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

MAPA: Módulo Assíncrono para Aceleração de Poda da Ferramenta CUDAlign

Matheus Trajano do Nascimento

Monografia apresentada como requisito parcial
para conclusão do Curso de Engenharia da Computação

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

Brasília
2023

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

MAPA: Módulo Assíncrono para Aceleração de Poda da Ferramenta CUDAlign

Matheus Trajano do Nascimento

Monografia apresentada como requisito parcial
para conclusão do Curso de Engenharia da Computação

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

Prof. Dr. Edison Ishikawa
Universidade de Brasília

Prof. Dr. Eduardo Adilio P. Alchieri
Universidade de Brasília

Prof. Dr. João Luiz Azevedo de Carvalho
Coordenador do Curso de Engenharia da Computação

Brasília, 18 de dezembro de 2023

Agradecimentos

Quero expressar gratidão à minha família, cujo apoio e compreensão tornaram possível a conclusão deste trabalho. Agradeço aos meus amigos por estarem presentes ao longo dessa jornada e proporcionarem momentos de leveza e descontração. Agradeço também a minha orientadora Profa. Dra. Alba Melo, pela orientação perspicaz, paciência e contribuições valiosas que foram fundamentais para o desenvolvimento deste trabalho. Obrigado a todos que, de alguma forma, compartilharam deste percurso acadêmico comigo.

Resumo

O alinhamento de sequências biológicas desempenha um papel essencial na bioinformática, sendo uma técnica fundamental no entendimento das moléculas de DNA, RNA e proteínas. O CUDAlign se destaca ao usar GPUs para obter alinhamentos ótimo de maneira eficiente. No CUDAlign 2.1, foi proposto o processo de poda, que envolve o descarte de blocos da matrizes de programação dinâmica que não contribuem para o alinhamento ótimo, acelerando o algoritmo de alinhamento. O módulo APA (Alinhamento com Poda Agilizada) foi concebido para aprimorar o processo de poda de blocos, gerando um escore heurístico que será utilizado como escore inicial do CUDAlign, buscando ampliar a área de poda. No entanto, a execução síncrona do APA introduz um *overhead* no tempo total de execução. Este trabalho apresenta o projeto do MAPA (Módulo de Aceleração de Poda Assíncrono), com o objetivo de eliminar esse *overhead*. Os resultados obtidos mostram um ganho de desempenho considerável em relação ao módulo APA.

Palavras-chave: alinhamento de sequências biológicas, poda de blocos, GPU

Abstract

Sequence alignment plays a crucial role in molecular biology and bioinformatics, serving as a fundamental technique in understanding DNA, RNA, and proteins. CUDAlign stands out by utilizing GPUs to efficiently achieve optimal alignments. In CUDAlign 2.1, the pruning process was introduced, involving the removal of the computation of the blocks from dynamic programming matrices that do not contribute to optimal alignment, thus accelerating the alignment algorithm. The APA module (Agile Pruning Alignment) was designed to enhance the block pruning process, generating a heuristic score used as the initial score for CUDAlign, aiming to expand the pruning area. However, the synchronous execution of APA introduces overhead in the overall execution time. This work presents the design of MAPA (Asynchronous Pruning Acceleration Module) with the goal of eliminating this overhead. The results obtained demonstrate a considerable performance improvement compared to the APA module.

Keywords: Sequence alignment, block pruning, GPU

Sumário

1 Introdução **1**

2 Alinhamento de Sequências Biológicas **4**

2.1 Definições 4

2.2 Algoritmos Exatos de Alinhamento 5

2.2.1 Algoritmo de Needleman–Wunsch 6

2.2.2 Algoritmo de Smith-Waterman 8

2.2.3 Algoritmo de Gotoh 9

2.2.4 Algoritmo de Myers-Miller 11

3 Evolução do CUDAlign **13**

3.1 CUDAlign 1.0 13

3.1.1 Paralelismo Externo 14

3.1.2 Paralelismo Interno 14

3.1.3 Otimizações 16

3.2 CUDAlign 2.0 17

3.3 CUDAlign 2.1 18

3.3.1 Processo de Poda 18

3.3.2 Otimização *Block Pruning* 19

3.4 CUDAlign 3.0 20

3.5 CUDAlign 4.0 20

3.5.1 *Pipelined Traceback* 21

3.5.2 *Incremental Speculative Traceback* 22

3.6 Módulo de Alinhamento com Poda Agilizada (APA) 22

4 Projeto do Módulo de Aceleração de Poda Assíncrono (MAPA) **24**

4.1 Objetivo 24

4.2 Visão Geral 25

4.3 Sub-Módulo de Preparação 26

4.4 Sub-Módulo de Execução em Paralelo 27

5 Resultados Experimentais **31**
5.1 Ambiente de Teste 31
5.2 Sequências Utilizadas 32
5.3 Comparação dos Resultados 32
5.3.1 Resultados Modo *Full* 33
5.3.2 Resultados Modo *Trimmed* 34
5.4 Resultados com Simulação de 90% do escore ótimo 38

6 Conclusão **41**

Referências **43**

Anexo **45**

I Artigo apresentado no WPOS/WCOMP 2023 **46**

Lista de Figuras

2.1 Ilustração do algoritmo de Needleman-Wunsch. 7
2.2 Matriz de Programação Dinâmica para o alinhamento global entre as sequências A e B 8
2.3 Alinhamento Global Ótimo por Needleman-Wunsch. 8
2.4 Matriz de Programação Dinâmica para o alinhamento local entre as sequências A e B 10
2.5 Alinhamento Local Ótimo das sequências A e B por Smith-Waterman. . . 10
2.6 Ilustração do algoritmo de Myers-Miller (Adaptado de [1]). 12

3.1 Matriz segmentada em blocos formando um *grid*. 15
3.2 Bloco de um *Grid* com 48 células e processado pelas *threads* T_0, T_1 e T_2. . 15
3.3 Delegação de Células do CUDAlign 1.0 [2]. 16
3.4 Etapas de execução do CUDAlign 2.0 [3]. 18
3.5 Definições geométricas do Block Pruning. [4]. 19
3.6 Divisão de colunas entre múltiplas GPUs. [5]. 20
3.7 Linhas de tempo para o *Pipelined Traceback* [5]. 21
3.8 Linhas de tempo para o *Incremental Speculative Traceback* [5]. 22

4.1 Arquitetura do MAPA 25
4.2 Algoritmo de funcionamento do MAPA. 27

5.1 Gráficos do tempo total de execução no modo *Full*. 36
5.2 Gráficos da porcentagem total de poda de blocos no modo *Full*. 36
5.3 Gráficos do tempo total de execução no modo *Trimmed*. 37
5.4 Gráficos da porcentagem total de poda de blocos no modo *Trimmed*. . . . 38
5.5 Gráficos do tempo total de execução da simulação. 39
5.6 Gráficos da porcentagem total de poda de blocos da simualação. 40

Lista de Tabelas

5.1 Tabela dos pares comparados e os nomes científicos dos organismos. 33
5.2 Tabela dos pares comparados, seus identificadores e respectivos escores. . . 33
5.3 Tempo de execução total e do BLASTn com o APA (síncrono) e o MAPA (assíncrono) no modo *Full*. 35
5.4 Quantidade de blocos podados (%) com o APA (síncrono) e o MAPA (assíncrono) no modo *Full*. 35
5.5 Tempo de execução total e do BLASTn com o APA (síncrono) e o MAPA (assíncrono) no modo *Trimmed*. 37
5.6 Quantidade de blocos podados (%) com o APA (síncrono) e o MAPA (assíncrono) no modo *Trimmed*. 37
5.7 Tempo total de execução e porcentagem de blocos podados da simulação. . 39

Capítulo 1

Introdução

Diante dos avanços tecnológicos, observou-se um aumento significativo na taxa de produção de dados biológicos, principalmente relacionados a moléculas orgânicas, como o DNA (ácido desoxirribonucleico), RNA (ácido ribonucleico) e proteínas. Dessa forma, a utilização de sistemas computacionais para lidar com esse grande volume de dados se tornou indispensável. Nesse contexto, a Bioinformática surgiu como uma disciplina localizada na intersecção entre a biologia e a informática. Um dos objetivos dessa área é a organização dessa vasta quantidade de informações biológicas, permitindo consultas e a inserção de novos dados. Além disso, dentro da Bioinformática, são desenvolvidas ferramentas capazes de analisar essas informações, gerando resultados que impulsionam o avanço das pesquisas na área [6].

Sequências biológicas são uma parte de uma macromolécula (polímero), como por exemplo o DNA e as proteínas, formadas por moléculas estruturais chamadas de monômeros [7]. Uma das tarefas mais fundamentais da Bioinformática é o alinhamento de sequências, onde duas ou mais sequências, que são representadas digitalmente como conjunto de caracteres, são comparadas com o objetivo de se encontrar padrões em comum entre as sequências alinhadas [8].

Para efetuar o alinhamento de sequências, foram criados diversos algoritmos, utilizando abordagens tanto exatas quanto heurísticas. Os algoritmos exatos são capazes de obter o alinhamento ótimo, que identifica a correspondência mais precisa entre as sequências comparadas, dentre eles, destacam-se o Needleman Wunsch [9], Smith Waterman [10], Gotoh [11] e Myers-Miller [1]. Esses algoritmos calculam o nível de similaridade entre as sequências por meio do processamento de uma matriz de programação dinâmica, utilizando um sistema de escore que atribui pesos aos *matches* (caracteres iguais), *missmatches* (caracteres distintos) ou *gaps* (espaçamentos adicionados a uma das sequências). Os algoritmos heurísticos buscam otimizar o desempenho, dessa forma, geram apenas resultados aproximados, que possuem algumas imprecisões de correspondência e, portanto,

não são totalmente corretos. Um dos algoritmos heurísticos mais conhecidos é o o BLAST [12] um dos que mais se destaca.

Apesar de conseguirem obter o alinhamento ótimo, algoritmos exatos são computacionalmente custosos, podendo levar bastante tempo para executar dependendo do tamanho das sequências comparadas. Buscando acelerar esse processo, foi desenvolvido a ferramenta CUDAlign [5], que aproveita a alta capacidade de processamento das GPUs (*Graphics Processing Units*), capazes de realizar operações com alto nível de paralelismo, para gerar o melhor alinhamento possível entre as sequências.

O CUDALign obtém o alinhamento ótimo ao executar o algoritmo de Smith-Waterman [10] em GPU, utilizando a plataforma CUDA [13] da NVIDIA. Foram desenvolvidas diversas versões dessa ferramenta ao longo do tempo. Na versão 2.1 do CUDAlign [4], foi introduzido a técnica de *Block Prunning*, onde blocos de células da matriz de programação dinâmica que não contribuem para o cálculo do alinhamento ótimo são descartados, evitando processamento desnecessário. O limiar que define quais desses blocos serão descartados é baseado no melhor escore calculado até o momento, que é inicializado com o valor zero no começo da execução do CUDALign.

Com o objetivo de agilizar ainda mais a execução do CUDALign, otimizando o processo de *Block Prunning*, foi desenvolvido o módulo APA (Módulo de Alinhamento com Poda Agilizada) [14]. Esse módulo inicia a execução do CUDALign com o valor do melhor escore diferente de zero, buscando ampliar a área de poda e, consequentemente, diminuir o tempo total de execução. O valor utilizado como melhor escore inicial é o escore gerado por um algoritmo heurístico de alinhamento, que é executado antes do CUDAlign. O algoritmo utilizado no APA foi o BLAST, mais especificamente a ferramenta BLASTn. Porém, por executar de forma síncrona, ou seja, o APA executa antes do CUDAlign, esse módulo acaba gerando um *overhead* no tempo total de execução, que muitas vezes pode acabar eliminando o ganho de tempo gerado pelo próprio módulo APA.

Portanto, buscando resolver esse problema de *overhead*, o objetivo do presente trabalho de graduação é projetar, implementar e avaliar uma solução assíncrona para o uso do escore heurístico na ferramenta CUDAlign. A solução proposta, chamada MAPA (Módulo de Aceleração de Poda Assíncrono), executa o BLASTn e o CUDAlign em paralelo e, ao final da execução do BLASTn, disponibiliza o escore heurístico em uma área de memória compartilhada, de onde o CUDAlign irá recuperá-lo e utilizá-lo para substituir o melhor escore calculado até então, caso seja maior.

O restante deste documento está organizado como se segue. O Capítulo 2 fornece definições básicas e explica os algoritmos utilizados no alinhamento de sequências. O Capítulo 3 mostra a evolução da ferramenta CUDAlign e apresenta o módulo APA. O Capítulo 4 mostra o processo de desenvolvimento do MAPA. O Capítulo 5 detalha como

foi configurado o ambiente de teste e analisa os resultados encontrados. Por fim, o Capítulo 6 mostra as conclusões e sugere trabalhos futuros.

Capítulo 2

Alinhamento de Sequências Biológicas

O alinhamento de sequências biológicas é uma das partes mais fundamentais da Bioinformática. Nesse capitulo, são abordados as definições básicas a respeito dessa área na Seção 2.1 e, em seguida, são apresentados na Seção 2.2 os algoritmos mais relevantes utilizados na comparação de sequências.

2.1 Definições

Sequência biológica é uma sequência formada por moléculas estruturais básicas chamadas monômeros e, dessa forma, uma sequência é parte de uma macromolécula (polímero) [7]. Dois exemplos conhecidos de polímeros são: o DNA, um tipo de ácido nucleico que porta todas as informações genéticas de um individuo, e as proteínas, estruturas compostas de aminoácidos que possuem papel importante nos processos biológicos dos organismos.

Um alfabeto pode ser definido como um conjunto finito de símbolos que representam os monômeros de uma sequência biológica, sendo que, no contexto computacional, esses símbolos são representados por caracteres de uma *string* [8]. O alfabeto $\alpha = \{A, C, T, G\}$ representa as bases nitrogenadas presentes nos nucleotídeos que formam o DNA, já os 20 aminoácidos que formam as proteínas são representados pelo alfabeto $\alpha = \{A, C, D, E, F, G, H, I, K, L, M, N, P, Q, R, S, T, V, W, Y\}$.

Novas sequências biológicas são criadas a partir de sequências já existentes por meio de (a) mutações comuns; ou (b) através de inserções/deleções em uma das sequências, de modo que podem ter se divergido em algum momento a partir de ancestrais comuns. Portanto, na análise de sequências, uma das tarefas mais básicas é descobrir se diferentes sequências estão relacionadas [15], uma vez que essa informação é útil para esclarecer questões a respeito da função, estrutura e evolução das sequências biológicas comparadas.

Na computação, essa análise de semelhança é feita através de um alinhamento de sequências, sendo esse um processo de comparação entre duas sequências com o objetivo de achar um conjunto ordenado de caracteres individuais ou padrões de caracteres presentes na mesma ordem em ambas as sequências [8]. Para isso, as sequências são escritas em duas linhas, permitindo o alinhamento de caracteres idênticos em uma mesma coluna (*match*), caracteres diferentes na mesma coluna (*mismatch*) ou de um caractere de uma das sequências com um *gap* na outra. Os *mismatches* representam mutações e os *gaps* inserções ou deleções de caracteres das sequências.

Para que se possa determinar a qualidade de um alinhamento é utilizado um sistema de escore aditivo, onde a pontuação total do alinhamento é dado pela soma do escore positivo ou negativo de cada par de caracteres alinhado (*match* ou *mismatch*) mais as penalidades devido a presença de *gaps*. A penalidade total de um *gap* de tamanho g pode ser calculada utilizando um *linear score* (Equação 2.1), onde o custo d é mesmo para todas as aparições de *gaps* ou através do *affine score* (Equação 2.2), em que há uma penalidade de abertura d para o primeiro *gap* e uma penalidade menor de extensão e para o restante [8].

$$\gamma(g) = -gd \tag{2.1}$$

$$\gamma(g) = -d - (g-1)e \tag{2.2}$$

Existem dois tipos principais de alinhamentos, o global e o local. O alinhamento global é utilizando quando são comparadas duas sequências que possuem o mesmo tamanho e são similares, nessa situação tenta-se alinhar toda a sequência, agrupando o máximo de pares de caracteres possíveis. Por outro lado, o alinhamento local é utilizado em sequências de tamanhos diferentes que são parecidas em algumas regiões mas com pouca similaridade em outras, dessa forma, apenas áreas com grandes quantidades de *matches* são alinhadas [8].

2.2 Algoritmos Exatos de Alinhamento

Existem inúmeras possibilidades de alinhamentos diferentes entre duas sequências quando se considera a presença de *gaps*. Desse modo, é computacionalmente impossível listar todas elas [15]. Portanto, são empregados algoritmos de programação dinâmica, que utilizam um sistema de escore aditivo descrito na Seção 2.1, para obter o alinhamento ótimo entre as sequências, que se trata do melhor alinhamento possível entre as sequências comparadas. Em outras palavras, constitui a solução que melhor atende aos critérios

estabelecidos para a comparação de sequências, buscando alcançar a máxima concordância entre os elementos analisados.

