O Pseudocódigo 5.2 apresenta o algoritmo utilizado para obter o melhor escore encontrado pelo *Lastz*. Como é possível especificar o padrão de pontuação a ser utilizado a partir de parâmetros e como o arquivo gerado já contém o escore compatível com o MASA-CUDAlign-MultiBP, não foi necessário criar uma função para calculá-lo. Dessa forma, bastou realizar uma iteração sobre as linhas do arquivo e buscar por aquelas contendo a *string* “a score=”, a qual indica o escore do alinhamento, de forma que, se o melhor escore é menor que o encontrado, basta substituir o melhor escore por este novo valor, retornando-o ao fim da iteração.

5.3.2 Sub-Módulo de Execução do MASA-CUDAlign-MultiBP

Com o melhor escore encontrado a partir das ferramentas de obtenção de alinhamento heurístico, o próximo passo a ser realizado foi transmitir este valor para a ferramenta MASA-CUDAlign-MultiBP. Primeiramente, a ferramenta precisou ser alterada para receber o melhor escore por meio dos parâmetros da linha de comando. Para tal, criou-se o parâmetro *initial-best*, que recebe o valor como uma *string*. Então, o valor passou a ser convertido para um número inteiro e armazenado na variável *initial_score* por meio do método *setBestScore*, sendo que, tanto a variável quanto o método fazem parte da classe *Job* do MASA-CUDAlign-MultiBP.

O estágio 1 da ferramenta originalmente inicializava a primeira GPU com o valor 0, assim como as demais GPUs, atribuindo este valor à variável *BestGlobal*. Desta forma, para inicializar a primeira GPU com o melhor escore heurístico, passou-se a atribuir o valor de *initial_score* do objeto da classe *Job* à variável *BestGlobal*. Assim, por meio da abordagem de anel utilizada no MultiBP, este valor passou a transmitido para as demais GPUs.

No entanto, ao testar o MASA-CUDAlign-MultiBP em duas GPUs e analisar o seu tempo de execução, notou-se que, apesar de aumentar significativamente a área de poda na segunda GPU, não houve um aumento de desempenho expressivo. Dessa forma, chegou-se à conclusão de que, como há uma intensa comunicação entre as GPUs referente às colunas de borda e, consequentemente, uma dependência da segunda GPU em relação à primeira, o ganho obtido pela poda na segunda GPU pouco importou. Por essa razão, foi adotada uma distribuição de colunas diferente da predefinida pelo MASA-CUDAlign-MultiBP (1,1) à partir do parâmetro *split*, o que resultou em uma notória melhoria de desempenho. Por exemplo, se definirmos o *split* como $1, 2$, a primeira GPU vai processar o primeiro 1/3 da matriz e a segunda GPU vai processar os últimos 2/3.

Após ser implementada a captação e utilização do melhor escore heurístico e definida a alteração do valor do split na ferramenta MASA-CUDAlign-MultiBP, precisou-se alterar o *script Slurm* utilizado na gerência do *cluster* de GPUs e submissão de *jobs*. Para tal, criou-se mais dois parâmetros de entrada para o *script*, os quais contém o escore e o *split* a ser enviado. Além disso, para inicializar a ferramenta nas GPUs com estes valores, foram adicionados os parâmetros *initial-best* e *split* à linha de comando utilizada.

Por fim, com o objetivo de enviar o melhor escore e o *split* ao *script*, foi necessário criar duas variáveis no *job* submetido, chamadas *score* e *split*. Assim, para alterar o valor atribuído a esta variável no arquivo, criou-se outra função no módulo desenvolvido neste TCC. O Pseudocódigo 5.3 representa o código criado. É feita uma iteração sobre as linhas do arquivo e é conferido se a linha contém as variáveis *score* ou *split*, substituindo a linha pela variável concatenada com o novo valor a ser atribuído. Ademais, é feita a escrita do arquivo com os novos valores.

5.4 Execução do MASA-CUDAlign-MultiBP no *Cluster* de GPUs da Nuvem AWS

A fim de executar o MASA-CUDAlign-MultiBP com o novo módulo, foi utilizado um *cluster* de GPUs hospedado na nuvem AWS, criado por meio da ferramenta AWS *ParallelCluster* (Seção 4.5). A Figura 5.2 apresenta uma visão abstrata do *cluster* utilizado no projeto.

Listing 5.3: Modifica o *job*.

```
1  function jobModifier(escore_valor, split_valor) {
2      lê o arquivo de job
3      for linha no arquivo:
4          if "score=" em linha:
5              linha = "score=" + escore_valor
6          else if "split=" em linha:
7              linha = "split=" + split_valor
8      escreve no arquivo de job
9  }
```

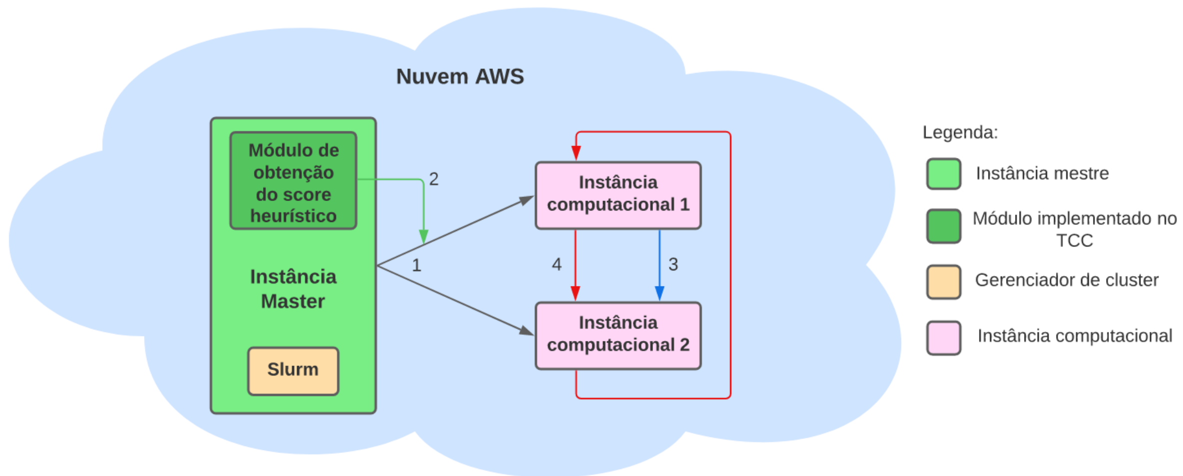


Figura 5.2: Integração do Projeto ao AWS Parallel Cluster.

Conforme mostra a figura, existem três instâncias que compõem o *cluster*, sendo elas uma instância mestre (*master*) e duas computacionais. A instância mestre não executa os *jobs*, de forma que seu maior objetivo seja gerenciar o *cluster* e escalonamento de *jobs* por meio do escalonador *Slurm* que está instalado na nuvem. Assim, a instância mestre inicia e gerencia a execução do MASA-CUDAlign-MultiBP nas instâncias computacionais, representado pelas setas pretas com o número 1.

Além da gerência da execução nas instâncias computacionais, a instância mestre executa o módulo de obtenção do escore heurístico desenvolvido neste TCC. A execução deste módulo é feita conforme o explicado na Seção 5.3, em que é obtido o melhor escore de um alinhamento à partir das ferramentas heurísticas (*MUMmer* ou *Lastz*) e, em seguida, este valor, juntamente ao *split*, é escrito no arquivo do *job* a ser executado. Então, no momento em que o *job* é submetido, o melhor escore heurístico é enviado ao MASA-CUDAlign-MultiBP como um parâmetro da linha de comando e, sendo assim, atribuído como o escore inicial do primeira GPU, presente na instância computacional 1. Este pro-

cesso é representado na Figura 5.2 por meio da seta verde com o número 2, que indica o envio do escore para a primeira instância computacional.

Ao ser submetido o *job* às instâncias computacionais, as colunas da matriz de alinhamento são distribuídas segundo a definição no parâmetro *split* e é inicializada a execução do estágio 1 do MASA-CUDAlign-MultiBP. Conforme detalhado na Seção 4.3, existem dois tipos de *threads* utilizadas para comunicação entre GPUs, sendo eles *thread* de comunicação (T_C) e *thread* de escore (T_S).

Representada pela seta em azul de número 3, a comunicação das *threads* T_C consiste na transferência dos valores referentes aos blocos das colunas de borda. Dessa forma, há apenas a comunicação no sentido da instância computacional 1 para a 2, visto que a instância 1 não necessita dos valores da coluna de borda da instância 2.

