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

Execução Automatizada e Flexível de Lotes de Comparações Paralelas de Sequências Biológicas

Pedro Lucas Pinto Andrade

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

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

Brasília
2022

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

Execução Automatizada e Flexível de Lotes de Comparações Paralelas de Sequências Biológicas

Pedro Lucas Pinto Andrade

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

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

Prof. Dr. Ricardo Pezzuol Jacobi
CIC/UnB

Prof.a Dr.a Aletéia Patrícia Favacho de Araújo
CIC/UnB

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

Brasília, 27 de setembro de 2022

Dedicatória

Dedico esse trabalho à minha família, que sempre me acompanhou nessa jornada, e aos meus amigos, que formaram quem eu sou hoje. Não estaria aqui se não fosse pela ajuda de todos vocês.

Agradecimentos

Agradeço à professora Alba, que me orientou e ensinou durante todo o processo de desenvolvimento deste trabalho. Não tenho dúvidas de que foi a melhor orientação que eu poderia ter.

Agradeço também aos meus amigos e à minha família, por fazerem parte da minha vida e estarem presentes sempre que precisei.

O presente trabalho foi realizado com apoio da Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES), por meio do Acesso ao Portal de Periódicos e com apoio do projeto CAPES-Pandemias, que disponibilizou a máquina com a placa gráfica (GPU).

Resumo

Este trabalho de graduação propõe uma ferramenta para obtenção de valores de parâmetros otimizados para a execução de alinhamentos de sequências biológicas, além da execução de lotes de alinhamentos, ambos de forma automatizada, utilizando a ferramenta MASA-CUDAlign. O objetivo da ferramenta proposta é auxiliar usuários do MASA-CUDAlign fornecendo valores adequados para os parâmetros de execução, obtidos por meio de *profiling*, de forma a otimizar o tempo de processamento que, no caso de sequências grandes, pode chegar à horas de execução por alinhamento. Além disso, a ferramenta também tem o objetivo de auxiliar os usuários fornecendo uma forma automatizada de executar lotes de múltiplos alinhamentos, sem a necessidade de interagir com a ferramenta entre cada alinhamento, e receber os resultados de forma organizada ao final da execução. Para testar a ferramenta proposta foi utilizada uma máquina com uma GPU NVIDIA GeForce RTX 2060, para executar alinhamentos com sequências reais de DNA de tamanho 1M, 3M, 4M, 6M e 8M, utilizando os valores de parâmetros já obtidos com a ferramentas proposta por meio de um *profiling*. Os resultados encontrados mostram que os valores de parâmetros obtidos pela ferramenta são adequados, alcançando o melhor desempenho para a maioria das comparações testadas, e um desempenho bem melhor do que os piores valores de parâmetros testados. Além disso, a execução em lote funcionou de forma satisfatória, realizando a interface com o MASA-CUDAlign, executando os múltiplos alinhamentos e entregando os resultados de forma organizada.

Palavras-chave: Comparação de sequências biológicas, MASA-CUDAlign, *profiling*, GPU

Abstract

This undergraduate work proposes a tool to obtain optimized values for execution parameters used in the alignment of biological sequences by the MASA-CUDAlign tool, as well as to use said tool to execute batches of alignments. The objective of the proposed tool is to help MASA-CUDAlign users by supplying adequate values for the execution parameters, obtained using profiling, to optimize the processing time which, in the case of large sequences, can be of multiple hours per alignment. In addition, it also has the objective of assisting the users by providing an automated method to execute batches of alignments, without the need to interact with the tool between each alignment, and gather all the results in an organized manner after the execution is done. To test the proposed tool, we used a desktop with an NVIDIA GeForce RTX 2060 GPU to execute alignments with real DNA sequences of sizes 1M, 3M, 4M, 6M and 8M, using the parameter values already obtained by the proposed tool with profiling. The collected results show that the parameter values obtained by the tool are adequate, achieving the best performance in most of the tested comparisons and a much better performance than the worst parameter values tested. Furthermore, the batch execution worked in a satisfactory way, interfacing with the MASA-CUDAlign tool and collecting and delivering the results in an organized manner.

Keywords: Biological sequences comparison, MASA-CUDAlign, profiling, GPU

Sumário

1 Introdução **1**

2 Comparação de Sequências Biológicas **3**

2.1 Sequências biológicas . . . 3

2.2 Alinhamento e escore . . . 3

2.3 Método da programação dinâmica e algoritmos para alinhamento de sequências . 4

2.3.1 Algoritmo de Needleman-Wunsch (NW) . . . 5

2.3.2 Algoritmo de Smith-Waterman (SW) . . . 5

2.3.3 Algoritmo de Gotoh . . . 6

2.3.4 Algoritmo de Hirschberg . . . 9

2.3.5 Algoritmo de Myers e Miller (MM) . . . 11

3 Ferramentas da Família CUDAlign **12**

3.1 CUDAlign em uma GPU . . . 13

3.1.1 CUDAlign 1.0 . . . 13

3.1.2 CUDAlign 2.0 . . . 13

3.1.3 CUDAlign 2.1 - Poda de Blocos (*Block Pruning* - BP) . . . 14

3.2 CUDAlign em múltiplas GPUs . . . 15

3.2.1 CUDAlign 3.0 . . . 15

3.2.2 CUDAlign 4.0 . . . 16

3.3 Arquitetura MASA . . . 16

3.4 MASA-CUDAlign-MultiBP . . . 19

3.4.1 Static-MultiBP . . . 19

3.4.2 Dynamic-MultiBP . . . 20

3.5 Parâmetros que afetam o desempenho do CUDAlign . . . 21

4 Projeto da Ferramenta para Execução Otimizada e Automatizada do CUDAlign **23**

4.1 Motivação . . . 23

4.2 Visão Geral . . . 24

4.3 Módulo Gerência (A) 26
4.4 Módulo Perfil (B) 28
4.4.1 Perfil completo 29
4.4.2 Perfil simplificado 30
4.5 Módulo Execução (C) 31
4.6 Módulo Resultado (D) 32
4.7 Implementação 33

5 Resultados Experimentais **34**
5.1 Sequências Utilizadas 34
5.2 Configuração do Ambiente 34
5.3 Configuração da Ferramenta 35
5.4 Execução do *Profiling* 36
5.5 Execuções com os Resultados do *Profiling* 38
5.6 Execução em Lote 42

6 Conclusão e Trabalhos Futuros **44**

Referências **46**

Lista de Figuras

2.1 Exemplo de alinhamento de duas sequências S_0 e S_1. 4
2.2 Alinhamento global com Needleman-Wunsch. 6
2.3 Alinhamento local com Smith-Waterman. 7
2.4 Alinhamento global com Gotoh. 8
2.5 Alinhamento local com Gotoh. 9
2.6 Matriz com o ponto médio ótimo e as duas sub-matrizes resultantes da divisão da inicial, representando a execução do algoritmo de Hirschberg - adaptado de [1]. 10

3.1 Processamento em paralelogramo no CUDAlign 1.0 [2]. 13
3.2 Poda de blocos em um alinhamento [3]. 15
3.3 Comunicação entre GPUs no CUDAlign 3.0 [4]. 16
3.4 *Incremental Speculative Traceback*, adaptado de [5]. 17
3.5 Módulos da arquitetura MASA, adaptado de [6]. 18
3.6 Compartilhamento de *score* no MultiBP [1]. 20
3.7 Estratégia dinâmica do MultiBP [1]. 21

4.1 Visão geral do projeto. 25
4.2 Sequência de execução de um *profiling*. 25
4.3 Sequência de execução de um lote de alinhamentos. 26
4.4 Tela inicial do gerenciador. 27
4.5 Estrutura do arquivo de parâmetros. 27
4.6 Tela do módulo Gerência na opção de alinhamentos. 28
4.7 Arquivo de configuração do lote de alinhamentos. 28
4.8 Tela do *profiling*. 28
4.9 Linha de comando para execução do CUDAlign com os parâmetros $B = 480$ e $T = 128$. 32
4.10 Comparação entre execuções com modo verboso desligado (esquerda) e ligado (direita). 32

5.1 Comandos utilizados para compilação do MASA CUDAlign. 35
5.2 Comandos utilizados para compilação da ferramenta proposta neste trabalho. . 36
5.3 Comparação do desempenho entre os melhores e piores parâmetros, e os parâmetros escolhidos pelo *profiling* da ferramenta. 40
5.4 Arquivo **configurações_de_alinhamentos** com 5 comparações configuradas para execução em lote. 42
5.5 Diretórios dos resultados da execução em lote das comparações da Tabela 5.1. . 43

Lista de Tabelas

4.1 Pares de parâmetros utilizados em um *profile*. 29
4.2 Pares de sequências utilizadas em um perfil completo. 30
4.3 Pares de sequências utilizadas em um perfil simplificado. 30

5.1 Pares de sequências utilizadas para avaliar os parâmetros escolhidos. 35
5.2 Melhores, piores e parâmetros médios para sequências de 1M x 1M (a). 36
5.3 Melhores, piores e parâmetros médios para sequências de 3M x 3M (a). 36
5.4 Melhores, piores e parâmetros médios para sequências de 5M x 5M (a). 37
5.5 Melhores, piores e parâmetros médios para sequências de 10M x 10M (a). . . . 37
5.6 Melhores, piores e parâmetros médios para sequências de 28M x 23M (a). . . . 37
5.7 Melhores, piores e parâmetros médios para sequências de 32M x 47M (a). . . . 37
5.8 Melhores valores de parâmetros encontrados pela ferramenta. 38
5.9 Desempenho de alinhamentos de 1M x 1M (b). 39
5.10 Desempenho de alinhamentos de 3M x 3M (b). 39
5.11 Desempenho de alinhamentos de 4M x 4M (b). 39
5.12 Desempenho de alinhamentos de 6M x 6M (b). 40
5.13 Desempenho de alinhamentos de 8M x 8M (b). 40

Capítulo 1

Introdução

A comparação de sequências biológicas é de grande importância na área da Bioinformática, pois ela produz uma métrica que descreve a similaridade entre as sequências [7]. Analisar a similaridade entre sequências de um organismo conhecido com as de outros diferentes permite entender melhor o funcionamento, a origem e a evolução desses organismos. As sequências, geralmente representadas como cadeias de caracteres, podem ser comparadas utilizando algoritmos de alinhamento, que consistem em parear os caracteres de duas sequências para analisar a similaridade entre essas cadeias. Entretanto, os algoritmos exatos demandam alto poder de processamento e, no caso de sequências longas, demoram muito tempo para serem serem executados.

Soluções utilizando GPUs (*Graphics Procesing Units*, ou placas gráficas) podem acelerar bastante o alinhamento. Assim, devido ao alto poder de processamento paralelo que esses dispositivos possuem, eles são uma alternativa muito boa para processar as matrizes utilizadas nos algoritmos de alinhamento. Contudo, essas soluções são muito sensíveis aos parâmetros de execução da GPU que determinam o número de blocos (B) e o número de *threads* (T) [8].

A ferramenta MASA-CUDAlign [5], [1] obtém alinhamentos ótimos de sequências biológicas utilizando GPUs para acelerar a computação, utilizando estratégias como a poda de blocos para evitar cálculos desnecessários e aumentar ainda mais o desempenho da execução. Além disso, a ferramenta também é capaz de utilizar múltiplas GPUs, de capacidades similares ou diferentes, para dividir a computação e aumentar o desempenho. Mas, como as outras ferramentas em GPU, é sensível aos parâmetros de entrada da execução, e esses parâmetros não são intuitivos para o usuário. Além disso, o usuário normalmente executa várias comparações de sequências e a ferramenta não fornece essa funcionalidade.

Portanto, o objetivo do presente trabalho de graduação é propor, implementar e avaliar uma ferramenta que faça as sugestões dos parâmetros de execução com base na GPU utilizada pelo usuário por meio de um *profiling*, e também permita a execução de várias comparações de maneira simples e acessível com uma execução em lote. O *profiling* consiste em executar a fer-

ramenta MASA-CUDAlign com variadas combinações de valores de parâmetros, com o intuito de coletar e analisar dados de desempenho de cada uma das combinações para que seja possível selecionar os valores que fornecem um desempenho adequado, reduzindo o tempo que o usuário gasta aguardando os alinhamentos. A execução em lote fornece uma forma acessível ao usuário para configurar e executar lotes de alinhamentos de sequências e coletar os resultados posteriormente, sem que seja necessário intervir entre cada execução, dando flexibilidade ao uso da ferramenta MASA-CUDAlign.

O restante deste documento está organizado como descrito a seguir. O Capítulo 2 descreve como são realizadas as comparações de sequências biológicas e alguns algoritmos exatos para obter os alinhamentos. O Capítulo 3 apresenta as ferramentas da família CUDAlign e o seu histórico de desenvolvimento. O Capítulo 4 apresenta o projeto da ferramenta proposta e o seu funcionamento. Os resultados obtidos com a nossa ferramenta são apresentados no Capítulo 5. Finalmente, o Capítulo 6 apresenta as conclusões deste trabalho de graduação e trabalhos futuros a serem realizados.

Capítulo 2

Comparação de Sequências Biológicas

2.1 Sequências biológicas

Uma sequência biológica é uma molécula formada por ácidos nucleicos ou amino ácidos, que por sua vez são agrupamentos de 3 ácidos nucleicos [9]. As sequências geralmente são codificadas com caracteres, para que possam ser analisadas e comparadas com o uso de algoritmos de alinhamento de sequências. Dependendo do tipo de sequência sendo tratada, diferentes alfabetos são utilizados para representá-la [7]:

- DNA: A (Adenina), C (Citosina), G (Guanina), T (Timina);
- RNA: A (Adenina), C (Citosina), G (Guanina), U (Uracila);
- Proteínas: A (Alanina), C (Cisteína), D (Aspartato), E (Glutamato), F (Fenilalanina), G (Glicina), H (Histidina), I (Isoleucina), K (Lisina), L (Leucina), M (Metionina), N (Asparagina), P (Prolina), Q (Glutamina), R (Arginina), S (Serina), T (Treonina), V (Valina), W (Triptofano), Y (Tirosina);

2.2 Alinhamento e escore

O alinhamento de sequências biológicas [7], ou *pairwise alignment*, é importante para obter informações sobre a função, estrutura ou evolução de uma sequência biológica. No alinhamento, os caracteres das sequências são pareados, de maneira a ressaltar as semelhanças. Em muitos casos, espaços (*gaps*) são introduzidos em uma das sequências para melhorar a qualidade do alinhamento. A Figura 2.1 ilustra um alinhamento entre duas sequências S_0 e S_1.

O escore (*score*) é um método de pontuação utilizado para avaliar os alinhamentos obtidos [7]. No caso de um algoritmo de programação dinâmica para alinhamento de sequências, uma pontuação é dada para cada caso possível na comparação de dois caracteres: *match* (acerto),

$S_0:$	**A**	**T**	**C**	**A**	**T**	**C**	
	|		|	|	|	|	
$S_1:$	**A**	_	**C**	**A**	**A**	**C**	
	$+1$	-2	$+1$	$+1$	-1	$+1$	$= +1(score)$

Figura 2.1: Exemplo de alinhamento de duas sequências S_0 e S_1.

mismatch (erro) e *gap* (espaço vazio). Na Figura 2.1, as pontuações associadas a *matches*, *mismatches* e *gaps* são, respectivamente, +1, -1 e -2.

Existem dois tipos básicos de alinhamento, global e local. O alinhamento global consiste em alinhar todos os caracteres das duas sequências. Isso implica que pode ser necessário alinhar uma grande parte de uma das sequências com espaços vazios, caso o tamanho das duas seja diferente. Portanto, o alinhamento global não é recomendado quando o comprimento das sequências é muito distinto.