Nas seções 2.2.1, 2.2.2, 2.2.3, 2.2.4 são apresentados os principais algoritmos de programação dinâmica para o alinhamento de sequências.

2.2.1 Algoritmo de Needleman–Wunsch

O algoritmo de Needleman-Wunsch [9] é um algoritmo exato, que gera o resultado ótimo para o problema do alinhamento global entre sequências, fornecendo o melhor alinhamento e seu score. A ideia é obter o alinhamento global ótimo através da técnica de programação dinâmica, que consiste em construir de maneira iterativa uma matriz composta de todos os alinhamentos ótimos possíveis entre subsequências das duas sequências de entrada. Esse algoritmo é dividido em duas etapas: geração da matriz de programação dinâmica e o *traceback*.

- **Geração da Matriz de Programação Dinâmica**

  A fim de obter o alinhamento entre duas sequências A e B, sendo seus elementos nas posições i e j representados respectivamente por a_i e b_j, é preciso construir a matriz de programação dinâmica F, indexada por i e j, onde cada célula $F(i, j)$ representa o score do alinhamento ótimo entre as subsequências $\{a_0...a_i\}$ e $\{b_0...b_j\}$. Dessa forma, o alinhamento global ótimo é construído com base nos *scores* ótimos obtidos a partir do alinhamento de subsequências menores.

  Portanto, a matriz é construída de maneira recursiva, inicializando o elemento $F(0, 0) = 0$ e preenchendo o restante das posições a partir do canto superior esquerdo até o canto inferior direito, onde, como ilustrado na Figura 2.1, cada *score* $F(i, j)$ possui três formas diferentes de ser obtido:

  1. A partir do alinhamento de a_j e b_j, sendo calculado com base no *score* do vizinho diagonal $(i-1, j-1)$ por $F(i, j) = F(i-1, j-1) + s(a_i, b_j)$.
  2. Com o alinhamento de b_j com um *gap* na sequência a, sendo o *score* dado por $F(i, j) = F(i, j-1) - d$, obtido com base na célula vizinha esquerda $(i, j-1)$.
  3. Alinhando a_j com um *gap* na sequência b e obtendo o *score* bom base no *score* da célula vizinha acima $(i-1, j)$, sendo esse dado por $F(i, j) = F(i-1, j) - d$.

  Será atribuído a $F(i, j)$ o maior dentre os três valores calculados. Em suma, toda essa relação é descrita matematicamente pela equação de recorrência (Equação 2.3), que é aplicada repetidamente até que a matriz seja preenchida. Porém, na primeira linha $(i = 0)$, os valores das células vizinhas $F(i-1, j)$ e $F(i-1, j-1)$ não estão

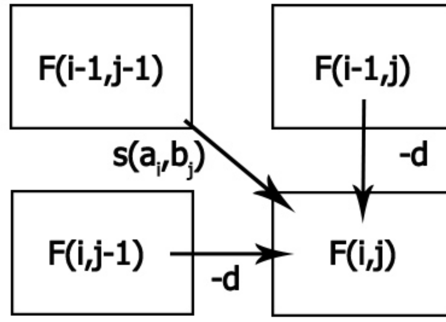

Figura 2.1: Ilustração do algoritmo de Needleman-Wunsch.

definidos, dessa forma, os scores dessa linha são calculados seguindo uma equação diferente: $F(0, j) = -jd$. De forma análoga, o mesmo acontece na primeira coluna da matriz, sendo assim, os scores dessa coluna são dados por: $F(i, 0) = -id$.

$$F(i,j) = max \begin{cases} F(i-1, j-1) + s(x_i, y_j), \\ F(i-1, j) - d, \\ F(i, j-1) - d \end{cases} \tag{2.3}$$

O *score* ótimo do alinhamento global entre as duas sequências $a_0...a_n$ e $b_0...b_m$ é dado na última célula da matriz $F(n, m)$.

- **Traceback:**

  O objetivo do *traceback* é, uma vez que matriz tenha sido calculada, achar o alinhamento global ótimo entre as sequências, para isso é preciso achar o caminho de escolhas definido pela equação de recorrência que leve desde a célula $F(n, m)$ até a origem da matriz. Dessa forma, durante a construção da matriz de programação dinâmica, é necessário identificar por meio de um ponteiro de qual das células vizinhas ($F(i-1, j-1)$, $F(i-1, j)$ ou $F(i, j-1)$) o valor do *score* de cada célula se originou. Uma observação importante é que dois ou mais alinhamentos ótimos podem existir em uma mesma matriz, uma vez que uma célula pode ter mais de um ponteiro caso a Equação 2.3 gere dois ou mais valores máximos iguais. Nesse caso, qualquer um dos caminhos pode ser escolhido arbitrariamente, todos eles irão gerar o alinhamento ótimo correto, porém, é uma convenção seguir os caminhos na diagonal durante o *traceback*, caso sejam uma das opções.

  A Figura 2.2 mostra o alinhamento global entre as sequências $A = \{A, T, G, A, A, C, T, A, C, G\}$ e $B = \{A, T, T, C, T, A, G, G, A, A\}$ utilizando algoritmo de Needleman-Wunsch, considerando a pontuação de $+1$ para *match*, -1 para

mismatch e penalidade de 2 para *gap*. O alinhamento resultante pode ser visto na Figura 2.3, o qual foi obtido a partir do *score* ótimo $F(10, 10) = -4$ e percorrendo o caminho até $F(0, 0) = 0$. Nesse exemplo, os caminhos diagonais foram priorizados, porém, é possível notar que existem outros alinhamentos ótimos, que poderiam ser obtidos seguindo os outros caminhos e também estariam corretos.

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

Figura 2.2: Matriz de Programação Dinâmica para o alinhamento global entre as sequências A e B

```
A  T  G  A  A  C  T  A  C  G  -  -
|  |              |  |  |     |
A  T  -  -  T  C  T  A  G  G  A  A
-------------------------------------
+1 +1 -2 -2 -1 +1 +1 +1 -1 +1 -2 -2  = -4
```

Figura 2.3: Alinhamento Global Ótimo por Needleman-Wunsch.

2.2.2 Algoritmo de Smith-Waterman

O alinhamento local ótimo entre duas sequências pode ser obtido através do algoritmo de Smith-Waterman [10], que funciona de maneira bastante similar ao algoritmo de Needleman-Wunsch, descrito na 2.2.1, porém, existem duas diferenças. Primeiro, como pode ser visto na (2.4), $F(i, j)$ irá assumir o valor de zero sempre que o resultado das

outras três expressões for negativo, o que corresponde a iniciar um novo alinhamento, sendo preferível fazer isso a continuar estendendo um alinhamento que possui um *score* negativo. Além disso, como consequência, a primeira linha e primeira coluna da matriz de programação dinâmica serão preenchidas com zero, diferente do que ocorria no alinhamento global.

$$F(i,j) = max \begin{cases} 0, \\ F(i-1,j-1) + s(x_i, y_j), \\ F(i-1,j) - d, \\ F(i,j-1) - d \end{cases} \tag{2.4}$$

Já a segunda diferença é que o *score* do alinhamento ótimo não será o último elemento $F(n,m)$, mas sim o maior *score* de toda matriz. Dessa forma, no algoritmo de Smith-Waterman o *traceback* irá começar a partir dessa célula, terminando ao encontrar uma célula qualquer com valor zero, que representa o começo do alinhamento.

Na Figura 2.4 o algoritmo de Smith-Waterman é utilizado no alinhamento local ótimo das sequências $A = \{A, T, G, A, A, C, T, A, C, G\}$ e $B = \{A, T, T, C, T, A, G, G, A, A\}$, com o mesmo sistema de pontuação utilizado no exemplo da 2.2.1. O maior *score* encontrado matriz foi 3, sendo que esse valor aparece em três células distintas: $F(6,8)$, $F(7,10)$ e $F(10,5)$. Nesse caso, o *traceback* pode começar de qualquer uma delas, então temos três possíveis alinhamentos locais ótimos, a Figura 2.5 mostra o alinhamento resultante ao escolher a célula $F(7,10)$ como ponto inicial.

2.2.3 Algoritmo de Gotoh

O sistema de *linear gap* utilizado no algoritmos de *Needleman-Wunsch* (Seção 2.2.1) e *Seção Smith-Waterman* (2.2.2) atribui a mesma penalidade para cada *gap* em uma sequência de *gaps*. Esse tipo de sistema não é ideal para sequências biológicas, uma vez que ele penaliza mais *gaps* longos, que são comuns de serem gerados atráves de mutações [15]. Logo, para obter um melhor resultado o algoritmo de Gotoh [11] propõe a modificação dos algoritmos anteriores para que seja utilizado o sistema de *affine gap* (Equação 2.2), onde é dado uma penalidade d para o *gap* de abertura e outra penalidade e, normalmente menor, para os *gaps* subsequentes.

Para a execução desse algoritmo são calculados três valores distintos para cada posição (i,j), onde cada um dos valores é armazenado em três matrizes distintas de tamanho nm, D, P e Q, definidas, respectivamente, pelas Equações 2.5, 2.6 e 2.7. Na matriz D cada posição corresponde ao melhor escore no caso do alinhamento (*match* ou *mismatch*) entre

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

Figura 2.4: Matriz de Programação Dinâmica para o alinhamento local entre as sequências A e B

```
 C   T   A   C   G
 |   |   |   |   |
 C   T   A   G   G
-------------------
+1  +1  +1  -1  +1  =  +3
```

Figura 2.5: Alinhamento Local Ótimo das sequências A e B por Smith-Waterman.

a_i e b_j, já a matriz P armazena em cada célula o escore da inserção de um *gap* na sequência A e, de maneira similar, Q armazena os escores referentes a *gaps* em B.

$$D(i,j) = max \begin{cases} D(i-1, j-1) + s(x_i, y_j), \\ P(i,j), \\ Q(i,j) \end{cases} \tag{2.5}$$

$$P(i,j) = max \begin{cases} D(i-1, j) - d, \\ P(i-1, j) - e, \end{cases} \tag{2.6}$$

$$Q(i,j) = max \begin{cases} D(i,j-1) - d, \\ Q(i,j-1) - e, \end{cases} \tag{2.7}$$

Em um alinhamento global, a primeira linha e coluna da matriz D são inicializadas como: $D(i,0) = \gamma(i)$ $(1 \leq i \leq n)$ e $D(0,j) = \gamma(j)$ $(1 \leq j \leq m)$, já a linha $P(0,j)$ e a coluna $Q(i,0)$ são indefinidas e portanto preenchidas com $-\infty$. No caso do alinhamento local, de maneira similar a Smith-Waterman, é necessário limitar a Equação 2.5 a valores maiores ou iguais a 0.

2.2.4 Algoritmo de Myers-Miller

O algoritmo de Gotoh (Seção 2.2.3) é capaz de resolver o problema de alinhamento de sequências com *affine gap*, porém, pelo fato da sua complexidade de espaço ser $\mathcal{O}(nm)$ e ser utilizado três matrizes, a quantidade de memória necessária pode se tornar muito grande. Para resolver esse problema, foi proposto o algoritmo de Myers-Miller [1], que utiliza o algoritmo proposto por Hischberg, capaz de encontrar a maior subsequência em comum *(LCS - Longest Common Subsequence)* em uma complexidade espacial linear [16], na solução de Gotoh. Portanto, esse método torna possível encontrar o alinhamento ótimo entre duas sequências em um espaço linear $\mathcal{O}(n+m)$.

A ideia do algoritmo é utilizar a técnica de dividir para conquistar, dividindo o cálculo das matrizes e obtendo o resultado final recursivamente. O primeiro passo do algoritmo é calcular a matriz de programação dinâmica entre as sequências A e B até a linha $\frac{m}{2}$, começando a partir do canto superior esquerdo, armazenando os *scores* das colunas dessa linha no vetor CC, caso eles terminem em *match* ou *mismatch* ou no vetor DD, caso terminem em *gaps*. Após isso, um processo similar é feito é calculando a matriz com as sequências invertidas, ou seja, começando do canto inferior direito e indo até a linha do meio, armazenando os valores da mesma forma nos vetores CC' e DD'.

No próximo passo, é calculado o *score* total $C(j)$ de cada coluna j na linha $\frac{m}{2}$ através da (2.8), sendo que, o maior valor entre os $C(j)$ é chamado de ponto médio e, como foi provado por Hischberg, faz parte do alinhamento ótimo.

$$C(j) = max \begin{cases} CC(j) + CC'(j), \\ DD(j) + DD'(j) \end{cases} \tag{2.8}$$

Esse ponto médio encontrando divide a matriz em duas menores, que então passam pelo mesmo processo descrito acima de maneira recursiva até que se chegue ao caso base, onde então o conjunto de pontos médios encontrados formam o alinhamento ótimo. Esse procedimento é ilustrado na Figura 2.6, onde é possível observar que, a medida que se

avança nas interações da recursão, algumas áreas da matriz não necessitam de serem calculadas novamente

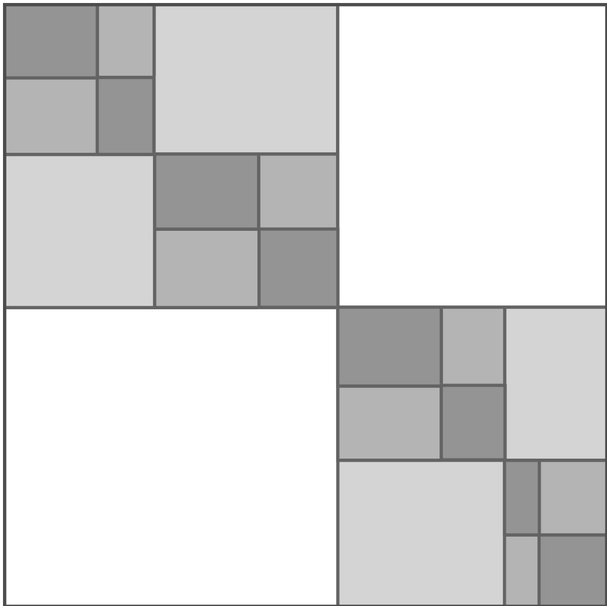

Figura 2.6: Ilustração do algoritmo de Myers-Miller (Adaptado de [1]).

Capítulo 3

Evolução do CUDAlign

Os algoritmos de programação dinâmica para comparação de sequências são capazes de gerar resultados ótimos, porém seu custo computacional é significativo, especialmente ao lidar com o alinhamento de sequências longas, uma vez que possuem complexidade de tempo $O(n^2)$. Para enfrentar esse desafio, uma abordagem promissora é aproveitar o potencial das GPUs (*Graphics Processing Units*), que são capazes de realizar operações com alto nível de paralelismo [17].

Sendo assim, foi desenvolvido o CUDAlign, uma ferramenta executada em GPU que é capaz de alinhar sequências longas em tempo reduzido. Este algoritmo foi desenvolvido com base na arquitetura CUDA (*Compute Unified Device Architecture*) da NVIDIA [13], que é uma API (*Application Programming Interface*) que permite a utilização de GPUs para processamento de propósito geral. Neste capítulo, são apresentadas as diferentes versões da ferramenta CUDAlign.

3.1 CUDAlign 1.0

O CUDAlign 1.0 [2] utiliza o algoritmo de Gotoh (Seção 2.2.3) para obter o escore ótimo no alinhamento de sequências. Nessa versão, o algoritmo é executado em memória linear, uma vez que a fase de *traceback* não é executada, dessa forma somente o escore é calculado e o alinhamento em si não é obtido.

Os elementos em uma mesma anti-diagonal de uma matriz de programação dinâmica são independentes entre si e, portanto, podem ser calculados simultâneamente. Sendo assim, no CUDAlign 1.0 foi utilizado a técnica de *wavefront*, onde o processamento das anti-diagonais da matriz é realizado em paralelo, seguindo um padrão em forma de ondas diagonais.

No CUDAlign 1.0, o *wavefront* é empregado em dois níveis distintos: paralelismo externo, abordado na Seção 3.1.1, e paralelismo interno, discutido na Seção 3.1.2.

3.1.1 Paralelismo Externo

No paralelismo externo, a matriz H de tamanho $m \times n$ é dividida em blocos, sendo cada bloco composto de $R \times C$ células, dessa forma, a matriz é segmentada em $\frac{m}{R} \times \frac{n}{C}$ blocos que formam um *grid* G. Essa divisão é ilustrada na Figura 3.1.