Quanto à comunicação das *threads* de escore (T_S), representada pelas setas em vermelho de número 4, esta corresponde ao envio do melhor escore local à GPU vizinha. A figura mostra a utilização da estratégia de anel desenvolvida no MASA-CUDAlign-MultiBP, em que a instância computacional 1 envia o seu melhor escore à 2. Já esta instância avalia se o escore recebido é maior que o seu próprio, substituindo-o caso seja e enviando o novo escore à sua vizinha, que é, novamente, a instância 1, repetindo o ciclo.

Ao fim da execução do estágio 1 da ferramenta MASA-CUDAlign-MultiBP, os resultados são recebidos a partir do *script* e escritos em arquivos na instância mestre. Assim, sendo possível encontrar o escore ótimo obtido, o tempo de execução, taxa de processamento em MCUPS (Milhões de Células da matriz Processadas por Segundo), entre outros dados resultantes do alinhamento.

Capítulo 6

Resultados Experimentais

Neste capítulo são apresentados os resultados obtidos em relação à adição do módulo desenvolvido neste trabalho ao MASA-CUDAlign-MultiBP. Na Seção 6.1 é apresentada a configuração do AWS Parallel Cluster, listando os passos e comandos utilizados. A Seção 6.2 traz as sequências utilizadas nos testes, assim como os escores obtidos em cada um dos alinhamentos. Por fim, a Seção 6.3 apresenta os resultados obtidos em relação ao tempo de execução, mostrando o efeito causado pela utilização do módulo de obtenção de escore heurístico.

6.1 Configuração do AWS *ParallelCluster*

A fim de executar a ferramenta MASA-CUDAlign-MultiBP, foi utilizado um *cluster* hospedado em uma nuvem AWS e criado pelo AWS *ParallelCluster* (Seção 4.5). Para tal, foi necessária a configuração do ambiente a partir de diversos comandos aqui explanados. A Figura 6.1 apresenta os principais comandos utilizados.

A criação do *cluster* se inicia pela instalação do AWS CLI e, em seguida, por sua configuração (comando 1). Nela se definem as credenciais utilizadas durante a interação entre usuário e AWS, sendo elas o ID da chave de acesso e a chave secreta de acesso, ambos atreladas à conta AWS do usuário. Além disso, também são configuradas a região *default* a ser utilizada e o formato de eventuais saídas geradas pelo AWS CLI, podendo ser *json* (o formato *default*), *yaml*, *text*, entre outros.

Após a configuração do CLI, deve ser instalado o AWS *ParallelCluster*, cuja versão utilizada neste projeto foi a 2.11.2. Durante a configuração do *ParallelCluster* (comando 2) são definidos os elementos do AWS EC2 que serão utilizados no *cluster*, sendo eles: *us-east-1* como região AWS; *slurm* como o escalonador de *jobs*; ubuntu1804 como o sistema operacional; dois como o tamanho mínimo e máximo de instâncias computacionais; g4dn.xlarge como o tipo das instâncias GPU mestre e computacionais; a criação automá-

1. aws configure
2. pcluster configure
3. pcluster create nome
4. pcluster ssh nome -i chave.pem
5. scp -i chave.pem projeto_tcc.zip ubuntu@IPV4_publico.compute-1.amazonaws.com:
6. ./configure –with-cuda-arch=sm_75
7. make
8. sudo apt install mummer
9. conda install -c bioconda lastz
10. python3 modulo.py -m diretório sequencia1.fasta sequencia2.fasta split
11. python3 modulo.py -l diretório sequencia1.fasta sequencia2.fasta split traceback threshold

Figura 6.1: Comandos para a configuração do *cluster* e execução do módulo do TCC.

tica de VPC (*Virtual Private Cloud*); e a criação da instância mestre em uma subrede separada das instâncias computacionais.

Entre os elementos AWS EC2 utilizados, destaca-se o tipo das instâncias g4dn.xlarge. Este modelo de instância se caracteriza por ser baseada em GPU e apresentar uma alta performance, apesar de seu baixo custo. Cada instância g4dn.xlarge é composta por 1 GPU NVIDIA T4 Tensor Core, que possui 16 GiB de memória e cuja arquitetura é Turing (Seção 3.1), 4 vCPUs (Unidade Central de Processamento Virtual), 16 GiB de memória RAM e 125 GB de armazenamento.

Após a correta configuração do *cluster*, é utilizado o comando 3 para lançar o *cluster* na nuvem AWS, em que *nome* destacado em vermelho representa o nome dado ao *cluster* a ser criado. Em seguida, utiliza-se o comando 4 para se conectar ao *cluster* definindo o seu *nome* e o caminho para o arquivo que contém a chave privada SSH. Após isso, a partir de outra janela do terminal, deve ser transferido ao *cluster* o arquivo *projeto_tcc.zip*, que contém o módulo desenvolvido neste projeto, a ferramenta MASA-CUDAlign-MultiBP, o *script Slurm* e o arquivo de *job*. Para tal, por meio do protocolo SCP, é utilizado o

comando 5, o qual, recebe como parâmetros o caminho para o arquivo que contém a chave privada, o caminho para o arquivo *projeto_tcc.zip* e o endereço DNS IPV4 público da instância mestre.

Após o recebimento do arquivo *projeto_tcc.zip*, é feita a extração do arquivo no terminal onde se está conectado ao *cluster*. Em seguida, deve-se acessar o diretório *multibp*, que contém a ferramenta MASA-CUDAlign-MultiBP, e compilar com a configuração correta para a GPU Tesla T4 da instância g4dn.xlarge (comandos 6 e 7). Por fim, fora do diretório *multibp*, devem ser instalados e descompactados os arquivos de sequências disponibilizados por Walisson Sousa, com a adição das sequências de 1M feita pelo autor deste trabalho, recuperadas por meio de um repositório do GitHub [48].

A fim de executar o módulo de obtenção de escore heurístico, é necessário instalar as ferramentas *MUMmer* e *Lastz*. Para instalar o *MUMmer* utiliza-se o comando 8, enquanto que, para utilizar o *Lastz* é necessário, primeiramente, configurar o sistema de gerenciamento de pacotes *Conda*. Após ser corretamente configurado, o *Conda* deve ser utilizado no comando 9, o qual realiza a instalação da ferramenta *Lastz*.

Por fim, os comandos 10 e 11 são utilizados para a execução do módulo de obtenção de escore heurístico (Subseção 5.3.1) com a ferramenta *MUMmer* e *Lastz*, respectivamente. Nestes comandos é necessário especificar o nome do diretório onde estão localizadas as sequências de uma comparação, o caminho para os arquivos *fasta* que contém as sequências e o *split* (Subseção 5.3.2) a ser feito entre as GPUs das instâncias computacionais. Além disso, no comando do *Lastz* também é possível especificar o tamanho de memória em MB utilizada para realizar a fase de *traceback* da ferramenta e o limite mínimo (*threshold*) de escore.

6.2 Sequências Utilizadas

Durante os testes da ferramenta MASA-CUDAlign-MultiBP com o módulo de obtenção de escore heurístico, feitas quatro comparações ao todo, cada qual com um par de sequências. As sequências foram cedidas por Walisson Sousa, com a adição do par de 1M feita pelo autor deste trabalho [48], que, por sua vez, foram obtidas por meio do *National Center for Biotechnology Information* (NCBI), onde são disponibilizadas sequências em formato *fasta* e muitos outros.

As quatro comparações foram feitas com sequências de tamanho 1M, 3M, 5M e 10M. O primeiro par, com sequências de 1M, é composto pelas sequências *Chlamydia trachomatis A/HAR-13* e a *Chlamydia trachomatis L2b/UCH-1/proctitis*, que são moderadamente semelhantes. O par de sequências de 3M é composto pelas sequências *Corynebacterium efficiens YS-314 DNA* e *Corynebacterium glutamicum ATCC 13032, IS fingerprint type*

Tabela 6.1: Sequências e os escores produzidos em suas comparações.

