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

CUDA-Parttree:

Estratégia Paralela em GPU para Alinhamento Múltiplo Heurístico de Milhares de Sequências

Cainã F. B. Razzolini

Dissertação apresentada como requisito parcial para
conclusão do Mestrado em Informática

Orientador
Prof. Dr. Alba Cristina Magalhães Alves de Melo

Brasília
2019

Ficha Catalográfica de Teses e Dissertações

Está página existe apenas para indicar onde a ficha catalográfica gerada para dissertações de mestrado e teses de doutorado defendidas na UnB. A Biblioteca Central é responsável pela ficha, mais informações nos sítios:

http://www.bce.unb.br
http://www.bce.unb.br/elaboracao-de-fichas-catalograficas-de-teses-e-dissertacoes

Esta página não deve ser inclusa na versão final do texto.

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

CUDA-Parttree:

Estratégia Paralela em GPU para Alinhamento Múltiplo Heurístico de Milhares de Sequências

Cainã F. B. Razzolini

Dissertação apresentada como requisito parcial para
conclusão do Mestrado em Informática

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

Prof. Dr. Philippe Olivier Alexandre Navaux
INF/UFRGS

Prof. Dr. Jacir Luiz Bordim
CIC/UnB

Prof. Dr. Bruno Luiggi Macchiavello Espinoza
Coordenador do Programa de Pós-graduação em Informática

Brasília, 20 de Agosto de 2019

Agradecimentos

Agradeço à minha família, que sempre apoiou minhas decisões e me deu suporte quando mais precisei. Agradeço ao meu pai que sempre fomentou minha curiosidade e é o meu maior exemplos. Em especial, agradeço a minha mãe, que sempre foi o meu suporte e me incentivou a prosseguir na vida acadêmica, a quem, no decorrer do meu mestrado, eu acabei perdendo, mas que mesmo nos momentos mais difíceis sempre conseguia achar forças para dar um sorriso inesquecível que ficará para sempre comigo.

À minha orientadora Alba, agradeço o tempo e esforço dedicados. Por todas as reuniões, revisões e orientações que fizeram desse trabalho o que é. Sobretudo, agradeço pelo suporte em dos momentos mais difíceis da minha vida, que me ajudou a seguir em frente e perseverar.

Resumo

O alinhamento de sequências biológicas é uma operação muito importante na Bioinformática e pode ser feito entre duas (alinhamento par-a-par) ou mais sequências (alinhamento múltiplo). O alinhamento múltiplo de sequências é um problema NP-completo e o algoritmo de programação dinâmica que obtém o alinhamento ótimo só é utilizado para conjuntos com poucas sequências (até 30). Por isso, são utilizados algoritmos heurísticos, que obtém alinhamento com acurácia aceitável em tempo hábil. Para conjuntos com dezenas de milhares de sequências são necessários algoritmos especificamente criados para esse fim. O Parttree é um algoritmo heurístico de alinhamento múltiplo de sequências para conjuntos com dezenas de milhares de sequências. Nessa dissertação de mestrado, propomos e avaliamos o CUDA-Parttree, uma estratégia de paralela que executa a primeira fase do Parttree (cálculo da matriz de distâncias com contagem de 6mers) parcialmente em GPU. Com essa estratégia, conseguimos reduzir consideravelmente o tempo de execução do cálculo das distâncias entre as sequências quando comparado ao Parttree. O CUDA-Parttree foi usado no alinhamento de 6 conjuntos de sequências reais de proteínas, como tamanho variando de 25.534 a 151.443 sequências, e 4 conjuntos de sequências sintéticas, variando de 10.000 a 100.000. O CUDA-Parttree conseguiu um *speedup* de 6,10x no cálculo da matriz de distâncias para o conjunto *Cyclodex_gly_tran* (50.280 sequências), reduzindo o tempo de execução de 33,94s para 5,57s. Contudo, as transformações de dados necessárias para o cálculo em GPU e para retornar a matriz de distâncias para o Parttree reduziram o *speedup* para 2,58x mais rápido que a versão em CPU. Com um conjunto de sequências sintéticas com 100.000 sequências, conseguimos um *speedup* de 4,46x, reduzindo o tempo de execução do cálculo de distâncias de 209,54s para 47,00s.

Palavras-chave: alinhamento múltiplo de sequências, GPU, contagem de kmers

Abstract

Biological sequence alignment is a very important task in Bioinformatics and it may be done between two (pairwise alignment) or more sequences (multiple sequence alignment). Multiple sequence alignment is an NP-complete problem and the dynamic programming algorithm that obtains the optimal alignment is used only for small sequence sets (up to 30 sequences). As such, heuristic algorithms are used for obtaining alignments with acceptable accuracy in a timely manner. For sets with tens of thousands of sequences, algorithms specifically designed for such sizes are necessary. Parttree is a state-of-the-art heuristic algorithm for multiple sequence alignment of tens of thousands os sequences. In this MSc dissertation, we propose and evaluate CUDA-Parttree, a parallel strategy that executes the first phase of Parttree (distance matrix calculation with 6mers) partially on GPU. Using this strategy, we were able to reduce the execution time of the distance calculation between all sequences compared to Parttree. CUDA-Parttree was used to align 6 real sequence sets, ranging from 25,534 to 151,443 sequences, and 4 synthetic sets, ranging from 10,000 to 100,000 CUDA-Parttree obtained a speedup of $6.10x$ on the distance matrix calculation for the *Cyclodex_gly_tran* (50,280 sequences) set, reducing the execution time from $33.94s$ to $5.57s$. Nevertheless, the data transformation required for the GPU calculation and to return the distance matrix to Parttree reduced the speedup to $2.59x$. With the sequence set *Syn_100000* ($100,000$ sequences), a speedup of $4.46x$ was attained, reducing execution time from $209.54s$ to $47.00s$.

Keywords: multiple sequence alignment, GPU, kmer counting

Sumário

1 Introdução **1**
1.1 Motivação 1
1.2 Objetivos 2
1.3 Estrutura do Documento 3

2 Alinhamento múltiplo de sequências biológicas **4**
2.1 Definições 4
2.1.1 Sequências 4
2.1.2 Alinhamento de sequências 5
2.2 Função de escore 5
2.2.1 Sum-of-pairs (SP) 5
2.2.2 Weighted Sum-of-pairs (WSP) 7
2.3 Alinhamento múltiplo exato de sequências 7
2.3.1 Algoritmo exato simples 7
2.3.2 Carrilo-Lipman 9
2.3.3 A-STAR (A*) 9
2.4 Métodos Heurísticos 11
2.4.1 Alinhamento progressivo 11
2.4.1.1 Visão Geral 11
2.4.1.2 Clustal W 13
2.4.1.3 MAFFT - FFT-NS-[1/2] 14
2.4.1.4 MSAProbs 17
2.4.2 Alinhamento por refinamento iterativo 18
2.4.2.1 Visão Geral 18
2.4.2.2 MAFFT - métodos iterativos 19
2.4.3 MAFFT Parttree - Alinhamento de milhares de sequências 19
2.5 Considerações Finais 20

3 Unidades de Processamento Gráfico(GPUs) 21
3.1 Arquiteturas das GPUs NVidia . . . 22
3.2 Evolução das GPUs da NVidia . . . 23
3.3 Arquitetura CUDA . . . 25

4 Algoritmos de MSA em plataformas de alto desempenho 29
4.1 CUDA Clustal W . . . 29
4.2 MAFFT - CUDA-Linsi . . . 30
4.3 CMSA . . . 32
4.4 MSAProbs-MPI . . . 33
4.5 FAMSA . . . 34
4.6 Quadro comparativo . . . 35

5 Projeto do CUDA-Parttree 37
5.1 Análise do Parttree . . . 38
5.2 Visão geral do CUDA-Parttree . . . 42
5.3 Conversão de dados (*Input*) . . . 44
5.4 Computação em GPU . . . 46
5.5 Recuperação de dados (Output) . . . 49
5.5.1 Considerações Finais . . . 49

6 Resultados Experimentais 51
6.1 Ambiente de Testes . . . 51
6.2 Conjuntos de sequências comparados . . . 51
6.3 Resultados obtidos . . . 53
6.3.1 Tempo de execução do CUDA-Parttree . . . 53
6.3.2 Speedup da computação . . . 54
6.3.3 *Profiling* do CUDA-Parttree . . . 55
6.3.4 Uso de memória . . . 57
6.3.5 Alinhamento de Sequências Sintéticas . . . 58
6.4 Comparação com o FAMSA . . . 59
6.5 Considerações Finais . . . 60

7 Conclusão e Trabalhos Futuros 62
7.1 Conclusão . . . 62
7.2 Trabalhos Futuros . . . 63

Referências 64

Lista de Figuras

2.1 Alinhamento de 12 sequências de proteínas. 5
2.2 Cálculo do Sum-of-pairs para 3 sequências. 6
2.3 Matriz de substituição PAM250. 7
2.4 Alinhamento entre 3 sequências. 9
2.5 Resultado da análise da FFT, possuindo 2 picos correspondentes a regiões homólogas. 15
2.6 Análise utilizando *sliding window*. 15
2.7 Exemplo de DP sobre os segmentos. 16
2.8 Espaço de busca da DP, área em cinza não é visitada. Cada ponto representa a posição de um segmento homólogo. 16

3.1 Comparativo do design de CPU e GPU. 22
3.2 Visão geral de arquitetura de um GPU. 23
3.3 Placa NVidia GTX 680 da arquitetuta Kepler com 1.536 CUDA *cores*. . . . 24
3.4 Placa NVidia GTX 980 da arquitetuta Maxwell com 2.048 CUDA *cores*. . . 25
3.5 Compilação de código CUDA C. 26
3.6 Organização hieráquica de *threads*. 27
3.7 Modelo de memória CUDA. 27

4.1 Cálculo da matriz de distâncias do CUDA Clustal W. 30
4.2 CUDA-LINSi framework. 30
4.3 Transformação de dados para GPU. 31

5.1 Conversão de uma sequência de 12 caracters em um vetor de 6mers. 40
5.2 Matriz de distâncias calculada pelo Parttree em um nível de recursão. Os números representam as áreas calculadas por cada chamada do algoritmo *calculateSimilarity*. 42
5.3 Fluxo do CUDA-Parttree. 43
5.4 Conversão de dados do CUDA-Parttree. 45

6.1 Proporção do tempo de execução das etapas do CUDA-Parttree. 56

6.2 Proporção do tempo de execução das etapas do CUDA-Parttree para as sequências sintéticas. 59

Lista de Tabelas

4.1 Comparativo entre os algoritmos de Estado da Arte. 35

6.1 Especificações de hardware e software do Ambiente 980. 51
6.2 Especificações de hardware e software do Ambiente 680. 52
6.3 Descrição dos conjuntos de sequências do extHomFam utilizados. 52
6.4 Descrição dos conjuntos de sequências sintéticas gerados. 52
6.5 Tempos de execução do *calculateSimilarity*, no Parttree e no CUDA-Parttree, no Ambiente 980. 53
6.6 Tempos de execução do *calculateSimilarity*, no Parttree e no CUDA-Parttree, no Ambiente 680. 54
6.7 Tempos de execução da computação da matriz de distâncias no Ambiente 980. 55
6.8 Tempos de execução da computação da matriz de distâncias no Ambiente 680. 55
6.9 Tempos de execução de cada etapa do CUDA-Parttree no Ambiente 980. . 56
6.10 Tempos de execução de cada etapa do CUDA-Parttree no Ambiente 680. . 57
6.11 Número de sequências base, sequências de referência e tamanho da matriz de distâncias. 57
6.12 Tempos de execução do *calculateSimilarity*, no Parttree e no CUDA-Parttree, no Ambiente 680. 58
6.13 Tempos de execução de cada etapa do CUDA-Parttree no Ambiente 980 para as sequências sintéticas. 58
6.14 Tempos de execução da computação da matriz de distâncias, no Parttree e no CUDA-Parttree, das sequências sintéticas 59
6.15 Tempos de execução totais do FAMSA e do CUDA-Parttree, no Ambiente 980 para os conjuntos de sequências reais. 60

Capítulo 1

Introdução

A Bioinformática é uma área de estudos interdisciplinar que visa desenvolver algoritmos e ferramentas para auxiliar biólogos na análise de dados, a fim de compreender a função e estrutura de sequências biológicas e obter informações sobre a evolução de organismos [1].

O alinhamento de sequências pode ser feito entre duas sequências, chamado alinhamento entre pares, ou mais que duas, chamado alinhamento múltiplo (*Multiple Sequence Alignment* - MSA). O alinhamento múltiplo de sequências é um problema computacionalmente desafiador, tendo sido provado NP-Completo [2]. Por isso é comum a utilização de algoritmos heurísticos, que apesar de não obterem o melhor resultado possível, conseguem resultados em tempo razoável com uma acurácia aceitável [3].

O MSA heurístico recebe um conjunto de sequências biológicas e retorna o alinhamento múltiplo, que ressalta regiões de similaridades e diferenças entre sequências. Geralmente é executado em 3 etapas: (1) cálculo da matriz de similaridades (ou matriz de distâncias), que retorna um escore para comparações entre pares de sequências; (2) construção da árvore guia a partir da matriz de similaridades e (3) obtenção do alinhamento.

1.1 Motivação

A evolução nos métodos de sequenciamento tem gerado bancos de dados genômicos cada vez maiores. Mesmo os métodos heurísticos, que buscam obter resultados de boa qualidade em um tempo de execução compatível com as necessidades dos biólogos, têm dificuldades em fazer o alinhamento múltiplo de conjuntos com dezenas de milhares de sequências em tempo hábil [4]. Até os algoritmos desenvolvidos especificamente para conjuntos com dezenas de milhares de sequências [5] [6] podem levar horas ou dias para obter resultados [4].

Visando obter um menor tempo de execução, diversas aplicações de Bioinformática tem usando a grande capacidade de paralelização ofertada pelas GPUs (*Graphics Processing Units*) [7]. Já existem implementações de algoritmos de alinhamento múltiplo utilizando

GPUs, mas a maioria dessas implementações é voltada para conjuntos com menos de 20.000 sequências.

O MAFFT é um pacote de programas que faz o alinhamento múltiplo de sequências com método heurístico usando FFT (*Fast Fourier Transform*) na fase 3, onde é obtido o alinhamento final [8]. Um dos programas que compõem o pacote MAFFT é o Parttree, que usa contagem de 6mers (subsequências de tamanho 6) em comum na etapa 1 e é capaz de comparar mais 100.000 sequências. O Parttree possui uma versão sequencial [8], que é bastante utilizada pela comunidade científica [9] [10] [11], porém seus tempos de execução podem chegar a horas. Até o momento, o Parttree não possui versão em GPU.

1.2 Objetivos

O objetivo da presente dissertação de mestrado é, portanto, propor e avaliar o CUDA-Parttree, uma estratégia paralela em GPU para alinhamento múltiplo heurístico de dezenas de milhares de sequências com o Parttree.

No Cuda-Parttree, optou-se por implementar a contagem de 6mers em GPU pois, além de ser uma rotina que exige alto poder computacional, a contagem de 6mers ou, de maneira mais genérica, a contagem de k-mers é usada em outros problemas de Bioinformática como a montagem de DNA (*DNA assembly*) [12].

Diferente da maioria das ferramentas para MSA, o Parttree executa a fase 1 (contagem de 6mers em comum) e a fase 2 (construção da árvore guia) simultaneamente, o que torna o projeto da contagem de 6mers em GPU particularmente desafiador. Para manter as *threads* em GPU ocupadas na maior parte do tempo, as sequências foram ordenadas por tamanho e foram introduzidos *paddings*, de maneira que todas as *threads* recebessem a mesma quantidade de trabalho. Além disso, a matriz de distâncias foi linearizada para que os acessos à memória de GPU fossem feitos de maneira otimizada.

O CUDA-Parttree foi implementado em C e CUDA e executado em duas GPUs (GeForce GTX980 Ti e GeForce GTX680). Foram utilizados 6 conjuntos de sequências, disponíveis no *benchmark extHomFam* [13], contendo entre 25.534 e 151.443 sequências, e 4 conjuntos sintéticos, gerados aleatoriamente, contendo entre 10.000 e 100.000 sequências para realização dos testes de desempenho. Os resultados mostram que conseguimos um *speedup* de até 6,2x na realização do cálculo da matriz de similaridades em GPU para o conjunto *bac_lusiferase* (25.534 sequências) quando comparado ao Parttree. Contudo, esse *speedup* foi afetado pelas transformações necessárias para adaptar as sequências ao cálculo em GPU e para transferir a matriz calculada para o formato usado nas demais etapas do Parttree, reduzindo o *speedup* total obtido, conseguindo um *speedup* de até 2,58x

para o conjunto *Cyclodex_gly_tran* (50.280 sequências). Ao comparar um conjunto de 100.000 sequências sintéticas, o *speedup* do cálculo da matriz de distâncias foi de 4,46x.

1.3 Estrutura do Documento

O restante desse documento está organizado da seguinte maneira. O Capítulo 2 discorre sobre o problema do alinhamento múltiplo de sequências, apresentando em seguida os principais algoritmos de alinhamento múltiplo exato e, por fim, os principais métodos e algoritmos de alinhamento múltiplo heurístico. O Capítulo 3 mostra a arquitetura de GPUs. O Capítulo 4 apresenta algumas abordagens para o alinhamento múltiplo heurístico em plataformas de alto desempenho. O Capítulo 5 contém o projeto da estratégia do CUDA-Parttree. O Capítulo 6 apresenta os experimentos realizados e a análise dos resultados obtidos. Por fim, o Capítulo 7 contém a conclusão e os trabalhos futuros.

Capítulo 2

Alinhamento múltiplo de sequências biológicas

O alinhamento de sequências é um dos problemas mais importantes da Bioinformática e consiste em organizar um conjunto de sequências de forma a explicitar regiões de similaridades entre elas. Estas similaridades são importantes pois indicam um possível relacionamento evolutivo e funcional entre as sequências [3].