A ideia do alinhamento local é alinhar as partes de uma sequência que possuem a maior densidade de semelhanças com a outra sequência. Dessa forma, não há problema caso as sequências tenham tamanhos diferentes.

2.3 Método da programação dinâmica e algoritmos para alinhamento de sequências

A programação dinâmica é um método computacional que consiste em dividir um problema em problemas menores e obter soluções ótimas para esses problemas menores. As soluções são salvas e utilizadas novamente, sem a necessidade de serem recalculadas, para estender o problema obtendo mais soluções até que se alcance o resultado do problema completo.

Ao unir as soluções ótimas dos sub-problemas, obtém-se a solução ótima para o problema inicial [10]. O método da programação dinâmica [7] consegue alinhar sequências levando em consideração *gaps* de tamanho arbitrário nas mesmas, para aumentar a quantidade de pares alinhados.

Para medir a qualidade de um alinhamento, é utilizado um sistema de *score* que penaliza pares incompatíveis e *gaps*, enquanto favorece pares compatíveis (Seção 2.2). Portanto, ao analisar cada par, existem 3 escolhas possíveis: alinhar os caracteres, alinhar o caractere da sequência S_0 com um *gap* na sequência S_1, alinhar o caractere da S_1 com um *gap* na S_0. Se em cada etapa for escolhida a opção que garante um alinhamento ótimo até aquele ponto, o alinhamento final também é ótimo [7].

Para realizar o alinhamento, uma matriz de programação dinâmica é utilizada. Cada célula guarda a melhor pontuação obtida até aquele ponto, podendo ter sido atingida devido a qualquer uma das 3 opções possíveis. Uma matriz auxiliar, ou a mesma matriz, guarda as posições e escolhas que contribuíram para essa pontuação.

Na fase 1 do alinhamento, a matriz de programação dinâmica é calculada e preenchida. Na fase 2 é realizado o *traceback*, no qual é encontrado o alinhamento que levou à pontuação ótima, com base nas posições e escolhas que levaram a essa pontuação.

2.3.1 Algoritmo de Needleman-Wunsch (NW)

O algoritmo de Needleman-Wunsch [7], [11], [12] é utilizado para obter um alinhamento global ótimo das sequências, seguindo o método da programação dinâmica. Quando toda a matriz for preenchida, a pontuação do alinhamento ótimo será a pontuação na última linha e coluna. O algoritmo maximiza os acertos em toda a extensão das sequências. A Equação 2.1 descreve como cada célula da matriz de programação dinâmica H é calculada pelo algoritmo, onde H_{ij} representa a pontuação da célula na linha i, coluna j; $s(S_0[i], S_1[j])$ é a pontuação obtida ao alinhar o caractere i da sequência S_0 ($S_0[i]$), com o caractere j da sequência S_1 ($S_1[j]$); e g é a penalidade do *gap*.

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

A Figura 2.2 mostra um alinhamento realizado entre as sequências S_0 = *GATCCTTT* e S_1 = *ATGATGAC*, utilizando o algoritmo NW, com as seguintes pontuações: *match* = +2, *mismatch* = -1, *gap* = -2. As células em negrito são as que compõem o alinhamento ótimo.

2.3.2 Algoritmo de Smith-Waterman (SW)

O algoritmo de Smith-Waterman [7], [13], [14] segue o método da programação dinâmica assim como o Needleman-Wunsch (Seção 2.3.1), mas com modificações no sistema de pontuação para obter alinhamentos locais. Para isso, quando um valor da matriz de programação dinâmica fica negativo, é trocado por zero. Dessa forma, o alinhamento até aquele ponto é, efetivamente, terminado. Portanto, ao realizar o *traceback*, isso é utilizado para delimitar os alinhamentos. A Equação 2.2 descreve como cada célula da matriz de programação dinâmica H é calculada pelo algoritmo de Smith-Waterman. Nessa equação, H_{ij} representa a pontuação da célula na linha i, coluna j; $s(S_0[i], S_1[j])$ é a pontuação obtida ao alinhar o caractere i da sequência S_0, com o caractere j da sequência S_1; e g é a pontuação do *gap*. A diferença da equação no algoritmo

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

(a) Matriz H.

_	**A**	**T**	**G**	**A**	**T**	**G**	**A**	**C**
	|	|			|			
G	**A**	**T**	**C**	**C**	**T**	**T**	**T**	_

(b) Alinhamento global ótimo.

Figura 2.2: Alinhamento global com Needleman-Wunsch.

de Smith-Waterman para a equação do algoritmo de Needleman-Wunsch se dá pela adição do valor 0 como opção na função de máximo. Dessa forma, caso os outros valores sejam negativos, o valor 0 é o escolhido, fazendo com que nenhuma célula da matriz de programação dinâmica assuma um valor negativo.

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

A Figura 2.3 mostra o alinhamento local ótimo realizado entre as sequências S_0 = *GATCCTTT* e S_1 = *ATGATGAC*, utilizando o algoritmo SW, com as seguintes pontuações: *match* = +2, *mismatch* = -1, *gap* = -2. As células em negrito são as que compõem o alinhamento local ótimo.

2.3.3 Algoritmo de Gotoh

Assim como os algoritmos de Needleman-Wunsch (Seção 2.3.1) e de Smith-Waterman (Seção 2.3.2), o algoritmo proposto por Gotoh [1], [15], [16] segue o modelo de programação dinâmica

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

(a) Matriz H.

```
G A T
| | |
G A T
```

(b) Alinhamento local ótimo.

Figura 2.3: Alinhamento local com Smith-Waterman.

para alinhar duas sequências biológicas. A diferença desse método está no modelo de *affine gap*. A ideia por trás do *affine gap* é que, biologicamente, um evento único de inserção ou remoção é mais provável do que vários pequenos eventos separados. Ou seja, faz mais sentido biológico favorecer *gaps* maiores e em menor quantidade, do que *gaps* menores e em maior quantidade. Para favorecer *gaps* maiores, o algoritmo de Gotoh utiliza penalidades diferentes para inserir um novo *gap* (penalidade maior) e para extender um *gap* já existente (penalidade menor).

A Equação 2.3 descreve como cada célula da matriz H é calculada pelo algoritmo de Gotoh. A matriz H representa as melhores pontuações obtidas em cada célula à medida que o alinhamento é realizado, e é análoga à matriz de programação dinâmica do algoritmo de Smith-Waterman ou Neddleman-Wunsch. A Equação 2.4 descreve como uma sequência de *gaps* na sequência S_1 são calculados, gerando a matriz E. A matriz E é responsável por armazenar as pontuações possíveis no caso de inserções ou extensões de *gaps* na sequência S_1. A Equação 2.5 é similar a Equação 2.4, mas considerando uma sequência de *gaps* na sequência S_0 e gerando a matriz F. A matriz F então é responsável por armazenar as pontuações possíveis no caso de inserções ou extensões de *gaps* na sequência S_0.

A Figura 2.4 mostra o alinhamento global ótimo obtido com as sequências S_0 = *GATTGACT* e S_1 = *GAGACTA*, utilizando o algoritmo de Gotoh, com as seguintes pontuações: *match* = +1, *mismatch* = -1, inserção de *gap* (G_{first}) = -4, extensão de *gap* (G_{ext}) = -1. As células em negrito na matriz H (Figura 2.4a) compõem o alinhamento ótimo.

$$H_{ij} = \max \begin{cases} H_{i-1,j-1} + s(S_0[i], S_1[j]), \\ E_{i,j}, \\ F_{i,j} \end{cases} \tag{2.3}$$

$$E_{ij} = \max \begin{cases} E_{i,j-1} + G_{ext}, \\ H_{i,j-1} + G_{first} \end{cases} \tag{2.4}$$

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

	*	G	A	G	A	C	T	A
*	**0**	−4	−5	−6	−7	−8	−9	−10
G	−4	**1**	−3	−4	−5	−6	−7	−8
A	−5	−3	**2**	−2	−3	−4	−5	−6
T	−6	−4	−2	1	−3	−4	−3	−6
T	−7	−5	**-3**	−3	0	−4	−3	−4
G	−8	−6	−4	**-2**	−4	−1	−5	−4
A	−9	−7	−5	−5	**-1**	−5	−2	−4
C	−10	−8	−6	−6	−5	**0**	−4	−3
T	−11	−9	−7	−7	−6	−4	**1**	< **-3**

(a) Matriz H.

	*	G	A	G	A	C	T	A
*	−	−	−	−	−	−	−	−
G	$-\infty$	−8	−3	−4	−5	−6	−7	−8
A	$-\infty$	−9	−7	−2	−3	−4	−5	−6
T	$-\infty$	−10	−8	−6	−3	−4	−5	−6
T	$-\infty$	−11	−9	−7	−7	−4	−5	−6
G	$-\infty$	−12	−10	−8	−6	−7	−5	−6
A	$-\infty$	−13	−11	−9	−9	−5	−6	−6
C	$-\infty$	−14	−12	−10	−10	−9	−4	−5
T	$-\infty$	−15	−13	−11	−11	−10	−8	**-3**

(b) Matriz E.

	*	G	A	G	A	C	T	A
*	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$
G	−	−8	−9	−10	−11	−12	−13	−14
A	−	−3	−7	−8	−9	−10	−11	−12
T	−	−4	**-2**	−6	−7	−8	−9	−10
T	−	−5	**-3**	−3	−7	−8	−7	−10
G	−	−6	−4	−4	−4	−8	−7	−8
A	−	−7	−5	−5	−5	−5	−8	−8
C	−	−8	−6	−6	−5	−6	−6	−8
T	−	−9	−7	−7	−6	−4	−7	−7

(c) Matriz F.

```
G  A  T  T  G  A  C  T  _
|  |        |  |  |  |
G  A  _  _  G  A  C  T  A
```

(d) Alinhamento global ótimo.

Figura 2.4: Alinhamento global com Gotoh.

O algoritmo de Gotoh encontra alinhamento globais, mas pode ser adaptado para encontrar alinhamentos locais [17], seguindo a mesma ideia do Smith-Waterman (Seção 2.3.2). Para tanto, é necessário adicionar um 0 na função de máximo da equação de recorrência principal (matriz H). Não é necessário inserir 0 nas equações de recorrência das matrizes que calculam os *gaps* (E e F). Dessa forma, como o SW, qualquer possível alinhamento na matriz é efetivamente terminado quando chega a um valor 0. A Equação 2.6 é a equação de recursão do algoritmo de Gotoh com a adição do valor 0 na matriz H. As outras equações utilizadas no algoritmo (Equação 2.4 e Equação 2.5) permanecem iguais.

$$H_{ij} = \max \begin{cases} H_{i-1,j-1} + s(S_0[i], S_1[j]), \\ E_{i,j}, \\ F_{i,j}, \\ 0 \end{cases} \tag{2.6}$$

A Figura 2.5 mostra o alinhamento local ótimo obtido com as sequências S_0 = *GATTGACT* e S_1 = *GAGACTA*, utilizando o algoritmo de Gotoh modificado para alinhamentos locais, com as seguintes pontuações: *match* = +1, *mismatch* = -1, inserção de *gap* = -4, extensão de *gap* = -1. Ou seja, as mesmas sequências e pontuações utilizadas no alinhamento global com Gotoh da Figura 2.4. As células em negrito na matriz H (Figura 2.5a) compõem o alinhamento local ótimo.

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

(a) Matriz H.

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

(b) Matriz E.

	*	**G**	**A**	**G**	**A**	**C**	**T**	**A**
*	−∞	−∞	−∞	−∞	−∞	−∞	−∞	−∞
G	−	−4	−4	−4	−4	−4	−4	−4
A	−	−3	−4	−3	−4	−4	−4	−4
T	−	−4	−2	−4	−2	−4	−4	−3
T	−	−4	−3	−3	−3	−3	−3	−4
G	−	−4	−4	−4	−4	−4	−2	−4
A	−	−3	−4	−3	−4	−4	−3	−3
C	−	−4	−2	−4	−2	−4	−4	−3
T	−	−4	−3	−3	−3	−1	−4	−4

(c) Matriz F.

G A C T

| | | |

G A C T

(d) Alinhamento local ótimo.

Figura 2.5: Alinhamento local com Gotoh.

2.3.4 Algoritmo de Hirschberg

O algoritmo proposto por Hirschberg [18], [19] é uma adaptação do algoritmo de Needleman-Wunsch (Seção 2.3.1) com o objetivo de reduzir a complexidade de memória. A complexidade de tempo, assim como o NW, é $O(nm)$, onde n e m são os tamanhos das sequências sendo alinhadas, ou seja, quadrática em relação ao tamanho das entradas. Entretanto, a complexidade

de espaço é $O(min(n,m))$, onde n e m são os tamanhos das sequências sendo alinhadas, ou seja, é linear em relação ao tamanho das entradas.

Para conseguir isso, o algoritmo leva em consideração que, para calcular uma linha de células da matriz de programação dinâmica, só são necessárias as linhas diretamente anterior e a atual. Isso significa que para calcular uma linha n só é necessário ter armazenado a própria linha n e a linha $n-1$. As outras linhas anteriores não são utilizadas no cálculo e não precisam ser armazenadas.

Para realizar o alinhamento somente com 2 linhas da matriz, são necessárias diversas alterações no método. O algoritmo de Hirschberg começa calculando a matriz de programação dinâmica H utilizando 2 linhas, até chegar na metade da matriz. A linha do meio assim obtida é armazenada. Então, o algoritmo é executado novamente, mas do final para o início da matriz (ou do final para o início das sequências), até a metade da matriz. Os *scores* obtidos na linha do meio nesse passo são somados com os *scores* obtidos na linha do meio no passo anterior, célula a célula. A célula com a maior pontuação é armazenada, pois ela representa um ponto médio ótimo (*crosspoint*) que faz parte do alinhamento [18], [19]. Agora, com esse ponto médio, é possível dividir as metades de cima à esquerda e de baixo à direita da matriz inicial em duas novas matrizes, excluindo as seções por onde o alinhamento não passa. O algoritmo então é executado recursivamente em cada nova matriz, armazenando pontos médios ótimos em cada execução. Ao final, esses pontos formam o alinhamento ótimo das sequências.

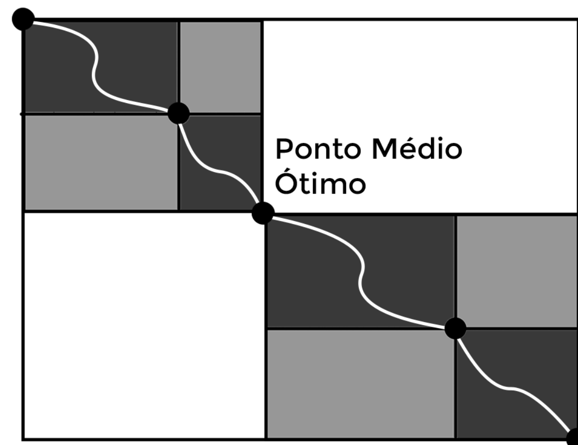

Figura 2.6: Matriz com o ponto médio ótimo e as duas sub-matrizes resultantes da divisão da inicial, representando a execução do algoritmo de Hirschberg - adaptado de [1].