Os valores de R e C são definidos a partir dos parâmetros de configuração B, T e α, sendo B a quantidade de blocos executados concorrentemente, T o número de *threads* por bloco e α o número de linhas da matriz que cada *thread* é responsável por executar. Esses parâmetros são escolhidos com base nas especificações da *GPU* utilizada e resultados empíricos. Nas Equações 3.1 e 3.2 são definidas as relações utilizadas a fim de obter R e C.

$$R = \alpha * T \tag{3.1}$$

$$C = \frac{n}{B} \tag{3.2}$$

Os blocos são então agrupados em anti-diagonais, chamadas de *diagonais externas*, sendo que os blocos que formam a anti-diagonal D_k são definidos pela Equação 3.3. As diagonais externas são processadas por meio da técnica de *wavefront*, para isso, a *CPU* invoca um *kernel* (função que pode ser executada em *GPU*) para cada diagonal D_k, onde são processados em paralelo todos os blocos pertencentes a essa diagonal. Após a *GPU* terminar de excetuar uma diagonal externa, a *CPU* envia a próxima para para ser executada, esse procedimento é repetido até que se execute todo o *grid* [18].

$$D_k = \{G_i j | i + j = k\} \tag{3.3}$$

3.1.2 Paralelismo Interno

O paralelismo interno ocorre dentro de cada bloco executado, onde diferentes *threads* processam as chamadas *diagonais internas* dos blocos. Uma diagonal interna é um conjunto de células definido pela Equação 3.4, onde os valores de *(i,j)* são relativos a célula superior esquerda do bloco [18].

$$d_k = \{(i, j) | [\frac{i}{\alpha}] + j = k\} \tag{3.4}$$

Cada um dos blocos possui T *threads*, sendo que a *thread* T_k é responsável por processar da linha αk até a $\alpha k + \alpha - 1$ do bloco. De maneira equivalente ao paralelismo entre blocos, todas as *threads* de um bloco calculam em paralelo a mesma diagonal interna, de modo

	0–8	8–16	16–24
0–6	$G_{0,0}$	$G_{0,1}$	$G_{0,2}$
6–12	$G_{1,0}$	$G_{1,1}$	$G_{1,2}$
12–18	$G_{2,0}$	$G_{2,1}$	$G_{2,2}$
18–24	$G_{3,0}$	$G_{3,1}$	$G_{3,2}$
24–30	$G_{4,0}$	$G_{4,1}$	$G_{4,2}$
30–36	$G_{5,0}$	$G_{5,1}$	$G_{5,2}$

D_3

Figura 3.1: Matriz segmentada em blocos formando um *grid*.

que uma *thread* T_i processa as células da coluna j ao mesmo tempo que a *thread* T_{i+1} processa células da coluna $j-1$.

A Figura 3.2 demonstra a execução de um único bloco da matriz, tendo 10 diagonais internas e considerando 3 *threads* por bloco, sendo cada uma responsável por 2 linhas. No inicio da execução, a diagonal d_0 possui apenas uma *thread* em execução, sendo que o paralelismo máximo é atingido apenas na diagonal d_2. Dessa forma, ao considerar os blocos retangulares, é necessário recomeçar o processo de *wavefront* com apenas uma *thread* para cada um dos blocos executados, atingindo novamente o paralelismo máximo apenas na diagonal d_{T-1}.

	d_0	d_1	d_2	d_3	d_4	d_5	d_6	d_7	
T_0	0,0	0,1	0,2	0,3	0,4	0,5	0,6	0,7	
	1,0	1,1	1,2	1,3	1,4	1,5	1,6	1,7	
T_1	2,0	2,1	2,2	2,3	2,4	2,5	2,6	2,7	d_8
	3,0	3,1	3,2	3,3	3,4	3,5	3,6	3,7	
T_2	4,0	4,1	4,2	4,3	4,4	4,5	4,6	4,7	d_9
	5,0	5,1	5,2	5,3	5,4	5,5	5,6	5,7	

Figura 3.2: Bloco de um *Grid* com 48 células e processado pelas *threads* T_0, T_1 e T_2.

3.1.3 Otimizações

Com objetivo de obter um melhor desempenho, mantendo todas as *threads* em execução por boa parte da matriz, foram propostas otimizações no algoritmo.

Na primeira otimização, chamada delegação de células, os blocos são considerados como paralelogramos no contexto do paralelismo interno, já que isso torna possível manter o paralelismo máximo durante a maior parte do cálculo da matriz. Dessa forma, no momento em que a primeira *thread* atingir a última coluna do bloco, todas as outras *threads* também encerram sua execução, deixando algumas células não processadas. Essa células pendentes são então delegadas a outro bloco, pertencente a próxima diagonal externa, que fica responsável por processa-las. Esse processo é ilustrado na figura Figura 3.3.

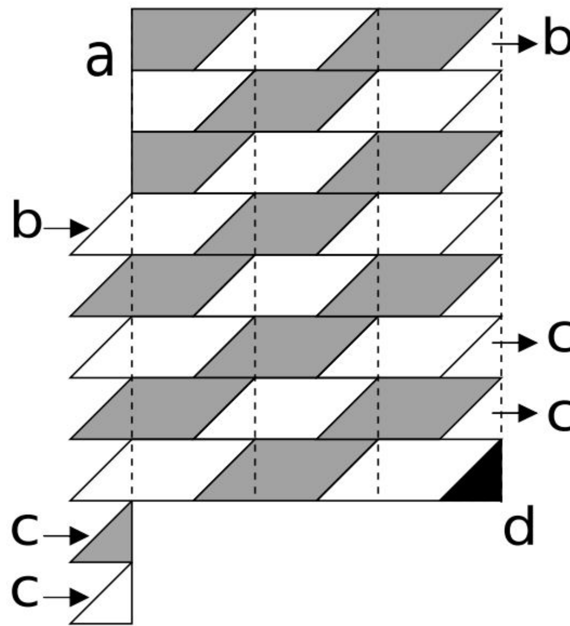

Figura 3.3: Delegação de Células do CUDAlign 1.0 [2].

Porém, há um problema na utilização do método apresentado: uma vez que os blocos de uma diagonal externa dependem dos valores calculados na diagonal anterior, as células pendentes podem prejudicar o fluxo de execução. Isso pode ocorrer caso essas células não sejam processadas pelo bloco a qual foram delegadas antes de serem consultadas por um dos outros bloco da mesma diagonal externa. Para resolver esse problema, foi proposta a otimização de fase, que divide a execução de uma diagonal externa em duas fases distintas [18].

A primeira é chamada *fase curta*, nela são processadas as primeiras $T - 1$ diagonais internas de cada bloco, calculando todas as células pendentes. Após isso, o controle é

devolvido a *CPU* para que seja feito a sincronização. Já na segunda fase, chamada *fase longa*, a *CPU* invoca novamente a *GPU* para que sejam processadas as $\frac{n}{B} - T$ diagonais restantes dos blocos.

3.2 CUDAlign 2.0

O *CUDAlign 2.0* [3] tem como objetivo obter em espaço linear o alinhamento completo de sequências longas. Para isso, são utilizados em conjunto os algoritmos de Gotoh (Seção 2.2.3) e Myers-Miller (Seção 2.2.4). A ideia principal do algoritmo é encontrar de maneira iterativa as coordenadas que fazem parte do alinhamento ótimo, até que que se obtenha alinhamento completo. Este algoritmo pode ser executado em seis etapas diferentes (Figura 3.4), sendo que algumas delas podem não ser necessárias dependendo das sequências de entrada.

1. **Obtenção do escore ótimo:** A matriz de programação dinâmica é calculada da mesma forma do CUDAlign 1.0 (Seção 3.1), com a única diferença sendo o armazenamento de algumas linhas especiais em disco. Ao final dessa etapa, obtém-se o escore ótimo, que representa o único *crosspoint* pertencente ao alinhamento ótimo conhecido até o momento.

2. ***Traceback* parcial:** O objetivo dessa etapa é encontrar mais coordenadas que pertençam ao alinhamento ótimo, utilizando uma versão modificada do Myers-Miller. Para isso, é feito um alinhamento reverso partindo do *crosspoint* obtido na etapa 1, com o proposito de buscar novos *crosspoints* do alinhamento ótimo que estão localizados em cima das linhas especiais. Visando um melhor desempenho, as *threads* executam as colunas ao invés das linhas, garantindo um menor espaço de processamento até que se ache um *crosspoint* [5]. Nessa etapa também são salvas algumas colunas especiais durante a execução e o processo é concluído quando chega ao início do alinhamento.

3. **Divisão de partições:** Nessa etapa, é realizado um procedimento similar à etapa anterior, com a diferença de que os pontos de início e fim das partições de busca já são definidos pelos *crosspoints* obtidos anteriormente. Dessa forma, é feito o alinhamento reverso dentro de cada partição de maneira independente, buscando *crosspoints* que cruzam as colunas especias.

4. **Obtenção do alinhamento completo:** Alinha as diferentes partições na CPU (*Central Processing Unit*) e concatena todos os resultados, formando o alinhamento ótimo completo.

5. **Vizualização:** Esta etapa é opcional e trata-se da visualização gráfica ou textual do alinhamento resultante.

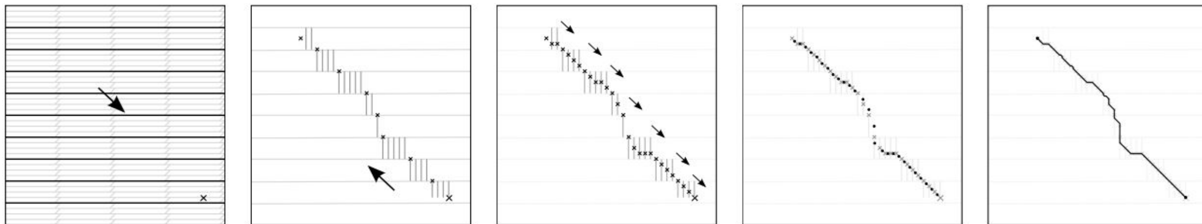

Figura 3.4: Etapas de execução do CUDAlign 2.0 [3].

3.3 CUDAlign 2.1

O primeiro estágio do CUDAlign 2.0 (Seção 3.2) é a etapa mais longa do algoritmo, uma vez que é calculado a matriz de programação dinâmica completa [5]. A fim de otimizar esse processo, foi desenvolvido o CUDAlign 2.1 [4], onde é implementada a técnica de poda de blocos (*Block Pruning*).

O *Block Pruning* tem como proposito evitar o cálculo desnecessário dos blocos de células que seguramente não fazem parte do alinhamento ótimo. Dessa forma, o processamento necessário no primeiro estágio é reduzido a medida que mais podas acontecem, sendo que a quantidade de blocos descartados depende da similaridade entre as sequências [5].

3.3.1 Processo de Poda

Considerando as sequências S_0 e S_1, de tamanho m e n respectivamente, $p(i,j)$ é a pontuação entre o alinhamento de $S_0[i]$ e $S_1[j]$, sendo $p(i,j) = ma$ em caso de *match* e $p(i,j) = mi$ em caso de *mismatch*.

A distância$-i$ (Δ_i) é definido como a distância entrea linha i e a última linha da matriz e, similarmente, a distância$-j$ (Δ_j) é a distância entre a coluna j e a última coluna da matriz. Uma célula na posição (i,j) é chamada de Δ_i-*cell* caso esteja mais próxima da última linha do que da última coluna ($\Delta_i < \Delta_j$) e Δ_j-*cell* se estiver mais próxima da ultima coluna do que da última linha ($\Delta_j < \Delta_i$).

Em um alinhamento local que passa pela célula (i,j) o escore máximo que pode ser obtido é dado pela Equação 3.5, sendo $H(i,j)$ o escore do alinhamento até essa célula. Uma vez que $H_{max}(i,j)$ é o maior escore possível, somamos um escore a $H(i,j)$ considerando um *match* perfeito entre as subsequências ainda não alinhadas $S_0[i...(i+min(\Delta i, \Delta j))]$

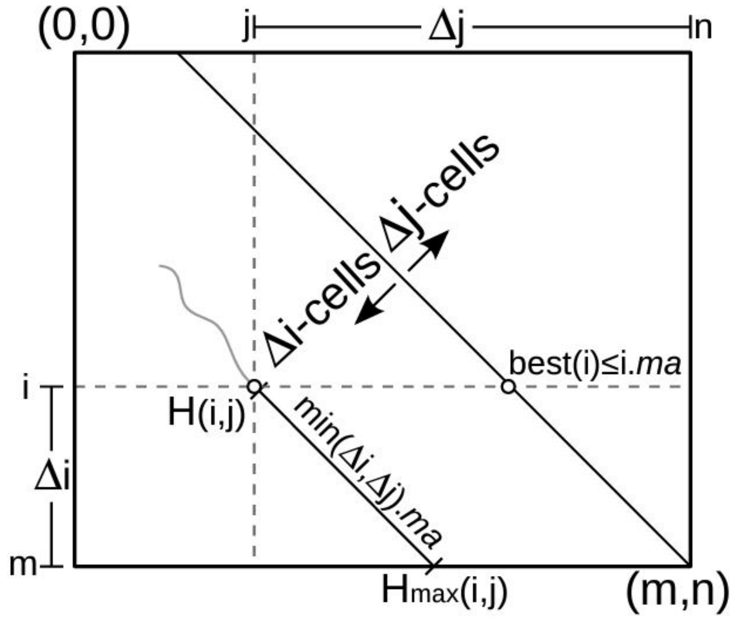

Figura 3.5: Definições geométricas do Block Pruning. [4].

e $S_1[j...(j + min(\Delta i, \Delta j))]$. Além disso, o maior escore possível de um alinhamento até uma célula da linha i é dado por $best(i) = i * ma$, uma vez que isso ocorre quando há um *match* perfeito entre as subsequências $S_0[1...i]$ e $S_1[1...i]$.

$$H_{max}(i, j) = H(i, j) + min(\Delta i, \Delta j) * ma \tag{3.5}$$

Portanto, o critério para que uma célula (i, j) seja podada, sem que prejudique a obtenção do alinhamento ótimo, é que $H_{max}(i, j) < best(i)$. É possível observar que se as células $(i, j - 1)$, $(i - 1, j)$ e $(i - 1, j - 1)$ forem podáveis, então consequentemente a célula (i,j) também será, dessa forma se torna possível descartar células sem que seja necessário calcula-las antes, reduzindo o custo computacional [5].

3.3.2 Otimização *Block Pruning*

A otimização *Block Pruning*, incluída no CUDAlign 2.1, utiliza uma versão adaptada do procedimento de poda descrito na Seção 3.3.1 aplicado a blocos de células. Nesse caso, a verificação da condição de poda é realizada apenas uma vez por bloco, considerando o pior cenário possível, onde valor de $H(i, j)$ da Equação 3.5 será o escore da célula localizada no canto superior direito do bloco [4]. No caso de H_{max} ser menor que o maior escore encontrado até o momento, o bloco será podado, dessa forma, já que não é preciso avaliar todas células de cada bloco, o *Block Pruning* é uma técnica bem rápida.

3.4 CUDAlign 3.0

Embora a otimização introduzida no CUDAlign 2.1 (Seção 3.3) tenha conseguido acelerar a obtenção do alinhamento ótimo, foi observado que o processamento de longas sequências ainda requer um tempo considerável. Sendo assim, o CUDAlign 3.0 [19] foi criado, apresentando uma nova arquitetura capaz de executar o alinhamento de sequências muito longas utilizando múltiplas GPUs, acelerando o tempo de cálculo da matriz de programação dinâmica.

Para cada uma das GPUs é atribuído um determinado número de colunas para serem processadas, sendo esse valor determinado de acordo com a capacidade de processamento de cada GPU. Dessa forma, o número de linhas processadas por segundo em cada GPU é igual, balanceando a distribuição de carga. A Figura 3.6 ilustra uma divisão uniforme de colunas, considerando 4 GPUs iguais.

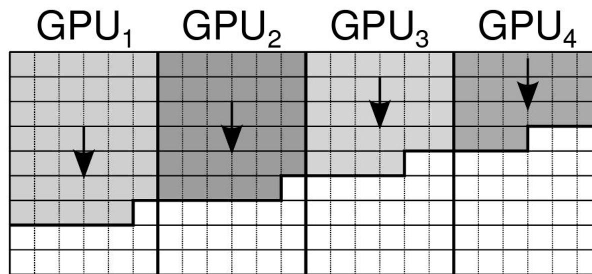

Figura 3.6: Divisão de colunas entre múltiplas GPUs. [5].

Ao executar o CUDAlign 3.0, cada uma das GPUs é atribuída a um processo do CPU. Esse processo é dividido em três *threads*: duas *threads* de comunicação, encarregadas de transferir as células entre diferentes processos, sendo que essa transferência é realizada de forma assíncrona por meio de *sockets*; e a *thread* gerente, responsável por gerenciar a GPU e transferir as células da matriz entre as *threads* de comunicação e a GPU [5].