Existem diversas maneiras de se realizar o alinhamento entre sequências. Por exemplo, pode-se utilizar apenas duas sequências, chamado alinhamento entre pares, ou mais de duas, chamado alinhamento múltiplo. Também é possível que o alinhamento seja feito considerando apenas as sequências em si, chamado alinhamento primário, ou que o alinhamento seja feito levando em consideração as estruturas 2D ou 3D das sequências, chamado de alinhamento estrutural [3]. Outra escolha a ser feita é se o alinhamento é feito sobre a extensão total das sequências, chamado alinhamento global, ou se apenas subsequências destas serão alinhadas, chamado de alinhamento local.

O presente capítulo abordará o alinhamento múltiplo de sequências considerando sua estrutura primária. Inicialmente, serão apresentadas as definições de sequência e escore. A seguir serão apresentados algoritmos que obtém o alinhamento múltiplo ótimo. Por fim, são apresentados os algoritmos heurísticos de alinhamento múltiplo.

2.1 Definições

2.1.1 Sequências

As sequências utilizadas nos alinhamentos são cadeias ordenadas de aminoácidos, no caso de proteínas, ou de nucleotídeos, para DNAs e RNAs. DNAs e RNAs são representados como sequências de caracteres do alfabeto $\{A, T, C, G\}$ e $\{A, U, C, G\}$, respecti-

vamente, e proteínas são representadas como uma sequência de caracteres do alfabeto $\{A, C, D, E, F, G, H, I, K, L, M, N, P, Q, R, S, T, V, W, Y\}$.

2.1.2 Alinhamento de sequências

Seja S um conjunto de sequências definido como $S = (a_1, ..., a_n)$, onde a_i é uma sequência de caracteres do alfabeto α, e $\beta = \alpha \cup \{-\}$, onde "$-$" é o símbolo de *gap*. Pode-se então representar um alinhamento global de sequências do conjunto $S = a_1, ..., a_n$ com o conjunto $\bar{S} = (\bar{a_1}, ..., \bar{a_n})$, onde $\bar{a_i}$ é obtido inserindo *gaps* até que todas as sequências tenham o mesmo tamanho [14]. A Figura 2.1 apresenta um exemplo de alinhamento múltiplo entre 12 sequências de proteínas.

```
HUMA        VLSPADKTNVKAAWGKVGAHAGEYGAEALERMFLSFPTTKTYFPHF DLSH     GS
HAOR        MLTDAEKKEVTALWGKAAGHGEEYGAEALERLFQAFPTTKTYFSHF DLSH     GS
HADK        VLSAADKTNVKGVFSKIGGHAEEYGAETLERMFIAYPQTKTYFPHF DLSH     GS
HBHU       VHLTPEEKSAVTALWGKV  NVDEVGGEALGRLLVVYPWTQRFFESFGDLSTPDAVMGN
HBOR       VHLSGGEKSAVTNLWGKV  NINELGGEALGRLLVVYPWTQRFFEAFGDLSSAGAVMGN
HBDK       VHWTAEEKQLITGLWGKV  NVADCGAEALARLLIVYPWTQRFFASFGNLSSPTAILGN
MYHU        GLSDGEWQLVLNVWGKVEADIPGHGQEVLIRLFKGHPETLEKFDKFKHLKSEDEMKAS
MYOR        GLSDGEWQLVLKVWGKVEGDLPGHGQEVLIRLFKTHPETLEKFDKFKGLKTEDEMKAS
IGLOB      SPLTADEASLVQSSWK    AVSHNEVEILAAVFAAYPDIQNKFSQFA1GKDLASIKDT
GPUGNI      ALTEKQEALLKQSWEVLKQNIPAHSLRLFALIIEAAPESKYVFSFLKDSNEIPE  NN
GPYL       GVLTDVQVALVKSSFEEFNANIPKNTHRFFTLVLEIAPGAKDLFSFLKGSSEVPQ  NN
GGZLB       MLDQQTINIIKATVPVLKEHGVTITTTFYKNLFAKHPEVRPLF    DMGRQE   SL
```

Figura 2.1: Alinhamento de 12 sequências de proteínas (Fonte: [14]).

Um alinhamento representa um conjunto de inserções, deleções e substituições sobre as sequências originais. Dado um conjunto de sequências, existem diversos alinhamentos possíveis e cabe à uma função de cálculo de escore atribuir uma pontuação ao alinhamento, como forma de medir sua qualidade.

2.2 Função de escore

A função de cálculo do escore de um alinhamento múltiplo é de fundamental importância para a qualidade do resultado obtido. Não existe hoje um consenso sobre qual a função a ser utilizada, havendo várias propostas [14]. Nessa seção, serão discutidas as duas funções mais utilizadas para atribuir um escore a um alinhamento múltiplo.

2.2.1 Sum-of-pairs (SP)

Uma das funções de escore mais conhecidas é o *Sum-of-pairs (SP)*, um método simples que não considera a dependência entre as colunas [3]. O escore do alinhamento múlti-

plo, $SP(M)$, é obtido através da soma dos escores dos alinhamentos entre os pares de sequências.

No alinhamento entre pares, as duas sequências estando já alinhadas, $\bar{S}_k$ e $\bar{S}_l$ são comparadas posição a posição, isto é, o i-ésimo caractere de $\bar{S}_k$, $\bar{S}_k{}^i$, com o i-ésimo da sequência $\bar{S}_l$, $\bar{S}_l{}^i$, até o fim das sequências. Para cada par $(\bar{S}_k, \bar{S}_l)$ é atribuído um escore (SC) e o escore final do alinhamento entre pares $(PA(\bar{S}_k, \bar{S}_l))$ é dado pela soma dos escores de cada posição do alinhamento. Assim temos as equações Equações 2.1 a 2.2, onde L é comprimento das sequências.

$$SP = \sum_{k<l} PA(\bar{S}_k, \bar{S}_l) \tag{2.1}$$

$$PA(S_k, S_l) = \sum_{i=0}^{L} SC(\bar{S}_k{}^i, \bar{S}_l{}^i) \tag{2.2}$$

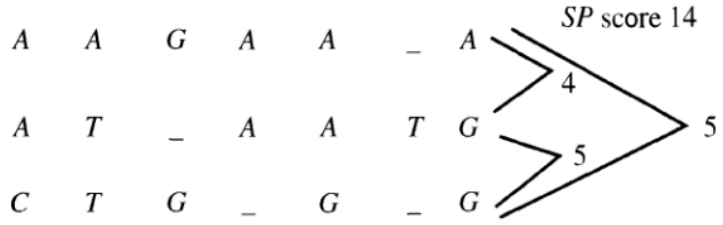

Figura 2.2: Cálculo do Sum-of-pairs para 3 sequências (Fonte: [14]).

No alinhamento de nucleotídeos (DNA e RNA), é geralmente atribuída uma pontuação única para *matches* (caracteres iguais) e *mismatches* (caracteres diferentes). A Figura 2.2 apresenta o alinhamento de entre 3 sequências de DNA, onde a pontuação para *matches* é 0 e para *mismatches* e *gaps* é 1.

No alinhamento de aminoácidos, o valor de $SC(\bar{S}_k{}^i, \bar{S}_l{}^i)$ é dado por uma matriz de substituição como a PAM [15] ou a BLOSUM [16], formuladas com base em dados biológicos com relevância estatística. A Figura 2.3 apresenta a matriz de substituição PAM250, onde o valor de cada célula representa a possibilidade de o amino ácido da linha ser substituído pelo da coluna.

Na literatura existem diferentes funções de escore no que tange os *gaps*, do mais simples em que $S(a, -)$ e $S(-, b)$ recebem um custo de *gap* fixo e $S(-, -)$ recebe 0, até mais complexas como *affine gap* que atribuem custos diferenciados de acordo com a posição do *gap* sendo inserido [14].

O SP torna simples o cálculo do escore de um alinhamento. Contudo, este método não possui uma boa justificativa probabilística já que considera que cada sequência descende das $N-1$ outras sequências e não de uma ancestral comum [3]. Isso acarreta na contagem

	-	C	S	T	P	A	G	N	D	E	Q	H	R	K	M	I	L	V	F	Y	W
-	0	12	12	12	12	12	12	12	12	12	12	12	12	12	12	12	12	12	12	12	12
C	12	05	17	19	20	19	20	21	22	22	22	20	21	22	22	19	23	19	21	17	25
S	12	17	14	16	16	16	16	16	17	17	18	18	17	17	19	18	20	18	20	20	19
T	12	19	16	14	17	16	17	17	17	17	18	18	18	17	18	17	19	17	20	20	22
P	12	20	16	17	11	16	18	18	18	18	17	17	17	18	19	19	20	18	22	22	23
A	12	19	16	16	16	15	16	17	17	17	17	18	19	18	18	18	19	17	21	20	23
G	12	20	16	17	18	16	12	17	16	17	18	19	20	19	20	20	21	18	22	22	24
N	12	21	16	17	18	17	17	15	15	16	16	15	17	16	19	19	20	19	21	19	21
D	12	22	17	17	18	17	16	15	13	14	15	16	18	17	20	19	21	19	23	21	24
E	12	22	17	17	18	17	17	16	14	13	15	16	18	17	19	19	20	19	22	21	24
Q	12	22	18	18	17	17	18	16	15	15	13	14	16	16	18	19	19	19	22	21	22
H	12	20	18	18	17	18	19	15	16	16	14	16	15	17	19	19	19	19	19	17	20
R	12	21	17	18	17	19	20	17	18	18	16	15	11	14	17	19	20	19	21	21	15
K	12	22	17	17	18	18	19	16	17	17	16	17	14	12	17	19	20	19	22	21	20
M	12	22	19	18	19	18	20	19	20	19	18	19	17	17	11	15	13	15	17	19	21
I	12	19	18	17	19	18	20	19	19	19	19	19	19	19	15	12	15	13	16	18	22
L	12	23	20	19	20	19	21	20	21	20	19	19	20	20	13	15	11	15	15	18	19
V	12	19	18	17	18	17	18	19	19	19	19	19	19	19	15	13	15	13	18	19	23
F	12	21	20	20	22	21	22	21	23	22	22	19	21	22	17	16	15	18	8	10	17
Y	12	17	20	20	22	20	22	19	21	21	21	17	21	21	19	18	18	19	10	07	17
W	12	25	19	22	23	23	24	21	24	24	22	20	15	20	21	22	19	23	17	17	00

Figura 2.3: Matriz de substituição PAM250 (Fonte: [15]).

múltipla de eventos evolucionários, agravando o problema com o crescimento do número de sequências consideradas. O modelo de *Weighted Sum-of-pairs* [17] foi proposto para levar em consideração essas limitações.

2.2.2 Weighted Sum-of-pairs (WSP)

Altschul, Carroll e Lipman [17] propuseram o uso de pesos no somatório dos valores de escore dos alinhamentos entre pares para que as distâncias evolucionárias entre as sequências permitam a geração de um alinhamento que melhor reflita o histórico evolucionário das sequências. Assim, o valor do escore de um alinhamento é dado pela Equação 2.3 [17]. Os pesos são atribuídos com base na distância entre as sequências em uma árvore evolucionária.

$$WSP = \sum_{i<j} w(i,j) \cdot PA(S_i, S_j) \tag{2.3}$$

Nesta equação Equação 2.3, S_i e S_j são as sequências no alinhamento e $w(i,j)$ o peso atribuído a este par.

2.3 Alinhamento múltiplo exato de sequências

2.3.1 Algoritmo exato simples

O problema do alinhamento múltiplo de sequências pode ser resolvido por um algoritmo exato conforme mostrado a seguir para um alinhamento de 3 sequências, podendo ser expandido para o número de sequências desejado [3].

Sejam S_1, S_2 e S_3 sequências de tamanhos n_1, n_2 e n_3, respectivamente, e $D(i,j,k)$ o escore do alinhamento ótimo usando SP (Seção 2.2.1) para as sequências $S_1[1,..,i]$, $S_2[1,..,j]$, $S_3[1,..,k]$. Considere que o escore para *match*, *mismatch* e *gap* são *smatch*, *smsis* e *sspace*, respectivamente. O algoritmo para calcular o alinhamento múltiplo é mostrado no Algoritmo 1 [1].

O alinhamento ótimo das três sequências usa uma matriz tridimensional de programação dinâmica em que cada célula (i,j,k), tal que $i \neq 0$, $j \neq 0$ e $k \neq 0$ e $i < n_1$,$j < n_2$ e $k < n_3$ deve considerar os 7 vizinhos anteriores para determinar o valor de $D(i,j,k)$. Para as demais células, sendo $D'_{1,2}(i,j)$ o valor de SP para as subsequências $S_1[1,..,i]$ e $S_2[1,..,j]$, define-se $D'_{1,3}(i,j)$ e $D'_{2,3}(i,j)$ por analogia [14], obtendo-se:

$$\begin{aligned} D(i,j,0) &= D'_{1,2}(i,j) + (i+j) + sspace \\ D(i,0,k) &= D'_{1,3}(i,k) + (i+k) + sspace \\ D(0,j,k) &= D'_{2,3}(j,k) + (j+k) + sspace \\ D(0,0,0) &= 0 \end{aligned} \tag{2.4}$$

Algoritmo 1 ExatoSimples(S_1, S_2, S_3)

```
1: for i = 1 → n1 do
2:     for j = 1 → n2 do
3:         for k = 1 → n3 do
4:             c_ij = score(S1, S2)
5:             c_ik = score(S1, S3)
6:             c_jk = score(S2, S3)
7:             d1 = D(i − 1, j − 1, k − 1) + c_ij + c_ik + c_jk
8:             d2 = D(i − 1, j − 1, k) + c_ij + sspace
9:             d3 = D(i − 1, j, k − 1) + c_ik + sspace
10:            d4 = D(i, j − 1, k − 1) + c_jk + sspace
11:            d5 = D(i − 1, j, k) + 2 ∗ sspace
12:            d6 = D(i, j − 1, k) + 2 ∗ sspace
13:            d7 = D(i, j, k − 1) + 2 ∗ sspace
14:
15:            D(i, j, k) = Min[d1, d2, d3, d4, d5, d6, d7]
16:        end for
17:    end for
18: end for
```

O Algoritmo 1 assume a inicialização em 0 conforme as Equações 2.4. Para cada sequência um laço é adicionado, nesse caso possuindo 3 laços. Uma vez dentro do laço, inicia-se o cálculo pelos *matches* e *mismatches*, linhas 4 a 20, e em seguida são consultadas as 7 células vizinhas, linhas 22 a 28. Ao final, $D(i,j,k)$ recebe o menor desses valores.

Considerando N sequências de tamanho L, o algoritmo constrói um matriz N-dimensional e exige um tempo de $\Theta(L^N)$ [14].

Esse método de programação dinâmica é capaz de gerar o alinhamento múltiplo ótimo para qualquer conjunto de sequências, mas o custo computacional torna inviável a sua utilização para conjuntos com um número médio de sequências (maior que 30 sequências).

De fato, o alinhamento múltiplo de sequências utilizando SP já foi provado como sendo NP-Completo [2].

2.3.2 Carrilo-Lipman

Carrilo e Lipman [18] mostraram a possibilidade de se limitar o espaço de busca onde se encontra o alinhamento ótimo usando limites que permitem que algumas células da matriz de programação dinâmica não sejam calculadas. Isso é feito utilizando-se o alinhamento exato dos pares de sequências e um escore gerado por um método de alinhamento múltiplo heurístico.

A soma dos escores dos alinhamentos exatos entre pares de todas as sequências é evidentemente sempre melhor ou igual à soma dos escores dos alinhamentos entre as sequências no alinhamento múltiplo, assim servindo como um limite superior. Já o escore obtido pelo método heurístico, apesar de não garantir a obtenção de um alinhamento ótimo, pode ser usado para definir um limite inferior para o alinhamento [1], isto é, o valor do escore do alinhamento obtido pelo método heurístico é sempre pior ou igual ao valor do escore do alinhamento ótimo.

2.3.3 A-STAR (A*)

O alinhamento múltiplo ótimo de sequências é equivalente à busca pelo melhor caminho em um grafo [18]. Essa equivalência pode ser observada da seguinte maneira: cada sequência a ser alinhada é considerada como um lado de um objeto multi-dimensional e todas compartilham uma origem em comum, o que gera um hipercubo de N-dimensões, onde N é o número de sequências a serem alinhadas. A Figura 2.4 ilustra um exemplo com 3 sequências.

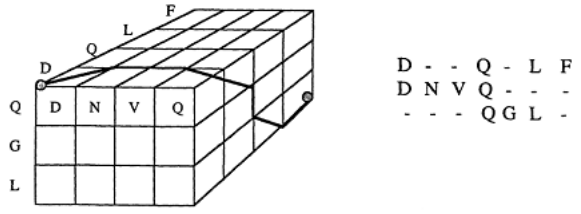

Figura 2.4: Alinhamento entre 3 sequências (Fonte: [18]).

No espaço delimitado dentro do hipercubo, um movimento no espaço N-dimensional que avance em uma dimensão representa a adição do caracter da sequência correspondente ao alinhamento e um movimento que não altere a posição em uma dada dimensão representa a inserção de um *gap*. Dessa forma, um caminho no espaço N-dimensional que

saia da origem e termine no vértice oposto à origem, chamado vértice final, equivale a um alinhamento que possui todos os resíduos de todas as sequências. Assim, o alinhamento ótimo é equivalente a encontrar o melhor caminho entre a origem e o vértice final, o que nesse caso é o alinhamento com o maior escore.

Explorando essa possibilidade, Spouge [19] propôs em 1989 o uso do algoritmo A* [20] para obter o alinhamento múltiplo de sequências. O A* é um algoritmo de busca de caminho em grafo do tipo *Best-first search* (BFS) que obtêm o caminho de melhor custo entre dois vértices utilizando heurísticas para limitar o espaço de busca. O A* utiliza a Equação 2.5 para definir qual o próximo nó a ser visitado.

$$f(N) = g(N) + h(N) \tag{2.5}$$

Nessa equação, $g(N)$ é custo do caminho da origem até o nó N e $h(N)$ é uma estimativa do custo do caminho do nó N até o final. O algoritmo então utiliza duas listas, uma para os nós já visitados, chamada *ClosedList*, inicialmente vazia, e uma para os nós que podem ser visitados a seguir, chamada *OpenList*, que começa contendo apenas o nó origem.