Para calcular alinhamentos locais utilizando o algoritmo de Hirschberg, é necessária uma etapa inicial adicional. É necessário primeiro utilizar o algoritmo de Smith-Waterman (Seção

2.3.2) para calcular o ponto final do alinhamento local ótimo, mas utilizando a técnica de armazenar apenas as linhas necessárias presente na Seção 2.3.4. Ao terminar, é armazenado esse ponto final e o algoritmo de SW é executado novamente, mas do final para o início da matriz, até que seja encontrado o ponto com o mesmo *score* que o ponto final armazenado. Esse ponto encontrado é o início do alinhamento desejado. Com essas duas células é possível delimitar uma área da matriz onde sabe-se que o alinhamento ótimo reside, ou seja, o problema do alinhamento local transforma-se no problema do alinhamento global. Com isso é possível executar o Hirschberg normalmente da forma descrita na presente seção, levando em consideração apenas a sub-matriz delimitada pelo início e fim do alinhamento.

2.3.5 Algoritmo de Myers e Miller (MM)

O algoritmo proposto por Myers e Miller [20] é uma adaptação do algoritmo de Gotoh (Seção 2.3.3), utilizando a estratégia apresentada por Hirschberg (Seção 2.3.4) para obter alinhamentos ótimos com o modelo de *affine gap* em complexidade de espaço linear.

O funcionamento do algoritmo é bastante similar ao algoritmo de Hirschberg (Seção 2.3.4). O algoritmo calcula as células da matriz de programação dinâmica, descartando as linhas assim que se tornam desnecessárias para continuar os cálculos. Ao chegar na metade da matriz, armazena a linha central. Depois, realiza o mesmo processo do final para o início da matriz, também parando na linha central. Assim, somando-se as duas versões da linha do meio, encontra o ponto médio ótimo e divide a matriz em duas menores, como ilustrado na Figura 2.6. Dessa forma, recursivamente, o alinhamento é calculado.

Entretanto, para o funcionamento correto do modelo de *affine gap*, são necessárias mudanças. Ao calcular a matriz em um sentido, caso um *gap* seja inserido, uma penalidade para a sua abertura será incluída no cálculo. Ao calcular a matriz no sentido inverso, caso seja inserido um *gap* nessa mesma seção da matriz, a penalidade será aplicada novamente como a abertura de um *gap*, como se ele estivesse sendo aberto duas vezes. Para tratar essa situação, o algoritmo MM utiliza vetores e cálculos adicionais, permitindo que a estratégia funcione para o modelo de *affine gap*.

Capítulo 3

Ferramentas da Família CUDAlign

O CUDAlign [6] é uma ferramenta para obter o alinhamento local ótimo entre duas sequências de DNA longas, com o modelo de *affine gap* (Seção 2.3.3) e com complexidade de espaço linear, em unidades de processamento gráfico (GPUs) que suportam *Compute Unified Device Architecture* (CUDA). CUDA [21] é uma plataforma para computação paralela desenvolvida pela NVIDIA para computação geral em GPUs. Enquanto unidades de processamento central (CPUs) são otimizadas para processamento sequencial, GPUs são otimizadas para processamento paralelo. Para encontrar o alinhamento, a ferramenta CUDAlign combina os algoritmos Gotoh (Seção 2.3.3) e MM (Seção 2.3.5) e iterativamente obtém as coordenadas do alinhamento ótimo, que são incrementalmente refinadas até obter o alinhamento completo.

O CUDAlign foi desenvolvido de maneira incremental. A sua primeira versão foi o CUDAlign 1.0, que calculava somente o *score* em uma GPU. Em seguida, foi desenvolvida a versão 2.0 [22], que calculava o *score* e obtinha o alinhamento em uma GPU. A versão 2.1 [3] também obtinha o *score* e o alinhamento em uma GPU, mas aplicava a estratégia de poda de blocos (*block pruning*) para acelerar a computação.

Apesar de entregar técnicas otimizadas e de executar mais rápido do que a maioria das ferramentas da época, o CUDAlign 2.1 ainda demorava horas ou mesmo dias, quando as sequências continham dezenas de milhões de pares de caracteres. Para otimizar esse tempo foi então necessária a evolução da ferramenta para execução em várias GPUs. O CUDAlign 3.0 [4] obtinha o *score* com várias GPUs e o CUDAlign 4.0 obtém o *score* e o alinhamento em várias GPUs [5]. Paralelamente ao desenvolvimento do CUDAlign 4.0, foi desenvolvida a arquitetura MASA [6], que re-restruturou o código do CUDAlign em duas partes - independente de dispositivo e dependente de dispositivo - permitindo a criação de implementações para CPU. O CUDAlign 4.0 foi então renomeado para MASA-CUDAlign 4.0 e essa versão ainda está em atividade. A última versão da ferramenta CUDAlign é o MASA-CUDAlign-MultiBP, que obtém o *score* em várias GPUs com poda de blocos.

3.1 CUDAlign em uma GPU

3.1.1 CUDAlign 1.0

A versão 1.0 do CUDAlign produz o *score* ótimo e as suas coordenadas na matriz de programação dinâmica, mas não recupera o alinhamento. Para que o cálculo seja feito na GPU, as células da matriz são agrupadas em blocos e processadas diagonalmente utilizando um padrão de paralelogramo, que é uma otimização proposta pelo CUDAlign. Esse padrão maximiza o paralelismo durante a execução em GPU [2]. Para resolver pendências nas bordas laterais da matriz que ocorrem devido ao padrão de paralelogramo, um bloco na próxima diagonal processa as células pendentes de um bloco da diagonal anterior, em um processo chamado delegação de células. Além disso, para resolver as pendências entre blocos de diagonais diferentes, foi realizada uma divisão na execução de um bloco em duas fases: a fase curta e a fase longa [2], [8]. A fase curta visa terminar de calcular as células pendentes, enquanto a fase longa visa processar o restante do bloco. Essa divisão de fases cria uma sincronia entre os blocos, garantindo que ao calcular um bloco, todas as suas pendências já foram calculadas anteriormente. A Figura 3.1 ilustra o padrão de paralelogramo.

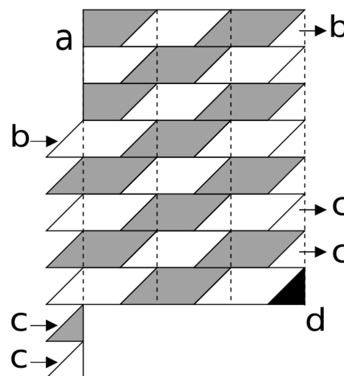

Figura 3.1: Processamento em paralelogramo no CUDAlign 1.0 [2].

3.1.2 CUDAlign 2.0

O CUDAlign 2.0 [22], [23] introduziu a funcionalidade para recuperar o alinhamento ótimo, além do *score* ótimo, que ainda não era realizada pelo CUDAlign 1.0. A sua execução é dividida em 6 estágios onde o primeiro estágio é responsável por calcular a matriz de programação dinâmica, os estágios 2 a 5 são responsáveis pelo *traceback* para encontrar o alinhamento, e o estágio 6 permite a visualização desse alinhamento.

O estágio 1 é o estágio mais computacionalmente intensivo e é executado em GPU. É responsável por calcular a matriz de Gotoh (Seção 2.3.3) em espaço linear, com o objetivo de

encontrar o *score* ótimo e a posição na matriz de programação dinâmica onde ele ocorre. Essa posição marca o final do alinhamento ótimo. Além da execução do CUDAlign 1.0 (Seção 3.1.1), esse estágio também armazena algumas linhas da matriz em disco, chamadas linhas especiais (SRA - *Special Rows Area*), para acelerar a computação nos próximos estágios. O número de linhas especiais é um parâmetro configurável do algoritmo. Quanto maior o número de linhas especiais, menor a área processada pelos estágios 2 e 3.

O estágio 2 computa a relação de recorrência para alinhamento semiglobal usando o algoritmo MM (Seção 2.3.5) em ordem reversa ao estágio 1, começando no final do alinhamento obtido. O objetivo é encontrar todos os pontos que pertencem ao alinhamento e cruzam as linhas especiais (*crosspoints*). As colunas especiais são salvas em disco.

Similarmente ao estágio 2, o estágio 3 utiliza as colunas armazenadas pelo estágio anterior para encontrar mais *crosspoints*, localizados dentro das partições que são áreas delimitadas por *crosspoints* consecutivos obtidas no estágio 2.

O estágio 4 executa o algoritmo MM (Seção 2.3.5) com múltiplas *threads* e é responsável por encontrar mais *crosspoints* que pertencem ao alinhamento, dentro das partições obtidas no estágio 3. O estágio 4 adiciona mais pontos à solução até que as partições estejam suficientemente pequenas. Como esse estágio executa-se muito rápido, sua execução é feita em CPU.

O estágio 5 executa o algoritmo NW (Seção 2.3.1) para alinhar as partições formadas pelos *crosspoints* encontrados no estágio 4 e concatena os resultados para obter o alinhamento ótimo completo. De maneira a se reduzir o espaço ocupado pelo alinhamento, o estágio 5 grava um arquivo binário que o representa. O estágio 5 também se executa em CPU.

Finalmente, o estágio 6 é opcional e converte o arquivo binário em um arquivo texto, para visualização.

3.1.3 CUDAlign 2.1 - Poda de Blocos (*Block Pruning* - BP)

O CUDAlign 2.1 [3], baseado no 2.0, teve como principal contribuição a poda de blocos, ou *block pruning*. Essa poda reduz a quantidade de células da matriz de programação dinâmica que precisam ser processadas, otimizando mais a execução da ferramenta.

Uma das otimizações mais importantes do CUDAlign é a poda de blocos, ou *block pruning* (BP), introduzida no CUDAlign 2.1 [3]. No CUDAlign as células da matriz de programação dinâmica são organizadas em blocos, observando a mesma relação de dependência existente entre as células. Ou seja, para calcular um bloco, só são necessários os blocos diretamente à esquerda e acima do mesmo. Um bloco é podado quando ele tem um *score* inicial tão baixo que é matematicamente impossível obter um alinhamento ótimo que o cruza. Portanto, não é necessário calcular as células pertencentes a esse bloco.

A Figura 3.2 ilustra a poda de blocos, onde a área em cinza não foi calculada.

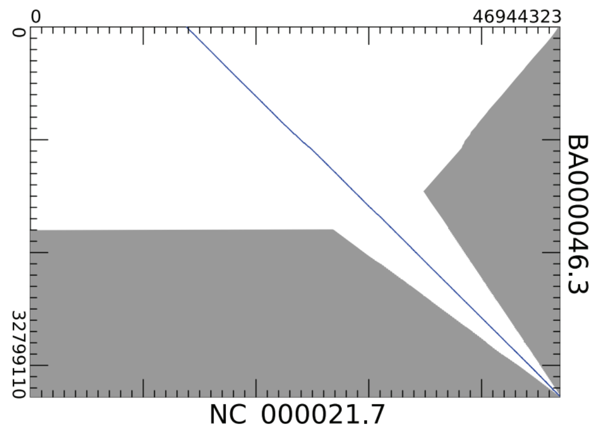

Figura 3.2: Poda de blocos em um alinhamento [3].

Como a verificação de poda é aplicada a um bloco de células e não a cada célula individual, o descarte de blocos é bem mais rápido que o procedimento de poda célula a célula. Entretanto, caso os blocos sejam muito grandes, torna-se mais difícil encontrar blocos inteiramente podáveis [3].

3.2 CUDAlign em múltiplas GPUs

3.2.1 CUDAlign 3.0

A versão 3.0 do CUDAlign foi uma evolução que permitiu executar o algoritmo de Gotoh (Seção 2.3.3) em múltiplas GPUs paralelamente, dividindo a matriz de programação dinâmica entre as GPUs. A proporção entre processamento e comunicação entre as GPUs foi cuidadosamente analisada para garantir bons resultados [4].

Para realizar a conexão entre as unidades de processamento gráfico, são utilizados *sockets* TCP, com as GPUs organizadas logicamente de forma linear, onde cada uma se comunica apenas com as vizinhas da esquerda e da direita. Cada GPU computa parte da matriz de programação dinâmica e transfere blocos de células da última coluna para a próxima GPU, observando a dependência para o cálculo de cada célula. A comunicação é realizada de forma assíncrona, utilizando *buffers* de entrada e saída de cada GPU. Se esses *buffers* ficam cheios, geram uma retenção na comunicação. Essa versão do algoritmo utilizando múltiplas GPUs obtém apenas o *score* ótimo e não obtém o alinhamento.

A Figura 3.3 ilustra o funcionamento do CUDAlign 3.0.

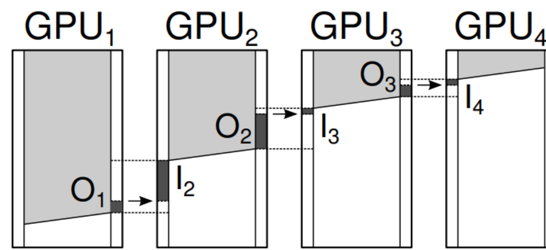

Figura 3.3: Comunicação entre GPUs no CUDAlign 3.0 [4].

3.2.2 CUDAlign 4.0

A versão 4.0 do CUDAlign mantém o funcionamento da versão 3.0 com algumas alterações, mas executa o *traceback* na matriz calculada para recuperar o alinhamento local ótimo [5]. Portanto, é capaz de obter o alinhamento ótimo executando em múltiplas GPUs.

Ao terminar de calcular suas células, uma GPU começa a calcular um *traceback* especulativo por meio da técnica *Incremental Speculative Traceback* (IST), assumindo que o ponto de maior *score* na borda (coluna mais à direita) é um *crosspoint*. Ao receber o *crosspoint* correto, compara com o utilizado no início de cada especulação. Caso sejam iguais, a especulação está correta. Caso contrário, os *crosspoints* corretos são calculados. Isso permite aproveitar o tempo ocioso que algumas GPUs teriam ao terminar de calcular suas partes da matriz [1] [5]. A Figura 3.4 ilustra a técnica *Incremental Speculative Traceback* utilizada no CUDAlign 4.0. Do ponto de vista da GPU_i, T_a é o tempo no qual ela fica ociosa, esperando que a GPU_{i-1} envie o primeiro bloco de células, T_1 é o tempo no qual a GPU_i calcula o estágio 1. Ao terminar de calcular sua parte da matriz no estágio 1, a GPU_i começa imediatamente a recuperar *crosspoints* de maneira especulativa, representado pelas linhas diagonais indo em direção á esquerda. Ao receber o *crosspoint* correto, a GPU_i verifica se houve acerto em sua especulação, executando os estágios 3 e 4. Finalmente, os estágios 5 e 6 são executados pela GPU_1, que conclui a execução.

Para realizar o cálculo em múltiplas GPUs no CUDAlign 3.0 e 4.0, foi necessário desabilitar a poda de blocos (Seção 3.1.3). Com cada GPU calculando uma parte da matriz, não existe uma visão global dos cálculos, necessária para a execução do *block pruning*.