	Sequência A		**Sequência B**		**Escore**	
Comp.	**Accession N.**	**Tam.**	**Accession N.**	**Tam.**	**Ótimo**	**Heurístico**
1M-1M	CP000051.1	1M	AM884177.2	1M	679278	222593
3M-3M	BA000035.2	3M	BX927147.1	3M	4226	3888
5M-5M	AE016879.1	5M	AE017225.1	5M	5220960	849975
10M-10M	NC_017186.1	10M	NC_014318.1	10M	10235188	6235710

4-5, que são muito diferentes. Já as sequências de 5M são muito semelhantes e correspondem a *Bacillus anthracis str. Ames* e a *Bacillus anthracis str. Sterne*. Por último, as sequências de 10M também são muito parecidas e são *Amycolatopsis mediterranei S699* e *Amycolatopsis mediterranei U32*.

A Tabela 6.1 apresenta os identificadores das sequências utilizadas (*accession number*), seus tamanhos e o escores ótimo, obtido pelo MASA-CUDAlign-MultiBP, e o heurístico, obtido pela ferramenta *Lastz*, para cada alinhamento. Além disso, destaca-se que todos os escores heurísticos *Lastz* foram obtidos com 2GB de memória para a operação de *traceback*.

6.3 Resultados do Módulo de Obtenção de Escore Heurístico

Nesta seção serão apresentados os resultados obtidos ao executar a ferramenta MASA-CUDAlign MultiBP com o módulo de obtenção de escore heurístico variando o escore inicial e o *split*. A notação utilizada para se referir ao *split* é a mesma utilizada na ferramenta, em que o valor corresponde a proporção de colunas da matriz de programação dinâmica atribuída para cada GPU. Por exemplo, o *split* 1,3 indica que a primeira GPU calcula as primeiras colunas (1/4 do total), enquanto que a segunda calcula os 3/4. Além disso, destaca-se que todos os testes foram realizados na núvem AWS com duas instâncias computacionais g4dn.xlarge.

6.3.1 Resultados obtidos com Escore igual a 0

A Tabela 6.2 apresenta a média do tempo, em segundos, e média de GCUPS obtidas ao executar a ferramenta MASA-CUDAlign-MultiBP com escore inicial 0 e variando o *split*. Para encontrar as médias e intervalos de confiança (apresentados na Subseção 6.3.3) de cada medição, foram feitas 5 execuções de cada.

Na comparação 1M-1M, o melhor tempo foi atingido por meio do *split* 1,2. A grande diferença de desempenho entre o *split* 1,1 e os demais pode indicar que, por apresentar uma área de poda maior que a primeira GPU, uma distribuição com maior área para a

Tabela 6.2: Tempos obtidos a partir do escore inicial 0.

Comparação	Escore Inicial	Split	Tempo (s)	GCUPS
1M-1M	0	1,1	5.04	214.79
1M-1M	0	1,2	**4.07**	**266.23**
1M-1M	0	1,3	4.64	234.55
1M-1M	0	2,3	4.40	244.35
3M-3M	0	1,1	**40.13**	**255.73**
3M-3M	0	1,2	55.25	186.91
3M-3M	0	1,3	60.95	169.21
3M-3M	0	2,3	47.12	218.75
5M-5M	0	1,1	79.08	345.21
5M-5M	0	1,2	72.17	382.98
5M-5M	0	1,3	79.36	347.02
5M-5M	0	2,3	**68.88**	**396.75**
10M-10M	0	1,1	280.93	373.88
10M-10M	0	1,2	239.91	436.78
10M-10M	0	1,3	271.60	385.14
10M-10M	0	2,3	**234.23**	**448.14**

segunda GPU pode melhorar o desempenho da ferramenta para esta comparação. Além disso, devido a intensa comunicação entre as GPUs acerca da coluna de borda, a segunda GPU pode ser atrasada pela primeira, de forma que o ganho de tempo obtido pela área de poda não seja usufruído com o *split* 1,1. Dessa forma, para esta comparação, ao distribuir uma área menor a primeira GPU, há um menor atraso e maior efeito da área de poda.

Quanto a comparação 3M-3M, o melhor tempo foi atingido com o *split* 1,1. Como as sequências desta comparação são muito diferentes, tanto o escore ótimo quanto o heurístico não alcançam um valor expressivo, de forma que a área de poda seja muito pequena. Devido a isto, a distribuição equilibrada da matriz entre as GPUs se mostrou a forma mais eficiente para esta comparação.

Por fim, ambas as comparações 5M-5M e 10M-10M apresentaram o melhor desempenho com o *split* 2,3. Nessas comparações as sequências são parecidas, de forma que seja obtido um escore que gere uma área de poda relevante. Assim, ao que indicam os resultados, a redistribuição 2,3 pode evitar que ocorra um grande atraso gerado pela transferência dos valores da coluna de borda, mas ainda mantendo uma distribuição próxima do equilíbrio entre as GPUs.

6.3.2 Resultados obtidos com Escore Inicial igual ao Escore Heurístico

A Tabela 6.3 apresenta as médias dos resultados obtidos ao executar o MASA-CUDAlign-MultiBP com a otimização de descarte de blocos, utilizando o escore inicial heurístico, encontrado com a ferramenta *Lastz*.

Os resultados do escore heurístico nas comparações 1M-1M e 3M-3M, se mostraram de maneira similar aos obtidos com o escore 0, os quais apresentaram melhores resultados com os *splits* 1,2 e 1,1. No entanto, quanto à comparação 1M-1M, é necessário destacar que a *thread* que compartilha os escores entre as GPUs é executada a cada 5 segundos. Dessa forma, como quase todas as execuções da comparação 1M-1M foram mais rápidas que 5 segundos, não houve o compartilhamento do escore inicial para a segunda GPU. Assim, a pouca melhoria de desempenho poderia vir apenas do incremento da área de poda da primeira GPU, que ainda não é grande pois as sequências são moderadamente semelhantes e o escore heurístico obtido não é próximo do ótimo.

Quanto às comparações 5M-5M e 10M-10M, os melhores tempos foram encontrados com os *splits* 1,2 e 1,3, respectivamente. Como o escore heurístico da comparação 10M-10M, quando comparado ao ótimo, é proporcionalmente maior que o do 5M-5M, esses resultados podem indicar que, para estas comparações, um maior escore inicial pode proporcionar um melhor aproveitamento da poda na segunda GPU, ao mesmo passo que diminui o atraso gerado pela transferência de valores da coluna de borda, o que é favorecido pelo *split* 1,3, o melhor para a comparação 10M-10M. No entanto, um menor escore heurístico gera uma menor área de poda, de forma que o ganho não valha o tempo perdido pela menor paralelização do processamento da matriz, o que faz com que o *split* 1,2 se mostre a melhor escolha para a comparação 5M-5M.

6.3.3 Comparação dos Tempos de Execução

Dados os resultados apresentados nas subseções 6.3.1 e 6.3.2, nesta subseção serão comparados os tempos obtidos em cada uma das comparações, variando o escore inicial e o *split*. Para cada caso feitas 5 execuções, de forma que fosse possível obter os intervalos de confiança com nível de confiança de 95%. Para apresentar a comparação entre as estratégias, foram criados gráficos de barra, em que o eixo x corresponde ao *split* utilizado e o eixo y ao tempo de execução em segundos. Além disso, as barras laranjas correspondem ao tempo encontrado com o escore inicial 0, enquanto que as barras azuis apresentam o tempo com escore inicial heurístico. Por fim, nos gráficos também estão presentes as diferenças percentuais dos tempos de execução com escore 0 em comparação ao escore heurístico.

Tabela 6.3: Tempos obtidos a partir do escore inicial heurístico.

Comparação	**Escore Inicial**	**Split**	**Tempo (s)**	**GCUPS**
1M-1M	222593	1,1	5.02	214.89
1M-1M	222593	1,2	**4.01**	**271.05**
1M-1M	222593	1,3	4.69	231.80
1M-1M	222593	2,3	4.41	245.89
3M-3M	3888	1,1	**42.20**	**247.04**
3M-3M	3888	1,2	55.44	186.54
3M-3M	3888	1,3	61.13	169.61
3M-3M	3888	2,3	47.56	217.50
5M-5M	849975	1,1	79.77	343.10
5M-5M	849975	1,2	**66.12**	**412.08**
5M-5M	849975	1,3	75.31	361.95
5M-5M	849975	2,3	68.04	402.08
10M-10M	6235710	1,1	259.81	406.08
10M-10M	6235710	1,2	208.14	503.71
10M-10M	6235710	1,3	**190.72**	**549.64**
10M-10M	6235710	2,3	229.34	461.37

A Figura 6.2 apresenta o gráfico de comparação da comparação 1M-1M. Nela é possível notar que os tempos se mantiveram, em geral, muito próximos. Isto pode ser atribuído ao fato das sequências alinhadas e o escore heurístico serem pequenos, de forma que o aumento da área de poda não gerou uma diminuição expressiva no tempo de execução. Ademais, em ambos os escores utilizados, o *split* 1,2 se mostrou o melhor para esta comparação.