O A* funciona em três etapas [3]:

- Na etapa *Find*, identifica-se o nó N_k da *OpenList* com melhor valor de $f(N)$.
- Na etapa *Stop*, caso N_k seja o nó final, o algoritmo encerra sua execução e retorna o alinhamento.
- Na etapa *Expand and reconcile*, o nó N_k é retirado da *OpenList* e adicionado à *ClosedList*. Os nós vizinhos à N_k, caso não estejam na *ClosedList*, são adicionados à *OpenList*, caso contrário são adicionados à *OpenList* apenas se o valor $g(N)$ for melhor que o valor quando foram adicionados à *ClosedList*. Por fim, retorna-se para a etapa de *Find*.

Esse algoritmo garante a obtenção do resultado ótimo desde que a heurística $h(n)$ nunca superestime o valor da distância real até o nó final (heurística admissível). Spouge [19] demonstrou que o limite inferior do método Carrillo-Lipman [18], $h_{2,all}$, é uma heurística admissível para o A*.

O A* é capaz de obter o alinhamento ótimo de sequências sem visitar todo o espaço de busca mas mesmo isso não é suficiente para tornar viável o alinhamento de grandes números de sequências, o que é de grande interesse para biólogos. Devido a essa limitação foram propostos algoritmos heurísticos para tentar obter bons resultados em um tempo viável.

2.4 Métodos Heurísticos

O alinhamento múltiplo de um grande número de sequências é de grande interesse pois permite a identificação de similaridades sutis ou dispersas entre as sequências. Contudo, como o alinhamento múltiplo de sequências utilizando SP foi provado NP-Completo [2], os métodos exatos não são viáveis nem para um número médio de sequências (maior que 30). Sendo assim, diversos métodos heurísticos foram propostos para tentar obter um alinhamento bom em tempo hábil.

Existem basicamente dois métodos para a obtenção de alinhamentos múltiplos heurísticos, o progressivo e o por refinamento iterativo, que são descritos nas seções Seções 2.4.1 e 2.4.2.

2.4.1 Alinhamento progressivo

2.4.1.1 Visão Geral

O alinhamento progressivo é um dos métodos heurísticos mais utilizados no alinhamento múltiplo de sequências. O alinhamento progressivo é construído iterativamente, alinhando-se uma sequência a um alinhamento já conhecido [3]. Diversos métodos de alinhamento progressivo foram propostos, com alguns aspectos particulares diferenciando-os: (a) a ordem com que selecionam as sequências a serem alinhadas, (b) a decisão de alinhar uma sequência a um alinhamento emergente ou gerar diversos alinhamentos e eventualmente alinhá-los entre si e (c) a forma como é feito o cálculo do escore.

Um dos métodos de alinhamento heurístico mais populares foi proposto por Feng e Doolittle [21] e serviu de base vários métodos de alinhamento progressivos posteriores como o Clustal [22]. Esse método consiste de três etapas relacionadas a seguir.

1. Cálculo de uma matriz de similaridade, contendo um valor representando as distâncias das sequências entre si. Os valores de distância geralmente são calculados a partir dos escores dos alinhamentos entre pares das sequências.
2. Construção de uma árvore-guia a partir da matriz de similaridade utilizando um algoritmo de clusterização.
3. Construção do alinhamento seguindo a ordem determinada pela árvore-guia.

Para o alinhamento entre sequências (etapa 1) são geralmente alinhados todos os pares de sequências ou somente algumas sequências são alinhadas a todas as demais. Nesse último caso, pode escolher uma única sequência e alinhá-las com todas as outras (*center-star* [3]) ou um subconjunto de sequências e alinhá-las par-a-par com todas as outras. Para

alinhar duas sequências, pode-se usar os algoritmos de programação dinâmica Needleman-Wunsh [23], Smith-Waterman [24], *Longest Common Subsequence [25]*, ou utilizar um método de contagem de sub-sequências em comum, como o *k-mers* [14].

Existem diversos algoritmos para se construir a árvore-guia a partir da matriz de similaridades (etapa 2), sendo o UPGMA (*Unweighted Pair Group Method and Arithmetic Mean*) [26] um dos mais utilizados. O UPGMA é um algoritmo de clusterização hierárquico aglomerativo (*bottom-up*). Inicialmente o UPGMA define cada sequência como um *cluster* em si e, a cada iteração do algoritmo, combina os dois *clusters* mais próximos em um *cluster* de nível mais alto. A distância entre os *clusters* é dada pela média das distâncias entre todos os elementos de cada *cluster*.

A terceira etapa do alinhamento progressivo pode ser realizada de diversas maneiras. Quando as sequências são alinhadas segundo a ordem da árvore-guia, pode ser necessário realizar o alinhamento de uma sequência a um grupo de sequências, o que é obtido realizando o alinhamento entre pares da nova sequência com todas as sequências já dentro do grupo e selecionando o alinhamento com escore mais alto. Também pode ser preciso realizar o alinhamento de dois grupos de sequências, o que é obtido realizando o alinhamento entre pares de todos os possíveis pares de sequências entre os dois grupos e selecionando aquele com o escore mais alto [3]. Outra possibilidade para a terceira etapa é utilizar o alinhamento com perfis, que consideram informações estatísticas dos caracteres encontrados em cada posição das sequências de um alinhamento, penalizando alinhamentos que possuam caracteres diferentes em regiões conservadas e diminuindo o custo de *gap* em regiões onde o *gap* é mais comum [3].

Existem dois principais problemas com a abordagem progressiva [3]: o problema do mínimo local e a escolha de parâmetros de alinhamento. O problema do mínimo local existe devido ao aspecto guloso do algoritmo, que de maneira gulosa adiciona as sequências mais próximas seguindo a árvore filogenética ao alinhamento, sem que existam garantias de que um resultado global ótimo seja atingido.

O problema da escolha de parâmetros advêm dos valores escolhidos para se qualificar um alinhamento. Em geral, o escore de um alinhamento entre proteínas é calculado utilizando-se uma matriz de substituição como PAM250 ou BLOSUM62 (Seção 2.2.1), e dois valores de custo para *gaps*, um para criar um novo *gap* e outro para estender um já existente (*affine gap*).

Essa abordagem funciona muito bem para sequências próximas por dois motivos. Em primeiro lugar, as matrizes obtidas na etapa 1 dão mais peso para identidades, o que permite que o resultado obtido seja bom para conjuntos próximos. Entretanto, em conjuntos de sequências muito divergentes, existirão muito mais divergências que identidades. Logo, os pesos atribuídos às divergências ganham muito mais relevância, com matrizes

distintas sendo mais adequadas para diferentes distâncias. O segundo motivo é que para sequências bastante divergentes o valor de *gap* é muito mais relevante para obtenção de um bom resultado do que em conjunto mais próximos [27].

2.4.1.2 Clustal W

O Clustal W [27] é um algoritmo de alinhamento progressivo muito popular e que evoluiu bastante desde sua proposta inicial. A primeira versão, chamada Clustal [22], utilizava uma versão modificada do algoritmo de Feng e Doolittle [21], em que a terceira etapa do alinhamento das sequências é feita utilizando um método de alinhamento de sequência-perfil e de alinhamento perfil-perfil [3].

O Clustal W trouxe algumas modificações que aprimoraram a acurácia do alinhamento obtido sem sacrificar a velocidade de execução. A primeira foi adição do uso de *affine-gap* no alinhamento entre pares feito na primeira etapa para construção da matriz de similaridade. Outra alteração foi na construção da árvore-guia utilizando o algoritmo de *neighbor-joining* [28] ao invés do UPGMA [26].

As mudanças mais significativas estão relacionadas abaixo:

- O uso do WSP (Seção 2.2.2) ao invés do SP (Seção 2.2.1) como função de escore.
- Foi adicionado ao algoritmo um mecanismo que escolhe entre duas matrizes de substituição dependendo na distância estimada entre duas sequências, utilizando uma matriz *hard*, que possui uma punição maior para diferenças, para sequências próximas, como a Blosum80 [16], e uma matriz *soft*, que é mais branda com diferenças, para sequências distantes, como a Blosum50 [16].
- A penalidade de *gap-open* no alinhamento de perfis recebeu um multiplicador dependente dos caracteres observados naquela posição. Essa penalidade foi obtida com base em *gaps* observados em alinhamentos estruturais [3].
- A penalidade por *gap* é amenizada se for gerada em um trecho com 5 ou mais caracteres que sejam hidrofílicos.
- As penalidades de *gap-open* e *gap-extend* são aumentadas caso não exista *gap* em uma coluna mas que exista um próximo do perfil. Isso visa forçar que *gaps* estejam no mesmo lugar no alinhamento.
- Se, durante o alinhamento progressivo, um escore de um alinhamento for muito baixo, a árvore-guia pode ser dinamicamente ajustada para postergar este alinhamento até que o perfil possua mais informação [3].

2.4.1.3 MAFFT - FFT-NS-[1/2]

O MAFFT [8] é um pacote de programas que oferece diversos métodos de alinhamento múltiplo, entre eles, os métodos FFT-NS. Esses métodos progressivos funcionam de maneira similar ao Clustal (Seção 2.4.1.2), contudo a construção do alinhamento final é feito utilizando FFT (*Fast Fourier Transform*). O método de cálculo de escore também foi modificado visando reduzir o tempo necessário à obtenção do alinhamento [8].

Construção do alinhamento usando FFT Para o alinhamento de proteínas, o MAFFT representa cada aminoácido a de uma sequência como um vetor constituído com o valor normalizado de seu volume e polaridade, calculados da seguinte maneira [8]: $\hat{v}(a) = [v(a) - \bar{v}(a)]/\sigma_v$ e $\hat{p}(a) = [p(a) - \bar{p}(a)]/\sigma_p$, em que $\bar{v}(a)$ e $\bar{p}(a)$ são os valores médios entre os 20 aminoácidos (Seção 2.1.1) e σ_v e σ_p são o desvio padrão do volume e polaridade respectivamente [8]. Desta forma, uma sequência de aminoácidos é descrita como uma sequência de vetores.

Os valores obtidos acima são utilizados para calcular a correlação $c(k)$ entre sequências conforme mostrado na Equação 2.6, que representa o grau de similaridade entre duas sequências com respeito a um deslocamento entre os seus inícios de k posições. Um alto valor de $c(k)$ pode indicar que as sequências possuem regiões homólogas [8].

$$c(k) = c_v(k) + c_p(k) \tag{2.6}$$

Na Equação 2.6, $c_v(k)$ e $c_p(k)$ são a correlação de volume e polaridade entre duas sequências, respectivamente, e são calculadas conforme mostrado na Equação 2.7, onde $\hat{v}_1(n)$ é o componente volume da sequência 1 de tamanho N na n-ésima posição e $\hat{v}_2(n)$ o mesmo componente para a sequência 2 de tamanho M.

$$c_v(k) = \sum_{1 \leq n \leq N, 1 \leq n+k \leq M} \hat{v}_1(n)\hat{v}_2(n+k) \tag{2.7}$$

Como em muitos casos $N \simeq M$, a complexidade de tempo é de $O(N^2)$. O uso da FFT permite resolver essa equação em $O(N \log N)$ [29]. Sendo assim, dados $V_1(m)$ e $V_2(m)$, que são as transformadas de Fourrier para $\hat{v}_1(n)$ e $\hat{v}_2(sn)$ (Equação 2.8 e Equação 2.9 respectivamente).

$$\hat{v}_1(n) \Leftrightarrow V_1(m) \tag{2.8}$$

$$\hat{v}_2(n) \Leftrightarrow V_2(m) \tag{2.9}$$

Pode-se expressar a correlação $c_v(k)$ como mostrada na Equação 2.10.

$$c_v(k) \Leftrightarrow V_1^*(m) \cdot V_2(m) \tag{2.10}$$

Nessa equação, $\Leftrightarrow$ representa o par da transformada e $*$ representa o conjugado complexo. O cálculo da correlação da polaridade é feito de maneira análoga.

A análise da FFT permite então encontrar regiões homólogas das sequências. A presença de picos na correlação $c(k)$ pode indicar a presença dessas regiões, conforme mostrado na Figura 2.5.

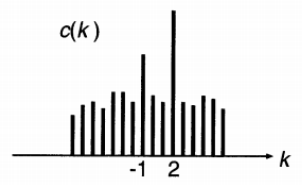

Figura 2.5: Resultado da análise da FFT, possuindo 2 picos correspondentes a regiões homólogas (Fonte: [8]).

Se a análise da FFT indicar a existência de regiões homólogas, é feita uma análise utilizando uma janela deslizante com 30 caracteres, conforme ilustrado na Figura 2.6. Calcula-se inicialmente o grau de homologia local dos 20 maiores picos na correlação $c(k)$. Segmentos de comprimento 30 cujo valor de escore ultrapasse um *threshold* determinado são considerados segmentos homólogos. Se segmentos homólogos forem adjacentes uns aos outros eles são combinados em um segmento único de comprimento até 150 caracteres. Caso o segmento contíguo ultrapasse 150 aminoácidos, ele é dividido.

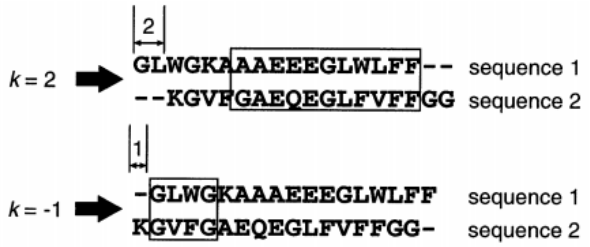

Figura 2.6: Análise utilizando *sliding window* (Fonte: [8]).

O alinhamento é então obtido organizando-se os segmentos homólogos de maneira consistente em ambas as sequências. Para tal, utiliza-se uma matriz $S_{ij}(1 \leq i, j \leq n)$, onde n é o número de segmentos homólogos, criada da seguinte maneira: se o i-ésimo segmento homólogo da primeira sequência corresponde ao j-ésimo segmento da segunda, então $S_{i,j}$ possui como valor o escore do segmento, caso contrário o valor é 0. Utiliza-se então um algoritmo de programação dinâmica (DP) para identificar o arranjo ótimo de

segmentos. A Figura 2.7 ilustra um exemplo com 5 segmentos, sendo que a ordem dos segmentos é diferente entre as sequências. O caminho ótimo depende então dos valores de S_{23} e S_{32}.

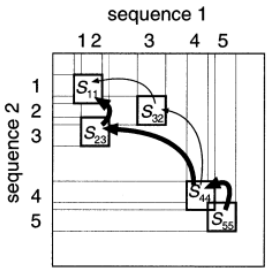

Figura 2.7: Exemplo de DP sobre os segmentos (Fonte: [8]).

O algoritmo de programação dinâmica divide a matriz de homologia em sub-matrizes limitadas pelas posições contendo os valores dos segmentos homólogos, dessa forma restringindo o espaço de busca do problema, conforme ilustrado na Figura 2.8.

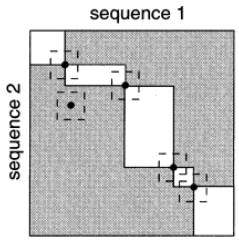

Figura 2.8: Espaço de busca da DP, área em cinza não é visitada. Cada ponto representa a posição de um segmento homólogo (Fonte: [8]).

Esse é o procedimento para o alinhamento entre um par de sequências, que foi expandido para realizar o alinhamento entre múltiplas sequências. Para tal é preciso expandir as equações de correlação de volume e de polaridade substituindo-se os valores de $\hat{v}_1(n)$ por $\hat{v}_{group1}(n)$ e de $\hat{p}_1(n)$ por $\hat{p}_{group1}(n)$, definidos nas Equações 2.11 e 2.12 [8].

$$\hat{v}_{group1}(n) = \sum_{i \in group1} w_i \cdot \hat{v}_i(n) \tag{2.11}$$

$$\hat{p}_{group1}(n) = \sum_{i \in group1} w_i \cdot \hat{p}_i(n) \tag{2.12}$$

O valores dos pesos w_i de cada sequência i são calculados da mesma forma que o Clustal W (Seção 2.4.1.2).

Esse método também pode ser aplicado sobre sequências de nucleotídeos convertendo cada aminoácido em um vetor de quatro dimensões cujas componentes são a frequência de A, T, G e C em cada coluna, ao invés de um vetor de volume e polaridade [8]. Assim, a correlação entre sequências é dada pela Equação 2.13.

$$c(k) = c_A(k) + c_T(k) + c_G(k) + c_C(k) \tag{2.13}$$

Método de *scoring* Visando aumentar a eficiência do alinhamento, o MAFFT utilizou um método de cálculo de escore próprio, utilizando uma matriz de substituição e uma penalidade por *gap* modificadas. Uma matriz de substituição normalizada, possuindo valores positivos e negativos é gerada utilizando a Equação 2.14 [8], sendo $average1 = \sum_a f_a M_{aa}$, $average2 = \sum_{a,b} f_a f_b M_{ab}$, M_{ab} uma matriz de substituição não alterada, f_a a frequência da ocorrência do aminoácido a e S^a o valor da penalidade por *gap extension*. Por padrão M_{ab} é a matriz de substituição PAM200 e f_a é a frequência de aminoácidos a, ambos de [30].

$$\hat{M}_{ab} = [(M_{ab} - average2)/(average1 - average2)] + S^a \tag{2.14}$$

2.4.1.4 MSAProbs

O MSAProbs [31] é um algoritmo de alinhamento progressivo de proteínas que utiliza *pair-hmm* - modelo escondido de Markov (*Hidden Markov Model - HMM*) e uma função de partição para calcular a probabilidade à posteriori. Também utiliza as técnicas de *weighted probabilistic consistency transformation* e alinhamento perfil-perfil com pesos para obter um resultado de alta acurácia [31].

Uma matriz de probabilidades a posteriori (*posterior probability matrix*) é uma matriz na qual cada elemento $a_{i,j}$ representa a probabilidade dos caracteres i da primeira sequência e j da segunda sequência estarem alinhados no alinhamento entre ambas [31]. Para calcular essa matriz, um *pair-HMM* utiliza o algoritmo *Forward and Backward* [3]. Uma outra matriz de probabilidades é calculada utilizando uma função de partição, que cria alinhamentos sub-ótimos (alinhamentos com *escore* próximo ao do alinhamento ótimo) utilizando programação dinâmica. Essas duas matrizes são combinadas para formar a matriz de probabilidades final.