3.3 Arquitetura MASA

A arquitetura MASA (*Multiplatform Architecture for Sequence Aligners*) [6] tem como objetivo prover uma infraestrutura flexível e customizável para desenvolver alinhadores de sequência em múltiplas plataformas de hardware e software. Para realizar isso, implementa módulos independentes de plataforma para que sejam reutilizados pelos módulos de plataformas específicas. Foi

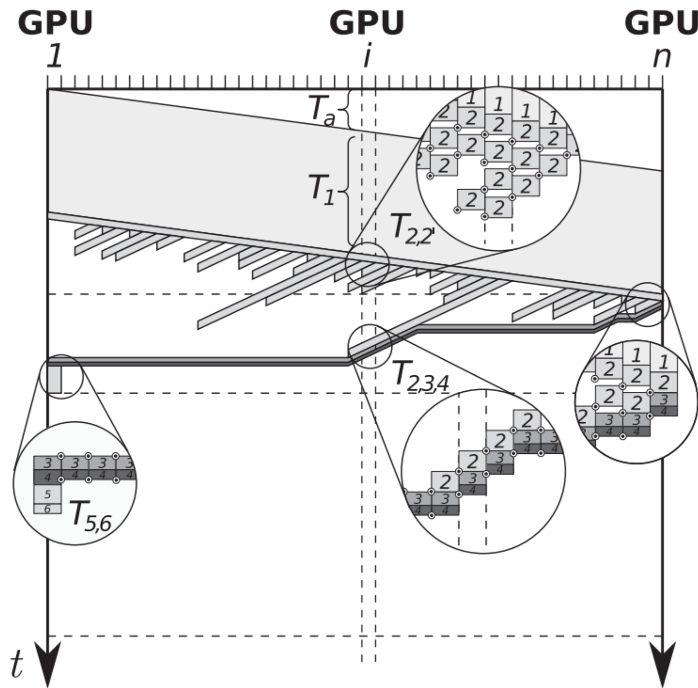

Figura 3.4: *Incremental Speculative Traceback*, adaptado de [5].

criada com base no algoritmo CUDAlign 2.1. Posteriormente, o CUDAlign 4.0 [5] foi implementado utilizando a arquitetura MASA. O CUDAlign passou a se chamar MASA-CUDAlign em 2016, quando a proposta da arquitetura foi publicada [6].

Grande parte do tempo de execução do algoritmo CUDAlign é gasto em *kernels* CUDA calculando a relação de recorrência. Essa é uma parte específica de plataforma, pois necessita de GPUs compatíveis com CUDA. Outras partes como operações de entrada/saída e coordenação de estágios são independentes de plataforma. O *block pruning* (Seção 3.1.3) e a estratégia de paralelização foram implementados de forma independente de plataforma na arquitetura MASA.

A organização da arquitetura MASA está dividida em 5 módulos, conforme apresentado na Figura 3.5. Os módulos de gerenciamento de dados, estatísticas e gerenciamento de estados são utilizados por todas as implementações, enquanto os módulos de *block pruning* e da estratégia de paralelização são customizáveis. Uma implementação MASA é a união de códigos dependentes e independentes de plataforma.

O módulo Gerenciamento de Dados (*Data Management*) é responsável por gerenciar dados de parâmetros do usuário, sequências de entrada, linhas e colunas especiais, alinhamento ótimo e *score*. Quando uma implementação MASA está sendo executada, ela se comunica com esse módulo para obter dados de entrada e armazenar resultados.

O módulo Estatísticas (*Statistics*) registra informações sobre a execução, como o tempo de

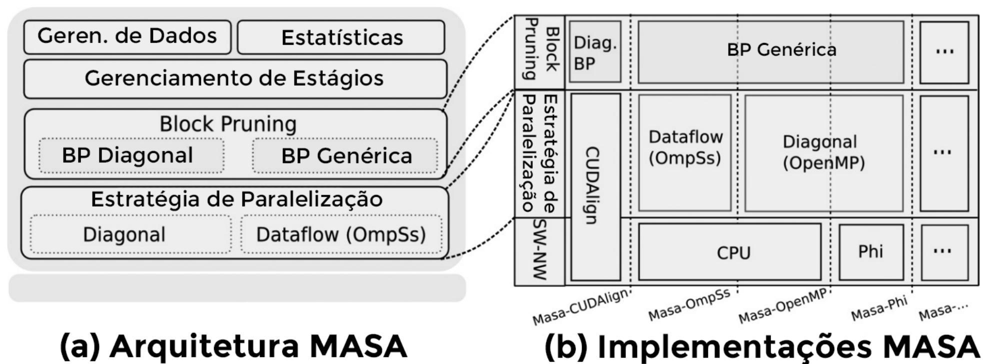

Figura 3.5: Módulos da arquitetura MASA, adaptado de [6].

execução de cada estágio, quantidade de memória e disco utilizada, porcentagem de blocos podados, entre outras.

O módulo Gerenciamento de Estágios (*Stage Management*) é responsável por coordenar a execução dos estágios 1 a 3, e por executar os estágios 4 a 6 na CPU. Durante os estágios 1 a 3, divide-se a matriz de programação dinâmica em partições e envia as partições para serem calculadas pelo código dependente de plataforma.

A otimização BP proposta no algoritmo CUDAlign 2.1 (Seção 3.1.3) foi redesenhada para uma versão independente de plataforma chamada de BP Diagonal. A BP Diagonal mantém em memória a janela de blocos não podáveis que devem ser computados em cada diagonal.

Uma nova versão chamada BP Genérica também foi criada, para acomodar diferentes níveis de paralelização de cada alinhador dependente de plataforma e suportar diferentes implementações de alinhadores. No BP Genérico o processamento dos blocos inicia-se quando as dependências são resolvidas, não sendo necessário esperar que o processamento da diagonal se complete. Isso dá origem ao padrão *dataflow*. Para implementá-lo, uma matriz mantém o estado podável de cada bloco, para determinar quando podem ser descartados.

De forma similar à estratégia de poda, as células são organizadas em blocos. A regra de dependência para o cálculo das células, presentes nas relações de recorrência do NW (Seção 2.3.1), SW (Seção 2.3.2) e Gotoh (Seção 2.3.3), se mantém entre os blocos. A partir disso, são possíveis duas estratégias de paralelização, diagonal e *dataflow*.

Na paralelização diagonal os cálculos começam no topo esquerdo da matriz e são propagados em diagonal pela matriz. Todos os blocos de uma antidiagonal podem ser calculados paralelamente devido às regras de dependência observadas ao agrupar as células em blocos. A limitação está nos frequentes pontos de sincronização ao final de cada diagonal computada.

Já na paralelização *dataflow* cada bloco de células é um nó. Quando as dependências desse nó são calculadas, ele está pronto para ser calculado. As dependências são resolvidas durante a execução, reduzindo a sincronização existente no método diagonal.

3.4 MASA-CUDAlign-MultiBP

O *framework* MultiBP [1], [24] tem como objetivo permitir executar a poda de blocos (Seção 3.1.3) em execuções com múltiplas GPUs obtendo *score* ótimo, o que não é possível no MASA-CUDAlign 4.0 (Seção 3.2). Para isso, o melhor *score* obtido em cada GPU até o momento é enviado periodicamente para a GPU à direita, com todas organizadas logicamente em um anel. Dessa forma, é criada uma visão global da execução em todas as GPUs, resolvendo a limitação que não permitia a execução do BP. Possui duas estratégias de balanceamento de carga entre as GPUs, dividindo o MultiBP em Static-MultiBP (Seção 3.4.1) e Dynamic-MultiBP (Seção 3.4.2). As duas estratégias podem ser utilizadas tanto em ambientes homogêneos (com GPUs similares), como em ambientes heterogêneos (com GPUs de diferentes capacidades), permitindo uma distribuição não equitativa de colunas entre os dispositivos.

O MultiBP foi integrado à ferramenta MASA-CUDAlign (Seção 3.3) como uma camada adicional, com mínima modificação no código e nas estruturas de dados da ferramenta. Para implementação, foram utilizados os conceitos de BP diagonal e de *buffers* de comunicação das GPUs.

3.4.1 Static-MultiBP

No Static-MultiBP [1], [24], cada GPU calcula um subconjunto de colunas das matrizes de Gotoh (Seção 2.3.3), só sendo necessário receber um bloco de dados da coluna da esquerda, enviada pela GPU à esquerda, e enviar um bloco de dados da coluna da direita para a GPU à direita, após ser calculada. Para comunicação entre as GPUs, são utilizadas três *threads* em CPU. Uma para a entrada, uma para a saída e uma para gerenciamento. Dessa forma, a comunicação entre as GPUs segue o modelo assíncrono presente no CUDAlign 3.0 e 4.0 (Seção 3.2). Cada GPU envia periodicamente o seu melhor *score* corrente para as outras, de forma a aumentar a porcentagem de blocos podados, pois o BP utiliza o melhor *score* corrente global para calcular se um bloco pode ou não ser descartado. Para realizar o compartilhamento dos *scores* entre as GPUs, sem impactar o compartilhamento das colunas pelas *threads* originais do MASA-CUDAlign (Seção 3.3), foram criadas novas *threads*. Cada uma dessas *threads* pertence a uma GPU e se conecta com outra formando uma topologia de anel. Periodicamente, cada *thread* envia o maior *score* local da sua GPU para a *thread* de destino, que compara com seu maior *score* local. Se for maior, o *score* local é atualizado para ser utilizado nas próximas operações do BP. A Figura 3.6 apresenta o compartilhamento de *score*, feito pelas *threads* T_s.

A distribuição de carga entre as GPUs no Static-MultiBP se dá de forma estática no início da execução, de acordo com uma divisão definida pelo usuário. Portanto, no caso de ambientes heterogêneos, é necessário definir uma divisão condizente com o poder de processamento de

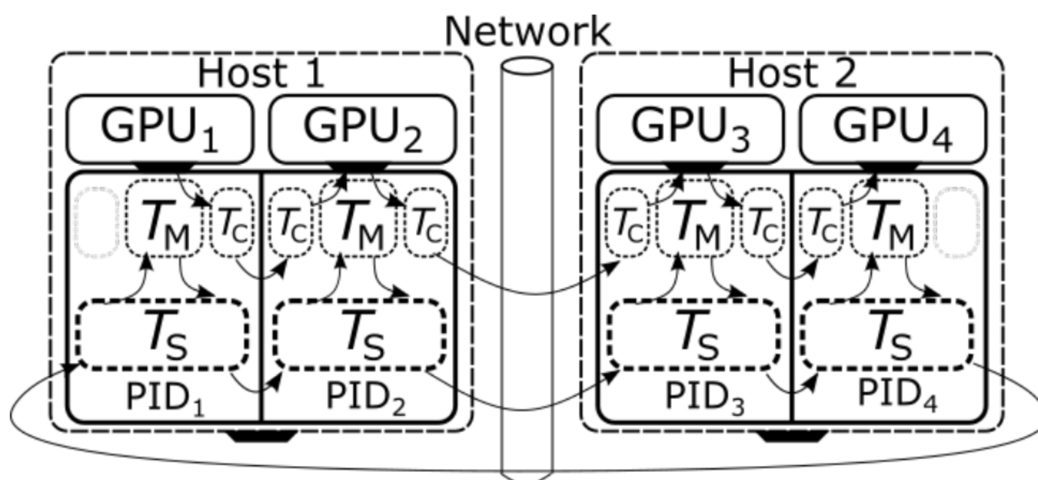

Figura 3.6: Compartilhamento de *score* no MultiBP [1].

cada GPU, evitando que uma ou mais fiquem ociosas aguardando os dados das últimas colunas processadas por outra GPU.

3.4.2 Dynamic-MultiBP

A poda de blocos, quando aplicada em ambientes com múltiplas GPUs utilizando o Static-MultiBP (Seção 3.4.1), pode causar um desbalanceamento na distribuição da carga de trabalho entre essas GPUs. No Static-MultiBP a matriz é dividida entre as GPUs no início da execução utilizando uma divisão definida pelo usuário. À medida que o processamento ocorre, o BP pode causar o descarte de blocos e a redução da carga em uma ou mais GPUs. Quanto mais similares são as sequências alinhadas, maior é a eficácia da poda de blocos e maior pode ser esse desbalanceamento. Uma carga de trabalho desbalanceada acarreta em GPUs ociosas enquanto outras estão sobrecarregadas, impactando negativamente no desempenho da ferramenta. Para evitar esse problema, estratégias de balanceamento dinâmico são empregadas na ferramenta Dynamic-MultiBP [1], [24].

A Figura 3.7 ilustra a estratégia dinâmica. Inicialmente, a distribuição de colunas é realizada de forma similar ao Static-MultiBP (Seção 3.4.1), mas apenas um subconjunto das colunas é distribuído. O usuário define um número de pontos de parada (*breakpoints*), que serão utilizados para dividir a matriz em seções. O cálculo da matriz então é realizado em ciclos, onde, em cada ciclo, uma seção da matriz é calculada. Ao chegar em um ponto de parada, os *buffers* das GPUs são avaliados e, caso necessário, as colunas são redistribuídas antes do próximo ciclo, tentando evitar o enchimento dos *buffers*. Depois da redistribuição, as GPUs se comunicam e a execução continua. Dessa forma, é possível balancear a carga de trabalho conforme a poda de blocos afeta o número de colunas a serem processadas, além de adaptar a carga para GPUs heterogêneas.

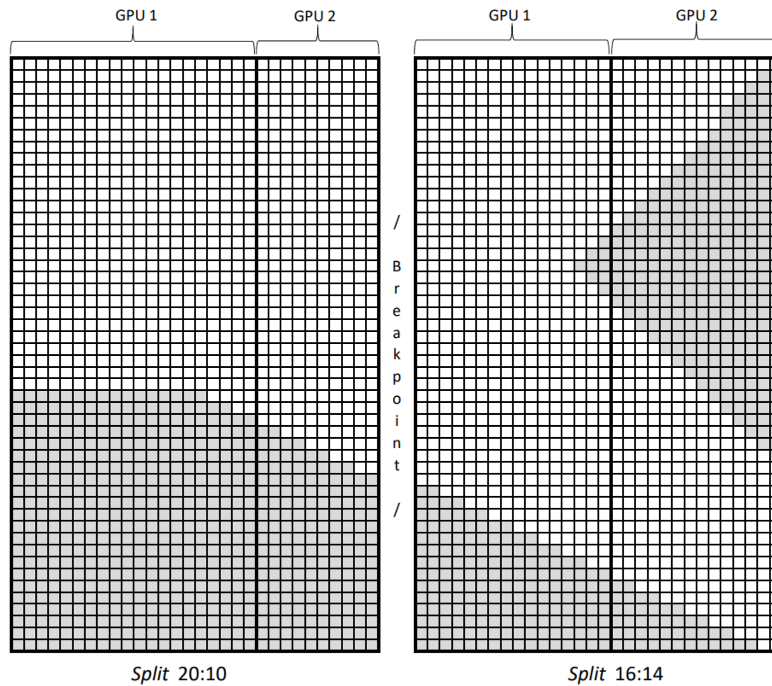

Figura 3.7: Estratégia dinâmica do MultiBP [1].

3.5 Parâmetros que afetam o desempenho do CUDAlign

Para executar a ferramenta CUDAlign é necessário definir parâmetros, além das sequências de entrada que serão alinhadas, que afetam diretamente o desempenho da execução, que são os números de blocos (B) e de *threads* (T) por bloco [8].

Um procedimento executado em GPU se chama *kernel*. Várias instâncias de um *kernel* são executadas por meio de *threads*. Cada *thread* possui um conjunto de registradores e uma memória local. Um bloco é composto por um conjunto de *threads* com acesso a uma memória compartilhada.

O número de blocos diz respeito a quantos blocos de *threads* serão executados concorrentemente na GPU. A GPU é composta por *streaming multiprocessors*, ou SMs, cada um composto por um conjunto de núcleos CUDA, no qual cada núcleo CUDA é uma unidade de execução para cálculos com números inteiros ou ponto flutuante.