3.5 CUDAlign 4.0

No CUDAlign 4.0 [20], foi proposta a utilização de múltiplas GPUs para a recuperação do alinhamento ótimo, que até então era feito de maneira sequencial. Buscando fazer isso de maneira eficiente, foram desenvolvidas duas estrategias para as etapas de *traceback*: *Pipelined Traceback* e *Incremental Speculative Traceback* .

3.5.1 *Pipelined Traceback*

Nessa estratégia, cada umas das GPUs executa um *pipeline* de três estágios para processar as etapas 2, 3 e 4 do CUDAlign (Seção 3.2).

Depois que a etapa 1 é finalizada, a última GPU começa a executar o primeiro estágio do seu *pipeline* (etapa 2 do CUDAlign), a partir do escore ótimo recebido da primeira etapa. Quando o *crosspoint* de uma coluna intermediaria é encontrando, ele é enviado a GPU à esquerda, que começa a executar a etapa 2 com o objetivo de encontrar o *crosspoint* da próxima partição.

Há uma relação de dependência entre as GPUs, uma vez que só podem começar a executar seu *pipeline* ao receberem o *crosspoint* calculado pela GPU vizinha na etapa 2. Dessa forma, todas as GPUs, com exceção da última, ficam ociosas entre o final da primeira etapa e incio da segunda. No momento em que essas dependências são resolvidas as GPUs podem então executar as etapas de 2 a 4 em seu *pipeline* [5]. As etapas 5,6 são executadas apenas na primeira GPU, uma vez que necessitam de muito poder computacional para executarem.

A Figura 3.7 ilustra a execução do CUDAlign com o *Pipelined Traceback*. O eixo x representa a quantidade de GPUs utilizadas e o eixo y o tempo. Os momentos de ociosidade são representados por t_a e t_b, já $t_{1...6}$ representam a execução das diferentes etapas do algoritmo.

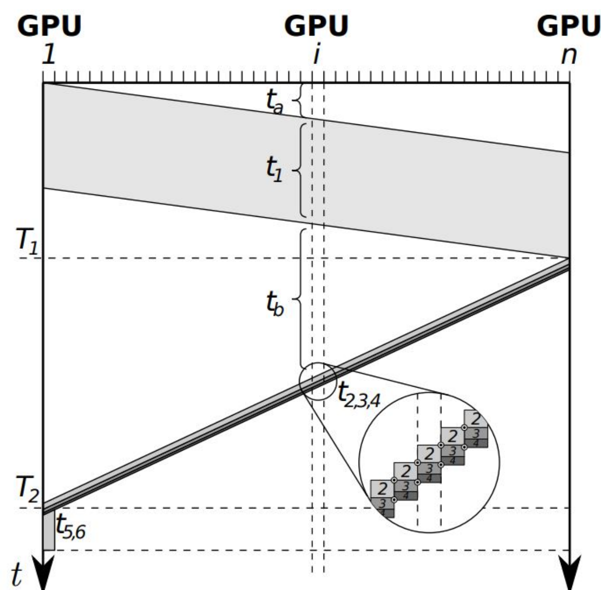

Figura 3.7: Linhas de tempo para o *Pipelined Traceback* [5].

3.5.2 *Incremental Speculative Traceback*

É possível observar na Figura 3.7 que as GPUs ficam ociosas após terminarem de processar suas colunas na primeira etapa (t_b), esperando o *crosspoint* enviado pela GPU a direita para que possam prosseguir a segunda etapa. Nesse tempo ocioso, as GPUs são utilizadas para buscar possíveis *crosspoints* em suas partições por meio de um *traceback* especulativo, considerando que o maior valor da coluna divisória é o *crosspoint* inicial da etapa 2 [21].

A execução do CUDAlign 4.0 utilizando essa estrategia é ilustrada na Figura 3.8, é possível notar que as GPUs podem gerar diversas especulações, uma vez que podem receber os *crosspoints* especulados das GPUs vizinhas. No momento em que uma GPU recebe o *crosspoint* correto, ou seja, aquele que de fato faz parte do alinhamento ótimo, ela irá verificar os resultados obtidos durante a especulação. Caso verifique que que está correto, não é necessário calcular novamente a sua partição, apenas tranferir os *crosspoints* para a esquerda [5].

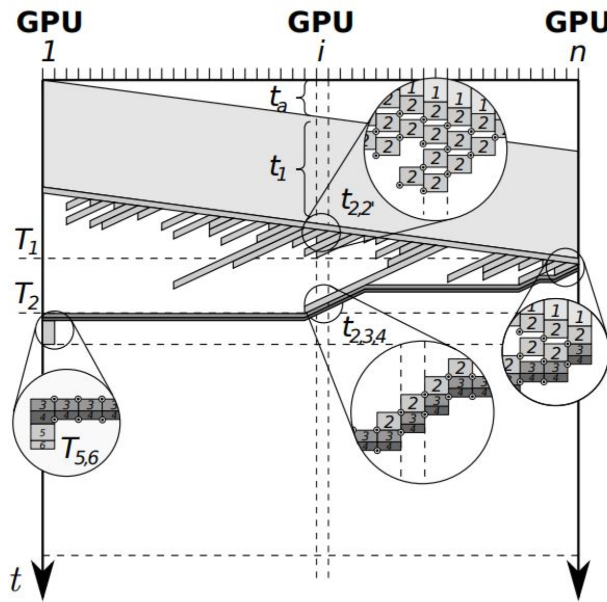

Figura 3.8: Linhas de tempo para o *Incremental Speculative Traceback* [5].

3.6 Módulo de Alinhamento com Poda Agilizada (APA)

O módulo de Alinhamento com Poda Agilizada (APA) [14] foi desenvolvido com o objetivo de otimizar o processo de poda no CUDAlign 2.1 (Seção 3.3), inicializando a execução da primeira etapa com um escore de poda próximo do ótimo. Essa abordagem permite a

antecipação de um escore que seria eventualmente alcançando durante a execução, o que viabiliza a eliminação de blocos de células no inicio da execução que não seriam podados normalmente. Tais blocos, apesar de não pertencerem ao alinhamento ótimo, acabariam sendo processados desnecessariamente por possuírem baixos escores e o escore de poda inicial ser igual a zero.

O APA propõem a utilização de um algoritmo heurístico de alinhamento para obtenção do escore de poda inicial, uma vez que são capazes de gerar alinhamentos com bons escores em um tempo relativamente curto quando comparados aos algoritmos exatos. Para isso, o APA utiliza o algoritmo BLAST (Basic Local Alignment Search Tool) [12], que executa um alinhamento local heurístico através de comparações das sequências contra bancos de dados e cálculos de significância estatística dos *matches*. Mais especificamente, foi utilizado o BLASTn, que faz buscas em uma base de nucleotídeos.

Com a adição do módulo APA, o processo de alinhamento é executado duas etapas, Preparação e Execução [14].

1. **Preparação:** Essa etapa é responsável por obter o escore inicial heurístico para o CUDAlign, dessa forma, o BLAST é invocado para que seja executado o alinhamento das sequências de entrada. Além disso, no inicio dessa etapa é possível escolher o modo de execução *Trim*, onde as sequências de entrada são aparadas para um tamanho de 2 Megabytes, acelerando o processo de obtenção do escore de poda.

2. **Execução:** Nessa etapa, CUDAlign 2.1 é invocado por linha de comando, recebendo o escore obtido na Preparação como parâmetro.

No geral, o módulo APA obteve resultados muito bons no alinhamento de sequências semelhantes, uma vez que essa similaridade possibilita a geração de escores heurísticos grandes e, portanto, uma área de poda maior [14]. Porém, em alguns casos, devido ao tempo de execução do APA em si houve pouca ou nenhuma redução no tempo de processamento. Essas situações ocorrem devido ao fato do módulo executar de maneira síncrona, ou seja, a execução do CUDAlign só se inicia quando a execução do BLAST termina, dessa forma, o tempo de execução do BLAST é adicionado ao tempo de execução do CUDAlign, gerando um *overhead* considerável.

Capítulo 4

Projeto do Módulo de Aceleração de Poda Assíncrono (MAPA)

Nesse capítulo será mostrado como se deu o processo de desenvolvimento do MAPA (Módulo de Aceleração de Poda Assíncrono), que busca solucionar o problema de *overhead* presente no módulo APA (Seção 3.6). Na Seção 4.1, é comentado a motivação para o desenvolvimento do MAPA. A Seção 4.2 mostra uma visão geral do funcionamento do MAPA, descrevendo sua arquitetura. Na Seção 4.3 é explicado de forma detalhada a etapa de preparação da arquitetura. Por fim, na Seção 4.4, é explicado o funcionamento da etapa de execução da arquitetura.

4.1 Objetivo

Conforme discutido na Seção 3.6, a execução síncrona do módulo APA antes do CUDAlign gera um *overhead*, potencialmente invalidando o ganho no tempo total de execução que o módulo poderia gerar. Além disso, nos casos em que o escore heurístico inicial é muito pequeno e, portanto, não faz nenhuma diferença no tempo de execução, o *overhead* gerado pelo BLASTn somente atrasa a execução do CUDAlign.

Logo, buscando solucionar esses problemas, surgiu a proposta de reformular o modo de execução do módulo APA, desenvolvendo uma ferramenta de aceleração de poda que funcione de forma assíncrona. A ideia base é executar o CUDAlign e o BLASTn paralelamente, transferindo o escore heurístico de forma assíncrona uma vez que ele for calculado pelo BLASTn. Dessa forma, mesmo se o escore gerado pelo BLASTn não contribuir significativamente para melhorias na poda, o tempo de execução do CUDAlign não deverá ser afetado, pois sua execução já estará em andamento.

Portanto, o objetivo desse trabalho de graduação foi projetar o MAPA (Módulo de Aceleração de Poda Assíncrono), um módulo de aceleração de poda que funcionasse de forma assíncrona.

4.2 Visão Geral

No projeto do MAPA, a implementação desenvolvida levou em consideração os problemas de concorrência, evitando soluções que pudessem afetar de alguma forma o desempenho do CUDAlign. Além disso, algumas das funcionalidades propostas no módulo APA (Seção 3.6) consideradas pertinentes foram incorporadas no módulo assíncrono, como os devidos ajustes. A linguagem escolhida para o desenvolvimento foi o C++, principalmente por uma questão de compatibilidade com o CUDAlign.

A arquitetura do MAPA (Figura 4.1) é baseada na arquitetura do APA (Seção 3.6) e possui também dois sub-módulos. Em vermelho estão as alterações do MAPA. No primeiro sub-módulo é feito a preparação dos dados de entrada, caso seja necessário, e os recursos necessários para a próxima etapa são instanciados. Já no segundo sub-módulo, o BLASTn e do CUDAlign são executados em paralelo, havendo a atualização assíncrona do escore heurístico no meio dessa etapa.

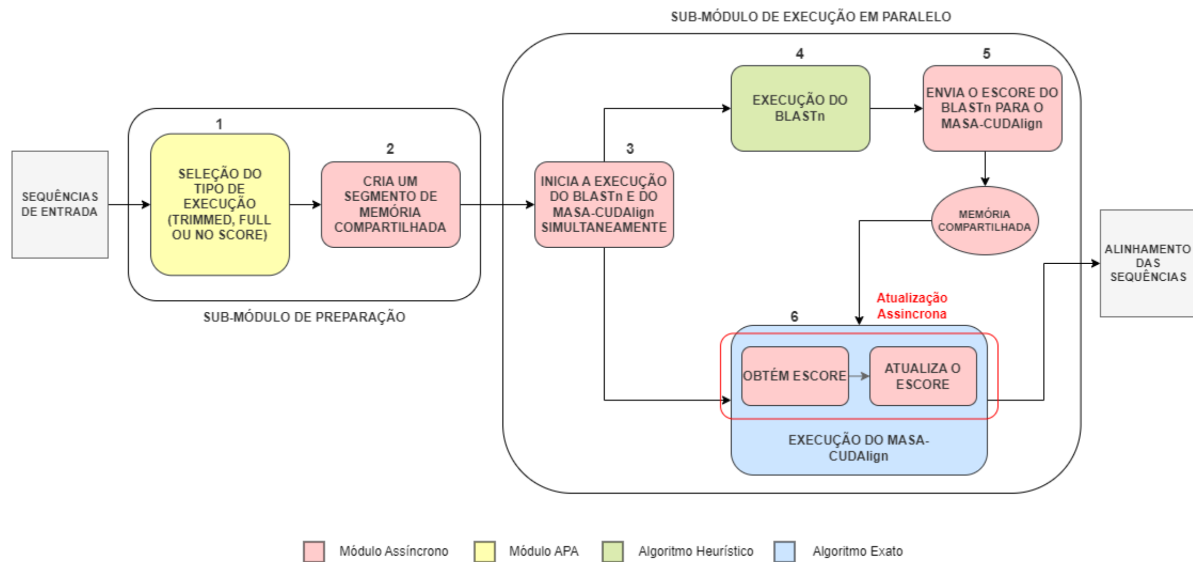

Figura 4.1: Arquitetura do MAPA

O sub-módulo de preparação possui duas tarefas principais:

1. **Seleção do tipo de execução:** assim como no módulo APA, o usuário pode escolher executar o módulo em três modos distintos: *trim*, *full* ou *no escore*.

2. **Criação do segmento de memória compartilhada:** nessa etapa, é criado o segmento de memória compartilhada para o compartilhamento do escore heurístico com o CUDALign.

O sub-módulo de execução é dividido em quatro etapas:

3. **Execução do CUDAlign e do BLASTn em paralelo:** nessa etapa, o CUDAlign e o BLASTn são colocados em execução simultaneamente.

4. **BLASTn:** essa etapa executa o BLASTn e obtém o alinhamento entre as sequências, obtendo o escore heurístico resultante e o escrevendo em um arquivo de saída.

5. **Envio do escore heurístico ao CUDAlign:** o escore heurístico gerado pelo BLASTn é lido do arquivo e escrito na memória compartilhada.

6. **CUDAlign** o CUDAlign executa normalmente, iniciando com o escore inicial padrão igual a zero. Durante a execução, verifica se um escore heurístico já foi atribuído à memória compartilhada e, se for o caso, substitui o escore calculado até o momento, caso o escore heurístico seja maior.

4.3 Sub-Módulo de Preparação

A primeira etapa desse sub-módulo é a seleção de modo de execução, uma das funcionalidades incorporada do modulo APA. Ao executar o MAPA em linha de comando, o usuário precisa indicar qual das seguintes linhas de execução deve ser seguida:

1. ***Trimmed:*** nessa linha de execução, o BLASTn gera o escore heurístico a partir de versões aparadas das sequências de entrada, que possuem um tamanho de 2 Megabytes. O objetivo dessa opção é reduzir o tempo de execução do BLASTn, levando em consideração que o aumento do escore é mais intenso na parte inicial das sequências [14].

2. ***Full:*** nesse modo, o escore heurístico é obtido a partir do processamento das sequências de entrada completas pelo BLASTn.

3. ***No Score:*** essa opção executa apenas o CUDAlign na sua forma padrão, sem a geração do escore heurístico.

Para selecionar o modo de execução, uma das seguintes *flags* deve ser passada como argumento de linha de comando: "*-t*" ou "*–trimm"*" para o modo *Trimmed*, "*-f*" ou "*–full*" para o modo *Full* e "*-ns*" ou "*–noscore*" para o modo *No Score*. Ainda nessa primeira

etapa da preparação, caso o modo *Trimmed* seja selecionado, serão geradas as versões aparadas das sequências de entrada, restringindo o tamanho das sequências ao tamanho de 2 Megabytes.

A segunda etapa do sub-módulo de preparação é a criação da área de memória compartilhada para a comunicação do escore heurístico. Optou-se pelo uso de memória compartilhada por ser um método rápido de transferência de dados entre processos. A região de memória instanciada possui dois campos: um número inteiro, onde o escore heurístico calculado é armazenado, e uma *flag*, responsável por indicar ao CUDAlign que o escore heurístico está disponível no primeiro campo.

4.4 Sub-Módulo de Execução em Paralelo

O desenvolvimento deste sub-módulo demandou uma atenção maior, principalmente na parte da atualização do escore ótimo do CUDAlign pelo heurístico. A utilização de mecanismos de sincronização no CUDAlign não foi considerada, uma vez que isso poderia impactar o desempenho da ferramenta. Logo, o maior desafio enfrentado foi pensar em uma forma de efetuar essa atualização de maneira rápida e sem causar problemas de concorrência.