A matriz de probabilidades obtida dessa combinação é utilizada para obter o escore de um alinhamento global ótimo entra duas sequências que é então usado para determinar a distância entre as mesmas. Com a matriz de distâncias calculada, o MSAProbs utiliza o UPGMA (Seção 2.4.1.1) para gerar a árvore-guia. Com a árvore-guia criada, os pesos

são atribuídos às sequências utilizando o método de pesos do Clustal W (Seção 2.4.1.2). Esses pesos são então utilizados para atualizar os valores das matrizes de probabilidades.

Por fim, o MSAProbs realiza o alinhamento progressivo das sequências, seguindo a ordem da árvore-guia e utilizando alinhamento entre perfis considerando os pesos. As matrizes de probabilidades dos perfis são calculadas utilizando as matrizes de probabilidades das suas sequências.

2.4.2 Alinhamento por refinamento iterativo

2.4.2.1 Visão Geral

Algoritmos de alinhamento por refinamento iterativo visam reduzir de erros introduzidos no alinhamento progressivo [1]. Em geral, esses algoritmos geram um alinhamento inicial, como por exemplo, utilizando um método progressivo, e, iterativamente, uma sequência é retirada do alinhamento e realinhada às demais sequências que permanecem no alinhamento, até que o resultado desse novo alinhamento não mais ofereça um aprimoramento significativo sobre o anterior.

O método de Barton e Sternberg [32], descrito a seguir, possui 4 passos onde o terceiro passo é executado de maneira iterativa e é um bom exemplo de como métodos de refinamento iterativo funcionam para o alinhamento múltiplo de N sequências.

1. Encontrar as duas sequências com a maior similaridade entre elas e alinhá-las utilizando um algoritmo de alinhamento entre pares.
2. Encontrar a sequência mais similar ao perfil do alinhamento realizado no passo anterior e alinhá-la utilizando um algoritmo de alinhamento sequência-perfil. Repete-se esse passo até que todas as sequências tenham sido adicionadas.
3. Retira-se a sequência x_1 do alinhamento e realiza-se o alinhamento sequência-perfil dela com um perfil das sequências remanescentes. Repete-se esse passo para as demais $x_2, ..., x_n$ sequências.
4. Repete-se o passo 3 um número fixo de vezes ou até que os escores obtidos deixem de melhorar mais que um determinado valor.

Em geral, métodos de refinamento iterativo conseguem resultados mais acurados que os de métodos progressivos, mas para isso tem um aumento significativo no tempo de execução.

2.4.2.2 MAFFT - métodos iterativos

O MAFFT (Seção 2.4.1.3) oferece alguns métodos de alinhamento por refinamento iterativo visando aumentar a acurácia dos alinhamentos. Nessa seção, serão detalhados o FFT-NS-i e o L-INS-i.

O FFT-NS-i é um método iterativo do MAFFT [8] que, após obter um alinhamento inicial utilizando FFT-NS-2 (Seção 2.4.1.3), divide as sequências em dois grupos e as realinha utilizando um algoritmo de *tree-dependent restricted partinioning* [33], repetindo esse processo até que não haja mais melhoria no escore obtido. A qualidade dos alinhamentos obtidos é calculada utilizando WSP (Seção 2.2.2).

O L-INS-i é um método de alinhamento por refinamento iterativo que utiliza o algoritmo Smith-Waterman [24] com *affine gap* para realização dos alinhamentos entre pares na fase 1. Além disso, o escore de cada alinhamento é definido como a soma do WSP (Seção 2.2.2) entre elas com um escore calculado baseado em consistência [34].

2.4.3 MAFFT Parttree - Alinhamento de milhares de sequências

O grande crescimento dos bancos de dados genômicos torna cada vez mais interessante o alinhamento utilizando conjuntos muito grandes de sequências. Contudo, mesmo utilizando métodos heurísticos (Seção 2.4.1), os alinhamentos com mais de 10.000 sequências geralmente exigem um tempo de execução muito grande.

Alinhamentos com grandes quantidades de sequências são limitados principalmente por dois fatores: o alinhamento entre pares entre todas as sequências e a construção da árvore-guia. No intuito de tornar viável o alinhamento de dezenas de milhares de sequências, o MAFFT adicionou o método *Parttree* [35] ao seu repertório. O *Parttree* [5] é um algoritmo de *devisive clustering* que constrói uma árvore-guia a partir de um conjunto de sequências não alinhadas em $O(N \log N)$.

Para tanto, alinha inicialmente todas as sequências à sequência mais longa. A seguir, define um subconjunto de sequências como sequências de referência e as alinha par-a-par entre si, construindo uma árvore preliminar. Em seguida, alinha par-a-par as sequências de referência com todas as demais sequências (sequências base) e então agrupa cada sequência base à sequência mais similar a ela. Para cada grupo é então executado uma chamada recursiva do Parttre, onde o grupo é usado como conjunto de sequências, e a árvore resultante da chamada recursiva é armazenada. Por fim, as árvores das chamadas recursivas são combinadas com a árvore preliminar e a árvore resultante é retornada.

O MAFFT utilizando *Parttree* foi capaz de alinhar grandes quantidades de sequências homólogas (60.000) com uma perda na qualidade média de apenas 3% [5] quando comparado ao alinhamento *golden standard* fornecido pela base de dados PFAM [36].

2.5 Considerações Finais

No presente capítulo foram apresentados o que são sequências biológicas, o alinhamento múltiplo de sequências e os principais métodos de cálculo do MSA heurístico.

O alinhamento múltiplo de sequências visa identificar o regiões de similaridades entre as sequências. Por se tratar de um problema NP-Completo, métodos heurísticos são utilizados para obter resultados em tempo hábil. Dois métodos de MSA heurísticos são utilizados, o progressivo e o por refinamento iterativo.

Diversos algoritmos de MSA heurísticos foram desenvolvidos. O Clustal W, o FFT-NS-2 e o MSAProbs são exemplos de métodos progressivos. O FFT-NS-i e o L-INS-i são exemplos de métodos por refinamento iterativo. Esses algoritmos calculam as distâncias entre todos os pares de sequências ($O(N^2)$), o que limita o seu uso para conjuntos com dezenas de milhares de sequências. O Parttree é um algoritmo que combina o cálculo de distâncias com a construção da árvore guia, utilizando uma abordagem recursiva que compara apenas algumas sequências de referência a todas as demais ($O(N \log N)$), reduzindo a complexidade do cálculo de distâncias.

Capítulo 3

Unidades de Processamento Gráfico(GPUs)

Durante várias décadas a evolução no desempenho de computadores foi definida pela aceleração do *clock* das CPUs (*Central Processing Unit*), o que permitia que mais operações fossem executadas no mesmo intervalo de tempo, impactando diretamente nos tempos de execução de programas. Essa abordagem contudo atingiu um limite por volta de 2003 devido ao alto consumo de energia e a dificuldade na dissipação de calor nos processadores [37].

Desde então, os fabricantes escolheram adotar o modelo *multicore*, em que vários processadores (*cores*) são utilizados em cada CPU para aumentar a capacidade de processamento. Esta abordagem, no entanto, diferentemente da aceleração do *clock*, que representava um ganho direto de desempenho sobre o código já existente, demanda que os programas sejam escritos explicitamente de forma paralela para obter todo o potencial da CPU [38].

Duas abordagens se destacaram no *design* de processamento paralelo: *multicore* e *manycore*. *Multicores* empregam *cores* complexos, com paralelismo a nível de instrução (*Instruction Level Parallelism* - ILP) sofisticado, visando reduzir bastante o tempo de execução de programas sequenciais. Lançados inicialmente com dois *cores*, hoje já existem processadores como o Intel Xeon W-3175X com até 28 *cores* e o IBM Power 9, com 22 *cores*.

A abordagem *manycore*, por outro lado, visa aumentar o *throughput* de aplicações paralelas [37] e por isso utiliza uma quantidade muito maior de *cores* para executar mais operações simultaneamente. Para tal, arquiteturas *manycore* empregam *cores* mais simples, com ILP limitado.

As GPUs (*Graphics Processing Units*) são exemplos de arquiteturas *manycore*. Tais arquiteturas podem possuir mais de 5.000 cores, como a placa NVidia Tesla V100, que

possui 5376 CUDA cores e que atinge até 15, 7 TFlops precisão simples [39], com desempenho teórico muito superior aos *multicores*. Essa capacidade tem feito cada vez mais desenvolvedores a utilizarem GPUs em aplicações que demandem um grande quantidades de cálculos [37].

A diferença de desempenho teórico entre CPUs e GPUs é resultado das diferentes filosofias de *design* empregadas em cada tipo de dispositivo. As CPUs são otimizadas para a execução de código sequencial, utilizando uma lógica de controle complexa para permitir que instruções de uma mesma *thread* sejam executadas em paralelo ou mesmo fora de ordem. Além disso, as CPUs possuem diversos níveis de memória *cache* com o intuito de minimizar o tempo de execução de cada *thread*. As GPUs, por outro lado, possuem lógica de controle mais simples e menos memória *cache*, o que permite que mais unidades de cálculo sejam instaladas na mesma área de *chip* quando comparada à CPU. A Figura 3.1 ilustra a diferença no *design* de CPUs e GPUs, onde é possível notar que a área dedicada à computação é maior nas GPUs do que nas CPUs.

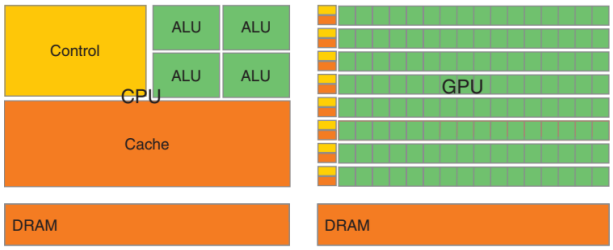

Figura 3.1: Comparativo do design de CPU e GPU (Fonte: [37]).

Inicialmente, para que programas pudessem utilizar a capacidade das GPUs era necessário que as computações fossem escritas como um problema de renderização. Esse modelo de programação impunha diversas restrições que tornavam o uso de GPUs mais difícil. Com o intuito de tornar mais acessível o uso de GPUs para outros tipos de problemas, foram propostas as GPGPUs (*General Purpose GPUs*), que podem ser usadas para a programação de aplicações genéricas, não estando limitadas à aplicações de renderização. Nessa linha, a NVIDIA propôs a arquitetura CUDA (*Compute Unified Device Architecture*), implementada inicialmente na placa GeForce 8800 GTX [40].

3.1 Arquiteturas das GPUs NVidia

A arquitetura das GPUs modernas consiste, geralmente, em um conjunto de *Streaming Multiprocessors* (SMs). Cada SM possui múltiplos *Streaming processors* (SPs), também conhecidos como CUDA *cores*, que compartilham a lógica de controle e uma cache de instruções. As GPUs possuem uma memória principal utilizada como memória global [37],

além de outros níveis de hierarquia de memória. A Figura 3.2 ilustra uma visão geral da arquitetura de uma GPU com 8 SMs, cada um com uma cache própria e 16 SPs.

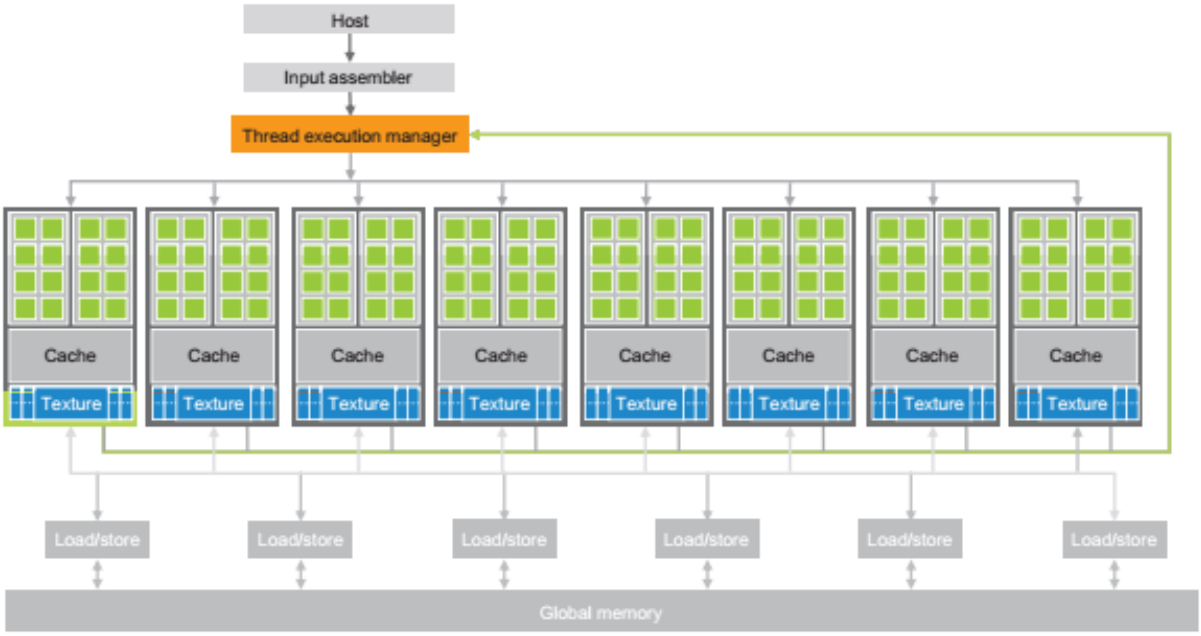

Figura 3.2: Visão geral de arquitetura de um GPU (Fonte: [37]).

A arquitetura de GPUs visa otimizar a paralelização de dados, ou seja, aplicar a mesma instrução a dados diferentes, sendo classificadas como SIMD (*Single Instruction Multiple Data*) na taxonomia de Flynn [41].

3.2 Evolução das GPUs da NVidia

A seguir são apresentadas em ordem cronológica as arquiteturas das placas gráficas NVIDIA: Kepler, Maxwell e Volta.

A arquitetura Kepler [43] foi proposta em 2012 e é uma evolução direta da Fermi, possui 4 GPCs (*Graphics Processing Clusters*), cada um com 2 SMXs (*Stream Multiprocessor*) com 192 *cores*, 65.536 registradores de 32 bits, 32 SFUs, 32 unidades de *Load/Store* e 16 unidades de textura cada, resultando em um total de 1.536 *cores*. A cache L1 configurável pode ser distribuída como 16KB/48KB, 32KB/32KB ou 48KB/16KB e cache L2 foi ampliada para 1536KB. A Figura 3.3 ilustra um placa GTX980 da arquitetura Maxwell.

A arquitetura Maxwell [44] foi proposta em 2014. Possui 16 SMMs (Maxwell SMs), cada um com 128 *cores* e 8 unidades de textura. A cache L2 foi aumentada para 2048KB. A Maxwell separou a cache L1 da memória compartilhada, dispondo de 96KB de memória

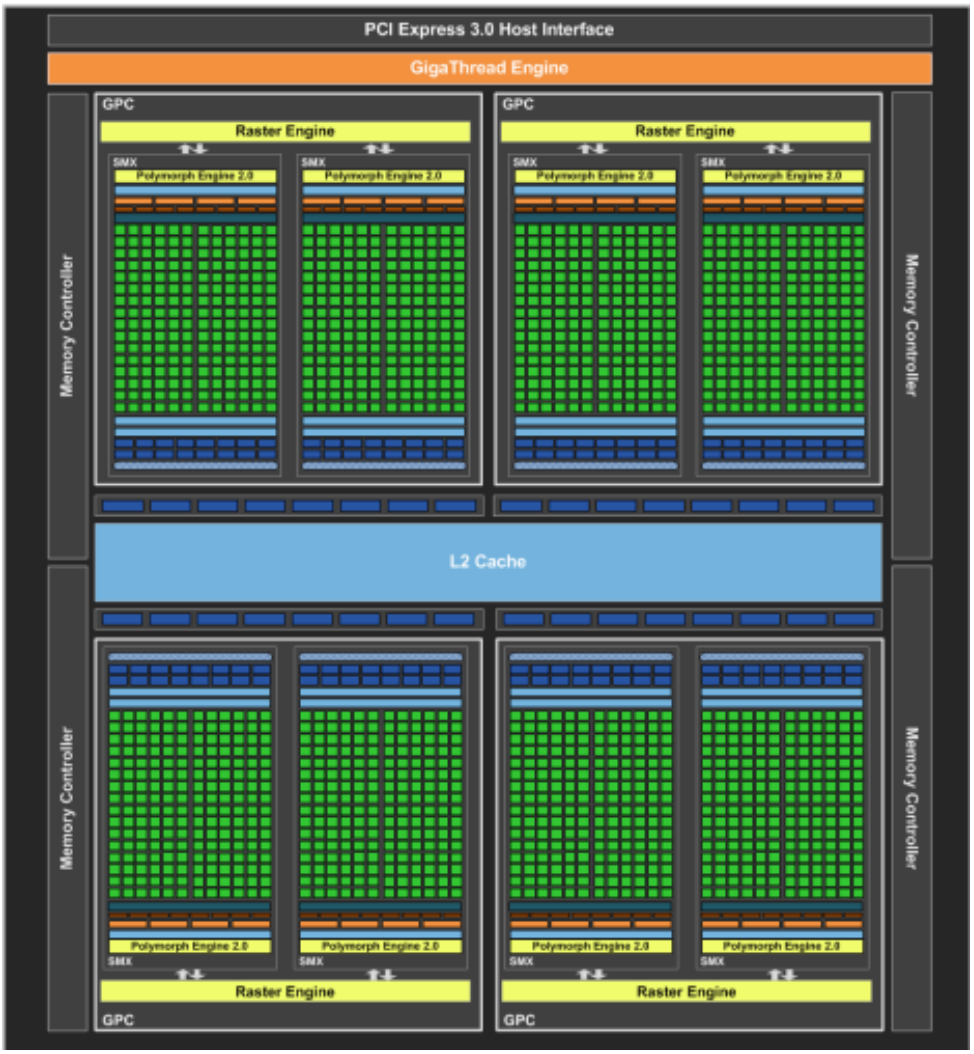

Figura 3.3: Placa NVidia GTX 680 da arquitetuta Kepler com 1.536 CUDA *cores* (Fonte: [43]).

compartilhada e a cache L1 foi unificada com memória de textura. A Figura 3.4 ilustra um placa GTX980 da arquitetura Maxwell.