No CUDAlign cada bloco tem um tempo de execução praticamente idêntico. Durante a execução de um bloco na arquitetura CUDA, o bloco fica residente no mesmo SM até que sua execução seja finalizada. O ideal é que o número de blocos selecionado seja múltiplo do número de SMs da GPU. Caso uma GPU possua 30 SMs, e supondo que cada SM consegue executar um bloco por vez, se o número de blocos escolhidos para serem executados de cada vez for 30, os SMs estarão sempre ocupados e o desempenho será muito bom. Caso o valor escolhido seja

31, uma execução calcularia os primeiros 30 blocos e o último bloco seria calculado em uma execução separada, deixando 29 SMs sem utilização. Na prática, cada SM é capaz de processar mais de um bloco paralelamente, aproveitando o tempo que um bloco fica parado esperando alguma operação de entrada e saída na memória. Entretanto, existe um limite de quantos blocos cada SM consegue processar de forma paralela [25]. Portanto, caso a GPU possua 30 SMs e o número de blocos escolhido seja 240, cada SM receberá 8 blocos para serem executados por vez, dividindo de forma ótima a carga de trabalho sem deixar SMs inutilizados.

Cada *thread* processa um grupo de células a serem calculadas. O número de *threads* máximo que podem ser executadas em um bloco depende do número de registradores e da quantidade de memória necessários demandados por um *kernel*. Como cada SM possui um número máximo de registradores para dividir entre as *threads*, a quantidade de registradores necessários para um *kernel* limita a quantidade de *threads* que podem ser utilizadas. Da mesma forma, a quantidade de *threads* limita quantos blocos podem ser executados paralelamente em um SM. Por exemplo, caso um bloco possua 128 *threads* e um *kernel* utilize 32 registradores, são necessários 32 X 128 = 4096 registradores para processar esse bloco. Caso um SM possua 8192 registradores, é possível processar 2 blocos por vez nesse SM [8].

Dessa forma, os valores escolhidos para os parâmetros de blocos e *threads* dependem um do outro. Portanto, eles são tratados como um par de parâmetros e não como dois parâmetros isolados.

Em todas as versões do CUDAlign, a determinação do número de blocos (B) e *threads* (T) é feita inteiramente pelo usuário. Caso nenhum valor seja informado, o CUDAlign utiliza o valor padrão $T = 128$ e calcula o valor de B com base no tamanho das sequências e no valor de T.

Capítulo 4

Projeto da Ferramenta para Execução Otimizada e Automatizada do CUDAlign

Este capítulo apresenta informações acerca do desenvolvimento do projeto da ferramenta proposta neste trabalho, cujo objetivo é auxiliar o usuário a determinar os melhores valores dos parâmetros B e T (Seção 3.5) para a execução do MASA-CUDAlign em sua máquina, assim como executar de maneira automática lotes de vários alinhamentos.

Como descrito na Seção 3.3, o CUDAlign passou a se chamar MASA-CUDAlign em 2016. Portanto, os nomes MASA-CUDAlign e CUDAlign nesse capítulo serão utilizados fazendo referência a mesma ferramenta.

4.1 Motivação

Como descrito na Seção 3.5, a escolha dos parâmetros de blocos e *threads* é de grande importância para a execução otimizada da ferramenta MASA-CUDAlign e pode afetar drasticamente o desempenho da mesma. Além disso, os parâmetros são muito dependentes da arquitetura da GPU utilizada para a execução, e devem ser determinados levando isso em consideração. Apesar de as recomendações da Seção 3.5 auxiliarem na escolha dos parâmetros, a análise empírica se mostra importante para refinar essa escolha, como mostrado por Sandes [8].

Uma maneira empírica de verificar os melhores parâmetros para a execução do CUDAlign em uma determinada GPU é construir um perfil (*profile*) [26]. *Profiling* consiste em executar vários alinhamentos com sequências de tamanhos variados, e, para cada tamanho de sequência, variar os números de blocos e *threads* e coletar os resultados de desempenho, criando um perfil de desempenho. Assim, é possível encontrar na prática parâmetros adequados de entrada para cada tamanho de sequência utilizado, para executar o CUDAlign em uma determinada GPU.

Considerando o tempo gasto para executar múltiplos alinhamentos de sequências e a organização necessária para coletar e comparar os resultados, realizar esse *profiling* é uma tarefa

bastante custosa. Assim sendo, o investimento inicial necessário pode parecer não valer a pena para o usuário.

Dessa forma, a ferramenta proposta no presente trabalho de graduação possui dois objetivos. Em primeiro lugar, a ferramenta constrói o perfil de desempenho, automatizando sua execução, coleta e processamento dos resultados, oferecendo como saída o número mais adequado de blocos e *threads* para várias comparações. Com isso o usuário pode utilizar o CUDAlign otimizado especificamente para sua GPU. Em segundo lugar, a ferramenta configura e executa lotes de alinhamentos de sequências de forma automatizada, deixando os resultados de todos os alinhamentos acessíveis ao usuário.

4.2 Visão Geral

A ferramenta foi implementada como uma interface entre o usuário e o MASA-CUDAlign, de forma a simplificar e auxiliar o usuário no uso do *software* de alinhamentos. Como apresentado na Figura 4.1, a ferramenta é composta por 4 módulos. O módulo **Gerência (A)** gerencia a execução da ferramenta e provê a interface com o usuário; o módulo **Perfil (B)** gerencia o processo de construir um perfil (*profile*); o módulo **Execução (C)** realiza a interface com o MASA-CUDAlign e executa alinhamentos; e finalmente o módulo **Resultado (D)** busca as estatísticas de resultado quando um alinhamento de *profiling* finaliza, limpa arquivos temporários e move os resultados de um alinhamento para uma pasta acessível ao usuário.

Além disso, também são utilizados 3 arquivos de texto para armazenamento de configurações: o arquivo **configurações_de_alinhamentos** é editado pelo usuário para configurar um lote de alinhamentos a ser realizado, o arquivo **melhores_parâmetros** armazena os melhores parâmetros para alinhar sequências de determinado tamanho, separados por modelo de GPU, obtidos em um *profiling*, e o arquivo **lista_de_execuções** armazena as execuções sendo realizadas, assim como seu estado (esperando ou executada), permitindo retomar a execução de um lote de alinhamentos caso a ferramenta seja fechada antes de finalizar.

Na Figura 4.1, os módulos em azul são os módulos implementados nesse projeto, enquanto os módulos em cinza são arquivos, podendo ser de resultados ou de configuração.

Inicialmente, o módulo **Gerência (A)** carrega os melhores parâmetros já obtidos pela ferramenta anteriormente, se houver (1). Após isso, exibe a interface principal para o usuário (2) e, se for escolhida a opção de *profiling*, a ferramenta executa o módulo **Perfil (B)** (3). Esse módulo então carrega uma lista de alinhamentos pré-configurados e executa o módulo **Execução (B)** (4), que executa cada um desses alinhamentos no CUDAlign (5). O módulo **Resultados (C)** recupera apenas as estatísticas de cada alinhamento (6). Ao terminar, o módulo **Perfil (B)** salva os parâmetros de desempenho no arquivo **melhores_parâmetros** (7). Essa sequência de passos pode ser visualizada na Figura 4.2.

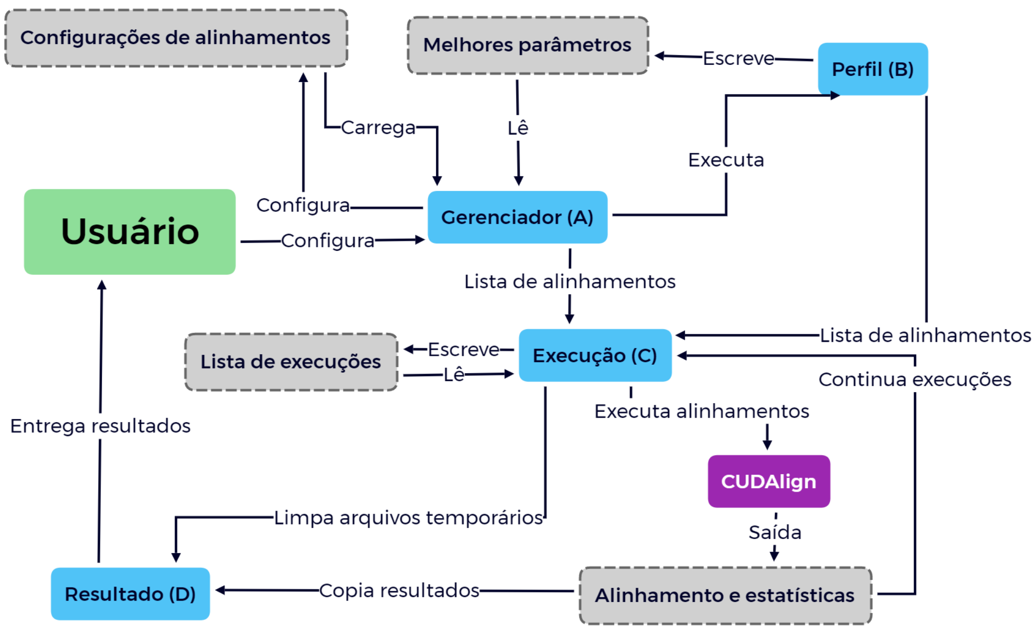

Figura 4.1: Visão geral do projeto.

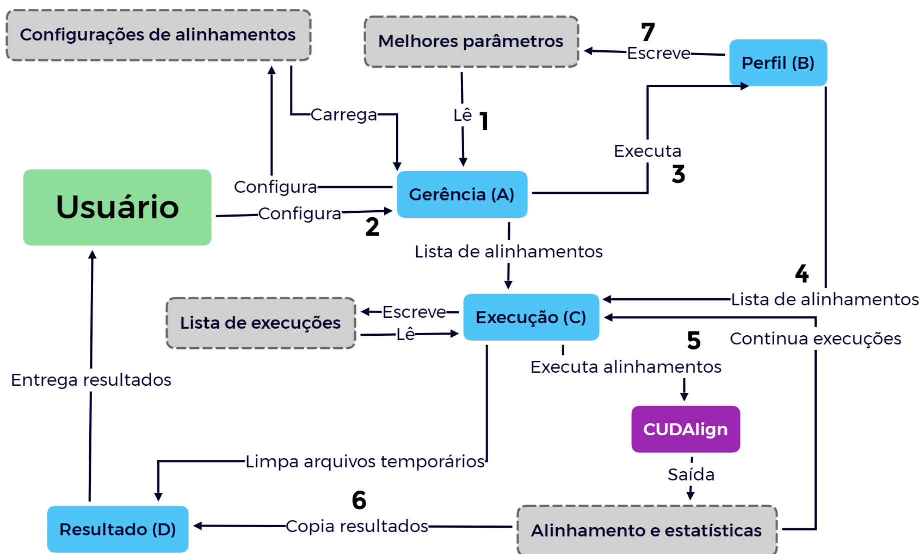

Figura 4.2: Sequência de execução de um *profiling*.

Novamente na interface principal, caso o *profiling* já tenha sido executado para a GPU em questão, o módulo **Gerência (A)** recupera os melhores parâmetros previamente calculados (1). Se o usuário escolher a opção de realizar alinhamentos, o módulo **Gerência (A)** carrega o lote

de alinhamentos do arquivo **configurações_de_alinhamentos** (2), que deve ser previamente fornecido pelo usuário. Então, o módulo **Execução (C)** (3), de maneira automatizada, executa o CUDAlign para cada par de sequências constantes no arquivo (4). Ao término de cada alinhamento, o CUDAlign escreve o alinhamento e as estatísticas em diretório temporário (5). O módulo de **Resultados (D)** entrega esses arquivos (alinhamento e estatísticas) para o usuário (6 e 7), à medida que cada alinhamento é finalizado, e limpa o diretório temporário do CUDAlign (8). Ao finalizar o último alinhamento do lote, a execução é terminada. Essa sequência de passos pode ser visualizada na Figura 4.3.

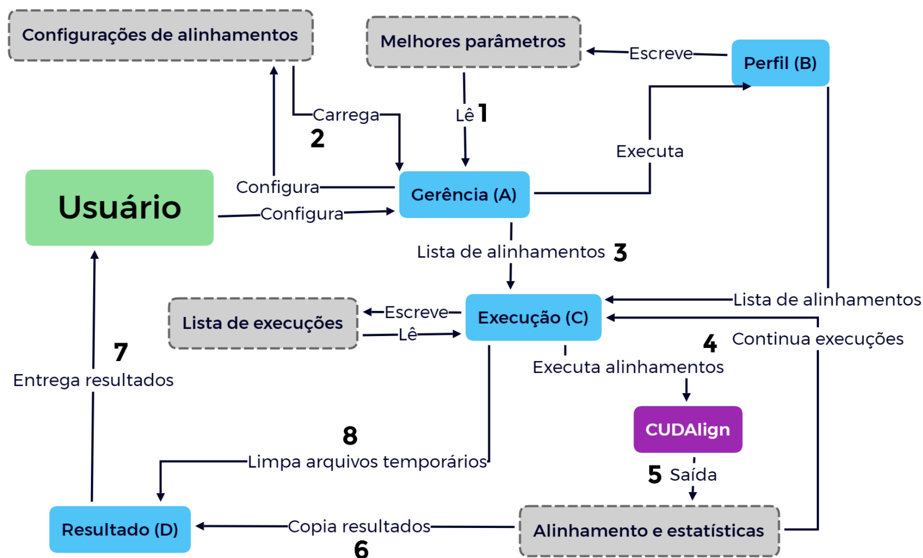

Figura 4.3: Sequência de execução de um lote de alinhamentos.

4.3 Módulo Gerência (A)

O módulo **Gerência** (Figura 4.1 (A)) é o módulo com o qual o usuário possui mais contato. É responsável por apresentar o menu e obter as escolhas do usuário. É possível escolher entre realizar alinhamentos (*Alignments*) ou realizar um *profile* (*Profiling*), conforme mostrado na Figura 4.4. Também é possível habilitar ou desabilitar o modo verboso, que afeta a quantidade de informações apresentadas pelo CUDAlign durante as suas execuções. O **Gerência** também é responsável por buscar informações sobre a GPU atual da máquina do usuário e acessar o arquivo **melhores_parâmetros** para carregar os melhores parâmetros, caso existam, para a GPU presente na máquina.

```
CUDAlign Profiler

1 - Alignments
2 - Profiling
3 - Enable Verbose Mode (More information during the executions) (DISABLED)
4 - Quit
```

Figura 4.4: Tela inicial do gerenciador.

Para carregar os melhores parâmetros, o módulo lê o arquivo **melhores_parâmetros** em busca da seção referente a GPU atual da máquina e, caso encontre, carrega os parâmetros previamente obtidos pelo *profiling* para a memória. A estrutura do arquivo de parâmetros é mostrada na Figura 4.5. Nessa figura está ilustrado que já foi executado o *profiling* para sequências de tamanho 1M, tendo sido obtidos $T = 128$ e $B = 256$ como melhores valores para a GPU NVIDIA GeForce RTX 2060.

```
#NVIDIA GeForce RTX 2060 # simple:1 complete:0
--size:1 threads:128 blocks:256
```

Figura 4.5: Estrutura do arquivo de parâmetros.

Caso o usuário escolha realizar alinhamentos, orientações sobre como configurar esses alinhamentos utilizando um arquivo de texto são apresentadas (Figura 4.6). O caminho e o nome dos arquivos das sequências a serem alinhadas em lote devem ser configurados nesse arquivo seguindo o formato apresentado na Figura 4.7 (duas sequências a serem alinhadas por linha do arquivo, separadas por um espaço em branco). Após inserir as configurações, o módulo **Gerência** carrega o arquivo editado pelo usuário e cria uma execução para cada alinhamento a ser realizado, formando uma lista de execuções. A lista é exibida ao usuário e, após sua confirmação, o módulo de **Execução** (Seção 4.5) é chamado para continuar o procedimento e realizar os alinhamentos.