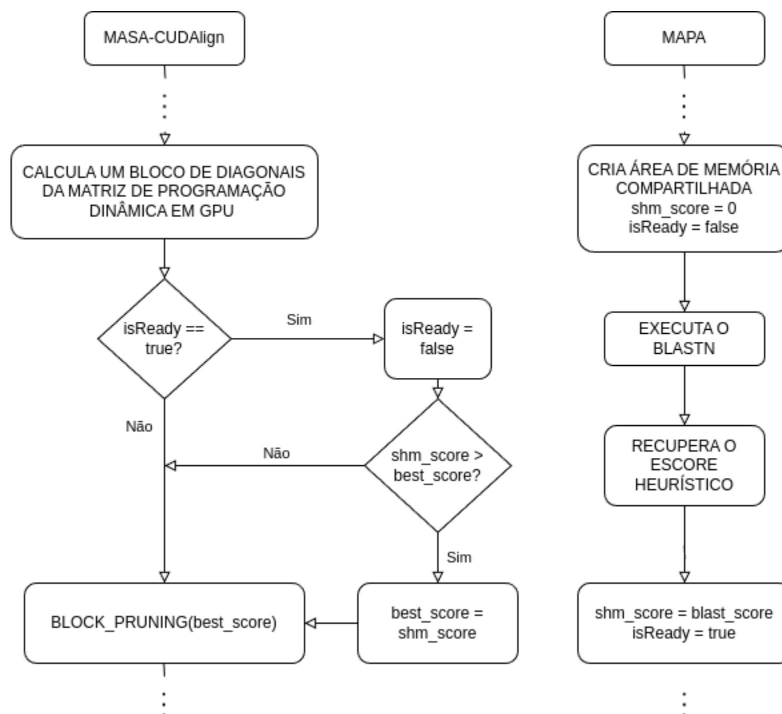

Figura 4.2: Algoritmo de funcionamento do MAPA.

Uma solução imediata seria utilizar *locks* ou semáforos para verificação do término do BLASTn e atualização da memória compartilhada e do melhor escore do CUDAlign. No entando, a obtenção e liberação do *lock* causaria um *overhead* no CUDAlign. Sendo assim, a solução implementada para resolver essa situação, mostrada no diagrama da Figura 4.2, foi fazer com que o próprio CUDAlign ficasse responsável por verificar a disponibilidade e atualizar o escore heurístico, adicionando uma pequena porção de código no CUDAlign contendo um *if* e uma atualização, com impacto desprezível no desempenho. Sendo assim, o escore heurístico, quando pronto, é apenas disponibilizado pelo MAPA na memória compartilhada e a *flag* é colocada em verdadeira, sem que nenhuma atualização seja realizada de imediato. Dentro do CUDAlign, foi incluído o código responsável pela verificação da *flag* e atribuição do escore heurístico a variável do melhor escore atual do CUDAlign no meio do fluxo de execução já existente, sem a necessidade de criar *threads* adicionais ou utilizar mecanismos de sincronização. A área de memória compartilhada é acessada pelo processo do MAPA, que a modifica para inserir o escore heurístico, e o CUDAlign, que faz apenas a leitura, dessa forma, não há problema de condição de corrida. Sendo assim, projetamos uma solução *lock-free* e assíncrona, fazendo do MAPA um módulo leve.

A lógica de atualização é responsável por atualizar o valor da variável que armazena o melhor escore calculado até o momento com o escore heurístico lido da memória compartilhada. Essa lógica foi implementada dentro do método *updateBestScore* da classe *AbstractBlockPruning* do CUDAlign. Esse método foi escolhido por ser o único local onde o atributo *bestScore*, que armazena o melhor escore encontrado até o momento dentro do CUDAlign, é alterado no fluxo original do algoritmo. Dessa forma, é garantido que o *bestScore* não será alterado em dois locais diferentes, já que o método *updateBestScore* será necessariamente chamado quando for necessário atualizar o escore ótimo.

Na arquitetura (Figura 4.1), o sub-módulo de execução inicia na etapa 3, onde o BLASTn e o CUDAlign são colocados em execução paralela. A criação desses dois novos processos é feita através da chamada de sistema *fork* e os programas são executados logo em seguida pela chamada de sistema *execl*. As sequências enviadas ao BLASTn podem ser completas ou aparadas, dependendo do modo de execução selecionado na preparação.

Na etapa 4, onde o BLASTn executa efetivamente, foram encontrados alguns problemas. Durante os testes, foi possível perceber em alguns casos um tempo estranhamente longo de execução do BLASTn, algumas vezes superando o tempo de execução do CUDAlign. Eventualmente, constatou-se que o problema estava no arquivo de saída, onde uma enorme gama de possibilidades diferentes de alinhamentos eram listadas, fazendo o processo de gravação do arquivo em disco algo extremamente demorado. Esse problema foi resolvido mudando as preferências de execução do BLASTn, indicando que somente o

melhor alinhamento deveria ser armazenado.

A etapa 5 é responsável por buscar o melhor escore heurístico gerado no arquivo de saída do BLASTn, atribuir esse valor na área de memória compartilhada e atualizar a *flag*, informando o CUDAlign que o escore está pronto. O método de busca pelo escore ótimo dentro do arquivo foi feito com base na solução implementada no módulo APA [14].

O Algoritmo 1 demonstra a execução das etapas 3, 4 e 5. Nas linhas 1 a 4 é criado um novo processo filho, onde o CUDALign é posto em execução. Já nas linhas 5 a 8, é criado o processo filho onde o BLASTn será executado. Na linha 9, o processo pai esperar a finalização do processo do BLASTn e, na linha 10, lê o escore heurístico do arquivo gerado pelo BLASTn e armazena na variável compartilhada. Na linha 11, a *flag* que indica que o escore está pronto, também em memória compartilhada, é colocada em *True* e, por fim, o processo esperar pela finalização do processo do BLASTn na linha 12.

Algorithm 1: Execução em paralelo dos algoritmos exato e heurístico

Input : Variável inteira $escoreHeuristico$ na memória compartilhada, variável booleana $escorePronto$ na memória compartilhada

Output: O escore ótimo gerado pelo CUDAlign

```
1  processoA_ID ← criarProcesso()                    // Fnção fork (C++).
2  if processoA_ID == 0 then
3  |  executaPrograma("CUDAlign")       // Troca executável do processo
   |   criado por meio do execl (C++).
4  end if
5  processoB_ID ← criarProcesso()
6  if processoB_ID == 0 then
7  |  executaPrograma("BLASTn")
8  end if
9  esperaProcessoFilho(processoB_ID)                 // Função wait (C++).
10 escoreHeuristico ← leArquivo()        // Leitura do arquivo gerado pelo
    BLASTn.
11 escorePronto ← True
12 esperaProcessoFilho(processoA_ID)
13 return
```

Por fim, a etapa 6 representa a execução do CUDAlign. Após área de memória compartilhada ser atualizada, o código de atualização assíncrona incluído dentro do CUDAlign irá detectar a mudança na próxima chamada da função *updateBestScore*. Nesse momento, o escore heurístico será lido da variável compartilhada e comparado tanto com o escore recebido como argumento nesse método quanto com o escore ótimo atual, gravado em *bestScore*. Caso seja maior que os dois, o escore heurístico será atribuído a variável *bestScore* e o CUDAlign continuará sua execução normalmente. Caso contrário, o escore cal-

culado pelo BLASTn é simplesmente descartado. Em ambos os casos a *flag* de verificação será colocada em falso, evitando repetir a mesma checagem pelo resto da execução.

Algorithm 2: Atualização assíncrona do *melhorEscore* feita dentro da função *updateBestEscore* do CUDAlign.

```
Input  : Variável inteira escoreHeuristico na memória compartilhada, variável
         booleana escorePronto na memória compartilhada
Output: melhorScore possivelmente atualizado.
1  /* ...  Função updateBestScore                                    */
2  /* Recebe o bestScore e score como argumento                      */
3  if escorePronto then
4  |  escorePronto ← False
5  |  if escoreHeuristico > bestScore & escoreHeuristico > score then
6  |  |  bestScore ← escoreHeuristico
7  |  end if
8  end if
9  /* Restante da função ...                                          */
10 return
```

A execução da etapa 6 é mostrada no Algoritmo 2, que representa um trecho da função *updateBestScore*. Na linha 3, é verificado através da *flag* em memória compartilhada se o escore heurístico já foi calculado, caso tenha sido, as linhas 4 a 7 são executadas. Na linha 4 a *flag* é colocada em *False*, para que esse trecho execute somente uma vez. Já na linha 5 é verificado se o escore heurístico gerado é maior que o *bestScore* atual e que o argumento *score*, recebido pela função *updateBestScore*. Caso seja, o a variável *bestScore* é atualizada na linha 6.

Capítulo 5

Resultados Experimentais

Com o intuito de analisar o desempenho do MAPA, foram feitos diversos testes utilizando diferentes sequências biológicas. Além disso, buscando estender mais a comparação dos módulos e entender melhor a otimização de poda, foram feitas simulações com base em um algoritmo heurístico ideal.

Na Seção 5.1 é descrito o ambiente onde o MAPA foi desenvolvido e testado, já a Seção 5.2 apresenta as sequências biológicas que foram utilizadas nos testes. Os resultados obtidos nos testes são mostrados e discutidos tanto na Seção 5.3 quanto na Seção 5.4.

5.1 Ambiente de Teste

Os testes foram realizados no Laboratório de Sistemas Integrados e Concorrentes (LAICO) do Departamento de Ciência da Computação (CIC) da Universidade de Brasília. A especificação da maquina utilizada inclui uma CPU Intel Core i7-9700 3.00GHz, 16GiB de memoria RAM e uma GPU NVidia GeForce RTX 2060. Além disso, o sistema operacional instalado nesse computador é o Linux Ubuntu 20.04.

Antes do inicio do desenvolvimento, foi feita a configuração do ambiente de desenvolvimento. A API CUDA 10.1 (Compute Unified Device Architecture), usada para a programação de GPUs NVidia, já estava instalada e configurada na máquina do laboratório, assim como o compilador de C/C++ GCC 9.4.0. A ferramenta BLASTn foi baixada do site *National Center of Biotechnology Information* (NCBI) [7] e instalada devidamente. Além disso, foi feita a clonagem dos repositórios *github* onde estão armazenados os códigos fonte do CUDALign [22] e do módulo APA [23].

O Listing 5.1 mostra o passo a passo de como executar o MAPA. Na etapa 1, o projeto do MAPA é clonado do repositório do *github* do autor desse trabalho. Na segunda etapa, é utilizado o comando *cd* para entrar no diretorio do código do CUDAlign, que está incluso no projeto. Na terceira etapa é feito a configuração do ambiente, indicando o diretório da

biblioteca CUDA e do compilador *nvcc* da NVIDIA. Na etapa 4 é executado o comando *make* para compilar o CUDAlign. Após isso, retorna-se ao diretório do anterior na quinta etapa e é feito a compilação do MAPA utilizando o comando da etapa 6. Por fim o MAPA pode ser executado utilizando o comando da etapa 8, recebendo a opção de execução e as sequências como entrada.

Listing 5.1: Comandos Bash

```
1 $ git clone https://github.com/trajano7/masa_cudalign_MAPA.git
2 $ cd masa_cudalign_MAPA/cudalign
3 $ ./configure --with-cuda=/usr/lib/cuda
4 --with-nvcc=/usr/lib/nvidia-cuda-toolkit/bin/
5 $ make
6 $ cd ..
7 $ g++ -std=c++17 mapa.cpp -o mapa
8 $ ./mapa [opcao de execucao] sequenciaA.fasta sequenciaB.fasta
```

5.2 Sequências Utilizadas

Os testes foram conduzidos com um conjunto de 14 sequências biológicas, sendo realizado um total de 8 alinhamentos entre pares distintos dessas sequências. Dentre esses alinhamentos, 6 deles foram os mesmos feitos nos testes do módulo APA [14], buscando comparar diretamente a diferença de desempenho. As sequências utilizadas são sequências de DNA de organismos reais, retirados do site do National Center for Biotechnology Information (NCBI).

A Tabela 5.1 apresenta os oito conjuntos de 2 sequências utilizados nos testes, exibindo o nome científico dos organismos, que inclui animais, plantas e bactérias, e o tamanho das sequências. Enquanto isso, na Tabela 5.2, são apresentados os identificadores das sequências de cada par, juntamente com o escore ótimo gerado pelo CUDAlign, além dos escores heurísticos gerados pelo BLASTn nos modos *Trimmed* e *Full*.

5.3 Comparação dos Resultados

Nesta seção, serão apresentados os resultados dos testes realizados no MAPA (assíncrono) e no módulo APA (síncrono), comparando o desempenho entre os dois módulos, tanto em relação ao tempo de execução quanto a porcentagem de poda. Os testes foram realizados nos três modos de execução, presentes nos dois módulos: *Trimmed*, *Full* e *No Score*, sendo que esse último foi executado em apenas um dos módulos, já que se trata apenas

Tabela 5.1: Tabela dos pares comparados e os nomes científicos dos organismos.

	Sequência A	**Sequência B**
Tamanhos	**Nome**	**Nome**
$3Mx3M$	*Corynebacterium efficiens*	*Corynebacterium glutamicum*
$5Mx5M$	*Bacillus anthracis str. Ames*	*Bacillus anthracis str. Sterne*
$5Mx7M$	*Bacillus anthracis str. Ames*	*Rhodopirellula baltica*
$10Mx10M$	*Amycolatopsis mediterranei U32*	*Amycolatopsis mediterranei S699*
$23Mx23M$	*Drosophila melanogaster (2L)*	*Drosophila melanogaster (3L)*
$40Mx40M$	*Vitis cinerea var. helleri x Vitis rupestris*	*Vitis cinerea var. helleri x Vitis riparia*
$50Mx33M$	*Homo sapiens*	*Pan troglodytes*
$50Mx50M$	*Homo sapiens*	*Gorilla gorilla gorilla*

Tabela 5.2: Tabela dos pares comparados, seus identificadores e respectivos escores.

	Sequência A	**Sequência B**	**Escores**		
Tamanhos	***Identificador***	***Identificador***	BLASTn (Trimm)	BLASTn (Full)	CUDAlign
$3Mx3M$	BA000035.2	BX927147.1	3888	3888	4226
$5Mx5M$	AE016879.1	AE017225.1	831053	849975	5220960
$5Mx7M$	NC_003997.3	NC_005027.1	NA	119	172
$10Mx10M$	NC_014318.1	NC_017186.1	2067218	6235710	10235188
$23Mx23M$	NT_033779.4	NT_037436.3	8153	9059	9063
$40Mx40M$	OX253915.1	OX254009.1	86695	119799	9247283
$50Mx33M$	NC_000021.7	BA000046.3	NA	119428	27206434
$50Mx50M$	NC_000021.7	NC_073245.1	NA	112021	26589372

da execução normal do CUDAlign, sem escore heurístico. Além disso, em relação ao tempo de execução, foi extraído dos resultados tanto o tempo total de execução, quanto o tempo de execução apenas do BLASTn, visando verificar melhor o tempo de *overhead*.

5.3.1 Resultados Modo *Full*

A Tabela 5.3 apresenta os tempos de execução do BLASTn e o tempo total de execução (BLASTn e Cudalign) para ambos os módulos no modo Full, síncrono e assíncrono. Na última coluna, é fornecido o tempo total de execução no modo *No Score*, no qual apenas o Cudalign é executado. Na Tabela 5.3, os melhores tempos estão assinalados destacados. Adicionalmente, a Figura 5.1 apresenta o tempo total de execução. Já a Tabela 5.4 contém a porcentagem total de blocos podados no modo Full, tanto para o módulo síncrono quanto para o assíncrono, além do modo *No Score*. A porcentagem de poda também é representada no gráfico da Figura 5.2.

Ao examinar os gráficos referentes ao tempo total de execução na Figura 5.1, observou-se que, nos alinhamentos *3Mx3M*, *5Mx7M* e *23Mx23M*, praticamente não houve diferença significativa no tempo total de execução entre os dois módulos e o modo *No Score*. Notou-se na Tabela 5.3 que o tempo do BLASTn nesses casos é muito baixo, resultando em um *overhead* insignificante em relação ao tempo total. A execução rápida do BLASTn pode

ser explicada pelo elevado nível de disparidade entre as sequências alinhadas, evidenciado pela baixa porcentagem de poda, conforme ilustrado no gráfico da Figura 5.2.

Os alinhamentos *5Mx5M* e *40Mx40M*, mostram resultados bastante favoráveis para o módulo assíncrono. Ao analisar o gráfico na Figura 5.1, é possível que, nesses dois casos, o tempo de execução total do módulo síncrono sofreu um *overhead* considerável em relação ao tempo do *No Score*. Já no módulo assíncrono, o tempo total de execução ficou abaixo do *no score*, porém ainda muito próximo, mostrando que ele foi capaz de eliminar o *overhead*. Na comparação de *40Mx40M* não houve nenhum aumento de poda e no alinhamento *5Mx5M* houve um leve aumento nos dois módulos 5.2). Mas, mesmo que os módulos tenham feito pouca ou nenhuma diferença nesses casos, o módulo assíncrono conseguiu manter o tempo próximo ao do *No Score*.