Após a Maxwell, foi proposta a arquitetura Pascal [45] em 2016 utilizando um placa de memória HBM2 (*High Bandwidth Memory*), que usa uma técnica de empilhamento em 3D, permitindo que mais memória seja alocada em uma área menor de *chip* e com uma largura de banda maior. No caso da Pascal é utilizada uma placa com 16GB de memória com largura de banda de 720 GB/s, possuindo 56 SMs, cada um com 64 *cores* de ponto flutuante precisão simples, 32 de precisão dupla, 4 unidades de textura e acesso a 64 KB de memória compartilhada. A cache L2 possui um tamanho total de 4096 KB.

A arquitetura Volta [39] é uma das mais recentes arquiteturas NVidia. Foi proposta em 2017 com o objetivo de atender os mercados de supercomputadores, *data centers* e

Figura 3.4: Placa NVidia GTX 980 da arquitetuta Maxwell com 2.048 CUDA *cores* (Fonte: [44]).

aplicações de *deep learning*. A Volta utiliza uma memória HBM2 de 16GB e uma largura de banda de 900 GB/s. A Volta possui 84 SMs, cada um com 64 *cores* para cálculo de ponto flutuante simples, 32 *cores* de ponto flutuante precisão dupla, 64 *cores* para inteiros, 8 *tensor cores*, um tipo de *core* específico para o treinamento de redes neurais, e 4 unidades de textura. Ao todo, são 5376 *cores* para operações de ponto flutuante, atingindo 7,8 Tflops (precisão dupla). Além disso, a cache L2 possui 6144 KB. A cache L1 foi combinada com a memória compartilhada, permitindo a configuração da distribuição, com uma capacidade de 128KB por SM.

3.3 Arquitetura CUDA

A arquitetura CUDA (*Compute Unified Device Architecture*), apresentada em 2006 [40], visava tornar mais acessível o uso de GPUs para computações genéricas (não relacionadas à aplicações gráficas). Ela introduziu um novo modelo genérico de programação paralela

com hierarquia de *threads*, sincronização utilizando barreiras e operações atômicas para auxiliar no controle de programas com alto grau de paralelismo.

A arquitetura CUDA requer a existência de um *host* (CPU) e um ou mais *devices* (GPUs). Dado isso, um programa CUDA consiste de código para o *host*, que é escrito de maneira padrão, e do código de *device* que utiliza palavras-chave específicas para identificá-lo. A compilação de um programa CUDA é então feita em etapas, onde inicialmente o compilador CUDA, como o NVCC, é utilizado para separar o código de *host* do código de *device*. O código *host* é então compilado pelo compilador padrão do usuário e executado em CPU enquanto que o código de *device* é compilado pelo NVCC e executado em GPU. Uma função de *device* que possa ser invocada de código *host* é chamada de *kernel* [37]. A Figura 3.5 ilustra esse processo.

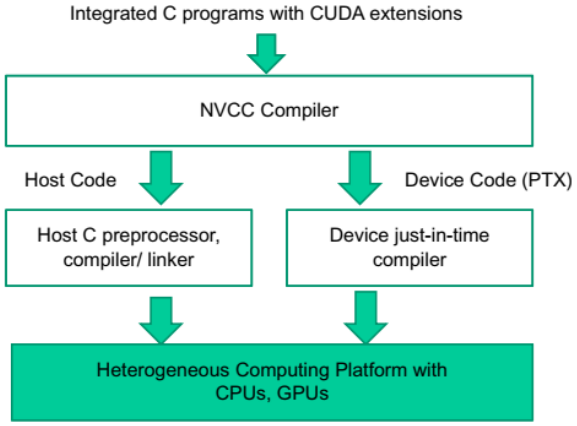

Figura 3.5: Compilação de código CUDA C (Fonte: [37]).

Quando um *kernel* é invocado, são geradas várias *threads* para executá-lo em vários *CUDA cores*. O conjunto de todas as *threads* é chamado de *grid*.

As *threads* em um *grid* são agrupadas de maneira hierárquica de forma a facilitar o seu controle. Cada *grid* é composto por um conjunto de blocos e cada bloco é composto por um conjunto de *threads*. Assim, para identificar uma *thread* é preciso saber o seu índice e o índice do seu bloco. *Grids* e blocos são estruturas tridimensionais, ou seja, um *Grid* é um *array* 3D de blocos e um bloco é um *array* 3D de *threads*. Essa organização é ilustrada na Figura 3.6.

Os recursos de uma GPU são distribuídos por blocos. Cada um dos SMs recebe um conjunto de blocos para executar e o escalonamento é realizado de forma que cada SM receba tantos blocos quanto seja possível executar simultaneamente. Contudo, existe um limite pré-definido dependente do *hardware*, e uma lista é mantida pelo escalonador com os blocos que ainda deverão ser executados. Uma vez que a execução de um bloco termine, um dos blocos na lista é atribuído a um SM [37].

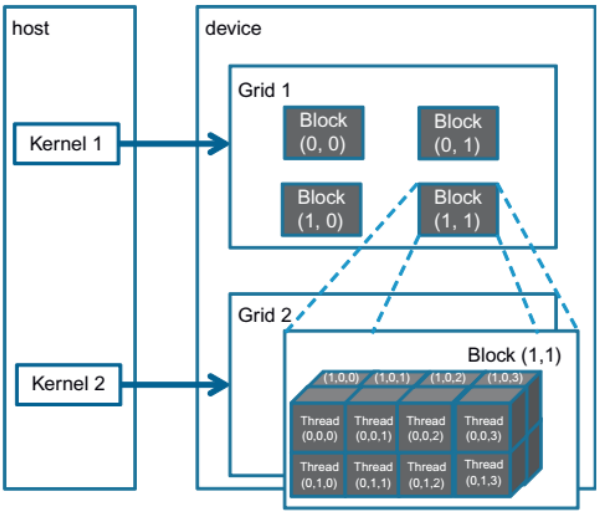

Figura 3.6: Organização hieráquica de *threads* (Fonte: [37]).

O escalonamento de *threads* de um bloco é feito em conjuntos de 32 *threads* por vez, chamado *warp*. Os SM executam as *threads* de uma *warp* segundo o modelo SIMD, ou seja, cada instrução é executada em todas as *threads* da *warp* e cada *thread* executa essa instrução sobre um conjunto de dados distinto, assim o tempo de execução é o mesmo para toda a *warp*.

A arquitetura CUDA faz uso de diversos tipos de memória distintos, com diferentes capacidades e visibilidades, para serem utilizadas conforme as necessidades dos programadores. A Figura 3.7 ilustra os diferentes tipos de memórias.

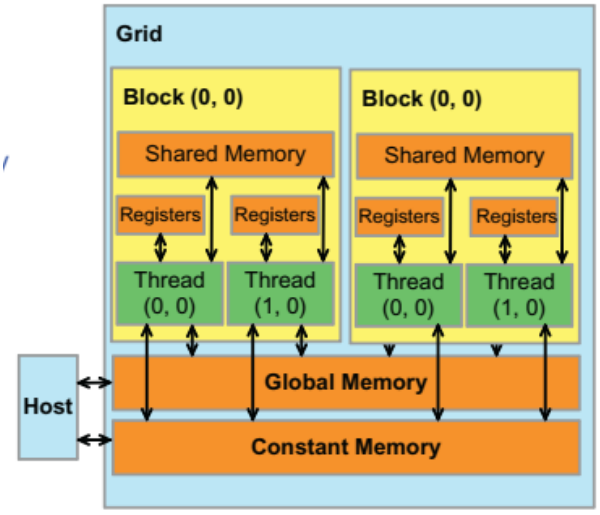

Figura 3.7: Modelo de memória CUDA (Fonte: [37]).

Na parte inferior da figura estão representadas as memórias global e constante que

podem ser escritas ou lidas pelo *host* utilizando as funções da API. Ambas podem ser acessadas por todas as *threads* do *grid*, sendo que a memória constante é *read-only*.

Os registradores, memória local e as memórias compartilhadas são memórias internas à GPU e possuem um latência muito menor que as memórias mencionadas no parágrafo anterior. Registradores e memória local são visíveis apenas às *threads* enquanto as memórias compartilhadas são visíveis por todas as *threads* do mesmo bloco.

O Algoritmo 2 apresenta o código de um somador de vetores utilizando CUDA.

Algoritmo 2 Algoritmo de soma de vetores, código de Device

```
1: function vecAddDevice(float A[256], float B[256], float C[256], intn)
2:     id = blockIdx.x * blockDim.x + threadIdx.x
3:     if (id<n) C[id] = A[id] + B[id]
4: end function
5: function vecAddHost(float A[256], float B[256])
6:     float C[256]
7:     alocaMemoria(A, B, C)
8:     enviaVetores(A, B)
9:     vecAdd <<< 8, 32 >>> (A, B, C, 256)
10:    copiaResultados(C)
11:    liberaMemoriaDevice(A, B, C)
12:    return C
13: end function
```

O código *host* (Algoritmo 2 linhas 5 a 13) aloca espaço na memória do *device* para os vetores A,B e C com a chamada *cudaMalloc*. Em seguida, na linha 8, envia os conteúdos dos vetores A e B da CPU para o *device* com a chamada *cudaMemcpy*. Na linha 9 é feita a invocação do *kernel*, com 8 blocos de 32 *threads* cada, o que significa que cada *thread* será responsável por calcular uma posição do vetor. Na linha 11, após o encerramento da execução do *kernel*, o resultado é copiado de memória *device* para a memória do *host* com a chamada *cudaMemcpy*. Por fim, na linha 11 é liberada a memória alocada em *device* utilizando a chamada *cudaFree*.

O código *device* (linhas 1 a 4) inicialmente calcula a id da *thread* (linha 2), obtido com calculando o produto da id do bloco (*blockIdx.x*) e do tamanho do bloco (*blockDim.x*) somado à id local da *thread* (*threadId.x*). Com a id calculada, é feita uma verificação para garantir que a posição a ser acessada no vetor é válida (linha 3) e, caso positivo, a *thread* calcula a soma das posições id dos vetores A e B e armazena o resultado em C (linha 3).

Capítulo 4

Algoritmos de MSA em plataformas de alto desempenho

A evolução nos métodos de sequenciamento fez com que quantidades cada vez maiores de sequências sejam obtidas em um curto período de tempo, o que gerou a necessidade de métodos de alinhamento rápidos capazes de lidar com múltiplas sequências. A fim de suprir essa necessidade, diversas abordagens foram propostas utilizando plataformas de alto desempenho como GPUs (*Graphic Processing Units*). Nessa seção serão apresentadas algumas das abordagens propostas nos últimos 4 anos.

4.1 CUDA Clustal W

O CUDA Clustal W [7], proposto em 2015, é uma implementação do Clustal W (Seção 2.4.1.2) utilizando GPU. A primeira fase do Clustal W, cálculo da matriz de distâncias, é paralelizada em GPU. Nessa fase, são feitas $\frac{N^2}{2}$ comparações par-a-par, onde N é o número de sequências, utilizando o algoritmo de Hirschberg [25] que calcula *Longest Common Subsequence (LCS)* em espaço linear. Cada bloco de *threads* da GPU recebe uma comparação par-a-par, que é executada pelas *threads* do mesmo bloco, que calculam cada anti-diagonal da matriz em paralelo, tanto no sentido *forward* quanto no sentido *backward* [7], como pode ser visto na Figura 4.1.

Como existem diversas invocações de *kernel* com transferência de dados de e para a CPU, os autores propuseram um pipeline assíncrono, com sobreposição das transferências e da computação.

O CUDA Clustal W foi testado em uma plataforma composta por um Intel Xeon 5550 2,67 GHz com 24GB DDR3 RAM e duas GPUs Tesla C2050 (arquitetura Kepler) com 448 CUDA cores cada. Os conjuntos de testes tinham até 1.000 sequências de até 1.523

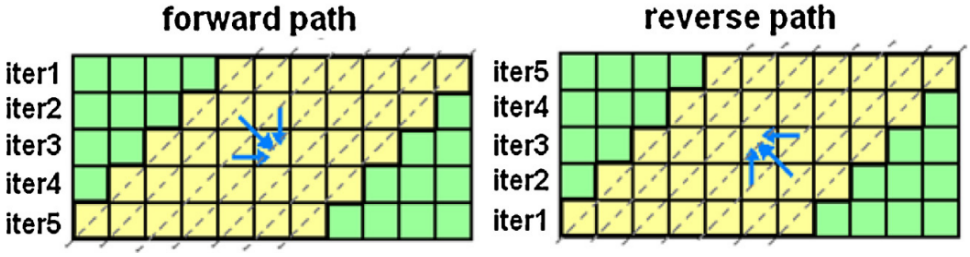

Figura 4.1: Cálculo da matriz de distâncias do CUDA Clustal W (Fonte: [7]).

aminoácidos. Quando comparado como Clustal W, que roda em CPU, o CUDA Clustal W conseguiu ser até 33x mais rápido.

4.2 MAFFT - CUDA-Linsi

Em 2015, Zhu et al. [46] apresentaram uma implementação do método L-INS-i do MAFFT (Seção 2.4.2.2) utilizando CUDA em GPUs NVidia.

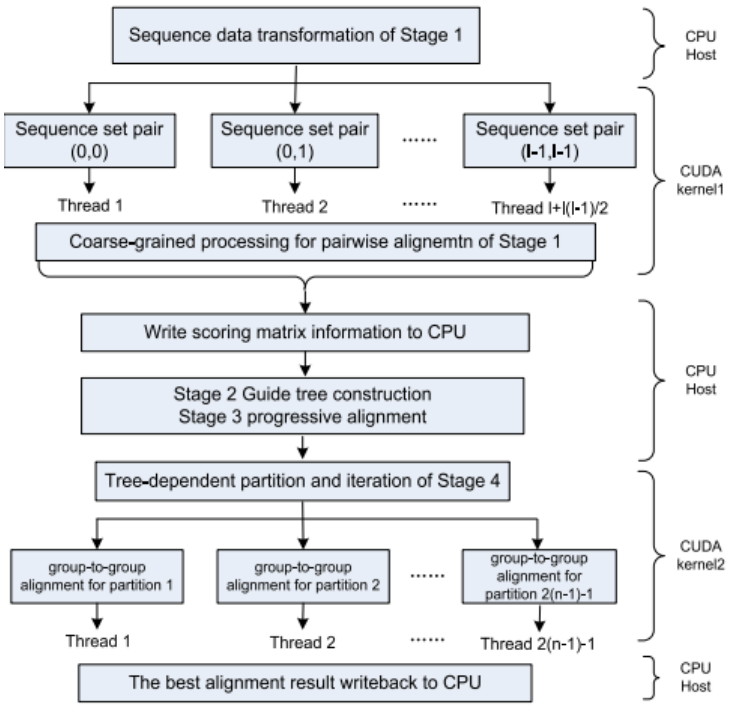

Figura 4.2: CUDA-LINSi framework (Fonte: [46]).

A Figura 4.2 ilustra o *framework* do CUDA-LINSi. O CUDA-LINSi paraleliza a execução das etapas 1 e 4 do L-INS-i, executando o alinhamento par-a-par e o refinamento iterativo em GPU. Na primeira etapa do CUDA-LINSi, as sequências são divididas em

conjuntos por um método de transformação de dados, que transforma as sequências em um formato que se adequa melhor ao funcionamento em GPU. Essa transformação possui 3 fases [46]:

1. Ordenação de carga de trabalho: as sequências são ordenadas por tamanho para minimizar a diferença de carga de trabalho entre *threads*.
2. Concatenação: as sequências são distribuídas em grupos com 16 sequências cada, em que sequências menores são concatenadas umas às outras até atingirem o tamanho próximo ao da maior sequência do grupo, sendo um símbolo de término de sequência inserido entre elas.
3. Intercalação: os grupos de sequências são armazenados de forma intercalada, sendo que cada subconjunto intercalado consiste de 8 bytes de caracteres de cada grupo.

A Figura 4.3 ilustra a transformação proposta pelo CUDA-LINSi.

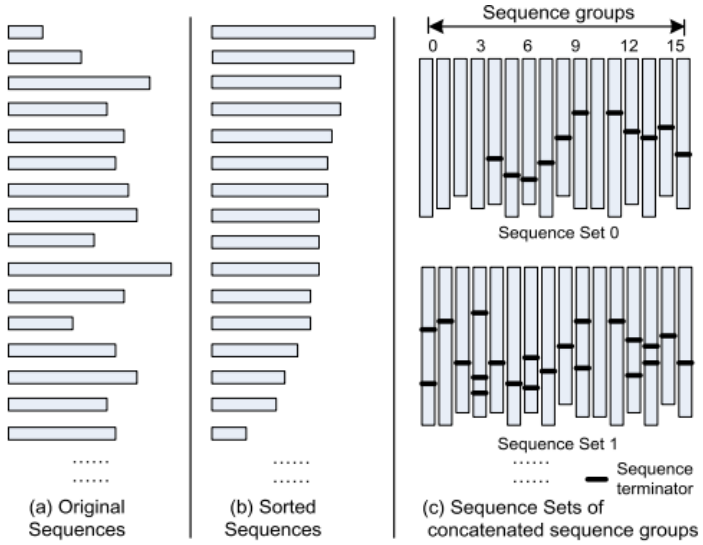

Figura 4.3: Transformação de dados para GPU (Fonte: [46]).

Na primeira etapa do L-INS-i, são feitos alinhamentos locais entre pares de sequências com *affine gap*, ação essa que foi paralelizada em um algoritmo *coarse-grained*. Nesse algoritmo, cada *thread* da GPU é responsável por um conjunto de sequências, com intuito de reutilizar memória entre as *threads*. As etapas 2 e 3 são executadas em CPU como no algoritmo original.

Na etapa 4 é realizado o refinamento do alinhamento de forma iterativa. As sequências alinhadas são utilizadas para construir uma árvore binária, na qual cada sequência é uma folha, que então é particionada em duas. Em seguida é realizado um alinhamento entre

grupos e caso o escore do alinhamento resultante seja melhor que o anterior, ele o substitui, repetindo-se o processo até que não haja mais ganho no escore.