Caso o usuário escolha realizar um *profiling*, um outro menu é exibido, contendo uma descrição do que é um perfil de desempenho e dos benefícios de realizar um (Figura 4.8). É possível escolher entre construir um perfil completo e um perfil simplificado. A diferença está na quantidade e na variedade de tamanhos de sequências utilizados. Também é exibido se algum dos perfis, simples ou completo, já foi realizado para essa GPU anteriormente. Caso já tenha sido, não há necessidade de realizar novamente visto que os melhores parâmetros para essa GPU já foram encontrados. A escolha do usuário nesse menu é passada para o módulo **Perfil** (Seção 4.4).

```
1 - Read from file
2 - Back
1

Setup all the alignments using the file 'alignment_setup.txt' in the project direc
tory.
On each line of the file, input the two sequences to be aligned, using the followi
ng format:

PATH_TO_SEQUENCE_A PATH_TO_SEQUENCE_B

Example:

.../seq/BA000035.2.fasta .../seq/BX927147.1.fasta

The sequences must be in the fasta format.
After running, the results of each alignment will be available in a folder inside
the ./results/ directory.
If the file is ready, press any key to continue...
```

Figura 4.6: Tela do módulo Gerência na opção de alinhamentos.

```
./seq/1M/AE002160.2.fasta ./seq/1M/CP000051.1.fasta
./seq/3M/BA000035.2.fasta ./seq/3M/BX927147.1.fasta
./seq/5M/AE016879.1.fasta ./seq/5M/AE017225.1.fasta
```

Figura 4.7: Arquivo de configuração do lote de alinhamentos.

```
───── Profiling Information ─────

    A profile is useful to obtain adequate parameters for your GPU, to ensure
that the alignments will be executed efficiently in your machine.
    A complete profile will run several executions with different sizes of seq
uences. It will give the best results, but it will take hours to run. It only
has to be executed once, unless the gpu in the machine is changed.
    A simple profile will run just a few executions and with small sequences.
The results will not be as good as a complete profile, but it should take only
 a few minutes to run.
--- It is recommended to run at least the simplified profiling once. ---

      Profiling Status

  Complete [ - NOT DONE]
     Simple [ - DONE]

1 - Complete profiling
2 - Simplified profiling
3 - Back
```

Figura 4.8: Tela do *profiling*.

4.4 Módulo Perfil (B)

O módulo de **Perfil** (Figura 4.1 (B)) é responsável por gerenciar e realizar *profiles* da GPU atual da máquina. A sua função é organizar e gerenciar toda a operação de um *profiling* de maneira

Tabela 4.1: Pares de parâmetros utilizados em um *profile*.

Threads	**Blocos**
128	256 480 512
256	256 480 512

automatizada. Primeiro, cria uma lista de execuções combinando as sequências e pares de parâmetros que devem ser testados, com base no tipo de perfil que será construído (simplificado ou completo). Então, chama o módulo de **Execução** (Seção 4.5) para realizar os alinhamentos, cujas estatísticas de desempenho serão colhidas pelo módulo de **Resultado** (Seção 4.6). Após isso, analisa os resultados para determinar quais foram os melhores parâmetros para cada tamanho de sequência testado, e grava esses valores no arquivo **melhores_parâmetros**.

Para gravar os valores de melhores parâmetros, o módulo lê o arquivo **melhores_parâmetros** em busca da seção referente à GPU atual, depois encontra se já existe uma linha referente ao tamanho de sequência cujos melhores parâmetros estão sendo gravados. Então armazena todas as linhas seguintes em um buffer temporário, substitui ou adiciona os novos melhores parâmetros e escreve de volta do buffer para o arquivo para evitar que alguma informação seja perdida. A estrutura do arquivo **melhores_parâmetros** é mostrada na Figura 4.5.

A análise dos resultados consiste em agrupar todas as execuções conforme o tamanho de sequência utilizado em cada uma e analisar os tempos de execução dentro de cada grupo. Dessa forma, é possível encontrar qual execução se mostrou a mais rápida para cada tamanho de sequência, assim como quais foram os parâmetros utilizados que geraram esse melhor desempenho.

Como mencionado anteriormente, o módulo pode construir dois tipos de perfis diferentes, o simplificado e o completo. Nos dois casos, são testados 6 pares de parâmetros, escolhidos conforme o apresentado na seção 3.5 e em testes realizados com uma GPU NVIDIA GeForce RTX 2060. Os pares de parâmetros podem ser vistos na Tabela 4.1.

4.4.1 Perfil completo

O perfil completo apresenta os melhores resultados devido à sua grande variedade e quantidade de alinhamentos realizados. Por esse mesmo motivo, é também o mais demorado. O tempo de execução é mostrado na Seção 5.4.

A lista de alinhamentos que são realizados nesse *profile* é composta por pares de sequências de 6 tamanhos diferentes, e cada par é alinhado utilizando 6 combinações de parâmetros

Tabela 4.2: Pares de sequências utilizadas em um perfil completo.

Tamanho das sequências	Identificadores das sequências
1M x 1M (a)	AE002160.2 CP000051.1
3M x 3M (a)	BA000035.2 BX927147.1
5M x 5M (a)	AE016879.1 AE017225.1
10M x 10M (a)	NC_014318.1 NC_017186.1
28M x 23M (a)	NT_037436.4 NT_033779.5
32M x 47M (a)	BA000046.3 NC_000021.9

Tabela 4.3: Pares de sequências utilizadas em um perfil simplificado.

Tamanho das sequências	Identificadores das sequências
1M x 1M (a)	AE002160.2 CP000051.1

diferentes, como apresentado na Tabela 4.1. Os pares de sequências podem ser visualizados na Tabela 4.2. O tamanho das sequências escolhidas é da ordem de milhões de pares de bases, ou seja, 1M equivale a 1 mega ou 1 milhão de caracteres. O identificador de uma sequência pode ser utilizado para encontrá-la no site do *National Center for Biotechnology Information* (NCBI) [27], que é um repositório internacional e público de sequências biológicas. Nesta tabela, as comparações estão marcadas com *(a)* para diferenciá-las das comparações usadas nas execuções reais (Capítulo 5).

4.4.2 Perfil simplificado

O perfil simplificado tem como objetivo obter valores apropriados de parâmetros para a GPU atual utilizando apenas um par de sequências de 1M (milhão de caracteres). Os resultados não são tão completos quanto os do *profile* completo, mas fornecem uma base sobre quais parâmetros utilizar com a GPU atual. Em compensação, seu tempo de execução é menor e mais adequado para máquinas com menor poder de processamento. O tempo de execução é mostrado na Seção 5.4.

As combinações de parâmetros são as mesmas do *profile* completo, mudando apenas a quantidade de pares de sequências utilizados. Apenas o par de sequências de 1M é alinhado, como pode ser visto na Tabela 4.3.

4.5 Módulo Execução (C)

O módulo de **Execução** (Figura 4.1 (C)) é responsável por executar a lista de alinhamentos que já está carregada em memória, sendo ela composta por alinhamentos do usuário ou por alinhamentos para *profiling*.

Ao iniciar, o módulo verifica se todos os arquivos das sequências configuradas em cada alinhamento estão presentes nos diretórios indicados. Se algum não estiver, indica o erro para o usuário e o programa é finalizado. O mesmo é feito para verificar se os arquivos executáveis do CUDAlign estão presentes. Tanto os arquivos do CUDAlign quanto os de sequências utilizadas para *profiling* são fornecidos junto com a ferramenta. No caso destes arquivos, a verificação tem a utilidade de confirmar que não foram removidos de seus diretórios erroneamente. No caso de arquivos de sequências providos pelo usuário, a verificação tem como utilidade encontrar algum erro de digitação ou mesmo a falta de um arquivo que o usuário acreditava estar presente, para que o erro não ocorra durante a execução de um lote de alinhamentos.

Durante a verificação de existência dos arquivos, o tamanho dos arquivos das sequências fornecidas pelo usuário é analisado. Para cada alinhamento, uma média entre o tamanho das duas sequências é tomado como o tamanho do alinhamento, para tomar a decisão sobre quais parâmetros utilizar no momento da execução.

Após realizar essa verificação, a lista de execuções é percorrida um elemento por vez. Em cada um, o tamanho registrado para o alinhamento é utilizado para buscar os melhores parâmetros para sequências desse tamanho. No caso de uma execução de *profiling*, os parâmetros já são pré-definidos pelo módulo de **Perfil** (Seção 4.4) e são mantidos. No caso de uma execução do usuário, os melhores parâmetros são buscados na tabela em memória, que foi carregada pelo módulo **Gerência** (Seção 4.3). Caso não exista um par de melhores parâmetros para o tamanho específico, o tamanho mais próximo é utilizado. Caso não exista nenhum parâmetro mais próximo, como é o caso que ocorre quando um alinhamento é realizado pelo usuário sem nunca ter realizado um *profile*, os valores padrões de 480 blocos e 256 *threads* são utilizados. Esses foram os melhores valores, em média, encontrados em uma análise empírica utilizando a GPU NVIDIA GeForce RTX 2060, conforme será mostrado no Capítulo 5.

Após determinar os melhores parâmetros para a comparação, uma linha de execução do CUDAlign é montada levando esses parâmetros em consideração, assim como as sequências que serão utilizadas e se o modo verboso está ou não habilitado. Caso esteja desabilitado, a impressão do CUDAlign é redirecionada para */dev/null*. Caso esteja habilitado, é impressa normalmente em tela. Essa diferença pode ser visualizada na Figura 4.10.

Um outro parâmetro é passado para a execução do CUDAlign, chamado *ram size*, referente ao espaço em memória RAM que o CUDAlign pode utilizar para armazenar as linhas especiais durante a execução. Para evitar problemas ocorridos ao deixar esse parâmetro com valores automáticos, é utilizado o tamanho de 5G em toda chamada ao CUDAlign. Caso necessário,

esse valor pode ser alterado pelo usuário antes da compilação da nossa ferramenta. Assim, com o comando pronto, é realizada uma chamada a função *linux system* para que a linha de comando relativa ao CUDAlign seja executada em um terminal, e a ferramenta fica esperando que essa execução finalize. Um exemplo do comando utilizado é mostrado na Figura 4.9.

```
./cudalign/128_threads/cudalign --ram-size=5G
    ./seq/1M/AE002160.2.fasta ./seq/1M/CP000051.1.fasta --blocks=480
```

Figura 4.9: Linha de comando para execução do CUDAlign com os parâmetros $B = 480$ e $T = 128$.

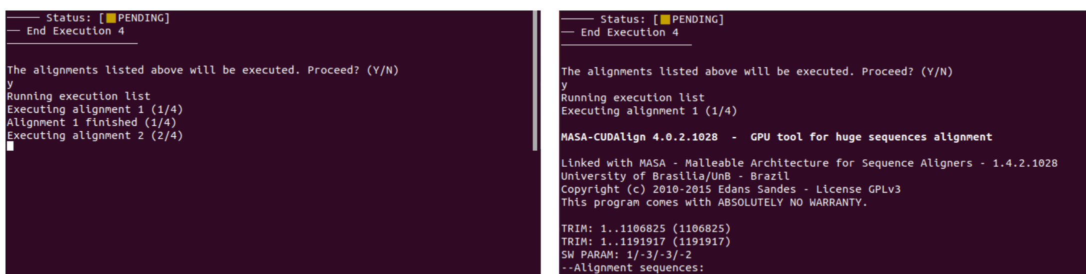

Figura 4.10: Comparação entre execuções com modo verboso desligado (esquerda) e ligado (direita).

Ao finalizar uma execução, o módulo de **Resultado** (Seção 4.6) é chamado, mas sua função é diferente no caso de um alinhamento de usuário e de um alinhamento de *profiling*. Essa diferença e sua função são melhor descritas na Seção 4.6. Após a execução do módulo de **Resultado**, no caso de um alinhamento do usuário, é apresentada uma mensagem indicando que os arquivos resultantes do alinhamento estão disponíveis na pasta de resultados. No caso de um alinhamento de *profiling*, as estatísticas de desempenho dessa execução já foram coletadas pelo módulo de **Resultado** para posterior análise e ficam armazenadas na lista de execuções em memória e no arquivo **lista_de_execuções**. Após isso, o próximo alinhamento é executado. Caso não haja mais nenhum, o módulo de **Execução** encerra seu funcionamento.

4.6 Módulo Resultado (D)

O módulo de **Resultado** (Figura 4.1 (D)) tem a função de finalizar uma execução após o término da operação do CUDAlign. A sua funcionalidade é diferente no caso de um alinhamento do usuário ou de um alinhamento de *profile*, como foi citado na seção 4.5.

Caso seja uma execução do usuário, o módulo de **Resultado** é responsável por copiar os arquivos de resultado do CUDAlign, contendo o alinhamento e as estatísticas geradas, para uma pasta acessível para o usuário dentro do diretório */results*. Para cada alinhamento, é criada

uma pasta com um nome contendo a data em que foi realizado, o horário e os nomes das duas sequências, para evitar que haja algum conflito entre os diretórios. Após copiar esses arquivos, o diretório temporário do CUDAlign é excluído, para evitar que hajam problemas entre execuções com diferentes tamanhos de sequência.

Caso seja uma execução de *profile*, a função do módulo é coletar as estatísticas de desempenho de uma execução para posterior análise, como mencionado na Seção 4.5. O módulo é responsável por acessar o arquivo de estatísticas gerado pelo CUDAlign ao final de um alinhamento, encontrar os dados referentes ao tempo gasto (em milisegundos) e ao número de células processadas por segundo (em MCups), e registrar os dados em sua execução referente na lista de execuções em memória. Dessa forma, o módulo de **Perfil** (Seção 4.4) pode verificar quais foram os melhores parâmetros assim que todos os resultados forem colhidos. Após registrar as estatísticas, o diretório temporário do CUDAlign é excluído para evitar problemas com o próximo alinhamento, assim como na execução de um alinhamento do usuário.

4.7 Implementação

A ferramenta proposta no presente trabalho de graduação foi implementada em C e utilizou a chamada *system* para a execução do MASA-CUDAlign. Adicionalmente, foi também utilizado o comando *nvidia-smi* para obter o modelo da GPU sendo utilizada. Além disso, os executáveis do MASA-CUDAlign são fornecidos já compilados para facilitar a utilização pelo usuário.

Capítulo 5

Resultados Experimentais

Neste capítulo são apresentados os resultados obtidos com a ferramenta de execução automatizada e com *profiling*, proposta no presente trabalho de graduação.

5.1 Sequências Utilizadas

Nas execuções foram usados dois conjuntos de comparações: (a) comparações para realizar o *profiling* e (b) comparações que utilizam os resultados do *profiling*. As sequências utlizadas para o *profiling* com a ferramenta são um subconjunto das sequências utilizadas por Figueiredo [1] e podem ser visualizadas na Tabela 4.2. Para testar a execução em lote da ferramenta e se os parâmetros escolhidos pela mesma são satisfatórios, foram utilizadas outras sequências, que podem ser visualizadas na Tabela 5.1, onde são apresentados os tamanhos das sequências e os identificadores para recuperação no banco público NCBI [27]. As sequências de 1M e 3M foram selecionadas por possuírem tamanhos similares às sequências utilizadas no *profiling*, enquanto que as sequências de 4M, 6M e 8M foram selecionadas por possuírem tamanhos diferentes dos utilizados no *profiling*, para avaliar o comportamento da ferramenta em ambos os casos.