No alinhamento *10Mx10M* das duas montagens dos genomas da bactéria *Amycolatopsis mediterranei* (U32 e S699), é possível observar no gráfico do tempo (Figura 5.1) que, mesmo com uma área de poda ligeiramente inferior (Figura 5.2), o MAPA conseguiu obter um tempo melhor que o APA, por conseguir eliminar o *overhead*. Esse foi o melhor resultado do MAPA no modo *Full*, conseguindo ficar com um tempo de execução abaixo do APA e do CUDAlign.

Analisando o gráfico na Figura 5.1, notou-se que nos últimos dois alinhamentos, *50Mx33M* e *50Mx50M*, os escores heurísticos gerados não são tão altos, quando comparados ao escore ótimo (Tabela 5.2), mesmos essas sequências possuindo um notável grau de similaridade. Dessa forma, a execução dos módulos não contribuiu muito para o aumento da área de poda, como pode ser visto no gráfico da Figura 5.2. Ainda assim, o tempo total de execução do BLASTn foi relativamente alto (Tabela 5.3), gerando um *overhead* grande no módulo APA, enquanto que o MAPA conseguiu eliminar totalmente esse *overhead*.

De forma geral, o tempo de execução do MAPA em todos os casos foi inferior, ou no máximo ligeiramente superior, ao tempo de execução do CUDAlign em todos os casos (Tabela 5.3 e Figura 5.1). Isso mostra que o módulo assíncrono conseguiu solucionar o problema de *overhead* que existe no módulo síncrono.

5.3.2 Resultados Modo *Trimmed*

Os resultados dos testes no modo *Trimmed* são mostrados na Tabela 5.5, que contém o tempo de execução do BLASTn e o tempo total de execução para os dois módulos, e na Tabela 5.6, onde é mostrado a porcentagem total de poda para o APA e o MAPA.

Analisando o gráfico da Figura 5.3, que apresenta o tempo total de execução dos testes no modo *Trimmed*, é possível notar que o módulo assíncrono ainda conseguiu obter um tempo levemente melhor que o síncrono nos alinhamentos *5Mx5M* e *10Mx10M*. Porém, de

Tabela 5.3: Tempo de execução total e do BLASTn com o APA (síncrono) e o MAPA (assíncrono) no modo *Full*.

	Assíncrono		**Síncrono**		**No Score**
Sequências Comparadas	**Tempo BLASTn (s)**	**Tempo Total (s)**	**Tempo BLASTn (s)**	**Tempo Total (s)**	**Tempo Total (s)**
3M x 3M	1,0	61,0	2,0	62,0	**60,0**
5M x 5M	6,0	**122,0**	5,0	131,0	123,0
5M x 7M	2,0	221,0	1,0	220,0	**219,0**
10M x 10M	13,0	**343,0**	14,0	364,0	428,0
23M X 23M	11,0	3291,0	11,0	3304,0	**3287,0**
40M X 40M	394,0	**7148,0**	395,0	7615,0	7226,0
50M X 33M	4810,0	7633,0	4670,0	12271,0	**7581,0**
50M X 50M	5623,0	**11482,0**	5625,0	17385,0	11552,0

Tabela 5.4: Quantidade de blocos podados (%) com o APA (síncrono) e o MAPA (assíncrono) no modo *Full*.

Sequências	**Blocos Podados (%)**		
Comparadas	**Assíncrono**	**Síncrono**	**No Score**
3M x 3M	0,12	0,12	0,12
5M x 5M	54,31	54,37	53,69
5M x 7M	0,0	0,0	0,0
10M x 10M	67,88	68,85	53,33
23M X 23M	0,04	0,04	0,04
40M X 40M	35,32	35,32	35,32
50M X 33M	34,78	34,78	34,78
50M X 50M	32,51	32,51	32,51

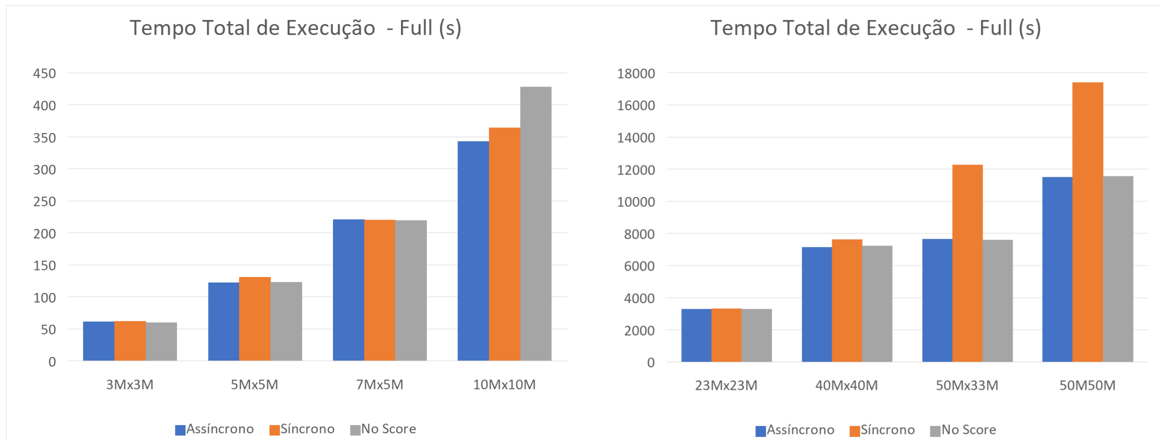

Figura 5.1: Gráficos do tempo total de execução no modo *Full*.

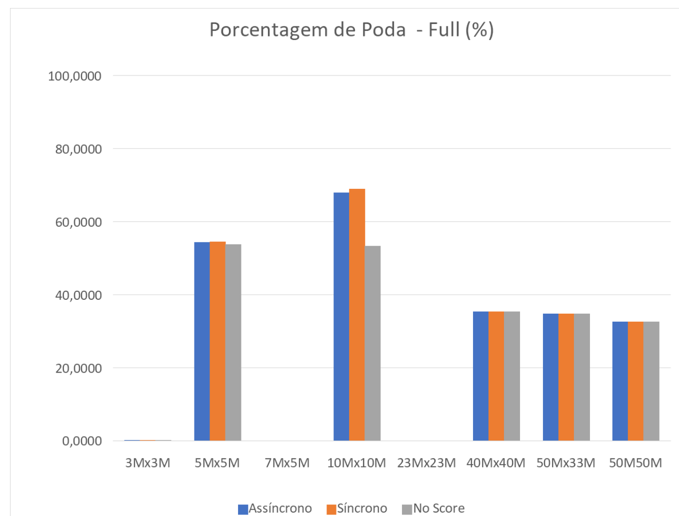

Figura 5.2: Gráficos da porcentagem total de poda de blocos no modo *Full*.

forma geral, notou-se o tempo de ambos os módulos e do modo *No Score* foi praticamente o mesmo em todos os outros pares de sequências alinhados. Isso ocorreu devido ao fato de, nesse modo de execução, o BLASTn estar limitado a executar com versões reduzidas das sequências de entrada, com apenas 2M de tamanho. Dessa forma, o tempo de execução do BLASTn é bem baixo, como pode ser visto na Tabela 5.5, o que gera um *overhead* insignificante.

O gráfico da Figura 5.4 mostra que a porcentagem total de poda de blocos deu uma reduzida significativa no alinhamento *10Mx10M*, quando comparado ao modo Full (Figura 5.2). Isso mostra que o modo *Trimmed* talvez não seja tão efetivo, uma vez que as sequências *trimed* utilizadas no algoritmo heurístico são muito curtas. Como o MAPA executa o BLASTn de forma assíncrona, com baixo *overhead*, concluímos que não há

ganho em se executar o MAPA no modo *trimmed*.

Tabela 5.5: Tempo de execução total e do BLASTn com o APA (síncrono) e o MAPA (assíncrono) no modo *Trimmed*.

	Assíncrono		**Síncrono**		**No Score**
Sequências Comparadas	**Tempo BLASTn (s)**	**Tempo Total (s)**	**Tempo BLASTn (s)**	**Tempo Total (s)**	**Tempo Total (s)**
3M x 3M	1,0	**60,0**	1,0	63,0	**60,0**
5M x 5M	2,0	**121,0**	2,0	128,0	123,0
5M x 7M	1,0	219,0	1,0	**217,0**	219,0
10M x 10M	2,0	413,0	3,0	427,0	428,0
23M X 23M	2,0	**3280,0**	2,0	3295,0	3287,0
40M X 40M	3,0	7157,0	2,0	7181,0	7226,0
50M X 33M	4,0	7628,0	2,0	**7573,0**	7581,0
50M X 50M	2,0	11540,0	2,0	**11530,0**	11552,0

Tabela 5.6: Quantidade de blocos podados (%) com o APA (síncrono) e o MAPA (assíncrono) no modo *Trimmed*.

Sequências	**Blocos Podados (%)**		
Comparadas	**Assíncrono**	**Síncrono**	**No Score**
3M x 3M	0,12	0,12	0,12
5M x 5M	54,35	54,35	53,69
5M x 7M	0,0	0,0	0,0
10M x 10M	54,91	54,91	53,33
23M X 23M	0,04	0,04	0,04
40M X 40M	35,32	35,32	35,32
50M X 33M	34,78	34,78	34,78
50M X 50M	32,51	32,51	32,51

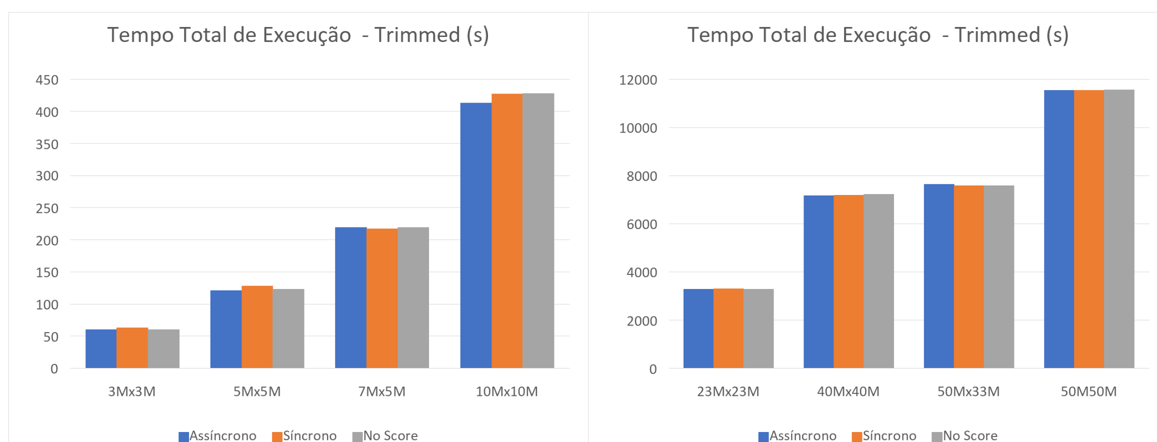

Figura 5.3: Gráficos do tempo total de execução no modo *Trimmed*.

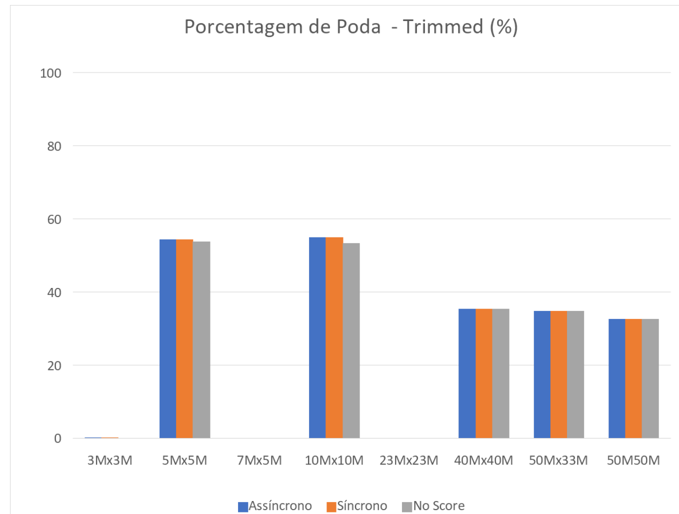

Figura 5.4: Gráficos da porcentagem total de poda de blocos no modo *Trimmed*.

5.4 Resultados com Simulação de 90% do escore ótimo

Além dos testes de comparação de desempenho da Seção 5.3, realizamos simulações de execução dos módulos, considerando a existência de um algoritmo heurístico ideal. Esse algoritmo teria a capacidade de gerar um escore equivalente a 90% do escore do CUDAlign (escore ótimo) no mesmo tempo de execução do BLASTn. A ideia dessa simulação é estudar o limite da otimização de poda utilizando um algoritmo heurístico e realizar mais comparações de desempenho entre o módulo APA e o MAPA. Nessas simulações, foi considerado o tempo de execução do BLASTn apenas no modo *Full*.

Analisando o gráfico na Figura 5.6, foi possível verificar que a porcentagem de blocos podados nos alinhamentos *3Mx3M*, *5Mx7M* e *23Mx23M* foi insignificante nas simulações, assim como nos testes do modo *Full* (Figura 5.2). Isso se manteve devido ao fato do escore ótimo desses alinhamentos ser muito baixo e o escore obtido pelo BLASTn para essas sequências já ser bastante próximo do ótimo (Tabela 5.2).

Nos alinhamentos *3Mx3M*, *5Mx7M*, *50Mx33M* e *50Mx50M* obtivemos um aumento significativo de poda (Figura 5.6), quando comparado a porcentagem de poda dos testes do modo *Full* (Figura 5.2). Em três desses alinhamentos, a poda chegou a ultrapassar a marca de 80%, mostrando que esse método de otimização de poda com escore inicial assíncrono possui bastante potencial. Devido a esse aumento na área de poda, o tempo de execução nos quatro casos (Figura 5.5) diminuiu significativamente em relação ao tempo de execução dos testes no modo *Full* (Figura 5.1).

Ao analisar apenas os alinhamentos *50Mx33M* e *50Mx50M*, foi possível notar que a

porcentagem de poda do APA foi consideravelmente maior do que a do MAPA. Essa poda adicional possivelmente ocorreu no inicio do alinhamento, uma vez que o módulo APA já inicia o CUDAlign com o escore heurístico pronto, enquanto do módulo assíncrono apenas disponibiliza no meio da execução. Porém, ao analisar o tempo de execução do APA (síncrono) e do MAPA (assíncrono) na Figura 5.5, percebe-se que, mesmo com uma porcentagem de poda bem maior, o módulo síncrono teve um tempo de execução significativamente maior do que o módulo assíncrono, devido ao problema de *overhead*.

Tabela 5.7: Tempo total de execução e porcentagem de blocos podados da simulação.

Sequências Comparadas	**Tempo de Execução (s)**		**Blocos Podados (%)**	
	Assíncrono	***Síncrono***	***Assíncrono***	***Síncrono***
3M x 3M	**62,0**	**62,0**	0,12	0,12
5M x 5M	**84,0**	93,0	80,53	80,53
5M x 7M	**215,0**	216,0	0,0	0,0
10M x 10M	**272,0**	296,0	80,59	81,31
23M X 23M	**3268,0**	3392,0	0,04	0,04
40M X 40M	**6777,0**	7118,0	37,79	38,51
50M X 33M	**6646,0**	8572,0	43,56	81,40
50M X 50M	**9217,0**	12680,0	50,56	68,85

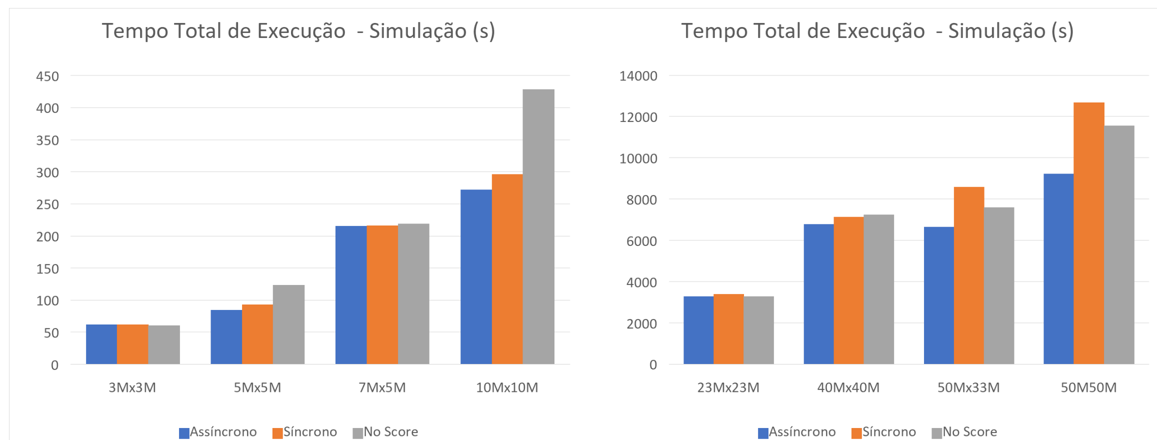

Figura 5.5: Gráficos do tempo total de execução da simulação.