O CUDA-LINSi utiliza um algoritmo *fine-grained* para realizar a etapa 4. As ações de particionar a árvore e fazer o alinhamento ente grupos são executadas em paralelo. Em um conjunto com n sequências, a árvore possui $2n - 2$ nós e, considerando que o nó raiz não é particionado, serão necessárias $2n - 3$ partições.

O objetivo do algoritmo *fine-grained* é obter o alinhamento de maior escore global e para tal é necessário obter o alinhamento de maior escore em cada iteração. Um *buffer* é declarado em memória compartilhada para armazenar o escore calculado por cada *thread* do bloco. O melhor resultado obtido é retornado à CPU.

O CUDA-LINSi obteve um *speedup* de $11,28x$ em um conjunto de teste com 200 sequências de tamanho médio de 4896 em relação ao L-INS-i em CPU, sem alterar a acurácia do algoritmo original. Foram utilizadas as GPUs Tesla C2050, Tesla M2090 e Tesla K20m e as CPUs AMD Opteron Processor 6143 (8cores) com 8GB RAM, Intel Xeon E5-2620 com 32GB RAM e Intel Xeon E5-2650 com 32GB RAM.

4.3 CMSA

O CMSA [47] foi proposto em 2017 por Chen et al. como um método para lidar com grandes quantidades de sequências de DNA/RNA. Para obter o resultado de forma mais rápida, o CMSA utiliza uma arquitetura heterogênea com CPU/GPU, realizando o balanceamento de carga entre elas. Além disso, o CMSA realiza um alinhamento baseado na estratégia *center-star* [3], na qual, ao invés de realizar o alinhamento entre todos os pares de sequências como é feito nos métodos heurísticos, todas as sequências são alinhadas com uma mesma sequência, diminuindo drasticamente o número de alinhamentos entre pares necessários.

A estratégia *center-star* funciona em três etapas [47]:

1. Identificação da sequência central.
2. Alinhamento entre pares de todas as demais sequências à sequência central.
3. Agrupamento dos *gaps* inseridos para gerar o alinhamento múltiplo.

O CMSA utiliza um método próprio de escolha da sequência central utilizando *bitmap* que é capaz de executar em $O(LN)$, onde L é o tamanho médio das sequências e N é o número de sequências, possibilitando uma execução muito mais rápida para conjuntos grandes de sequências. Inicialmente, as sequências são divididas segmentos de 8 caracteres e cada caracter é codificado utilizando dois bits e, portanto, cada segmento é codificado em

16 bits. O valor decimal equivalente a esses 16 bits é então utilizado como índice em um *array* de ocorrências, que é preenchido percorrendo-se todas as sequências e incrementando esse *array* a cada ocorrência do segmento, com a limitação de que um segmento só é contado até uma vez por sequência.

O *array* é utilizado para identificar a sequência central, escolhida como sendo a sequência cujo somatório dos valores de ocorrência de seus segmentos for máxima, o que pode ser feito em $O(LN)$. Isso reduz significantemente o custo da identificação da sequência central em conjuntos grandes de sequências.

A arquitetura heterogênea utilizada visa fazer uso eficiente das capacidades da CPU e da GPU. Para tal, uma etapa de pré-computação é adicionada ao algoritmo, na qual conjuntos de sequências de mesmo tamanho são submetidos à CPU e a GPU para a realização do alinhamento com a sequência central. A razão entre os tempos de execução é então utilizada para definir o balanceamento de carga, isto é, se a CPU e à GPU levaram respectivamente t_1 e t_2 para finalizar o alinhamento, então, a CPU receberá $\frac{n}{R+1}$ sequências e a GPU $\frac{Rn}{R+1}$ sequências, com $R = \frac{t1}{t2}$. O alinhamento entre as sequências é feito utilizando o K-Band [48], um algoritmo de programação dinâmica (DP) que utiliza heurísticas para reduzir a área calculada da matriz.

O ambiente de teste utilizado foi composto da CPU Intel Xeon E5-2620 (6 cores) 2,4 GHz e da GPU NVidia K40 (2880 CUDA Cores). Conjuntos de até 500.000 sequências foram comparados. O tempo de execução do CMSA (CPU) foi comparado com o HAlign [49], que também usa o método center-star e o CMSA obteve menores tempos de execução.

Quando comparado a outros algoritmos do estado da arte de alinhamento múltiplo como o MAFFT (Seções 2.4.1.3 e 2.4.2.2) e o Kalign [50], o CMSA obteve o melhor desempenho entre todos e a sua acurácia mostrou-se superior ao HAlign, que também utiliza *center-star*. O MAFFT obteve alinhamentos mais acurados que o CMSA nos menores conjuntos testados, contudo ele não foi capaz de processar os conjuntos maiores, com até 500.000 sequências.

4.4 MSAProbs-MPI

O MSAProbs-MPI [51] é uma implementação MPI/OpenMP do MSAProbs (Seção 2.4.1.4). O MSAProbs-MPI utiliza um cluster de CPUs multi-core para paralelizar as duas etapas mais custosas do algoritmo: (a) o cálculo da matriz de probabilidades a posteriori e de distâncias par-a-par e (b) a atribuição de pesos às matrizes de probabilidade a posteriori (*Weighted consistency transformation os all posterior probability matrices*).

Inicialmente, cada nó é responsável pela leitura de um conjunto de sequências, divididas igualmente entre todos os nós. No estágio 1, os nós calculam as matrizes de probabilidades a posteriori e as distâncias par-a-par entre a sequências sob sua responsabilidade. As matrizes são salvas na memória dos nós e as distâncias são agregadas em um nó, chamado p_0, que então constrói a árvore-guia e faz o *broadcast* dos pesos das sequências resultantes (estágio 2).

No terceiro estágio, as matrizes de probabilidades são transformadas de acordo com os pesos calculados por p_0. Essas transformações consistem de produtos de matrizes, os quais dependem de matrizes armazenadas em outros nós. Para lidar com essa dependência, foi utilizado um algoritmo de comunicação em anel. Para tal, esse estágio foi dividido em $P-1$ etapas, onde P é o número de nós. Em cada etapa i, um nó p recebe em um *buffer* as matrizes armazenadas no nó $(p+i) \mod P$ e realiza as transformações. Como para conjuntos grandes de sequências esses *buffers* são um gargalo, por isso foi adicionada a possibilidade de subdividir as matrizes em blocos, aumentando o número de mensagens enviadas, mas possibilitando o uso de *buffers* menores. Por fim, o nó p_0 agrega as matrizes transformadas e realiza o alinhamento.

Os resultados experimentais foram obtidos em um cluster de 32 nós, onde cada nó era composto por 2 Intel Haswell 2680, totalizando 24 cores e 128 GB RAM por nó. Foram comparadas até 1.543 sequências de tamanho médio 885.

O MSAProbs-MPI obteve, em um *cluster* com 32 nós (784 *cores*), um *speedup* de até $15x$ em relação ao MSAProbs, obtendo exatamente o mesmo alinhamento. A distribuição do primeiro estágio foi a que gerou maior aceleração uma vez que o *overhead* decorrente de comunicação é significantemente inferior ao do terceiro estágio.

4.5 FAMSA

O FAMSA (*Fast and Accurate Multiple Sequence Alignment*) [4], proposto em 2016, é uma estratégia de alinhamento por refinamento iterativo que usa instruções vetoriais AVX da CPU Intel para calcular o alinhamento múltiplo entre dezenas de milhares de sequências com *affine gap*. A primeira fase do algoritmo (cálculo da matriz de distâncias) é executada por múltiplas *threads* usando AVX com o algoritmo de *Longest Common Subsequence (LCS)*, explorando o paralelismo diagonal (*wavefront*) com operações *bit-parallel*. Cada *thread* se executa em um CPU Core, usando operações vetoriais. Além disso, o FAMSA foi também implementado em OpenCL, que permite o seu uso em GPUs da NVidia e da AMD.

Para calcular a árvore-guia (etapa 2), o FAMSA uso o algoritmo SLINK [52]. O SLINK é incremental, o que permite que as etapas 1 e 2 sejam executadas da maneira simultânea.

Na fase 3 (construção do alinhamento), diversas versões especializadas do algoritmo genérico de programação dinâmica são utilizadas para casos de conjuntos de sequências muito parecidas. Para calcular a localização dos caracteres, os autores usam uma representação "*gapped*", onde um *array* de caracteres e um *array* contendo o número de *gaps* são usados para representar o perfil do alinhamento, reduzindo bastante a quantidade de memória e o tempo necessários para a computação. A fase de refinamento é aplicada somente para conjuntos pequenos de subsequências ($N <= 1000$).

O FAMSA foi comparado às ferramentas Clustal Omega [6], MAFFT Parttree (Seção 2.4.3) e MAFFT DPParttree [5], dentre outros. O *benchmark* utilizado foi o extHomFam [13], que possui conjuntos de até 415.519 sequências. Os testes foram executados em uma estação de trabalho contendo 2 Intel Xeon E5-2670v3 2,3 GHz (12 cores cada), 128GB RAM e uma GPU NVidia Quadro M6000 (3072 CUDA Cores).

O FAMSA em CPU, utilizando instruções vetoriais AVX, apresentou o melhor tempo de execução e a melhor acurácia que os todos os demais algoritmos comparados, com exceção de um caso com conjunto pequenos (de até 3.000 sequências), em que o Clustal Omega [6] conseguiu melhor acurácia, com base no *golden standard* do extHomFam. O desempenho da versão em GPU ficou aquém do desempenho da CPU (24 *cores*).

4.6 Quadro comparativo

Tabela 4.1: Comparativo entre os algoritmos de Estado da Arte.

Artigo	Ano	Algoritmo de distância	Número máximo sequências	Plataforma
CUDA Clustal W	2015	LCS	1.000	GPU
CUDA-Linsi	2015	Smith-Waterman	200	GPU
CMSA	2017	Center-star/ K-band	500.000	GPU
MSAProsbs-MPI	2016	HMM	1.543	MPI/OpenMP
FAMSA	2016	LCS	415.519	AVX ou GPU

Foram analisados 5 trabalhos recentes de alinhamento múltiplo de sequências em plataformas de alto desempenho. Os trabalhos usam diversos algoritmos de alinhamento par-a-par (LCS, Smith-Waterman, K-band, HMM). Três dos algoritmos comparam números considerados pequenos de sequências (até 1.543) e dois algoritmos comparam um número grande sequências (até 500.000). Como plataforma de alto desempenho, a maioria dos algoritmos usa GPU. O MSAProbs-MPI usa *clusters* de computadores (MPI e

OpenMP) e o FAMSA usa instruções vetoriais (AVX) ou GPU. No artigo do FAMSA, os melhores resultados apresentados foram obtidos usando instruções vetoriais de CPU.

O CUDA-Parttree, proposto nesta Dissertação de Mestrado, usa a contagem de 6mers em comum como algoritmo de cálculo de distâncias e compara um grande número de sequências (até 151.443). A nosso conhecimento, essa é a primeira proposta de uma abordagem de contagem de 6mers em GPU.

Capítulo 5

Projeto do CUDA-Parttree

Devido à rápido evolução de equipamentos para sequenciamento genômico, conjuntos cada vez maiores de sequências estão sendo gerados. Por exemplo, o conjunto de transportadores ABC do PFAM (*Protein Families Database*) [36] possui aproximadamente 30.000 sequências. Outro exemplo é o *Ribosomal Database Project release 9* [53] que possui mais de 200.000 sequencias de SSU rRNA (*Small subunit ribosomal* RNA).

A maioria dos métodos de alinhamento múltiplo de sequências utiliza uma árvore-guia [5] para determinar a ordem na qual sequências serão alinhadas umas às outras. Na construção de uma árvore-guia é comum realizar inicialmente o alinhamento entre pares de todas as sequências (Seção 2.4.1), gerando uma matriz de distâncias que é utilizada por um algoritmo de clusterização para gerar a árvore-guia.

A etapa do cálculo de distâncias entre sequências possui características que a tornam interessante do ponto de vista de paralelização pois existem grandes quantidades de cálculos independentes entre si, ou seja, que podem ser realizados simultaneamente sem necessidade de comunicação.

Para calcular a matriz de distâncias entre todas as sequências são normalmente necessárias $\frac{N(N-1)}{2}$ operações ($O(N^2)$). Para conjuntos com dezenas de milhares de sequências isso significa da ordem de bilhões de comparações.

O MAFFT Parttree (Seção 2.4.3), apesar de diferir dos algoritmos de alinhamento progressivo tradicionais ao juntar o cálculo das distâncias com a construção da árvore-guia, comporta-se de maneira parecida às outras ferramentas na etapa de cálculo da matriz de distâncias e, por isso, pode obter um ganho de desempenho calculando as distâncias entre sequências em paralelo.

Com o intuito de acelerar ainda mais o cálculo de matriz de distâncias, o MAFFT Parttree (Seção 2.4.3) faz o alinhamento de n sequências de referência com todas as outras, reduzindo a complexidade para $O(N \log N)$. Além disso, o alinhamento de duas sequências é feito com a contagem de 6mers em comum, que bem mais rápido que o alinhamento

usando programação dinâmica. Mesmo assim, o tempo de execução do Parttree pode ser muito longo caso dezenas de milhares de sequências sejam comparadas.

O cálculo da matriz de distâncias engloba diversas operações par-a-par, que são independentes entre si. Por essa razão, a grande maioria das propostas da literatura (Capítulo 4) paraleliza essa etapa. O MAFFT Parttree usa um método distinto das demais ferramentas (contagem de 6mers) que, a nosso conhecimento não foi implementado em GPU.

O presente capítulo descreve o projeto do CUDA-Parttree, que executa a contagem de 6mers para alinhamento múltiplo de sequências (MSA) em GPU. Na Seção 5.1 é feita a análise do código do Parttree. A Seção 5.2 apresenta uma visão geral do CUDA-Parttree. A seguir, as Seções 5.3 a 5.5 detalham o seus principais módulos.

5.1 Análise do Parttree

O MAFFT Parttree (Seção 2.4.3) é um pacote de programas de alinhamento múltiplo de sequências que oferece diversas opções em várias etapas da realização do alinhamento. No caso do cálculo de distâncias, o método padrão utilizado é a contagem de 6mers, onde 6mers são subsequências de 6 caracteres em comum entre as sequências.

Enquanto diversos algoritmos de alinhamento múltiplo progressivo calculam as distâncias entre todos os pares de sequências, o Parttree, por ser utilizado em conjuntos com dezenas de milhares de sequências, restringe o número de comparações realizadas.

Conforme apresentado na Seção 2.4.3, o Parttree utiliza uma abordagem recursiva, em que são realizados 3 cálculos de distâncias em cada nível da recursão: (a) um entre a sequência mais comprida e todas as demais; (b) um entre as n sequências de referência entre si e (c) um último entre as n sequências de referência e as $N-n$ restantes. O método do cálculo de distâncias empregado é sempre o mesmo, variando apenas o conjunto de dados utilizado.

As chamadas à rotina de cálculo de alinhamento podem ser visualizadas nas linhas 2, 5 e 16 no Algoritmo 3 como *calculateSimilarity*.

Na linha 1, identifica-se a sequência mais longa l e com ela é feito o cálculo de similaridade com todas as demais (linha 2). Nas linhas 3 e 4 são escolhidas $n-1$ sequências, sendo uma a sequência menos similar à mais longa e $n-2$ escolhidas de forma aleatória, que junto da sequência l formam o conjunto de sequências de referência. A seguir na linha 5, é calculada a similaridade entre as n sequências de referência. Nas linhas de 6 a 10 é realizada a filtragem das sequências de referência em que, caso haja sequências com similaridade maior que um valor pré-determinado (MIN_DIFF), apenas a sequência mais longa é mantida, retirando-se as demais do conjunto de sequências de referência.

Algoritmo 3 PartTree(baseSeqs,N,n)

```
1: l ← longest(baseSeqs)
2: calculateSimilarity(l, baseSeqs)
3: least ← leastSimilar(baseSeqs − l, l)
4: refSeqs ← {l, least} ∪ pickRandom(n, baseSeqs − {l, least})
5: calculateSimilarity(refSeqs, refSeqs)
6: for a, b ∈ refSeqs do
7:     if similarity(a, b) < MIN_DIFF then
8:         refSeqs ← refSeqs − shortest(a, b)
9:     end if
10: end for
11: n' ← len(refSeqs)
12: T ← buildTree(refSeqs)
13: if n' >= N then
14:     return T
15: end if
16: calculateSimilarity(refSeqs, baseSeqs − refSeqs)
17: groups ← groupToMostSimilar(refSeqs, baseSeqs)
18: for seq ∈ refSeqs do
19:     N ← len(group[seq])
20:     trees[seq] ← PartTree(groups[seq], N, n)
21: end for
22: return combineTrees(T, trees)
```

Após a filtragem, o conjunto de sequências de referência, contendo n' sequências, é utilizado para a construção da árvore T (linha 12). Caso $n' >= N$, ou seja, existam menos sequências no conjunto a ser alinhado que o número de sequências de referência após a filtragem, a recursão encerra retornando T (linha 13 a 15). Caso contrário, é realizado o cálculo de similaridade entre as sequências de referência e as demais (sequências base) (linha 16) e, em seguida, cada sequência base é agrupada à sequência de referência mais similar a si (linha 17). Nas linha 18 a 21, é realizada a chamada recursiva do Parttree, sendo cada grupo criado utilizado como conjunto de sequências base da chamada recursiva. Por fim, na linha 22, as árvores retornadas pelas chamadas recursivas são combinadas com a árvore T e a árvore resultante é retornada.

A primeira chamada ao cálculo de distâncias (linha 2 do Algoritmo 3) realiza a comparação $O(N)$ vezes, pois compara a sequência mais comprida a todas as demais. A segunda chamada à rotina *calculateSimilarity*, linha 5, realiza $O(n^2)$ comparações, onde n é o número de sequências de referência. A última chamada à rotina *calculateSimilarity* realiza $O(nN)$ na linha 16. Como por padrão $N >> n$, a última chamada realiza muito mais comparações que as outras duas somadas, porém bem menos que $\frac{N^2}{2}$ dos algoritmos progressivos.