5.2 Configuração do Ambiente

O ambiente utilizado para a execução da ferramenta foi uma máquina instalada no Laboratório de Sistemas Integrados e Concorrentes (LAICO), presente na Universidade de Brasília. A máquina utilizada possui 1 GPU NVIDIA GeForce RTX 2060 (arquitetura Turing, 1920 núcleos CUDA, 1,36 GHz com turbo máximo de 1,68 GHz, 6GB de memória a 14Gbps), 1 CPU Intel i7 9700 (8 núcleos, 3 GHz) e 16 GB de memória RAM, e utiliza o sistema operacional Ubuntu 20.04 LTS.

Tabela 5.1: Pares de sequências utilizadas para avaliar os parâmetros escolhidos.

Tamanho das sequências	**Identificadores das sequências**
1M x 1M (b)	JANKHN010000019.1 JANKHN010000507.1
3M x 3M (b)	CP100650.1 NZ_JANUDR010000003.1
4M x 4M (b)	CP103782.1 CP103784.1
6M x 6M (b)	NZ_JANULM010000001.1 NZ_JANUMG010000001.1
8M x 8M (b)	NW_026095751.1 NZ_NWMZ01000001.1

```
cd masa-cudalign
./configure --with-cuda=[PATH_CUDA] --with-nvcc=[PATH_NVCC]
    --with-threads=[THREADS]
make
```

Figura 5.1: Comandos utilizados para compilação do MASA CUDAlign.

5.3 Configuração da Ferramenta

A ferramenta foi desenvolvida utilizando a linguagem C e possui os executáveis do MASA-CUDAlign já compilados para facilitar a utilização pelo usuário. Para obter os executáveis do CUDAlign, foi necessário compilar o código encontrado no repositório original [28]. A versão utilizada foi o MASA-CUDAlign 4.0.

Como citado na Seção 4.1, para executar o *profiling* e obter os melhores parâmetros para a execução da ferramenta na máquina do usuário, é necessário variar os parâmetros B e T de entrada. O parâmetro B (número de blocos) é informado no momento da execução do alinhamento, mas o parâmetro T (número de *threads*) deve ser informado no momento da compilação do CUDAlign. Portanto, foi necessário compilar e gerar dois executáveis do CUDAlign, um com o parâmetro T configurado para 128 *threads* e outro configurado para 256 *threads*, que são os dois valores utilizados no *profiling*, como mostrado na Figura 4.1. Para compilar as duas instâncias do CUDAlign, foram utilizados os comandos apresentados na Figura 5.1. Nesta figura, o parâmetro *[PATH_CUDA]* deve ser substituído pelo diretório onde o arquivo *libcudart.so* se encontra, *[PATH_NVCC]* deve ser substituído pelo diretório onde o compilador *nvcc* se encontra e *[THREADS]* deve ser substituído pelo número de *threads* desejado. Como mencionado, esses executáveis já são fornecidos com a ferramenta desenvolvida neste trabalho de graduação, e só precisam ser compilados novamente se houver alguma incompatibilidade com a máquina do usuário.

```
cd cudalign_manager
make
./manager
```

Figura 5.2: Comandos utilizados para compilação da ferramenta proposta neste trabalho.

Tabela 5.2: Melhores, piores e parâmetros médios para sequências de 1M x 1M (a).

Resultado	**Threads**	**Blocos**	**Tempo (ms)**	**MCups**
Melhor	128	480	10050	111500
Pior	64	64	20768	53958
Médio	64	128	14798	75725

Para utilizar a nossa ferramenta, é necessário obter os arquivos de implementação da mesma, que estão disponíveis em um repositório no GitHub [29]. Para compilar e executar a ferramenta desenvolvida é necessário executar os comandos mostrados na Figura 5.2. Após iniciar a ferramenta, a interação do usuário se dá pela interface provida pelo programa, como mostrado na Seção 4.3.

5.4 Execução do *Profiling*

Como apresentado na Seção 4.4, algumas combinações de valores de parâmetros B e T foram escolhidas para serem utilizadas no *profiling* automatizado da ferramenta. Essas combinações foram selecionadas com base em resultados obtidos de um *profiling* manual do CUDAlign, com uma variação maior desses valores. Para obter esses resultados, um *script bash* foi utilizado para executar o CUDAlign múltiplas vezes e coletar os dados para posterior análise. As sequências utilizadas nessas execuções foram as mesmas que as utilizadas na etapa de *profiling* da ferramenta (Tabela 4.2).

As Tabelas 5.2 a 5.7 apresentam o tempo em milissegundos e os MCups obtidos no melhor e no pior alinhamento, assim como no alinhamento médio, e os parâmetros utilizados para obter esses resultados para cada tamanho de sequência utilizado, obtidos na máquina descrita na Seção 5.2.

Tabela 5.3: Melhores, piores e parâmetros médios para sequências de 3M x 3M (a).

Resultado	**Threads**	**Blocos**	**Tempo (ms)**	**MCups**
Melhor	256	480	59732	172953
Pior	64	128	111934	92294
Médio	64	256	81333	127020

Tabela 5.4: Melhores, piores e parâmetros médios para sequências de 5M x 5M (a).

Resultado	Threads	Blocos	Tempo (ms)	MCups
Melhor	256	512	116426	234755
Pior	64	128	246139	111041
Médio	64	384	155540	175720

Tabela 5.5: Melhores, piores e parâmetros médios para sequências de 10M x 10M (a).

Resultado	Threads	Blocos	Tempo (ms)	MCups
Melhor	256	512	374703	279663
Pior	64	128	873136	120016
Médio	64	384	528965	198105

Tabela 5.6: Melhores, piores e parâmetros médios para sequências de 28M x 23M (a).

Resultado	Threads	Blocos	Tempo (ms)	MCups
Melhor	256	480	3542928	186562
Pior	64	128	6946021	95158
Médio	64	256	5008512	131970

Tabela 5.7: Melhores, piores e parâmetros médios para sequências de 32M x 47M (a).

Resultado	Threads	Blocos	Tempo (ms)	MCups
Melhor	256	512	5285870	289837
Pior	64	128	12089291	126727
Médio	64	432	7266942	210821

Tabela 5.8: Melhores valores de parâmetros encontrados pela ferramenta.

Tamanho das sequências	**Threads**	**Blocos**
1M x 1M (a)	128	480
3M x 3M (a)	256	480
5M x 5M (a)	256	512
10M x 10M (a)	256	512
28M x 23M (a)	256	480
32M x 47M (a)	256	512

Como pode ser visto, dependendo dos valores dos parâmetros B e T, existe uma variação de cerca de 100% no tempo de execução. Por exemplo, conforme a Tabela 5.2, na comparação 1M x 1M (a), o tempo de execução é 10 segundos ($B = 480$, $T = 128$) e 20,7 segundos ($B = 64$, $T = 64$) e, na comparação 32M x 47M (a), o tempo de execução é reduzido de 3,35 horas ($B = 128$, $T = 64$) para 1,46 horas ($B = 512$, $T = 256$), conforme ilustrado na Tabela 5.7. Com base nesse *profile*, escolheu-se os pares de valores para os parâmetros que foram utilizados no *profiling* realizado pela ferramenta. Os melhores valores de parâmetros para a execução do CUDAlign em uma GPU NVIDIA GeForce RTX 2060 são mostrados na Tabela 5.8.

A execução do *profiling* simplificado foi rápida, pois se utiliza somente a comparação 1M x 1M (a), conforme a Tabela 4.3. Na máquina do ambiente de testes LAICO, a execução demorou aproximadamente 61 segundos para construir o *profile*, permitindo o uso da ferramenta com os parâmetros selecionados após esse tempo. Já para o *profiling* completo, foi necessário um tempo considerável. A construção do *profile* demorou aproximadamente 61115 segundos, ou seja, 16 horas, 58 minutos e 36 segundos, utilizando uma GPU NVIDIA GeForce RTX 2060. Durante esse tempo, a ferramenta executou todos os alinhamentos configurados para o *profiling* completo, como explicado na Seção 4.4, e construiu o *profile* de forma automatizada. O *profiling* só é executado uma vez para cada GPU e, após esse tempo, a ferramenta pode ser utilizada para executar os alinhamentos desejados pelo usuário de forma otimizada.

5.5 Execuções com os Resultados do *Profiling*

De forma a avaliar o impacto do *profiling* em alinhamentos do usuário, a ferramenta desenvolvida foi utilizada para executar alinhamentos com sequências diferentes das utilizadas pelo *profiling*, apresentadas na Tabela 5.1 e marcadas com *(b)*, utilizando como parâmetros B e T os mesmos pares de valores utilizados pelo *profiler*. Os tempos em milissegundos e os MCups dessas execuções são apresentados nas Tabelas 5.9 a 5.13. As linhas em negrito correspondem à melhor combinação de parâmetros previamente escolhida pela nossa ferramenta.

Como pode ser visto na Tabela 5.9, a nossa ferramenta escolheu os valores $B = 480$ e $T = 128$ para as comparações 1M x 1M (b) (Tabela 5.1), obtendo tempo de execução 8,9 segundos.

Tabela 5.9: Desempenho de alinhamentos de 1M x 1M (b).

Threads	**Blocos**	**Tempo (ms)**	**MCups**
128	256	9277	142201
128	**480**	**8960**	**147228**
128	512	9202	143353
256	256	8638	152723
256	480	8710	151459
256	512	8909	148077

Tabela 5.10: Desempenho de alinhamentos de 3M x 3M (b).

Threads	**Blocos**	**Tempo (ms)**	**MCups**
128	256	81098	154402
128	480	77458	161657
128	512	79844	156828
256	256	77043	162529
256	**480**	**71651**	**174759**
256	512	76074	164599

Ao se comparar com o melhor tempo (8,6 segundos) e o pior (9,27 segundos), verificamos que a nossa ferramenta conseguiu um tempo próximo da média, se considerados todos os tempos da Tabela 5.9.

Nas comparações 3M x 3M (b) (Tabela 5.10), a ferramenta escolheu os parâmetros $B = 480$, $T = 256$, e obteve o melhor tempo de execução (71,6 segundos), com uma redução de quase 10 segundos em relação à pior escolha. Nas comparações 4M x 4M (b) (Tabela 5.11) os melhores valores de B e T obtidos pela ferramenta também obtiveram o menor tempo de execução (93,1 segundos), com uma redução de mais de 10 segundos em relação à pior escolha. A comparação de 6M x 6M (b) também apresenta o menor tempo de execução com os valores escolhidos pela ferramenta (Tabela 5.12), com redução de cerca de 30 segundos em relação à pior escolha. Finalmente, para as comparações 8M x 8M (b), a ferramenta apresentou tempos de execução próximos à média, se considerados o melhor e o pior tempo de execução (Tabela 5.13).

Tabela 5.11: Desempenho de alinhamentos de 4M x 4M (b).

Threads	**Blocos**	**Tempo (ms)**	**MCups**
128	256	106148	151568
128	480	97577	164883
128	512	97306	165343
256	256	98408	163491
256	480	93335	172377
256	**512**	**93184**	**172656**

Tabela 5.12: Desempenho de alinhamentos de 6M x 6M (b).

Threads	Blocos	Tempo (ms)	MCups
128	256	243062	155001
128	480	223084	168882
128	512	223294	168723
256	256	225113	167360
256	480	213901	176132
256	**512**	**213600**	**176380**

Tabela 5.13: Desempenho de alinhamentos de 8M x 8M (b).

Threads	Blocos	Tempo (ms)	MCups
128	256	410116	159542
128	480	374397	174763
128	512	389017	168195
256	256	380273	172063
256	480	357906	182816
256	**512**	**371831**	**175969**

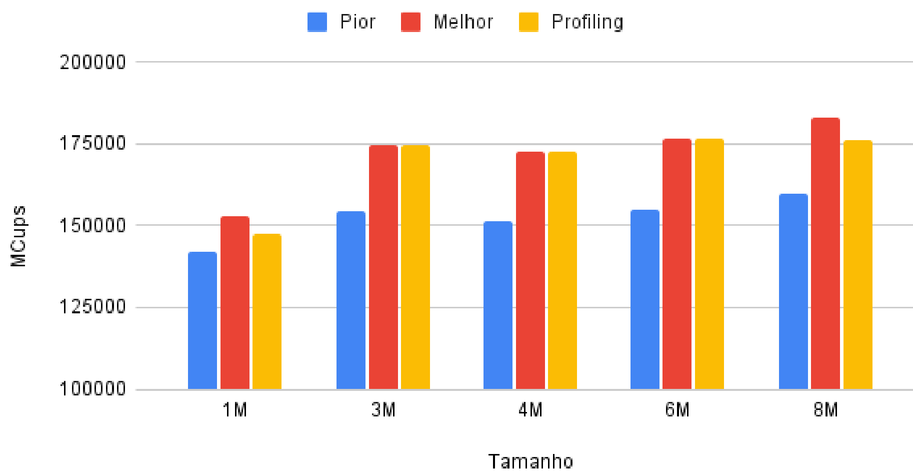

Figura 5.3: Comparação do desempenho entre os melhores e piores parâmetros, e os parâmetros escolhidos pelo *profiling* da ferramenta.

A Figura 5.3 mostra os desempenhos da melhor, da pior e da combinação de parâmetros escolhida pelo *profiling* da ferramenta, para cada tamanho de sequência testado.

Nos alinhamentos de sequências de 1M x 1M (b), os parâmetros escolhidos pela ferramenta apresentaram um desempenho aproximadamente 4% pior do que o melhor, que foi obtido utilizando os parâmetros $T = 256$ e $B = 256$. Contudo, apresentaram um desempenho aproximadamente 3,5% melhor do que a pior combinação de parâmetros, que foi $T = 128$ e $B = 256$. As sequências de 1M mostraram uma variação menor entre cada combinação de parâmetros,

quando comparadas às sequências de outros tamanhos. Os parâmetros escolhidos pela ferramenta são diferentes da média escolhida para os outros tamanhos (Tabela 5.8), o que indica que realizar o *profile* apenas com esse tamanho de sequência traz resultados incompletos, como mencionado na Seção 4.4.2.

É possível observar também que o MCups obtido com a execução de 1M x 1M (b), ou seja, das sequências para validação (Tabela 5.9), foi 32% maior do que a execução com as sequências de *profiling* (Tabela 5.2), no alinhamento com $T = 128$ e $B = 480$. Analisando o alinhamento resultante, é possível observar que as sequências utilizadas para validação possuem uma similaridade muito menor do que as utilizadas para o *profiling*, implicando em um estágio 1 mais demorado (27% mais demorado) pois quanto menor o *score*, menor o efeito do *block pruning* na quantidade de células que devem ser calculadas (Seção 3.1.3). Entretanto, essa similaridade baixa também implica em um *traceback* muito mais rápido (apenas 12 milissegundos contra 3055 milissegundos, ou 254 vezes mais rápido), devido ao alinhamento pequeno que deve ser recuperado.

Nos alinhamentos de sequências de 3M (b), os parâmetros escolhidos pela ferramenta foram aproximadamente 6% melhores do que a segunda melhor combinação ($T = 256$ e $B = 512$). Além disso, também foram aproximadamente 13,1% melhores do que a pior combinação ($T = 128$ e $B = 256$).

Para as sequências de 4M (b) e de 6M (b), os parâmetros escolhidos pela ferramenta foram aproximadamente 0,1% melhores do que a segunda melhor combinação ($T = 256$ e $B = 480$ em ambos os tamanhos). Essa pequena diferença ocorre devido à similaridade entre os resultados obtidos com a primeira e a segunda melhor combinação. Comparando os resultados obtidos com os piores parâmetros, os parâmetros selecionados pela ferramenta foram aproximadamente 13,9% e 13,8% melhores para as sequências de 4M (b) e 6M (b), respectivamente. Em ambos os casos, a pior combinação de parâmetros foi $T = 128$ e $B = 256$.