A Figura 6.3 apresenta o gráfico de barra dos tempos de execução com a comparação 3M-3M. Ao analisá-lo, é notório o melhor desempenho com o *split* 1,1, que representa uma divisão equilibrada da área da matriz entre as GPUs. Conforme explicado nas subseções anteriores, isto pode ser justificado pois, como as sequências são muito diferentes entre si, se obtém um escore pequeno e, consequentemente, não é gerada uma área de poda expressiva. Dessa forma, a melhoria no tempo de execução obtida na poda da segunda GPU não compensa a perda de paralelização ao redistribuir as áreas da matriz. Além disso, a variação do escore não parece gerar tempos diferenciáveis entre si.

Na Figura 6.4, onde são mostrados os resultados da comparação 5M-5M, é possível notar que nos *splits* 1,2 e 1,3 há uma clara diminuição do tempo de execução ao utilizar o escore inicial heurístico, em comparação ao 0. Isto pode indicar que, como as sequências desta comparação são mais compridas e parecidas entre si, é encontrado um escore

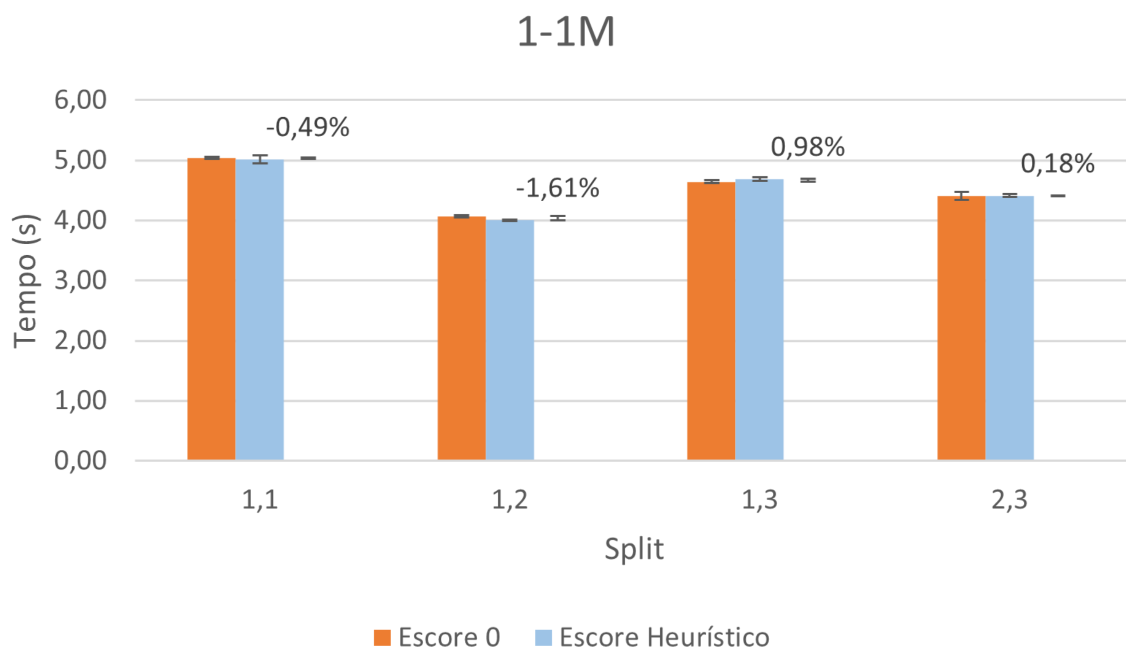


Figura 6.2: Desempenho na comparação 1M-1M.

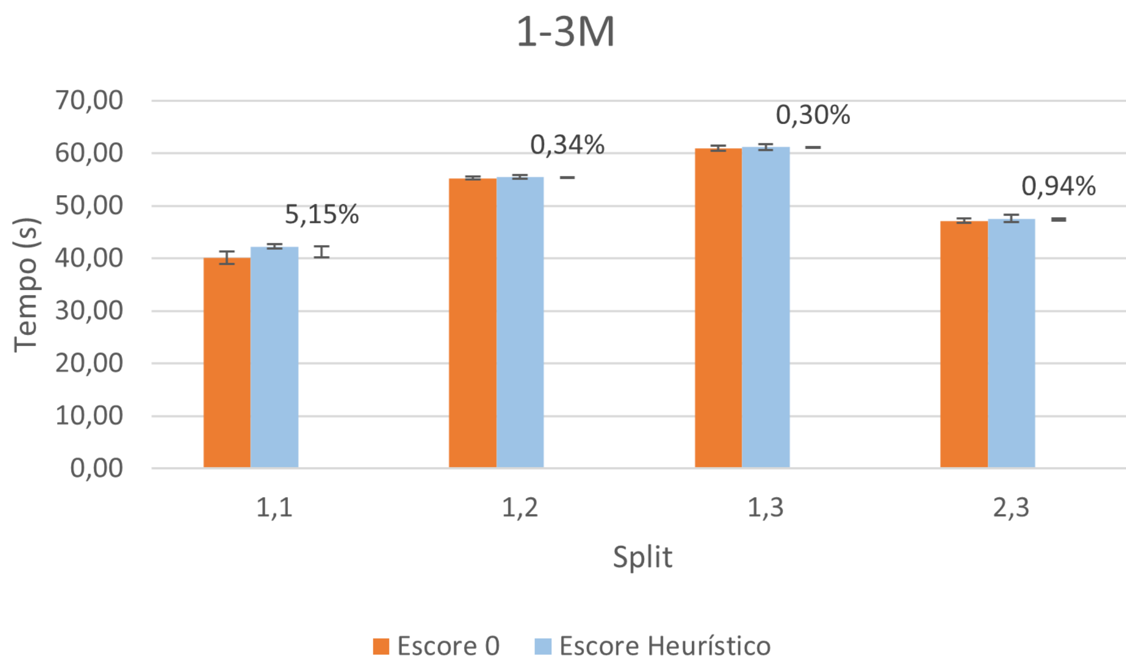


Figura 6.3: Desempenho na comparação 3M-3M.

heurístico que causa uma poda significativa, porém ainda pequena, na segunda GPU, resultando em um menor tempo de execução. Assim, ao que indicam os resultados, os *splits* 1,2 e 1,3 se mostram impactantes por diminuírem o tempo de espera causado pela transferência da coluna de borda e, por sua vez, gerado pela diminuição da área da matriz destinada à primeira GPU.

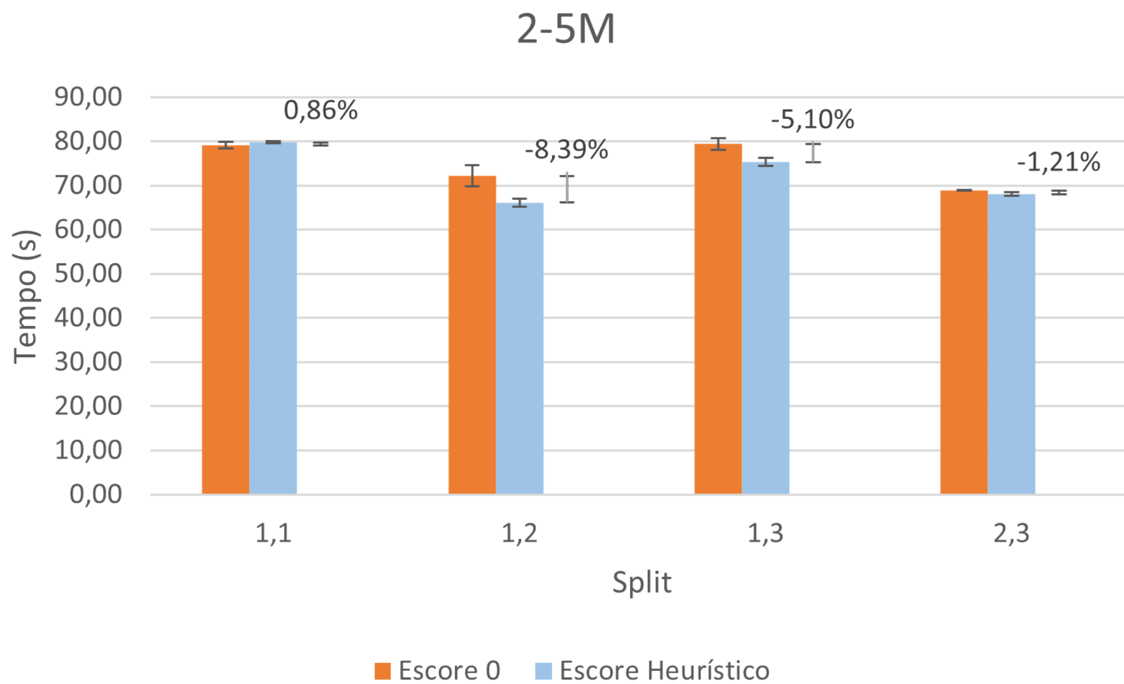


Figura 6.4: Desempenho na comparação 5M-5M.