O algoritmo de cálculo de distâncias (*calculateSimilarity*) agrupa cada aminácido em um de 6 grupos, definidos com base nos valores da matriz de substituição PAM250 (Seção 2.2.1), e os representa como um inteiro de 0 a 5. Em seguida, cada 6mer, subsequência de 6 caracteres, é representado como um inteiro calculado usando a Equação 5.1, possuindo valor de 0 a $6^6 - 1$. Dessa forma, cada sequência de aminoácidos é convertida em um vetor de 6mers representados por inteiros. A Figura 5.1 ilustra o processo de conversão de caracteres em 6mers.

$$kmer = \sum_{i=0}^{5} 6^i * s[i] \tag{5.1}$$

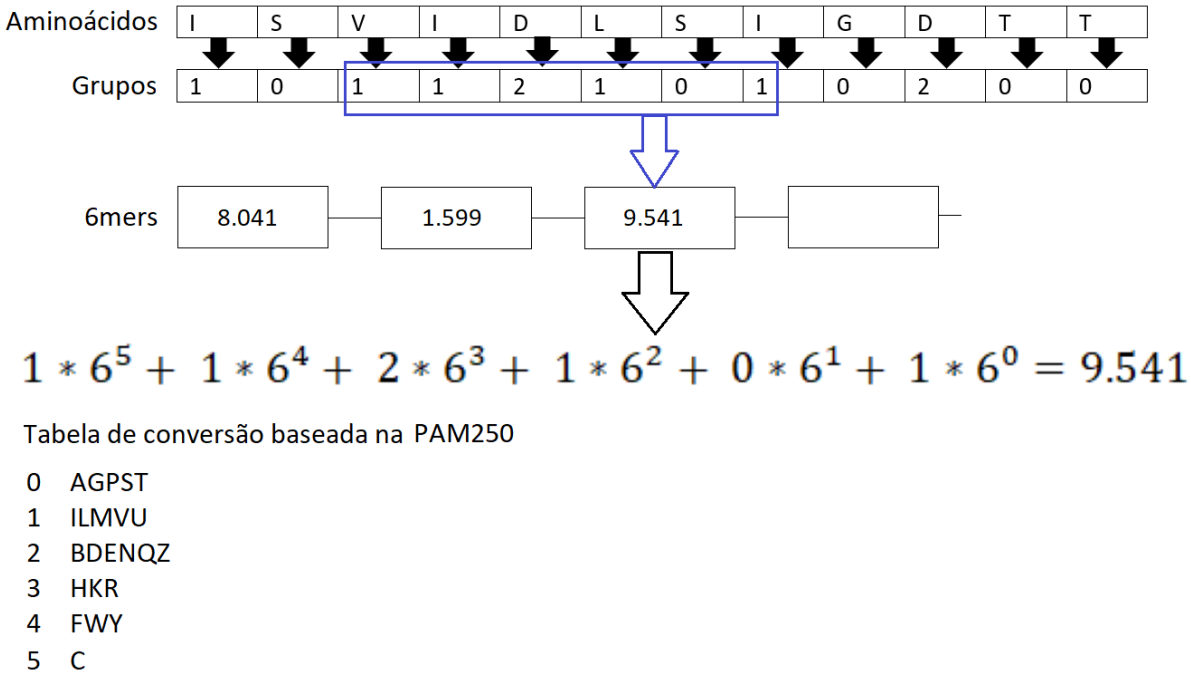

Figura 5.1: Conversão de uma sequência de 12 caracters em um vetor de 6mers.

O cálculo de similaridade, apresentado nos Algoritmos 4 e 5, possui um conjunto de sequências de referência e um conjunto de sequências base. Para cada sequência de referência é calculado o número de 6mers em comum com cada sequência base.

A primeira etapa do cálculo de similaridade é a construção de um vetor, denominado mapa de referência (*refMap*), com 6^6 posições, em que cada posição i representa o número de vezes em que o 6mer i apareceu na sequência, que pode ser visto nas linhas de 3 a 7 do Algoritmo 4 e na Figura 5.1. Uma vez obtido o mapa de referência, inicia-se a iteração sobre as sequências base, apresentada no Algoritmo 5 (*calculate6mers*).

Para cada sequência base é criado um vetor espelho (*mirrorMap*), Algoritmo 5 linha 2, possuindo as mesmas 6^6 posições, mas inicialmente com todas as posições zeradas. Esse

Algoritmo 4 calculateSimilarity(refSeq,baseSeq)

```
1: distMtx[n * (N − n)] = [0, 0...0]
2: for r ∈ refSeq do
3:     refMap[6^6] = [0, 0...0]
4:     for i = 0 → length(r) do
5:         k ← r[i]
6:         refMap[k] + +
7:     end for
8:     calculate6mers(refMap, baseSeqs, dstMtx)
9: end for
10: return distMtx
```

Algoritmo 5 calculate6mers(refMap,baseSeq,dstMtx)

```
1: for s ∈ baseSeqs do
2:     mirrorMap[6^6] = [0, 0...0]
3:     for i = 0 → length(s) do
4:         k ← s[i]
5:         if refMap[k] > mirrorMap[k] then
6:             distMtx[r][s] + +
7:             mirrorMap[k] + +
8:         end if
9:     end for
10: end for
```

vetor será utilizado para garantir que caso um 6mer apareça mais vezes na sequência base do que na sequência de referência o valor excedente não seja considerado.

Em seguida, para cada 6mer da sequência base, representado pelo inteiro k, são comparados o mapa e referência e o espelho. Caso o valor na posição k do vetor espelho seja menor que o valor da posição k do mapa de referência, o número de 6mers em comum e o valor no vetor espelho são incrementados, caso contrário nada é alterado pois o número de vezes em que o 6mer k apareceu na sequência base já excedeu o da sequência de referência (linhas 3 a 9).

O uso da recursão (linha 20 do Algoritmo 3) significa que essas chamadas são realizadas múltiplas vezes, diminuindo o tamanho dos conjuntos a cada nível da recursão. De fato, para instâncias do Parttree com um grande número de sequências de referência, o tamanho dos conjuntos a serem alinhados diminui tanto entre níveis de recursão que muito poucas operações são realizadas, se comparado ao primeiro nível da recursão.

A Figura 5.2 ilustra a matriz de distâncias calculada pelo Parttree, em que cada linha representa uma sequência de referência e cada coluna uma sequência base. Cada área marcada por um número na figura representa as células da matriz que são calculadas em cada uma das chamadas do algoritmo *calculateSimilarity* (Algoritmo 4). A área 1

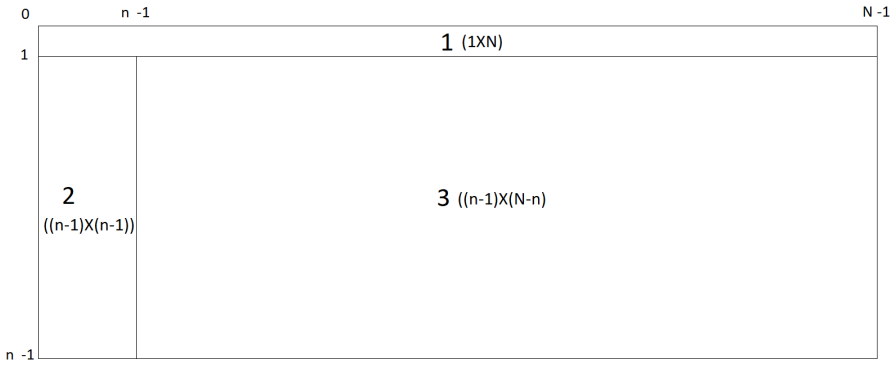

Figura 5.2: Matriz de distâncias calculada pelo Parttree em um nível de recursão. Os números representam as áreas calculadas por cada chamada do algoritmo *calculateSimilarity*.

representa o alinhamento da sequência mais longa a todas as N sequências (linha 2 do Algoritmo 3). A área 2 representa o alinhamento das n sequências de referência entre si (linha 5). Por fim, a área 3 representa o alinhamento das n sequências de referência com as $N - n$ sequências base (linha 16).

Considerando um conjunto com 100.000 sequências e 1.000 sequências de referência, a área 1 possui 100.000 células. Já a área 2, considerando que existam 1.000 sequências de referência após a filtragem, possui $(1.000)^2 = 1.000.000$ de células. Finalmente, a área 3 possui então $1.000 * (100.000 - 1.000) = 99.000.000$ de células. Assim, nesse exemplo, a área 3 representa 99% da área a ser calculada. Além disso, considerando uma distribuição aleatória, no segundo nível de recursão cada grupo é responsável por calcular o alinhamento de $\frac{100.000}{1.000} = 100$ sequências.

Tendo em vista isso, foi decidido fazer o cálculo paralelo da comparação entre as sequências de referência e as sequências base no primeiro nível da recursão (linha 16 do Algoritmo 3 e área 3 da Figura 5.2). Dado que os conjuntos de sequências para os quais o Parttree foi projetado possuem dezenas de milhares de sequências, essa etapa pode requerer milhões de comparações, o que tornou interessante o uso de GPUs com sua capacidade de paralelização massiva.

5.2 Visão geral do CUDA-Parttree

A versão proposta na presente Dissertação de Mestrado, batizada de CUDA-Parttree, faz o cálculo das distâncias entre as sequências de referência e as demais sequências do primeiro nível de recursão (linha 16 do Algoritmo 3) em GPU.

Conforme apresentado na Seção 3.3 sobre CUDA, as GPUs NVIDIA utilizam uma arquitetura específica. Assim, diversas alterações foram projetadas para melhor acomodar o cálculo de distâncias ao CUDA.

O objetivo principal da nossa proposta é reduzir o tempo de cálculo da matriz de distâncias na primeira recursão, quando comparado ao código sequencial original (MAFFT Parttree). Deve ser garantido também que a mesma matriz de distâncias seja obtida.

O cálculo da matriz de distâncias foi dividido no CUDA-Parttree em três fases: conversão de dados, computação em GPU e recuperação de dados, como pode ser visto no Algoritmo 6. As etapas do algoritmo são ilustradas na Figura 5.3.

Algoritmo 6 CudaParttree(baseSeqs,refSeqs,n,N)

1: $sequencesSetup(baseSeqs, baseSeqsCPT, secLengthVec, secAddressVec, positionMap)$ // conversão de dados
2: $calculateSimilarityCudaParttree(secLengthVec, secAddressVec, N,$
$n, refSeqs, baseSeqsCPT, dstMtx)$ // computação em GPU
3: $returnDataCudaParttree(distMtx, positionMap)$ //recuperação de dados
4: **return**

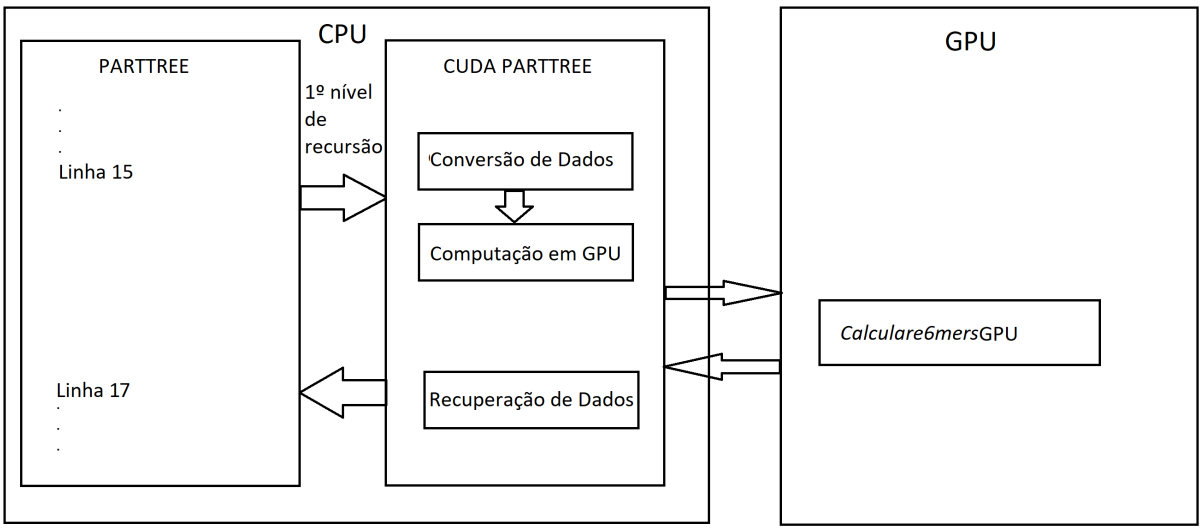

Figura 5.3: Fluxo do CUDA-Parttree.

No Algoritmo 6 linha 1, a rotina *sequencesSetup* recebe as sequências base, faz as transformações necessárias para sua utilização em GPU e as envia para GPU, produzindo as sequências tratadas (*baseSeqsCPT*) e dois vetores (*secLengthVec* e *secAddressVec*) utilizados para localizar as sequências em GPU e o vetor *positionMap* que mapeia cada sequência à sua posição original. Na linha 2, a rotina *calculateSimilarityCUDAParttree* recebe os dados produzidos pela rotina *setupSequences* (*baseSeqsCPT*, *secLengthVec* e *secAddressVec*), as sequências de referência (*refSeqs*) e os tamanhos dos conjuntos de sequências base (N) e de sequências de referência (n) e produz a matriz de distâncias (distMtx), contendo o

resultado da contagem de 6mers feita em GPU. Na linha 3, a rotina *returnDataCudaParttree* recebe a matriz de distâncias (*distMtx*) e o vetor de posições (*positionMap*) e retorna as distâncias calculadas ao Parttree (Algoritmo 3). As Seções 5.3 a 5.5 apresentam o detalhamento de cada rotina.

5.3 Conversão de dados (*Input*)

A fase de conversão de dados é responsável por adaptar as sequências e enviá-las à GPU. A fim de obter bom desempenho em GPU, são feitas algumas alterações nos dados recebidos do programa original. A primeira mudança consiste na forma como as sequências são armazenadas.

No código original, para cada par de sequências de referência e sequência base é necessário utilizar um vetor espelho para garantir que a contagem de 6mers esteja correta. Como o código é sequencial, é possível reutilizar o mesmo vetor espelho, apenas sendo necessário retornar o seus valores para 0 após o cálculo de cada par. Já no CUDA-Parttrre os cálculos serão feitos em paralelo, o que exige que, para cada par sendo calculado, haja um vetor espelho diferente. Considerando que cada vetor espelho possui 6^6 posições de inteiros, cada vetor espelho ocupa 186KB, e que os conjuntos utilizados podem possuir dezenas ou até centenas de milhares de sequências, a quantidade de memória exigida seria proibitiva ou seria necessário adicionar um lógica de controle para reutilizar um número pré-determinado de vetores espelho, o que exigiria a introdução de mecanismos de comunicação entre *threads*, que prejudicariam o paralelismo.

Com a finalidade de utilizar menos memória sem prejudicar o paralelismo, a forma como as sequências são armazenadas foi alterada. No código original, as sequências são armazenadas como sequências de inteiros, onde cada inteiro está contido intervalo $[0, 6^6[$ e representa um 6mer, que aparece no vetor na mesma ordem que na sequência original.

A nova forma de representação utilizada é um vetor de *structs*, onde cada *struct* é composta de dois inteiros, um para armazenar o 6mer e outro para indicar quantas vezes ele aparece naquela sequência. Essa forma de representação descarta a posição onde o 6mer aparece, mas como o único fator que importa no algoritmo é a contagem, isso não impacta o resultado. A Figura 5.4 ilustra o processo de conversão de dados de sequência de caracteres para vetor de *structs* de 6mers.

Com as sequências representadas no formato de *struct*, não é mais necessário a utilização de vetores espelho e a contagem de 6mers continua sendo feita em $O(L)$, onde L é o comprimento do vetor de 6mers. Para tal, basta percorrer o vetor linearmente e comparar o número de vezes em que o 6mer aparece na sequência base e na sequência referência e somar o valor do menor deles ao total de 6mers em comum. Dessa forma, qualquer 6mer

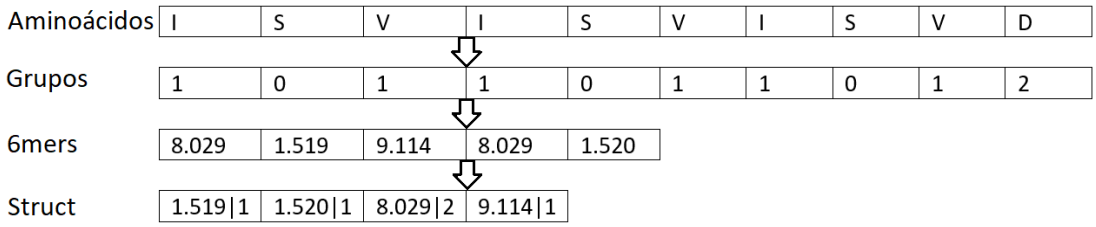

Figura 5.4: Conversão de dados do CUDA-Parttree.

que não exista na sequência base não afetará o resultado, qualquer 6mer que não exista na sequência de referência mas exista na base somará o valor 0 ao total e, para os 6mers que existem em ambas, apenas o menor número será adicionado.

Essa forma alterada do cálculo de similaridades adiciona um *overhead* de conversão das sequências, que é feito em $O(L * N)$, contudo ela elimina a necessidade de se alocar memória e de zerar o vetor espelho, o que era feito em $O(L * (N - n) * n)$, pois era executado para cada par de sequência base e sequência de referência.

Após a conversão das sequências para o novo formato algumas operações são realizadas a fim de melhor adaptá-las ao processamento em GPU. Conforme discutido na Seção 3.3, GPUs possuem estruturas chamadas *warps* que são conjuntos de *threads*, geralmente 32, que devem todas executar o mesmo código. Isso significa que, quando existem caminhos divergentes na execução de instruções, como os criados com a utilização de *ifs*, primeiro todas as *threads* da *warp* para as quais a condição é verdadeira são executadas enquanto as demais ficam ociosas. Em seguida, as *threads* para as quais a condição é falsa são executadas e as demais passam a ficar ociosas. Por isso, para obter o melhor do paralelismo em GPUs é ideal reduzir o número de caminhos divergentes tanto quanto possível [54].