Para as sequências de 8M (b), os parâmetros escolhidos pela ferramenta foram aproximadamente 4% piores do que a melhor combinação de parâmetros ($T = 256$ e $B = 480$). O *profiling* da ferramenta não utiliza sequências de 8M, portanto os parâmetros escolhidos como os melhores nessa execução são os parâmetros para o tamanho mais próximo (Tabela 5.8). Nesse caso, os parâmetros para 10M são utilizados. Portanto, a diferença de desempenho entre a melhor combinação de parâmetros selecionada pela ferramenta e a melhor combinação observada na prática, pode ter ocorrido devido à diferença no comportamento das sequências de 8M (b) para as sequências de 10M (a). Para evitar essa diferença, é possível adicionar mais variação de tamanhos no *profiling* da ferramenta, mas isso implica em um *profiling* mais demorado. Portanto, é preciso manter um equilíbrio entre tempo e qualidade dos resultados. Vale ressaltar que a combinação de parâmetros escolhida pela ferramenta foi a segunda melhor nos alinhamentos com as sequências de 8M (b). Essa escolha foi aproximadamente 10,2% melhor do que a pior

combinação de parâmetros ($T = 128$ e $B = 256$).

É possível observar que, nas sequências maiores que 1M, as melhores combinações de parâmetros mostram-se parecidas, tanto com as sequências utilizadas no *profiling* (Tabelas 5.2 a 5.7) quanto com as utilizadas para validação (5.9 a 5.13). O melhor valor para T foi 256, enquanto o melhor valor para B variou entre 480 e 512, dependendo dos tamanhos das sequências utilizadas. Em alguns casos, a diferença entre $B = 480$ e $B = 512$ foi muito pequena (0,1%), como nos alinhamentos com sequências de 4M (b) (Tabela 5.11) e 6M (b) (Tabela 5.12). Contudo, em casos como de comparações de 3M x 3M (b) e 8M x 8M (b) a diferença foi mais considerável (4%). Em todos os casos, os melhores parâmetros escolhidos pela ferramenta foram melhores do que os piores parâmetros testados. Para comparações de 3M x 3M (b), 4M x 4M (b), 6M x 6M (b) e 8M x 8M (b) essa diferença foi de mais de 10%, enquanto para 1M foi de 3,5%.

Em suma, consideramos que a ferramenta proposta atingiu os objetivos e foi capaz de escolher valores apropriados para os parâmetros B e T.

5.6 Execução em Lote

De forma a avaliar o impacto da execução em lote, a ferramenta desenvolvida foi utilizada para executar um conjunto de comparações de forma automatizada, utilizando as mesmas comparações descritas na Tabela 5.1. O arquivo **configurações_de_alinhamentos** foi preenchido com as comparações a serem realizadas, como pode ser visto na Figura 5.4.

```
1   ../test_sequences/1M/JANKHN010000019.1.fasta ../test_sequences/1M/JANKHN010000507.1.fasta
2   ../test_sequences/3M/CP100650.1.fasta ../test_sequences/3M/NZ_JANUDR010000003.1.fasta
3   ../test_sequences/4M/CP103782.1.fasta ../test_sequences/4M/CP103784.1.fasta
4   ../test_sequences/6M/NZ_JANULM010000001.1.fasta ../test_sequences/6M/NZ_JANUMG010000001.1.fasta
5   ../test_sequences/8M/NZ_NWMZ01000001.1.fasta ../test_sequences/8M/NW_026095751.1.fasta
```

Figura 5.4: Arquivo **configurações_de_alinhamentos** com 5 comparações configuradas para execução em lote.

A ferramenta carregou as comparações e suas sequências e, de forma automatizada, executou o MASA-CUDAlign para cada uma. Ao final de cada execução, os arquivos de estatísticas e do alinhamento resultante foram disponibilizados em um diretório próprio para cada comparação, como pode ser visto na Figura 5.5. O nome do diretório do resultado de uma comparação contém a data e o horário do momento em que o alinhamento foi finalizado, além dos nomes das sequências comparadas, para que seja possível diferenciar os resultados.

Cada uma das comparações executadas utilizou o parâmetro adequado, selecionado de forma automatizada pela ferramenta com base no *profiling* previamente executado. O tempo total de execução foi de 759,2 segundos, ou 12,7 minutos, aproximadamente. Caso fossem utilizados as piores escolhas de parâmetros para cada uma das comparações, o tempo total de

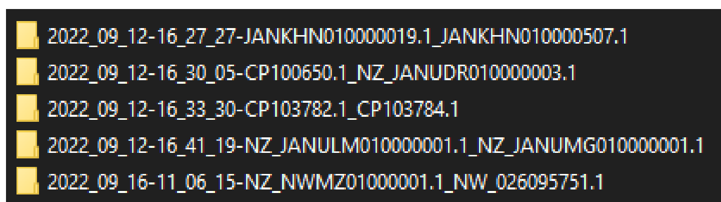

Figura 5.5: Diretórios dos resultados da execução em lote das comparações da Tabela 5.1.

execução seria de 849,7 segundos, ou 14,2 minutos, aproximadamente. Ou seja, a escolha de parâmetros da ferramenta resultou em um tempo total de execução de 1,5 minuto mais rápido do que o tempo total de execução utilizando os piores parâmetros, para esse caso de 5 comparações descritas na Tabela 5.1.

Portanto, considera-se que a ferramenta atingiu os objetivos da execução em lote, sendo capaz de executar um conjunto de comparações de forma totalmente automatizada, coletando os resultados de forma organizada para facilitar a posterior análise do usuário. Além disso, cada comparação do lote foi executada utilizando parâmetros adequados obtidos pelo *profiling*, resultando em comparações otimizadas.

Capítulo 6

Conclusão e Trabalhos Futuros

O presente trabalho de graduação propôs, implementou e avaliou uma ferramenta para sugerir os parâmetros B e T de execução em GPU para o MASA-CUDAlign, com base em um *profiling* da GPU do usuário. Além disso, a ferramenta proposta fornece uma maneira acessível de configurar, executar e coletar os resultados de lotes de alinhamentos de sequências. A ferramenta foi implementada utilizando 4 módulos responsáveis por apresentar a interface ao usuário, gerenciar as execuções em lote, construir o *profile* e coletar resultados e fornecê-los ao usuário, além de arquivos para o armazenamento de configurações.

Os resultados obtidos com execuções em uma GPU NVIDIA GeForce RTX 2060 mostraram que, de 5 tamanhos de sequências testados (1M, 3M, 4M, 6M e 8M), a ferramenta sugeriu os melhores valores de parâmetros para 3 tamanhos. Além disso, os parâmetros sugeridos foram melhores do que a pior combinação de parâmetros em todos os 5 tamanhos testados. Portanto, o uso do *profiling* completo se mostrou vantajoso. É importante observar que a diferença de desempenho entre a melhor combinação de parâmetros e a segunda melhor combinação foram pequenas, para todos os tamanhos de sequências testados. Contudo, o impacto se torna maior à medida que mais execuções são realizadas, assim como execuções com sequências de tamanho maior. Dessa forma, a escolha dos valores apropriados de parâmetros se torna mais importante de forma proporcional ao uso da ferramenta. Quanto mais alinhamentos precisam ser executados, maior o impacto que a melhor combinação de parâmetros possui no tempo total de execução.

Além disso, foi possível observar que as sequências de 1M possuem um comportamento diferente das sequências maiores testadas. Os melhores parâmetros obtidos para sequências desse tamanho foram bem diferentes dos melhores parâmetros selecionados para os outros tamanhos de sequências. Os resultados obtidos executando sequências maiores, com a melhor combinação de parâmetros para sequências de 1M, são medianos, pois são melhores do que a pior combinação obtida para cada tamanho (7,5% em média), mas também são piores do que a melhor combinação (5,5% em média) e piores do que as selecionadas para esses tamanhos pela

ferramenta. Portanto, utilizar apenas sequências de 1M para construir um *profile*, como é o caso do *profiling* simples, é rápido, mas os resultados são incompletos.

A execução em lote se mostrou uma funcionalidade útil, facilitando o uso do MASA-CUDAlign para múltiplos alinhamentos. O impacto da ferramenta foi maior conforme o tamanho das sequências testadas aumentou. Dessa forma, foi possível remover a interação do usuário com a ferramenta entre cada alinhamento, efetivamente removendo o tempo gasto coletando resultados e configurando um próximo alinhamento para execução. Além disso, também facilitou a organização da lista de alinhamentos a serem realizados, removendo a necessidade de acompanhar manualmente quais alinhamentos já foram obtidos, e a análise de resultados posterior, devido à coleta e organização automática dos dados.

Diante do trabalho realizado e dos resultados apresentados, sugere-se como trabalhos futuros:

- Avaliar o uso de um tamanho de sequência maior que 1M para o *profiling* simplificado, para evitar o comportamento obtido com as sequências desse tamanho, mas ainda fornecer uma opção de *profiling* rápido para máquinas com menor poder de processamento. É possível também avaliar a adição de uma opção de *profiling* médio, com mais sequências que o *profiling* simplificado mas menos que o completo, de forma a fornecer um *profile* intermediário que combine pontos positivos dos dois tipos fornecidos com a ferramenta;
- Compatibilizar e avaliar o uso da ferramenta proposta com o MASA-CUDAlign executado em múltiplas GPUs. A ferramenta foi desenvolvida e testada em um ambiente com apenas uma GPU, mas o MASA-CUDAlign 4.0 é capaz de utilizar múltiplas GPUs para acelerar a computação. Logo, é interessante modificar a ferramenta para adicionar a capacidade de executar o CUDAlign com múltiplas GPUs, e gerenciar os melhores parâmetros corretamente para esses casos, para que possa ser utilizada em máquinas com essa característica;
- Avaliar o uso de mais variações de tamanhos de sequências no *profiling* completo. Realizar mais alinhamentos com mais tamanhos de sequências traria resultados mais completos, para reduzir os casos nos quais é preciso utilizar os parâmetros escolhidos para um tamanho mais próximo, e aumentar os casos nos quais os parâmetros escolhidos são os melhores. Entretanto, isso aumentaria ainda mais o tempo de execução necessário para o *profiling* completo, e deve ser avaliado se a qualidade do resultado compensaria o investimento de tempo maior.

Referências

[1] Figueiredo 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, Brasil, 2021. https://repositorio.unb.br/handle/10482/41495, acesso em 2022-03-30. ix, 1, 6, 10, 16, 19, 20, 21, 34

[2] Sandes, Edans Flavius O. e Alba Cristina M.A. de Melo: *Cudalign: Using gpu to accelerate the comparison of megabase genomic sequences*. SIGPLAN Not., 45(5):137–146, jan 2010, ISSN 0362-1340. https://doi.org/10.1145/1837853.1693473. ix, 13

[3] 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. ix, 12, 14, 15

[4] 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. ix, 12, 15, 16

[5] 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. ix, 1, 12, 16, 17

[6] De O. Sandes, Edans F., Guillermo Miranda, Xavier Martorell, Eduard Ayguade, George Teodoro e Alba C. M. A. De Melo: *Masa: A multiplatform architecture for sequence aligners with block pruning*. ACM Trans. Parallel Comput., 2(4), fevereiro 2016, ISSN 2329-4949. https://doi.org/10.1145/2858656. ix, 12, 16, 17, 18

[7] Mount, D.W. e Howard Hughes Medical Institute Endowed Library Fund: *Bioinformatics: Sequence and Genome Analysis*. Bioinformatics: Sequence and Genome Analysis. Cold Spring Harbor Laboratory Press, 2001, ISBN 9780879695972. https://books.google.com.br/books?id=bYQoAQAAMAAJ. 1, 3, 4, 5

[8] Oliveira Sandes, Edans Flávius de: *Comparação paralela de sequências biológicas longas utilizando Unidades de Processamento Gráfico (GPUs)*. Tese de Doutoramento, Universidade de Brasília, Brasil, 2011. https://repositorio.unb.br/handle/10482/10022, acesso em 2022-08-21. 1, 13, 21, 22, 23

[9] *Biological sequences*. https://www.ncbi.nlm.nih.gov/IEB/ToolBox/SDKDOCS/BIOSEQ.HTML, acesso em 2022-03-30. 3

[10] Denardo, Eric V.: *Dynamic Programming: Models and Applications*. Dover Books on Computer Science, 2003, ISBN 9780486428109. 4

[11] 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. 5

[12] Computer Science, University of Freiburg Department of: *Freiburg RNA Tools - Teaching - Needleman-Wunsch*. http://rna.informatik.uni-freiburg.de/Teaching/index.jsp?toolName=Needleman-Wunsch, acesso em 2022-03-30. 5

[13] 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. 5

[14] Computer Science, University of Freiburg Department of: *Freiburg RNA Tools - Teaching - Smith-Waterman*. http://rna.informatik.uni-freiburg.de/Teaching/index.jsp?toolName=Smith-Waterman, acesso em 2022-03-30. 5

[15] 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. 6

[16] Computer Science, University of Freiburg Department of: *Freiburg RNA Tools - Teaching - Gotoh*. http://rna.informatik.uni-freiburg.de/Teaching/index.jsp?toolName=Gotoh, acesso em 2022-03-30. 6

[17] Computer Science, University of Freiburg Department of: *Freiburg RNA Tools - Teaching - Gotoh (Local)*. http://rna.informatik.uni-freiburg.de/Teaching/index.jsp?toolName=Gotoh%20(Local), acesso em 2022-03-30. 8

[18] 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. 9, 10

[19] Computer Science, University of Freiburg Department of: *Freiburg RNA Tools - Teaching - Hirschberg*. http://rna.informatik.uni-freiburg.de/Teaching/index.jsp?toolName=Hirschberg, acesso em 2022-03-30. 9, 10

[20] Myers, Eugene W. e Webb Miller: *Optimal alignments in linear space*. Bioinformatics, 4(1):11–17, março 1988, ISSN 1367-4803. https://doi.org/10.1093/bioinformatics/4.1.11. 11

[21] NVIDIA: *Cuda Zone*. https://developer.nvidia.com/cuda-zone, acesso em 2022-05-01. 12

[22] 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. 12, 13

[23] 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, Brasil, 2015. https://repositorio.unb.br/handle/10482/20248, acesso em 2022-08-25. 13

[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. 19, 20

[25] NVIDIA: *Achieved Occupancy*. https://docs.nvidia.com/gameworks/content/developertools/desktop/analysis/report/cudaexperiments/kernellevel/achievedoccupancy.htm, acesso em 2022-09-04. 22

[26] Coutinho, Bruno Rocha, George Luiz Medeiros Teodoro, Rafael Sachetto Oliveira, Dorgival Olavo Guedes Neto e Renato Antonio Celso Ferreira: *Profiling general purpose gpu applications*. Em *2009 21st International Symposium on Computer Architecture and High Performance Computing*, páginas 11–18, 2009. 23

[27] Medicine, National Library of: *National Center for Biotechnology Information*. https://www.ncbi.nlm.nih.gov, acesso em 2022-08-21. 30, 34

[28] Sandes, Edans: *Repositório do MASA CUDAlign*. https://github.com/edanssandes/MASA-CUDAlign, acesso em 2022-09-18. 35

[29] Andrade, Pedro Lucas Pinto: *Repositório da Ferramenta de Profiling e Execução Automatizados do CUDAlign*. https://github.com/pelu97/cudalign_manager, acesso em 2022-09-24. 36