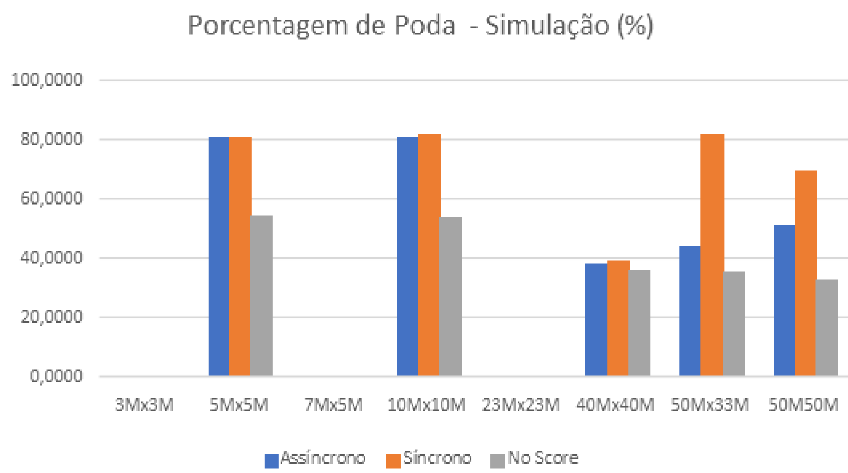

Figura 5.6: Gráficos da porcentagem total de poda de blocos da simulação.

Capítulo 6

Conclusão

O presente trabalho de graduação propôs, implementou e avaliou a ferramenta MAPA, com o objetivo de solucionar o problema de *overhead* do módulo APA. Para isso, o MAPA executa o CUDAlign e BLASTn de maneira simultânea e, ao final da execução do BLASTn, disponibiliza o escore heurístico gerado de forma assíncrona para o CUDAlign.

Os resultados dos testes no modo *Full* mostraram que, para os alinhamentos *3Mx3M*, *7Mx7M* e *23Mx23M*, a baixa semelhança entre as sequências resultou em uma variação muito baixa entre o tempo de execução dos módulos e o modo *No Score*. Já nos alinhamentos *5Mx5M* e *40Mx40M*, os resultados mostraram que o módulo síncrono sofreu um *ovehead*, mesmo com baixas porcentagens de poda, enquanto o MAPA conseguiu manter o tempo de execução próximo ao do CUDAlign puro (*No Score*). As maiores diferenças entre os módulos foram observadas nos resultados dos alinhamentos *50Mx33M* e *50Mx50M*, onde os tempos de execução do BLASTn foram elevados, apesar de haver aumento na área de poda, gerando um *overhead* alto no módulo APA. O MAPA, por sua vez, conseguiu reduzir o efeito do *overhead*, mantendo seu tempo de execução próximo ao do modo *No Score*. Já os testes do modo *Trimmed* mostraram que, no geral, o tempo de execução foi praticamente o mesmo nos dois módulos e no modo *No Score*, devido ao fato das sequências aparadas serem muito pequenas e não gerarem bons escores. O modo *Trimmed* se mostrou dispensável para o MAPA, uma vez que o ele possui um baixo *overhead*.

Os resultados das simulações mostram um aumento significativo de poda em alguns casos, chegando a ultrapassar 80%. Isso mostrou que esse método de otimização de poda com escore inicial possui bastante potencial. Além disso, foi observado nos testes dos alinhamentos *50Mx50M* e *50Mx33M* que, mesmo com uma porcentagem de poda bem menor, o MAPA conseguiu obter um tempo de execução melhor que o APA, por conseguir diminuir o *overhead*. De maneira geral, os resultados dos testes mostraram que o MAPA conseguiu reduzir bastante o problema de *overhead*, cumprindo seu objetivo.

Como trabalhos futuros, sugerimos sugerimos a expansão do MAPA para sua utilização na versão do CUDAlign destinada a execução em múltiplas GPUs [24]. Além disso, seria interessante explorar alternativas ao BLASTn, procurando por algoritmos que possam gerar melhores escores, uma vez que observamos escores heurísticos baixos nos alinhamentos de *40Mx40M*, *50Mx33M* e *50Mx50M*, apesar do alto grau de semelhança entre as sequências.

Referências

[1] Myers, Eugene Wimberly e Webb Miller: *Optimal alignments in linear space*. Computer applications in the biosciences : CABIOS, 4 1:11–7, 1988. viii, 1, 11, 12

[2] Sandes, Edans e Alba Melo: *Cudalign: Using gpu to accelerate the comparison of megabase genomic sequences*. Sigplan Notices - SIGPLAN, 45:137–146, maio 2010. viii, 13, 16

[3] O. Sandes, Edans Flavius de e Alba Cristina M.A. de Melo: *Smith-waterman alignment of huge sequences with gpu in linear space*. Em *2011 IEEE International Parallel & Distributed Processing Symposium*, páginas 1199–1211, 2011. viii, 17, 18

[4] O. Sandes, Edans Flavius de e Alba Cristina M.A. de Melo: *Retrieving smith-waterman alignments with optimizations for megabase biological sequences using gpu*. IEEE Transactions on Parallel and Distributed Systems, 24(5):1009–1021, 2013. viii, 2, 18, 19

[5] Oliveira Sandes, Edans Flávius de: *Algoritmos paralelos exatos e otimizações para alinhamento de sequências biológicas longas em plataformas de alto desempenho*. Tese de Doutoramento, Universidade de Brasília, Brasília, Brasil, 2015. `https://repositorio.unb.br/handle/10482/20248`. viii, 2, 17, 18, 19, 20, 21, 22

[6] Luscombe, Nicholas, Dov Greenbaum e M Gerstein: *What is bioinformatics? a proposed definition and overview of the field*. Methods of information in medicine, 40:346–58, fevereiro 2001. 1

[7] *Biological Sequences*, 2007. `https://www.ncbi.nlm.nih.gov/IEB/ToolBox/SDKDOCS/BIOSEQ.HTML`, [Online; accessed 20. Jul. 2023]. 1, 4, 31

[8] Mount, David W.: *Bioinformatics: Sequence and Genome Analysis*. Cold Spring Harbor Laboratory Series. Cold Spring Harbor Laboratory Press, ISBN 9780879697129. 1, 4, 5

[9] Needleman, Saul B. e Christian D. Wunsch: *A general method applicable to the search for similarities in the amino acid sequence of two proteins*. Journal of Molecular Biology, 48(3):443–453, 1970, ISSN 0022-2836. `https://www.sciencedirect.com/science/article/pii/0022283670900574`. 1, 6

[10] 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, 8

[11] Gotoh, Osamu: *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`. 1, 9

[12] 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`. 2, 23

[13] Oh, Fred: *What is cuda: Nvidia official blog*, Jan 2022. `https://blogs.nvidia.com/blog/2012/09/10/what-is-cuda-2/`, [Online; accessed 20. Jul. 2023]. 2, 13

[14] Pinho, Matheus Augusto S.: *Alinhamento paralelo de sequências biológicas com poda agilizada em gpu.* Monografia, Universidade de Brasília, 2023. 2, 22, 23, 26, 29, 32

[15] Durbin, Richard, Sean R. Eddy, Anders Krogh e Graeme Mitchison: *Biological Sequence Analysis: Probabilistic Models of Proteins and Nucleic Acids.* Cambridge University Press, 1998. 4, 5, 9

[16] Hirschberg, D. S.: *A linear space algorithm for computing maximal common subsequences.* Commun. ACM, 18(6):341–343, jun 1975, ISSN 0001-0782. `https://doi.org/10.1145/360825.360861`. 11

[17] Sandes, Edans Flavius De Oliveira, Azzedine Boukerche e Alba Cristina Magalhaes Alves De Melo: *Parallel optimal pairwise biological sequence comparison: Algorithms, platforms, and classification.* ACM Comput. Surv., 48(4), mar 2016, ISSN 0360-0300. `https://doi.org/10.1145/2893488`. 13

[18] Oliveira Sandes, Edans Flávius de: *Comparação paralela de sequências biológicas longas utilizando unidades de processamento gráfico (gpus).* Tese de Mestrado, Universidade de Brasília, Brasília, Brasil, 2011. `https://repositorio.unb.br/handle/10482/10022`. 14, 16

[19] 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.* Em *2014 14th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, páginas 160–169, 2014. 20

[20] Sandes, Edans Flavius de Oliveira, Guillermo Miranda, Xavier Martorell, Eduard Ayguade, George Teodoro e Alba Cristina Magalhaes Melo: *Cudalign 4.0: Incremental speculative traceback for exact chromosome-wide alignment in gpu clusters.* IEEE Transactions on Parallel and Distributed Systems, 27(10):2838–2850, 2016. 20

[21] Figueirêdo Júnior, Marco Antônio Caldas de: *Comparação paralela de sequências biológicas em múltiplas GPUs com descarte de blocos e estratégias de distribuição de carga.* Tese de Doutoramento, Universidade de Brasília, Brasília, Brasil, 2021. viii, 2021. `https://repositorio.unb.br/handle/10482/41495`. 22

[22] Sandes, Edans Flavius de Oliveira e Miranda: *Masa-CUDAlign.* `https://github.com/edanssandes/MASA-CUDAlign`, [Online; accessed 10. Aug. 2023]. 31

[23] Pinho, Matheus Augusto S.: *Masa-Cudalign-APA*. `https://github.com/mataugustosp/masa-cudalign-apa`, [Online; accessed 10. Aug. 2023]. 31

[24] Figueiredo, Marco, Joao Paulo Navarro, Edans F. O. Sandes, George Teodoro e Alba C. M. A. Melo: *Parallel fine-grained comparison of long dna sequences in homogeneous and heterogeneous gpu platforms with pruning*. IEEE Transactions on Parallel and Distributed Systems, 32(12):3053–3065, 2021. 42

Anexo I

Artigo apresentado no WPOS/WCOMP 2023

MAPA: Módulo Assíncrono para Aceleração de Poda da Ferramenta CUDAlign

Matheus Trajano do Nascimento
Departamento de Ciência da Computação - CIC
Universidade de Brasília - UnB
Brasília, Brasil
matheus.trajano48@hotmail.com

Alba Cristina Magalhaes Alves de Melo
Departamento de Ciência da Computação - CIC
Universidade de Brasília - UnB
Brasília, Brasil
alves@unb.br

***Abstract*—A análise de sequências biológicas desempenha um papel essencial na biologia molecular e bioinformática, pois produz o alinhamento de sequências que é uma técnica fundamental no entendimento de DNA, RNA e proteínas. O módulo APA (Alinhamento com Poda Agilizada) foi concebido para aprimorar o processo de poda da ferramenta de alinhamento CUDAlign, utilizando um escore heurístico como ponto de partida no CUDAlign. No entanto, a execução síncrona do APA introduz um *overhead* no tempo total de execução. Este trabalho apresenta o projeto do MAPA (Módulo de Aceleração de Poda Assíncrono), com o objetivo de eliminar esse *overhead*. Os resultados obtidos mostram um ganho de desempenho considerável em relação ao módulo APA.**

***Index Terms*—comparação de sequências biológicas, poda, GPU**

I. Introdução

A análise de sequências biológicas é crucial em biologia molecular e bioinformática, com o alinhamento de sequências sendo uma técnica central no entendimento de DNA, RNA e proteínas. O CUDAlign se destaca ao usar GPUs para obter alinhamentos ótimo de maneira eficiente. No CUDAlign 2.1 [3], foi proposto o processo de poda, que envolve a remoção de partes não essenciais das matrizes de programação dinâmica para acelerar o algoritmo de alinhamento. O módulo APA [6] buscou otimizar a poda, gerando um escore heurístico que será utilizado como escore inicial do CUDAlign. No entanto, a execução síncrona do algoritmo heurístico faz com que o APA cause um *overhead*. O objetivo deste trabalho é desenvolver um módulo de aceleração de poda assíncrono para eliminar esse *overhead*.

II. Módulo APA

A. CUDAlign

O CUDAlign é uma ferramenta desenvolvida com o propósito de obter o alinhamento ótimo entre sequências biológicas por meio da programação dinâmica. Para otimizar esse processo, o CUDAlign utiliza GPUs, que são capazes de executar operações com alto paralelismo, no cálculo das matrizes de programação dinâmica.

- **CUDALign 1.0:** Calcula o escore ótimo do alinhamento de sequências por meio do algoritmo de Gotoh, porém, não fornece o alinhamento em si. Utiliza a técnica de *wavefront*, calculando os elementos das anti-diagonais da matriz em paralelo [1].
- **CUDALign 2.0:** Estende o CUDAlign 1.0, permitindo a obtenção do alinhamento completo de sequências. Utiliza o algoritmo de Gotoh e Myers-Miller em conjunto, com uma abordagem iterativa para encontrar as coordenadas do alinhamento ótimo [2].
- **CUDAlign 2.1:** Introduz a técnica de "Block Pruning" para otimizar o processo de poda de blocos de células que não contribuem para o alinhamento ótimo, economizando cálculos desnecessários de blocos de baixo escore [3].
- **CUDAlign 3.0:** Introduz a capacidade de usar várias GPUs para acelerar o cálculo da matriz de programação dinâmica. Divide as colunas da matriz entre várias GPUs para balancear a carga de processamento [4].
- **CUDAlign 4.0:** Introduz o "Incremental Speculative Traceback" para otimizar as etapas de "traceback" do algoritmo. Isso permite que várias GPUs trabalhem juntas na recuperação do alinhamento ótimo, que até então era feito de forma sequencial [5].

B. Módulo de Alinhamento com Poda Agilizada - APA

O módulo APA [6] foi desenvolvido com o objetivo de otimizar o processo de poda do CUDALign 2.1 (Seção II-A). A proposta é iniciar a execução do CUDAlign com um escore inicial diferente de zero, permitindo eliminar blocos de células no início da execução, que não seriam podados normalmente. O escore inicial, por sua vez, é obtido por meio de um algoritmo de alinhamento heurístico, mais especificamente a ferramenta BLAST (Basic Local Alignment Search Tool) [7], que é capaz de gerar bons alinhamentos em tempo razoável.

O módulo APA demonstrou bom desempenho no alinhamento de sequências semelhantes, gerando escores heurísticos altos e permitindo uma poda mais ampla. Porém, devido ao fato do módulo executar de maneira síncrona, ou seja, a execução do CUDAlign só se inicia quando a execução do BLASTn termina, o tempo de execução do BLAST é adicionado ao tempo de execução do CUDAlign, gerando um *overhead* considerável.

III. Projeto do Módulo Assíncrono

A. *Motivação*

A execução síncrona do módulo APA (Seção II-B) antes do MASA-CUDAlign pode invalidar o ganho de desempenho que o módulo proporcionaria, devido ao *overhead* gerado. Buscando solucionar esse problema, surgiu a proposta de reformular modo de execução do APA, desenvolvendo o MAPA (Módulo de Aceleração de Poda Assíncrono). A ideia é executar o CUDAlign e o BLASTn em paralelo, transferindo o escore heurístico somente quando for calculado pelo BLASTn, de forma assíncrona, eliminado o *overhead* do APA.

B. *Projeto do MAPA*

No projeto do MAPA, a implementação desenvolvida levou em consideração os problemas de concorrência, evitando soluções que pudessem afetar de alguma forma o desempenho do MASA-CUDAlign. Além disso, algumas das funcionalidades propostas no módulo APA consideradas pertinentes foram incorporadas no módulo assíncrono. Por uma questão de compatibilidade com o MASA-CUDAlign, a linguagem escolhida para o desenvolvimento foi o C++.

A arquitetura do módulo assíncrono (Fig. 1) é baseada na arquitetura do APA (Seção II-B) e possui também dois sub-módulos. As funcionalidades do módulo APA que foram mantidas estão em amarelo, já as novas funcionalidades implementadas para o módulo assíncrono estão representadas em vermelho.