Quanto à Figura 6.5, é notória a disparidade de tempo entre os escores. Isto se deve ao grande comprimento das sequências e grande escore heurístico, pois, a partir dele, é criada uma área de poda muito significativa durante a execução da ferramenta. Outro detalhe a se destacar é o melhor *split* não ser o mesmo entre os escores. Como o escore heurístico foi mais próximo do ótimo, a área de poda da segunda GPU se tornou muito grande, de forma que uma maior distribuição da matriz para esta GPU se mostrou muito efetiva, ou seja, com o *split* 1,3. No entanto, na execução com o escore 0 mostrou-se mais interessante uma distribuição próxima do equilíbrio, ou seja, o 2,3.

Para cada comparação e escores, em adição ao melhor tempo e seu respectivo *split*, a Tabela 6.4 apresenta o *speedup* obtido ao utilizar o escore heurístico encontrado pela ferramenta *Lastz*. Na tabela, é possível notar que os *speedups* obtido nas comparações 1M-1M e 5M-5M foram pequenos. Já na comparação 3M-3M, com sequências muito diferentes, houve uma diminuição de desempenho, mas isto é justificável pelo grande

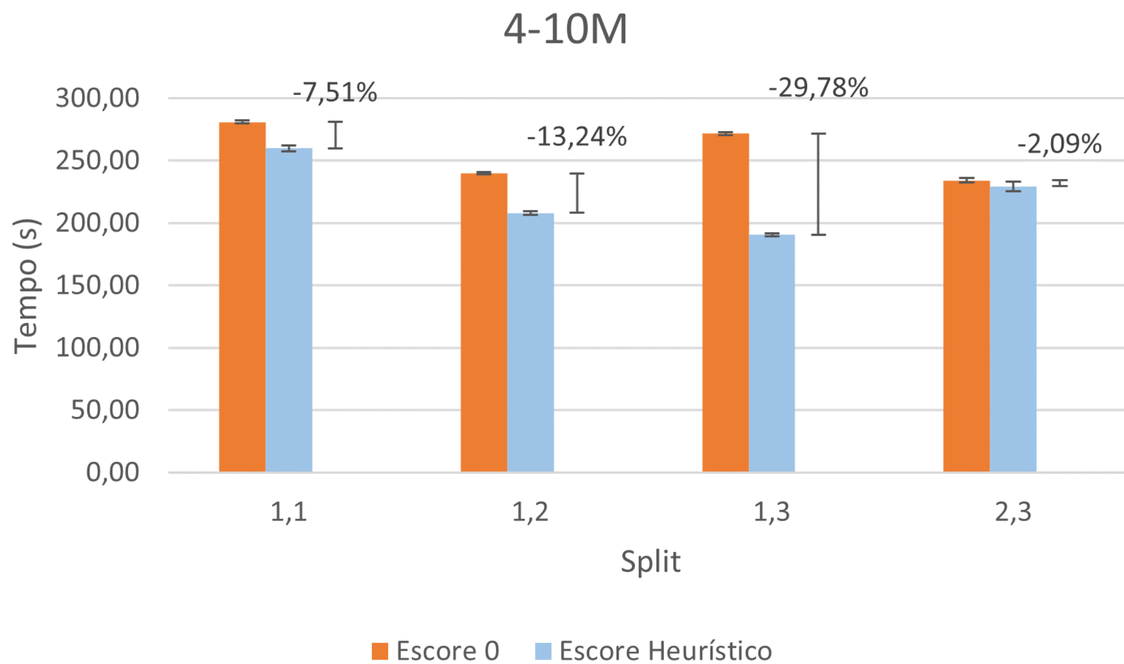


Figura 6.5: Desempenho na comparação 10M-10M.

Tabela 6.4: *Speedup* ao utilizar o escore heurístico.

	Escore 0		**Escore Heurístico**		
Comparação	**Split**	**Tempo (s)**	**Split**	**Tempo (s)**	***Speedup***
1M-1M	1,2	4.07	1,2	4.01	1.02
3M-3M	1,1	40.13	1,1	42.20	0.95
5M-5M	2,3	68.88	1,2	66.12	1.04
10M-10M	2,3	234.23	1,3	190.72	1.23

intervalo de confiança e variância encontrados nas amostras com escore 0. Por fim, na comparação 10M-10M houve uma melhoria de 1.23x. No entanto, é necessário destacar que o tempo de execução com escore inicial heurístico desconsidera o *overhead* causado pelo módulo de obtenção de escore heurístico.

Por último, foram feitos testes para identificar a melhoria de desempenho obtido ao utilizar o escore ótimo como inicial. A partir de tal valor inicial, a ferramenta MASA-CUDAlign-MultiBP pode atingir o máximo da área de poda da matriz, assim, melhorando enormemente seu desempenho. Assim, para realizar esta comparação, foram utilizados os melhores *splits* com escore heurístico de cada comparação.

A Tabela 6.5 apresenta os tempos, que são as médias de 5 execuções, e o *speedup* obtido ao utilizar o escore inicial ótimo. Primeiramente, a comparação 1M-1M apresentou pouca diferença quanto ao tempo de execução e, consequentemente, ao *speedup*. Assim como

Tabela 6.5: *Speedup* ao utilizar o escore ótimo.

	Escore 0		Escore Ótimo		
Comparação	**Split**	**Tempo (s)**	**Split**	**Tempo (s)**	***Speedup***
1M-1M	1,2	4.07	1,2	4.09	1.00
3M-3M	1,1	40.13	1,1	40.68	0.99
5M-5M	2,3	68.88	1,2	33.81	2.04
10M-10M	2,3	234.23	1,3	101.24	2.31

com o escore heurístico, a execução foi mais rápida que 5 segundos, de forma que o escore ótimo não tenha sido compartilhado para a segunda GPU. Além disso, como as sequências são moderadamente parecidas, ambos os escores ótimo e heurístico não são grandes, de forma que a área de poda gerada a partir deles também não seja grande. Assim, observa-se que, para a comparação 1M-1M, o aumento da área de poda para apenas a primeira GPU não surtiu efeitos satisfatórios.

Quanto a comparação 3M-3M, por ser composto por sequências muito diferentes e, consequentemente, gerando menores escores ótimo e heurístico, a comparação 3M-3M pode corroborar com a suposição de que escores pequenos pouco aumentam a área de poda, de forma que não diminua o tempo de execução. Por fim, as comparações 5M-5M e 10M-10M foram executados em um tempo muito menor, apresentando impressionantes 2.04x e 2.31x de *speedup*.

Assim, os resultados mostram, para estas comparações, a influência que escores iniciais e o *split* possuem em relação ao desempenho da ferramenta MASA-CUDAlign-MultiBP. No entanto, ainda se torna necessário um entendimento concreto acerca da decisão de qual o *split* a ser utilizado em cada situação e, sobretudo, a identificação de uma ferramenta capaz de encontrar escores heurísticos melhores que as ferramentas *Lastz* e *MUMmer*.

6.3.4 Área de Poda

Nesta subseção serão apresentados os gráficos gerados pela ferramenta MASA-CUDAlign MultiBP com os melhores *splits* de cada caso, seguindo a Tabela 6.4. Para cada comparação foram criados dois gráficos, em que o primeiro corresponde à execução com escore inicial 0 e o segundo com escore inicial heurístico. Cada gráfico corresponde a matriz de programação dinâmica das sequências, sendo que as áreas brancas foram processadas durante a execução da ferramenta, enquanto que as áreas pretas foram podadas, ou seja, não foram processadas. Além disso, a linha vermelha representa o *split* escolhido, dividindo a matriz entre as duas GPUs utilizadas.

A Figura 6.6 apresenta as áreas de poda da comparação 1M-1M com os dois escores (zero e escore heurístico). Nela é possível notar que houve um aumento da área de poda

com o escore heurístico apenas na área da GPU 1. Conforme explanado na Subseção 6.3.2, isto ocorre devido ao pequeno escore heurístico obtido e ao curto tempo de execução nessa comparação, que, por ser menor que o intervalo de 5 segundos de quando a *thread* que compartilha escores entre GPUs é ativada, faz com que o escore inicial heurístico não seja compartilhado com a segunda GPU. Por isso, não há mudança alguma na área de poda da segunda GPU ao alterar o escore inicial. Dessa forma, ao que indica o gráfico, por haver apenas uma pequena mudança na área de poda da primeira GPU e por ser uma matriz pequena, os resultados obtidos não apresentaram um *speedup* significativo.

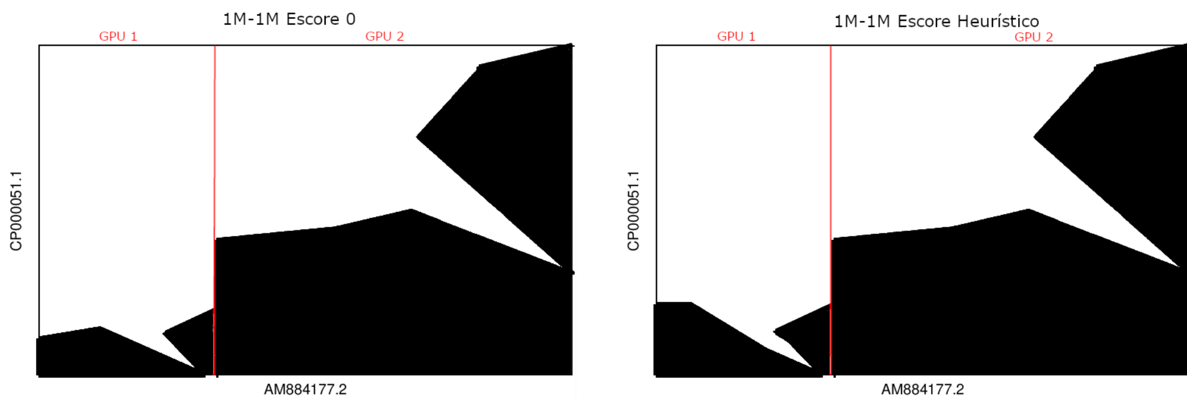


Figura 6.6: Área de poda da comparação 1M-1M com o escore 0 e o heurístico.

Na Figura 6.7, onde são apresentados os gráficos da comparação 3M-3M, é evidenciada a quase nula área de poda, que, por ser minúscula, não pôde ser representada nesse nível de precisão nos gráficos de poda. Conforme explicado nas subseções anteriores, como as sequências desta comparação são muito diferentes entre si, ambos os escores ótimo e heurístico são muito pequenos, de forma que não seja feita a poda de forma efetiva. Assim, a área de poda é representada pela linha reta, que indica a área de poda quase nula.

Na Figura 6.8 é exibida a mudança gerada ao alterar o *split* e o escore utilizados na comparação 5M-5M. Nela, é possível notar que, apesar de diminuir levemente a área de poda obtida pela GPU 1, há também um aumento na área de poda da GPU 2, o que torna a maior distribuição de área para a GPU 2 algo importante para melhorar o desempenho. Além disso, assim como na Figura 6.8, a Figura 6.9 apresenta uma mudança significativa ao alterar o *split* e o escore. Porém, neste caso, para o escore heurístico, há um aumento percentual na área de poda em ambas as GPUs, sobretudo na GPU 2, o que pode justificar o grande aumento de *speedup* obtido nesta comparação.

Por fim, a Tabela 6.6 reúne os percentuais das áreas podadas nas figuras anteriores. Exceto na comparação 3M-3M, é notável o aumento da área de poda obtido ao modificar

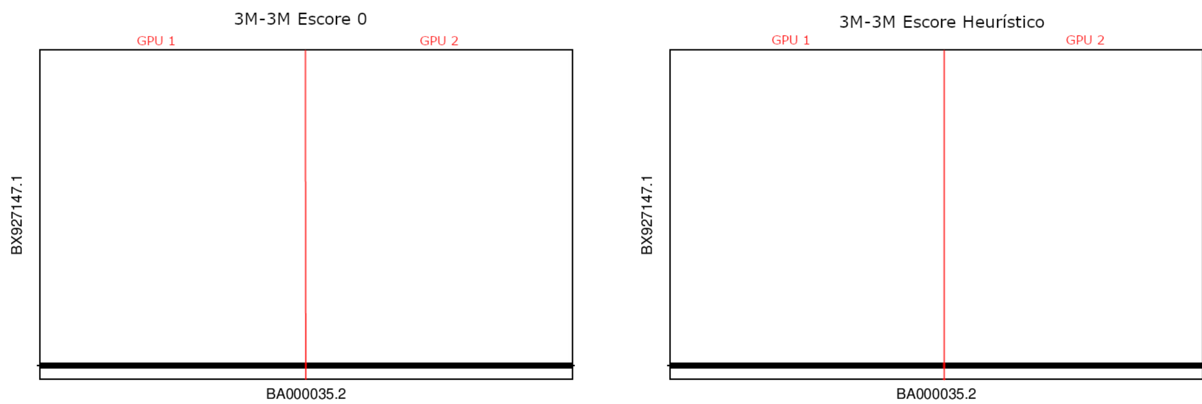


Figura 6.7: Área de poda da comparação 3M-3M com o escore 0 e o heurístico.

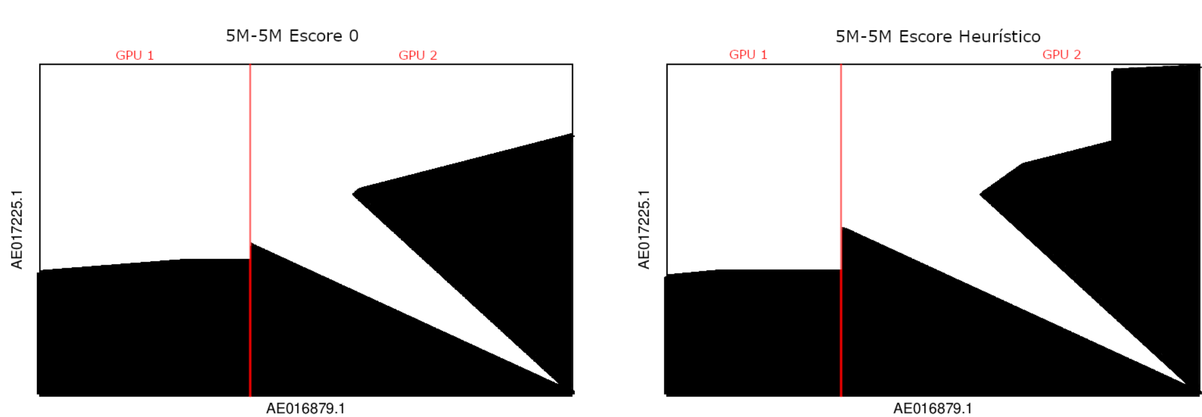


Figura 6.8: Área de poda da comparação 5M-5M com o escore 0 e o heurístico.

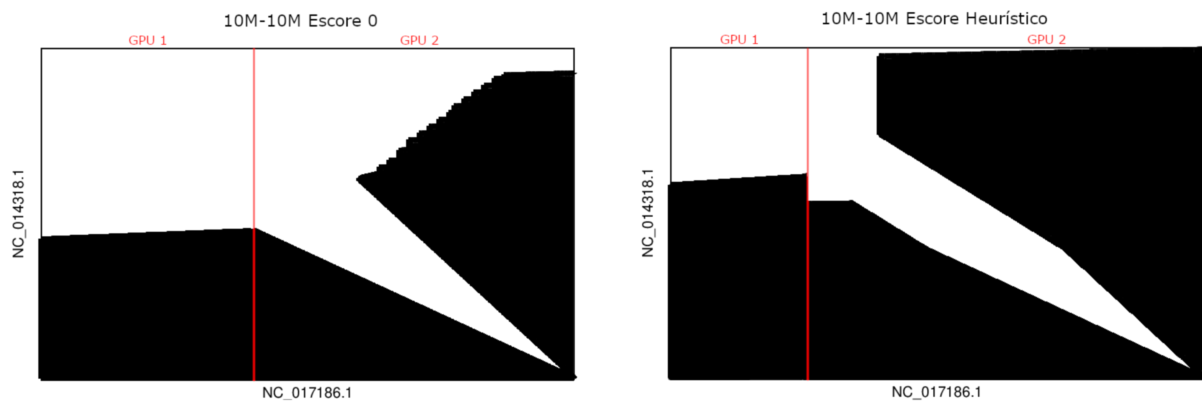


Figura 6.9: Área de poda da comparação 10M-10M com o escore 0 e o heurístico.

o *split* e escore inicial para as comparações realizadas, sobretudo na 10M-10M. Assim, conforme os resultados encontrados, é possível observar que, para essas comparações,

Tabela 6.6: Percentual de área podada para cada comparação.