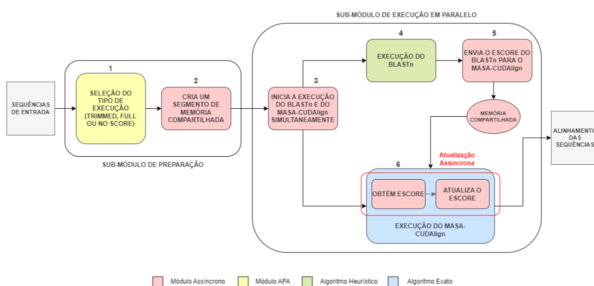

Fig. 1. Arquitetura do módulo assíncrono

1) Sub-Módulo de Preparação: O sub-módulo de preparação envolve as etapas 1 e 2 da arquitetura. Na primeira etapa é feita seleção do modo de execução, uma funcionalidade incorporada do módulo APA. Ao usar o módulo assíncrono na linha de comando, o usuário pode escolher entre três modos de execução:

- ***-trimm* ou *-t*:** Execução do BLASTn com sequências aparadas (tamanho de 2 Megabytes).
- ***-full* ou *-f*:** Obtenção do escore heurístico a partir do processamento das sequências de entrada completas.
- ***-noscore* ou *-ns*:** Execução padrão do MASA-CUDAlign sem geração do escore heurístico.

Na segunda etapa, é criada uma área de memória compartilhada para a comunicação do escore heurístico. A região de memória contém um número inteiro para o escore e uma *flag* para notificar o MASA-CUDAlign sobre a disponibilidade do escore.

2) Sub-Módulo de Execução: O sub-módulo de execução se inicia na etapa 3, onde o BLASTn e o CUDAlign são colocados em execução paralela. A criação desses dois processos e a execução dos programas é feito através das chamadas de sistema *fork* e *execl*, respectivamente. A etapa 4 trata-se da execução do BLASTn, que gera um arquivo de saída contendo o escore heurístico calculado. A etapa 5 é responsável por buscar o melhor escore heurístico gerado no arquivo de saída do BLASTn, atribuir esse valor na área de memória compartilhada e atualizar a *flag*, informando o MASA-CUDAlign que o escore está pronto.

Por fim, a etapa 6 representa a execução do MASA-CUDAlign. O código de verificação da *flag* e a atualização do escore foram incorporados ao fluxo de execução do CUDAlign, sem a necessidade de criar threads adicionais. Isso evitou a necessidade de utilizar mecanismos de sincronização, mantendo o MAPA leve. A lógica de atualização do escore ótimo foi implementada no método ”updateBestScore” da classe ”AbstractBlockPruning” no MASA-CUDAlign. Esse método foi escolhido pois é o único que pode modificar o atributo ”bestScore”, variável que armazena o melhor escore encontrado até o momento diretamente, dessa forma, é assegurado a consistência na atualização do escore ótimo.

Após área de memória compartilhada ser atualizada, o código de atualização assíncrona incluído dentro do MASA-CUDAlign irá detectar a mudança na próxima chamada da função *updateBestScore*. Nesse momento, o escore heurístico será lido da variável compartilhada e comparado tanto com o escore recebido como argumento nesse método quanto com o escore ótimo atual, gravado em *bestScore*. Caso seja maior que os dois, o escore heurístico será atribuído a variável *bestScore* e o CUDAling continuará sua execução normalmente.

IV. Resultados

Nessa seção, serão apresentados os resultados dos testes feitos, buscando principalmente comparar o desempenho entre o módulo APA (síncrono) e o MAPA (assíncrono). Os testes foram realizados no Laboratório de Sistemas Integrados e Concorrentes (LAICO) do Departamento de Ciência da Computação (CIC) da Universidade de Brasília. As especificações da máquina utilizada incluem uma CPU Intel Core i7-9700 3.00GHz, 16GiB de memória RAM e uma GPU NVidia GeForce RTX 2060.

Além dos testes padrões feitos com o objetivo de comparar o desempenho do módulo assíncrono e síncrono, também foram realizados testes onde foi simulado que o escore heurístico atingiria 90% do escore exato. O objetivo dessas simulações foi tentar verificar até que ponto a utilização dos módulos com escore heurístico poderia aumentar a área de poda.

A. *Sequências Utilizadas*

Os testes foram conduzidos com um conjunto de 14 sequências distintas, sendo realizado um total de 8 comparações entre pares dessas sequências. Dentre essas

comparações, 6 delas foram as mesmas feitas nos testes do módulo APA [6], buscando comparar diretamente a diferença de desempenho.

As sequências utilizadas são segmentos de DNA de organismos reais, retirados do site do National Center for Biotechnology Information (NCBI). A Tabela I exibe as comparações realizadas, apresentando o nome científico dos organismos, o tamanho das sequências e os escores resultantes dos alinhamentos. Isso inclui os escores heurísticos obtidos pelo BLASTn nos modos *trimmed* e *full*, além dos escores exatos provenientes da execução do MASA-Cudalign.

B. Comparação dos resultados

A Tabela II mostra o tempo de execução somente do BLASTn e o tempo de execução total (BLASTn e MASA-Cudalign) para o módulo síncrono (APA) e para o módulo assíncrono, incluindo os modos de execução: *trimmed* e *full*. Além disso, a última coluna apresenta o tempo total de execução no modo *no score*, onde apenas o MASA-Cudalign é executado. Já na tabela III é apresentando a porcentagem de poda para todos os testes feitos, tanto no modulo assíncrono quanto no síncrono.

Os resultados das comparações *3Mx3M*, *5Mx7M* e *23Mx23M* nas linhas um, três e cinco, respectivamente, na Tabela II mostram que praticamente não houve diferença no tempo total de execução entre os dois módulos e o modo *no score*. É possível notar que nos três casos o tempo de execução do BLASTn foi bem baixo, mesmo para as sequências relativamente grandes de 23M. Dessa forma, o *overhead* gerado não foi tão relevante em relação ao tempo total, oque justificaria a pouca variação de tempo observada.

A segunda linha da Tabela II mostra resultados favoráveis para o módulo assíncrono no alinhamento dos genomas das bactérias *Bacillus anthracis str. Ames* e *Bacillus anthracis str. Sterne*. É possível notar que o tempo de execução total do módulo síncrono sofreu um *overhead* em relação ao tempo do *no score*, tanto no modo *trimmed* quanto no *full*. Já no módulo assíncrono, devido a eliminação do *overhead* gerado pelo BLASTn, o tempo total de execução foi similar ao *no score*. Dessa forma, mesmo com um baixo aumento na porcentagem de poda (Tabela III), o módulo assíncrono conseguiu manter o tempo original da execução *no score*.

O resultado da comparação de duas montagens dos genomas da bactéria *Amycolatopsis mediterranei* (U32 e S699) são mostrados na quarta linha da Tabela II. No caso do modo *Full*, é possível observar que, mesmo com uma área de poda ligeiramente inferior (Tabela III), o módulo assíncrono conseguiu obter um tempo menor que o síncrono, pelo fato de ter eliminando o *overhead*. Já no caso do modo *trimmed*, a execução dos módulos fizeram pouca diferença na porcentagem de poda, como pode ser visto na quarta linha da Tabela III. Apesar disso, ainda é possível notar a eliminação do *overhead* no módulo assíncrono. Enquanto que no módulo síncrono o tempo de execução se iguala ao modo *no score*.

As linhas sete, oito e nove da Tabela II mostram os resultados dos últimos três alinhamentos testados. Os pares de sequências comparadas nesses três casos possuem um elevado grau de similaridade entre si, sendo assim, os escores exatos obtidos são altos, como pode ser visto na Tabela I. Já os escores heurísticos obtidos pelo BLASTn não foram grandes o suficiente, quando comparados ao escore exato. Consequentemente a execução do algoritmo heurístico não contribuiu para o aumento na área de poda (Tabela III). Porém, mesmo que os escores heurísticos não tenham sido bons o suficiente para aumentar a poda, eles ainda foram relativamente altos, demandando um tempo considerável de execução do BLASTn para serem obtidos, gerando um *overhead* grande. Ainda assim, é possível observar na Tabela II que o módulo assíncrono foi capaz de eliminar por completo esse *overhead*, enquanto que no módulo síncrono ele foi somado ao tempo de execução do CUDAlign, aumentando significativamente o tempo total de execução.

Ao compararmos a execução com o escore Full nos módulos síncrono e assíncrono, notamos que o desempenho do assíncrono é maior para todos os tamanhos de sequência.

C. Resultados das Simulações de Escores Heurísticos

A Tabela IV mostra os resultados obtidos ao simular um escore heurístico de 90% do escore ótimo. Esses testes foram todos feitos utilizando o modo *Full* dos módulos assíncrono e síncrono, considerando que os tempos de execução do BLASTn são os mesmos da execução original, apresentados na Tabela II.

Como esperado, o resultado das simulações das comparações *3Mx3M*, *5Mx7M* e *23Mx23M* não mudaram em nada em relação a aqueles vistos na Seção IV-B. Isso se manteve devido ao fato do escore ótimo desses alinhamentos ser muito baixo e o escore obtido pelo BLASTn para essas sequências já ser bastante próximo do ótimo (Tabela I).

Nos alinhamentos *5Mx5M* e *10Mx10M* obtivemos um aumento grande da área de poda tanto no módulo assíncrono como no síncrono. Consequentemente, tivemos uma diminuição do tempo de execução total nos dois casos. Já no caso do alinhamento *40Mx40M* houve apenas um leve aumento na área de poda nos dois módulos.

Por fim, é possível observar nas comparações *50Mx33M* e *50Mx50M* que ocorreu um aumento de poda no módulo síncrono consideravelmente maior do que no módulo assíncrono. Essa poda adicional possivelmente ocorreu no inicio do alinhamento, uma vez que o módulo APA já inicia o MASA-CUDAlign com o escore heurístico pronto, enquanto do módulo assíncrono apenas disponibiliza no meio da execução.

Observando os resultados da simulação na Tabela IV, é possível notar que o módulo assíncrono obteve um menor tempo de execução em todos os casos, mesmo naqueles onde o módulo APA conseguiu uma maior porcentagem de poda.

V. Conclusão

Os testes realizados no módulo assíncrono mostraram ganhos consideráveis de desempenho em relação ao módulo APA. Tanto nos casos onde ocorreu aumento de poda

TABLE I

SEQUÊNCIAS UTILIZADAS NOS TESTES E SEUS RESPECTIVOS ESCORES OBTIDOS NO ALGORITMO HEURÍSTICO (*trimmed* E *full*) E EXATO (*no score*)

Tam.	Sequência A	Sequência B	Escore BLATSn (Full)		Escore CUDAlign
	Nome	Nome	Trimmed	Full	
3M x 3M	*Corynebacterium efficiens*	*Corynebacterium glutamicum*	3888	3888	4226
5M x 5M	*Bacillus anthracis str. Ames*	*Bacillus anthracis str. Sterne*	831053	849975	5220960
5M x 7M	*Bacillus anthracis str. Ames*	*Rhodopirellula baltica*	NA	119	172
10M x 10M	*Amycolatopsis mediterranei U32*	*Amycolatopsis mediterranei S699*	2067218	6235710	10235188
23M X 23M	*Drosophila melanogaster (2L)*	*Drosophila melanogaster (3L)*	8153	9059	9063
40M X 40M	*Vitis cinerea var. helleri x Vitis rupestris*	*Vitis cinerea var. helleri x Vitis riparia*	86695	119799	9247283
50M X 33M	*Homo sapiens*	*Pan troglodytes*	-INF	119428	27206434
50M X 50M	*Homo sapiens*	*Gorilla gorilla gorilla*	-INF	112021	26589372

[a]Escore do CUDAlign foi obtido pela execução do modo *no score* de um dos módulos.

TABLE II

TEMPO DE EXECUÇÃO TOTAL E DO BLASTN COM O APA (SÍNCRONO) E O MAPA (ASSÍNCRONO)

Sequências Comparadas	Assíncrono				Síncrono				
	Tempo BLASTn (s)		Tempo Total (s)		Tempo BLASTn (s)		Tempo Total (s)		Tempo Total (s)
	Trimmed	***Full***	***Trimmed***	***Full***	***Trimmed***	***Full***	***Trimmed***	***Full***	***No Score***
3M x 3M	1,0	1,0	**60,0**	61,0	1,0	2,0	63,0	62,0	**60,0**
5M x 5M	2,0	6,0	**121,0**	122,0	2,0	5,0	128,0	131,0	123,0
5M x 7M	1,0	2,0	219,0	221,0	1,0	1,0	**217,0**	220,0	219,0
10M x 10M	2,0	13,0	413,0	**343,0**	3,0	14,0	427,0	364,0	428,0
23M X 23M	2,0	11,0	**3280,0**	3291,0	2,0	11,0	3295,0	3304,0	3287,0
40M X 40M	3,0	394,0	7157,0	**7148,0**	2,0	395,0	7181,0	7615,0	7226,0
50M X 33M	4,0	4810,0	7628,0	7633,0	2,0	4670,0	**7573,0**	12271,0	7581,0
50M X 50M	2,0	5623,0	11540,0	11482,0	2,0	5625,0	**11530,0**	17385,0	11552,0

[a]Os elementos em negrito representam o menor tempo total de cada comparação.

TABLE III

QUANTIDADE DE BLOCOS PODADOS (%) COM O APA (SÍNCRONO) E O MAPA (ASSÍNCRONO)

Sequências Comparadas	Blocos Podados (%)				
	Assíncrono		Síncrono		
	Trimmed	***Full***	***Trimmed***	***Full***	**No Score**
3M x 3M	0,12	0,12	0,12	0,12	0,12
5M x 5M	54,35	54,31	54,35	54,37	53,69
5M x 7M	0,0	0,0	0,0	0,0	0,0
10M x 10M	54,91	67,88	54,91	68,85	53,33
23M X 23M	0,04	0,04	0,04	0,04	0,04
40M X 40M	35,32	35,32	35,32	35,32	35,32
50M X 33M	34,78	34,78	34,78	34,78	34,78
50M X 50M	32,51	32,51	32,51	32,51	32,51

TABLE IV

TESTE DO MASA-CUDALIGN SIMULANDO UM ESCORE HEURÍSTICO DE 90% DO ESCORE ÓTIMO

Sequências Comparadas	Tempo de Execução (s)		Blocos Podados (%)	
	Async	***Sync***	***Async***	***Sync***
3M x 3M	**62,0**	**62,0**	0,12	0,12
5M x 5M	**84,0**	93,0	80,53	80,53
5M x 7M	**215,0**	216,0	0,0	0,0
10M x 10M	**272,0**	296,0	80,59	81,31
23M X 23M	**3268,0**	3392,0	0,04	0,04
40M X 40M	**6777,0**	7118,0	37,79	38,51
50M X 33M	**6646,0**	8572,0	43,56	81,40
50M X 50M	**9217,0**	12680,0	50,56	68,85

[a]Todos os resultados foram obtidos no modo *Full*.

(*5Mx5M* e *10Mx10M*) quanto nos casos onde não ocorreu (*40Mx40M*, *50Mx33M* e *50Mx50M*), devido a baixa eficiência do alinhamento feito pelo BLASTn, o módulo assíncrono conseguiu eliminar o *overhead* que está presente nos resultados do módulo síncrono.

Nas simulações do escore heurístico em 90% do escore ótimo também foram obtidos resultados favoráveis para o módulo assíncrono. Além disso, foi possível observar que é possível atingir porcentagens de poda acima de 80% na execução do algortimo com uma *GPU*.

REFERENCES

[1] Sandes, E. & Melo, A. CUDAlign: Using GPU to Accelerate the Comparison of Megabase Genomic Sequences. *Sigplan Notices - SIGPLAN*. **45** pp. 137-146 (2010,5)

[2] O. Sandes, E. & Melo, A. Smith-Waterman Alignment of Huge Sequences with GPU in Linear Space. *2011 IEEE International Parallel & Distributed Processing Symposium*. pp. 1199-1211 (2011)

[3] O. Sandes, E. & Melo, A. Retrieving Smith-Waterman Alignments with Optimizations for Megabase Biological Sequences Using GPU. *IEEE Transactions On Parallel And Distributed Systems*. **24**, 1009-1021 (2013)

[4] O. Sandes, E., Miranda, G., Melo, A., Martorell, X. & Ayguadé, E. CUDAlign 3.0: Parallel Biological Sequence Comparison in Large GPU Clusters. *2014 14th IEEE/ACM International Symposium On Cluster, Cloud And Grid Computing*. pp. 160-169 (2014)

[5] Sandes, E., Miranda, G., Martorell, X., Ayguade, E., Teodoro, G. & Melo, A. CUDAlign 4.0: Incremental Speculative Traceback for Exact Chromosome-Wide Alignment in GPU Clusters. *IEEE Transactions On Parallel And Distributed Systems*. **27**, 2838-2850 (2016)

[6] Pinho, M. Alinhamento Paralelo de Sequências Biológicas com Poda Agilizada em GPU. (Monografia, Universidade de Brasília,2023)

[7] Altschul, S., Gish, W., Miller, W., Myers, E. & Lipman, D. Basic local alignment search tool. *Journal Of Molecular Biology*. **215**, 403-410 (1990), https://www.sciencedirect.com/science/article/pii/S0022283605803602