	Escore 0					**Escore Heurístico**			
Comp.	**Split**	**GPU 1**	**GPU 2**	***Total***		**Split**	**GPU 1**	***GPU 2***	***Total***
1M-1M	1,2	11.6%	70.9%	51.1%		1,2	15.1%	70.9%	52.3%
3M-3M	1,1	0.1%	0.1%	0.1%		1,1	0.1%	0.1%	0.1%
5M-5M	2,3	38.8%	49.1%	44.9%		1,2	33.2%	55.7%	48.2%
10M-10M	2,3	43.3%	57.1%	51.5%		1,3	60.7%	74.9%	71.3%

pode-se haver uma melhoria na área de poda, quando a estratégia de otimização da área de poda é utilizada em combinação com a modificação do *split*.

Capítulo 7

Conclusão e Trabalhos Futuros

O presente Trabalho de Graduação apresentou o projeto, a implementação e a avaliação do Módulo de Otimização da Área de Descarte, acoplado à ferramenta MASA-CUDAlign-MultiBP. A implementação consistiu em um módulo que obtém um escore heurístico a partir da ferramenta *Lastz* e transmite este valor para a primeira GPU, que o compartilha gradativamente para as demais, utilizando a estratégia de anel. Para a realização dos testes, foi utilizado um *cluster* na nuvem AWS com uma instância mestre, com a qual é executado o módulo de obtenção de escore heurístico, e duas instâncias computacionais, cada qual com uma GPU NVIDIA T4 Tensor Core, as quais são responsáveis pela execução da ferramenta MASA-CUDAlign-MultiBP e, consequentemente, compartilhamento do escore entre si.

Apesar de se supor originalmente que apenas o compartilhamento do escore inicial heurístico já resultaria em uma melhoria de desempenho, notou-se que esta mudança sozinha não apresentou resultados muito relevantes. Dessa forma, adicionou-se o fator de encontrar um *split* adequado para a comparação, pois objetivou-se encontrar o melhor equilíbrio entre a perda de tempo da segunda GPU devido à espera pela coluna de borda vindo da primeira GPU (com *split* 1,1), e a perda de desempenho resultante pela menor paralelização decorrente da disparidade de número de colunas processadas entre GPUs (como o *split* 1,3, por exemplo).

Quanto aos resultados obtidos, por ser composta por sequências muito parecidas e ter sido encontrado um escore heurístico bom, a comparação 10M-10M apresentou uma grande melhoria em tempo de execução e área de poda. Ademais, os melhores *splits* diferentes entre o escore 0 e o heurístico (2,3 e 1,3, respectivamente) podem indicar que, com escores iniciais grandes, a perda de paralelização pouco afeta os ganhos obtidos pelo aumento da área de poda e diminuição de espera pela coluna de borda. Na comparação 5M-5M, composta por sequências parecidas, o escore heurístico encontrado pela ferramenta *Lastz* foi pequeno em comparação ao ótimo, resultando em uma melhoria de

desempenho e aumento de área de poda pequenos. Além disso, notou-se que os melhores *splits* com escore 0 e heurístico foram 2,3 e 1,2, respectivamente, o que pode indicar que a perda de paralelização com *split* 1,2 é compensada pela diminuição da espera pela coluna de borda, para esta comparação.

Na comparação 1M-1M, composta por sequências muito curtas, com melhores *splits* iguais (1,2) e escore inicial de tamanho moderado, houve uma melhoria de desempenho muito pequena. Isto também pode ter sido corroborado pelo fato do tempo de execução desta comparação ser menor que 5 segundos, o intervalo em que ocorre o compartilhamento de escore entre as GPUs, de forma que o aumento da área de poda só tenha ocorrido na primeira GPU. Por fim, devido ao fato de ser composta por sequências muito diferentes, na comparação 3M-3M tanto o escore heurístico quanto o ótimo são pequenos, de forma que a área de poda seja praticamente inexistente, resultando em melhora alguma. Assim, os resultados obtidos mostram a potencial influência que o escore inicial e o *split* possuem na execução da ferramenta MASA-CUDAlign-MultiBP.

Por fim, como trabalhos futuros sugere-se:

- A implementação da paralelização entre o módulo de obtenção do escore heurístico e o MASA-CUDAlign-MultiBP: isto se deve ao fato de haver um *overhead* gerado pela execução da ferramenta heurística, que é menor, porém, proporcional ao tempo de execução com a ferramenta ótima. No entanto, por meio da paralelização, se tornaria possível executar a ferramenta heurística simultaneamente ao MASA-CUDAlign-MultiBP e, quando fosse encontrado o escore heurístico, este seria transmitido à primeira GPU executando a ferramenta. Assim, o *overhead* originário da execução sequencial deixaria de existir.

- O estudo acerca das ferramentas de obtenção de escore heurístico: apesar de fornecer bons escores heurísticos em algumas das comparações, a ferramenta *Lastz* deixou a desejar na comparação 5M-5M. Além disso, o tempo de execução do *Lastz* muitas vezes chegava próximo do tempo de execução do MASA-CUDAlign-MultiBP, de forma que o *overhead* causado pudesse encobrir os possíveis ganhos. Por último, em comparações de sequências muito grandes, como 47M, a ferramenta apresenta um erro ao retornar o escore heurístico 0. Assim, seria interessante um estudo mais detalhado acerca de ferramentas heurísticas, a fim de encontrar uma que apresente resultados mais promissores.

- O estudo acerca do split adequado para cada situação: ao longo do desenvolvimento do projeto notou-se o grande impacto que a mudança de *split* apresenta no desempenho da ferramenta MASA-CUDAlign-MultiBP. Dessa forma, seria interessante ser feito um estudo para investigar a correlação entre o *split*, o escore inicial

e as sequências da comparação, de forma a encontrar um *split* adequado para cada situação.

Referências

[1] Durbin, Richard, Sean R. Eddy, Anders Krogh e Graeme Mitchison: *Biological Sequence Analysis: Probabilistic Models of Proteins and Nucleic Acids*. Cambridge University Press, 1998. ix, 5, 6

[2] Myers, E. e W. Miller: *Optimal alignments in linear space*. Computer applications in the biosciences : CABIOS, 4 1:11–7, 1988. ix, 10, 11

[3] Nvidia: *Nvidia tesla v100 gpu architecture*. https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf, August 2017. (Accessed on 12/10/2021). ix, 13, 14, 15

[4] Jia, Zhe, Marco Maggioni, Benjamin Staiger e Daniele Paolo Scarpazza: *Dissecting the nvidia volta gpu architecture via microbenchmarking*. ArXiv, abs/1804.06826, 2018. ix, 14, 16

[5] Nvidia: *Nvidia cuda architecture, introduction overview*. https://developer.download.nvidia.com/compute/cuda/docs/CUDA_Architecture_Overview.pdf, April 2009. (Accessed on 19/10/2021). ix, 16

[6] Nvidia: *Cuda c++ programming guide*. https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html, September 2021. (Accessed on 19/10/2021). ix, 17, 18

[7] Landaverde, Raphael, Tiansheng Zhang, Ayse K. Coskun e Martin Herbordt: *An investigation of unified memory access performance in cuda*. Em *2014 IEEE High Performance Extreme Computing Conference (HPEC)*, páginas 1–6, 2014. ix, 19

[8] Harris, Mark: *How nvlink will enable faster, easier multi-gpu computing*, November 2014. https://developer.nvidia.com/blog/how-nvlink-will-enable-faster-easier-multi-gpu-computing/, (Accessed on 25/10/2021). ix, 20

[9] Sandes, E. F. O. e A. C. M. A. Melo.: *Algoritmos paralelos exatos e otimizações para alinhamento de sequências biológicas longas em plataformas de alto desempenho*. Tese de Doutorado. Universidade de Brasília, Brasília, Brasil, 2016. https://repositorio.unb.br/handle/10482/20248. ix, 2, 6, 12, 22, 23, 24, 26, 27, 28, 29, 30, 31, 35

[10] Jr., Marco Figueiredo e Alba C. M. A. Melo: *Comparação paralela de sequências biológicas em múltiplas gpus com descarte de blocos e estratégias de distribuição de*

carga. Tese de Doutorado. Universidade de Brasília, Brasília, Brasil, 2021. ix, xi, 2, 12, 13, 16, 31, 32, 34, 36, 40

[11] Mount, David W.: *Bioinformatics: Sequence and Genome.* Cold Spring Harbor Laboratory Press,U.S., 2004. 1, 4, 5

[12] Pearson, W R e D J Lipman: *Improved tools for biological sequence comparison.* Proceedings of the National Academy of Sciences, 85(8):2444–2448, 1988. `https://www.pnas.org/doi/abs/10.1073/pnas.85.8.2444`. 1

[13] Altschul, Stephen F., Warren Gish, Webb Miller, Eugene W. Myers e David J. Lipman: *Basic local alignment search tool.* Journal of Molecular Biology, 215(3):403–410, 1990, ISSN 0022-2836. `https://www.sciencedirect.com/science/article/pii/S0022283605803602`. 1

[14] Needleman, S. B. e 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 3:443–53, 1970. 1, 5

[15] Smith, T.F. e M.S. Waterman: *Identification of common molecular subsequences.* Journal of Molecular Biology, 147(1):195–197, 1981, ISSN 0022-2836. `https://www.sciencedirect.com/science/article/pii/0022283681900875`. 1, 2, 7

[16] Manavski, Svetlin e Giorgio Valle: *Cuda compatible gpu cards as efficient hardware accelerators for smith-waterman sequence alignment.* BMC Bioinformatics, 9, 2008. 2

[17] Liu, Yongchao, Douglas L. Maskell e Bertil Schmidt: *Cudasw++: optimizing smith-waterman sequence database searches for cuda-enabled graphics processing units.* BMC Research Notes, 2, 2009. 2

[18] Gotoh, O.: *An improved algorithm for matching biological sequences.* Journal of Molecular Biology, 162(3):705–708, 1982, ISSN 0022-2836. `https://www.sciencedirect.com/science/article/pii/0022283682903989`. 9

[19] Hirschberg, D.: *A linear space algorithm for computing maximal common subsequences.* Communications of the ACM, 18:341 – 343, 1975. 10

[20] Nickolls, John e William J. Dally: *The gpu computing era.* IEEE Micro, 30(2):56–69, 2010. 12, 13

[21] Patterson, David A. e John L. Hennessy: *Computer Organization and Design, Fifth Edition: The Hardware/Software Interface.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 5th edição, 2013, ISBN 0124077269. 14

[22] Harris, Mark: *Inside pascal: Nvidia's newest computing platform.* `https://developer.nvidia.com/blog/inside-pascal/`, April 2016. (Accessed on 26/10/2021). 14

[23] Nvidia: *Nvidia cuda compute unified device architecture*. https://developer.download.nvidia.com/compute/cuda/1.0/NVIDIA_CUDA_Programming_Guide_1.0.pdf, June 2007. (Accessed on 19/10/2021). 15

[24] Nvidia: *Nvidia cuda toolkit v6.0*, February 2014. https://developer.download.nvidia.com/compute/cuda/6_0/rel/docs/CUDA_Toolkit_Release_Notes.pdf, (Accessed on 24/10/2021). 19

[25] Sakharnykh, Nikolay: *Maximizing unified memory performance in cuda*, November 2017. https://developer.nvidia.com/blog/maximizing-unified-memory-performance-cuda/, (Accessed on 24/10/2021). 19

[26] Foley, Denis e John Danskin: *Ultra-performance pascal gpu and nvlink interconnect*. IEEE Micro, 37(2):7–17, 2017. 20

[27] *Nvidia visual profiler*. https://developer.nvidia.com/nvidia-visual-profiler. (Accessed on 26/10/2021). 21

[28] *Cuda-gdb*. https://developer.nvidia.com/cuda-gdb. (Accessed on 26/10/2021). 21

[29] *Cuda-memcheck*. https://developer.nvidia.com/cuda-memcheck. (Accessed on 26/10/2021). 21

[30] *Developing a linux kernel module using gpudirect rdma*. https://docs.nvidia.com/cuda/gpudirect-rdma/index.html, October 2021. (Accessed on 26/10/2021). 21

[31] Harris, M. e K. Perelygin: *Cooperative groups: Flexible cuda thread programming*. https://developer.nvidia.com/blog/cooperative-groups/, October 2017. (Accessed on 26/10/2021). 21

[32] Ramarao, Pramod: *Cuda 11 features revealed*. https://developer.nvidia.com/blog/cuda-11-features-revealed/, May 2020. (Accessed on 26/10/2021). 21

[33] Sandes., E. F. O.: *Comparação paralela de sequências biológicas longas utilizando unidades de processamento gráfico (gpus)*. Dissertação de Mestrado. Universidade de Brasília, Brasília, Brasil, 2011. 22

[34] Sandes, E. F. O. e A. C. M. A. Melo.: *Smith-waterman alignment of huge sequences with gpu in linear space*. Em *2011 IEEE International Parallel Distributed Processing Symposium*, páginas 1199–1211, 2011. 23

[35] Sandes, E. F. O. e A. C. M. A. Melo.: *Retrieving smith-waterman alignments with optimizations for megabase biological sequences using gpu*. IEEE Transactions on Parallel and Distributed Systems, 24(5):1009–1021, 2013. 25

[36] O. Sandes, Edans F. de, Guillermo Miranda, Alba C.M.A. de Melo, Xavier Martorell e Eduard Ayguadé: *Cudalign 3.0: Parallel biological sequence comparison in large gpu clusters*. páginas 160–169, 2014. 25

[37] Jr., Marco Figueiredo: *Masa-opencl : comparação paralela de sequências biológicas longas em gpu.* Dissertação de Mestrado. Universidade de Brasília, Brasília, Brasil, 2015. 31

[38] Lopes, Rafael A., Samuel Thibault e Alba C. M. A. Melo: *Masa-starpu: Parallel sequence comparison with multiple scheduling policies and pruning.* páginas 225–232, 2020. 31

[39] Augonnet, Cédric, Samuel Thibault, Raymond Namyst e Pierre André Wacrenier: *Starpu: A unified platform for task scheduling on heterogeneous multicore architectures.* Volume 23, agosto 2009, ISBN 978-3-642-03868-6. 31

[40] Dagum, L. e R. Menon: *Openmp: an industry standard api for shared-memory programming.* IEEE Computational Science and Engineering, 5(1):46–55, 1998. 33, 34

[41] *Openmp application programming interface.* `https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5.0.pdf`, November 2018. (Accessed on 09/28/2021). 34

[42] Leite, Alessandro Ferreira: *A user-centered and autonomic multi-cloud architecture for high performance computing applications.* Tese (Doutorado em Informática) - Universidade de Brasília, Brasília, Brasil, 2014. 35

[43] *Amazon elastic compute cloud - user guide for linux instances.* `https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/ec2-ug.pdf#concepts`. (Accessed on 23/02/2022). 35

[44] *Aws command line interface - user guide for version 2.* `https://docs.aws.amazon.com/cli/latest/userguide/aws-cli.pdf#cli-configure-quickstart`. (Accessed on 10/03/2022). 36

[45] *Aws parallelcluster - aws parallelcluster user guide.* `https://docs.aws.amazon.com/parallelcluster/latest/ug/aws-parallelcluster-ug.pdf#parallelcluster-version-2`. (Accessed on 06/03/2022). 36

[46] *The mummer 3 manual.* `http://mummer.sourceforge.net/manual/`. (Accessed on 02/03/2022). 40, 41

[47] *Lastz - release 1.04.15.* `https://lastz.github.io/lastz/`, August 2021. (Accessed on 02/03/2022). 40, 43

[48] W. Sousa, P. Mizuno: *Sequences.* `https://github.com/pvvm/sequences`, March 2022. (Accessed on 10/03/2022). 49