Uma das formas de fazer isso é garantir que as *threads* de uma *warp* executem sobre conjuntos de dados de tamanhos iguais já que ao trabalhar com vetores é necessário saber se o fim do vetor já foi atingido. Para conseguir isso, inicialmente o CUDA-Parttree ordena as sequências de acordo com o seu comprimento, mantendo um vetor com as posições originais das sequências, e em seguida adiciona um *padding* para que cada conjunto de sequências de tamanho igual ao tamanho da *warp* (denominado seção), possua o mesmo comprimento. O *padding* adicionado consiste de um *6mer* 0 com contagem igual a 0, assim não afetando o resultado obtido algoritmo de contagem. Nesse processo de adição do *padding*, são armazenados os comprimentos das sequências de cada seção em um vetor de comprimento $N/warpsize$.

Por fim, os dados são enviados para a GPU. A forma mais comum de utilização de GPUs recomenda que vetores em memória global da GPU possuam apenas uma dimensão. Por isso, o vetor de sequências foi linearizado e um vetor adicional contendo as posições

de início de cada seção de sequências é criado para ser enviado à GPU. Sendo assim, são enviados à GPU três vetores: o vetor linearizado de sequências, um vetor com os comprimentos de cada seção de sequencias e um vetor com as posições iniciais de seção.

Algoritmo 7 sequencesSetup(baseSeqs)

1: $convertSequences(baseSeqs, baseSeqsCPT)$
2: $sortSequences(baseSeqsStruct, sortedBaseSeqs, positionMap)$
3: $addPadding(sortedBaseSeqs, paddeddSortedBaseSeqs)$
4: $calculateSectionSize(paddeddSortedBaseSeqs, sectionLengthVec)$
5: $calculateSectionAddress(paddeddSortedBaseSeqs, sectionAddressVec)$
6: $baseSeqsCPT \leftarrow paddeddSortedBaseSeqs$
7: $sendToGPU(baseSeqsCPT)$

O Algoritmo 7 apresenta o algoritmo de conversão de dados (*input*). Na linha 1, as sequências são convertidas para *structs*. Em seguida, os vetores de 6mers, agora representados como *structs*, são ordenados (linha 2), mantendo-se em um vetor auxiliar $positionMap$ a posição original das sequências. No próximo passo (linha 3), é adicionado o *padding* às sequências para que todas as sequências de uma seção possuam o mesmo comprimento. A seguir, são calculados os comprimentos (linha 4) e a posição inicial no vetor linearizado (linha 5) de cada seção. Por fim, o vetor linearizado é enviado à GPU na linha 7.

5.4 Computação em GPU

A fase de computação em GPU consiste na construção dos mapas de referência (refMap) em CPU, do envio desses mapas para a GPU, a invocação do *kernel* e por fim a recuperação da matriz distâncias da GPU. O Algoritmo 8 representa o algoritmo utilizado.

Para cada sequência de referência escolhida pelo Parttree é necessária a construção do mapa de referência que assim como no código original, contém em cada posição o número de vezes em que um 6mer apareceu na sequência. Como o número de sequências de referência mantidas após a etapa de filtragem (n') é limitado superiormente por n, o parâmetro informado pelo usuário, a quantidade de memória pode ser acomodada em GPU.

Todos os mapas de referência são armazenados em um vetor linearizado que será enviado à GPU. Para tal, ao construir o mapa é necessário calcular a posição inicial do mapa no vetor linearizado (Algoritmo 8 linha 4). A seguir, para cada 6mer na sequência de referência é recuperado o número de vezes em que ele apareceu e o valor é preenchido no mapa (linhas 5 a 9). Na *struct* utilizada, *kmer* é o valor inteiro que identifica o 6mer e *count* é o número de vezes que ele apareceu naquela sequência.

Algoritmo 8 calculateSimilarityCudaParttree(secLengthVec,secAddressVec ,N,n,refSeqs,baseSeqs,distMtx)

1: $distMtx[n * (N - n)] = [0, 0...0]$
2: $refMap[6^6 * n] = [0, 0...0]$
3: **for** $i = 0 \rightarrow n$ **do**
4: $\quad addr = i * 6^6$
5: $\quad$ **for** $j = 0 \rightarrow length(refSeqs[i])$ **do**
6: $\quad\quad r \leftarrow refSeqs[i][j]$
7: $\quad\quad k \leftarrow r.kmer$
8: $\quad\quad refMap[addr + k] = r.count$
9: $\quad$ **end for**
10: **end for**
11: $calculate6mers <<< B, T >>> (secLengthVec, secAddrVec, N, n, refMap, baseSeqs, distMtx)$
12: $retrieveFromGPU(distMtx)$
13: $freeMemoryGPU()$

A invocação do *kernel* é feita utilizando como parâmetros B blocos com T *threads* e um *grid* contendo $\lceil N/T \rceil$ blocos (linha 11).

Cada *thread* em GPU é responsável por calcular os 6mers em comum entre uma sequência base e todas as sequências de referência. A adição do *padding* garante que todas as *threads* de um *warp* trabalharão com sequências base de mesmo comprimento e, como a iteração ocorre sobre o vetor contendo os mapas de referência que possui um comprimento fixo de $6^6 * n'$, é garantido que não serão criados caminhos divergentes referentes a esses valores.

O Algoritmo 9 representa a contagem de 6mers em comum em GPU desenvolvido nessa dissertação. Ele usa como entrada o número total de sequências base (N), um vetor contendo os comprimentos das seções ($sectionLengthVec$), um vetor contendo a posição inicial das seções no vetor de sequência ($sectionaddrVec$), o número de sequências de referência restantes após a fase de filtragem (n), o vetor de mapas de referência ($refMap$), o vetor linearizado de sequências ($baseSeqs$), e a matriz de contagem que armazenará os resultados ($distMtx$).

Na linha 1 é calculado o identificador da sequência base pela qual a *thread* será responsável, $seqId$. Na linha 2 é feito o teste para garantir que o $seqId$ corresponde a uma sequência válida.

A linha 3 contém o cálculo para obter o identificador da seção à qual sequência base está relacionada e na linha 4 é calculado a posição da sequência dentro da seção.

A linha 5 obtém a posição inicial da seção no vetor de sequências e na linha 6 o comprimento da sequência base. Na linha 7 é calculado a posição da sequência no vetor de sequências.

Algoritmo 9 calculate6mersGPU(sectionLengthVec,sectionAddrVec,N,n,refMap, baseSeqs,distMtx)

```
1:  seqId = blockIdx.x * blockDim.x + threadIdx.x
2:  if seqId < N then
3:      sectionId = ⌊seqId/WARPSIZE⌋
4:      offset = sectionId%WARPSIZE
5:      sectionAddr = sectionAddrVec[seqId]
6:      sectionLength = sectionLengthVec[seqId]
7:      sequenceAddr = sectionAddr + offset * sectionLength
8:      for refSeqId = 0 → n do
9:          mapAddress = refMap + (refSeqId * MAPSIZE)
10:         distance = 0
11:         for j = 0 → sectionLength do
12:             baseCount = baseSeq[sequenceAddr + j].count
13:             refCount = mapAddress[baseSeq[sequenceAddr + j].kmer]
14:             if baseCount <= refCount then
15:                 distance += baseCount
16:             else
17:                 distance += refCount
18:             end if
19:         end for
20:         index = N * refSeqId + seqId
21:         distMtx[index] = distance
22:     end for
23: end if
```

O *loop* na linha 8 é responsável pela iteração sobre as sequências de referência. A iteração inicia com o cálculo da posição inicial do mapa da sequência de referência no vetor de mapas.

O *loop* iniciado na linha 11 e encerrado na linha 19 é responsável pela iteração sobre todos os 6mers da sequência base. Conforme apresentado no Algoritmo 8, são considerados dois valores para serem somados ao valor da contagem, o número de vezes em que o 6mer apareceu na sequência base (linha 12) e o número de vezes em que ele apareceu na sequência de referência (linha 13). Apenas o maior valor é somado à contagem, linhas 14 a 18.

O valor de 6mers em comum é então salvo na posição adequada da matriz de resultados *distMtx* (linha 21), que nesse caso é uma matriz unidimensional contendo uma versão linearizada de uma matriz bidimensional $N * n'$, onde cada linha representa uma sequência de referência e cada coluna uma sequência base.

Essa fase termina quando a matriz de distâncias ($dstMtx$) que está na memória global da GPU é transferida para a memória da CPU (linha 12 do Algoritmo 8) e a memória da GPU é liberada (linha 13).

5.5 Recuperação de dados (Output)

A fase de recuperação de dados consiste em ajustar as distâncias calculadas entre as sequências e retornar os dados obtidos em GPU para o fluxo normal do algoritmo Parttree.

Algoritmo 10 returnDataCudaParttree(distMtx,positionMap)

```
1: for i = 0 → n' do
2:     for j = 0 → N do
3:         count = distMtx[i * N + positonMap[j]]
4:         finalDistanceMatrix[i][j] = (1 − count)/MIN(T_ii, T_jj) * lenfac
5:     end for
6: end for
7: return finalDistanceMatrix
```

Essas duas funções são executadas concomitantemente, conforme apresentado no Algoritmo 10. As linhas 1 e 2 iteram o algoritmo sobre as sequências de referência e as sequências base, respectivamente.

Na linha 3, o resultado da comparação de 6mers, que está armazenado na matriz linearizada ($dstMtx$), é recuperado. Contudo, é importante lembrar que as sequências recebidas pelo CUDA-Parttree foram ordenadas por comprimento na fase de carga de dados, o que significa que, para poder retornar os dados na mesma ordem em que o Parttree enviou, é preciso fazer o ajuste da posição. Este ajuste é feito com o auxílio de vetor contendo as posições originais das sequências, denominado $positionMap$. Por exemplo, para uma sequência k que originalmente estava na posição 0 e que após a ordenação foi para a posição 3 tem-se $positionMap[0] = 3$. Isso permite que o ajuste de endereços seja feito em $O(1)$ para cada sequência.

Na linha 4 é aplicada a fórmula que calcula a distância entre sequências, utilizando o número de 6mers em comum, $count$, T_{ii} o número de 6mers na sequência i, T_{jj} o número de 6mers na sequência j e um valor $lenfac$ obtido como uma razão dos comprimentos de i e j.

Por fim, a matriz com as distâncias calculadas é retornada ao Parttree, que então segue seu fluxo original.

5.5.1 Considerações Finais

O MAFFT Parttree é um algoritmo de alinhamento múltiplo de milhares de sequências que recursivamente constrói a árvore-guia a partir de um conjunto de sequências não alinhadas. A distância entre as sequências é calculada utilizando o número de *6mers* em comum e apenas um conjunto de $n << N$ sequências de referência é alinhada a todas as demais por chamada de recursão.

O cálculo de distâncias é de interesse para paralelização pois possui uma grande quantidade de cálculos independentes entre si. Como o Parttree executa o cálculo da matriz de distâncias em três chamadas por nível de recursão, apenas a terceira chamada, que corresponde a maior parte da matriz, é calculada em GPU. O número de sequências alinhadas reduz significantemente entre níveis de recursão e, por isso, foi decidido paralelizar apenas o primeiro nível.

O CUDA-Parttree paraleliza a terceira chamada do cálculo da matriz de distâncias no primeiro nível de recursão. O algoritmo foi dividido em três fases: (a) Conversão de dados, responsável por preparar as sequências para execução em GPU; (b) Computação em GPU, responsável por realizar o cálculo da matriz de distâncias em GPU e recuperar o resultado para CPU; e (c) Recuperação de dados, responsável por retornar o resultado para o Parttree.

Capítulo 6

Resultados Experimentais

Neste capítulo são apresentados os experimentos realizados para avaliar o desempenho do CUDA-Parttree. Na Seção 6.1 são apresentados os ambientes de teste utilizados. Na Seção 6.2 são apresentados os conjuntos de sequências utilizados na realização dos experimentos. Na Seção 6.3 são apresentados os experimentos realizados e os seus resultados. Por fim, na Seção 6.5 é realizada uma discussão sobre os resultados obtidos.

6.1 Ambiente de Testes

O CUDA-Parttree foi implementado em CUDA C e testado em dois ambientes de teste (Ambinete 680 e Ambiente 980), disponíveis no Laboratório de Sistemas Integrados e Concorrentes (LAICO) da Universidade de Brasília. As configurações de CPU, GPU e de software utilizadas em cada ambiente são descritas nas Tabelas 6.1 e 6.2.

Tabela 6.1: Especificações de hardware e software do Ambiente 980.

CPU	GPU	Software
Intel Core i7-3770 CPU, 3.40GHz 4 Cores 128KB L1 Cache 8GB RAM 1 TB Disco	GeForce GTX 980 Ti 2816 CUDA cores Arquitetura Maxwell 1,19 GHz 6GB RAM	CentOS Linux 7 nvcc release 8.0 gcc version 4.8.5 CUDA Toolkit 8.0

6.2 Conjuntos de sequências comparados

Os testes de desempenho foram realizados utilizando 6 conjuntos de sequências do grupo *xLarge* disponíveis no *benchmark extHomFam* [13]. A Tabela 6.3 apresenta os conjuntos escolhidos. Na primeira coluna é informado o ID do conjunto, conforme utilizado no

Tabela 6.2: Especificações de hardware e software do Ambiente 680.

CPU	GPU	Software
Intel Core i7-3770 CPU, 3.40GHz 4 Cores 128KB L1 Cache 8GB RAM 1 TB Disco	GeForce GTX 680 1536 CUDA cores Arquitetura Kepler 1,01 GHz 2GB RAM	Ubuntu 12.04.2 LTS nvcc release 6.5 gcc version 4.6.3 CUDA Toolkit 6.5

extHomFam. A segunda coluna apresenta o número de sequências de cada conjunto. As colunas 3, 4 e 5 apresentam os tamanhos da menor sequência, da maior sequência e o tamanho médio das sequências, respectivamente. Por fim, a coluna 6 apresenta o tamanho do *dataset*, em MB.

Tabela 6.3: Descrição dos conjuntos de sequências do extHomFam utilizados.

ID	Número de sequências	Tamanho Menor sequência	Tamanho Maior sequência	Tamanho médio	Tamanho dataset(MB)
bac_luciferase	25534	30	405	294	8,4
Peptidase_M24	25574	23	354	218	6,4
fn3	49584	21	157	84	5,7
Cyclodex_gly_tran	50280	18	686	190	11,2
mdd	104692	24	387	120	15,7
HATPase_c	151443	31	243	115	21,9

Foram também utilizadas 4 conjuntos de sequências sintéticas, geradas aleatoriamente, com tamanho variando de 10.000 a 100.000, com sequências de 130 a 149 caracteres, como pode ser visto na Tabela 6.4. Na primeira coluna é informado o ID do conjunto. A segunda coluna apresenta o número de sequências de cada conjunto. As colunas 3, 4 e 5 apresentam os tamanhos da menor sequência, da maior sequência e o tamanho médio das sequências, respectivamente. Por fim, a coluna 6 apresenta o tamanho do *dataset*, em MB.

Tabela 6.4: Descrição dos conjuntos de sequências sintéticas gerados.

ID	Número de sequências	Tamanho Menor sequência	Tamanho Maior sequência	Tamanho médio	Tamanho dataset(MB)
syn_10000	10000	130	149	140	1,5
syn_25000	25000	130	149	140	3,8
syn_50000	50000	130	149	140	7,6
syn_100000	100000	130	149	140	15,2

6.3 Resultados obtidos

Para a execução dos experimentos foram utilizados blocos com 256 *threads* de GPU por bloco, $T = 256$, e um *grid* com tantos blocos quanto necessário para cobrir todas as sequências, ou seja, $B = N/T$, com N sendo o número de sequências do conjunto. Esses valores foram definidos empiricamente.

Quanto maior for o número de sequências de referência melhor será a qualidade final da árvore gerada [3], contudo isso também implica em um crescimento no tempo execução e da memória necessários para a construção da árvore-guia. Por isso, foi decidido utilizar como número máximo de sequências de referência, n, o valor de 5.000, que foi o valor mais alto que os nossos ambientes de teste foram capazes de lidar. Cabe lembrar que o número de sequências de referência realmente utilizado n' é menor ou igual a n (Seção 5.1).

O tamanho da seção (Seção 5.3) foi definido como 32 conforme o tamanho da *warp* das GPUs utilizadas.

Por fim, os demais parâmetros utilizados foram os definidos por padrão na execução do Parttree, conforme apresentado na página do autor [55].

6.3.1 Tempo de execução do CUDA-Parttree

Para avaliar o ganho do CUDA-Parttree sobre o Parttree original, medimos o tempo de execução do CUDA-Parttree (Figura 5.3 e Algoritmo 6) e o tempo de execução da rotina *calculateSimilarity* do Parttree (Algoritmo 3 linha 16 e Algoritmo 4), pois a rotina *calculateSimilarity* do Parttree foi substituída pelo CUDA-Parttree na execução em GPU.

Ambas as rotinas fazem o alinhamento entre as n' sequências de referência e as N sequências base. No Parttree, isso compreende a geração dos mapas de referência e as comparações dos mapas com as sequências base. Já no CUDA-Parttree, isso engloba a conversão dos dados, a computação em GPU e o retorno dos dados ao Parttree (Seções 5.3 a 5.5).

Tabela 6.5: Tempos de execução do *calculateSimilarity*, no Parttree e no CUDA-Parttree, no Ambiente 980.

ID	Parttree(s)	CUDA Parttree(s)	Speedup total
bac_luciferase	15,47	6,48	2,39x
Peptidase_M24	10,29	5,64	1,83x
fn3	27,17	13,47	2,02x
Cyclodex_gly_tran	33,94	13,18	2,58x
mdd	54,02	24,96	2,16x
HATPase_c	121,56	68,65	1,77x

A Tabela 6.5 apresenta, para os seis conjuntos de teste da Tabela 6.3, os tempos de execução do cálculo das distâncias entre as sequências base e as sequências de referência (área 3 da Figura 5.2) do Parttree (CPU 1 core), CUDA-Parttree e o *speedup* do CUDA-Parttree sobre o Parttree. Como pode ser visto, o CUDA-Parttree conseguiu um *speedup* médio de 2,12x em relação ao tempo do Parttree no Ambiente 980 e o melhor *speedup* foi 2,58x para o conjunto *Cyclodex_gly_tran*. Nesse caso, o tempo de execução foi reduzido de 33,94s para 13,18s.

Tabela 6.6: Tempos de execução do *calculateSimilarity*, no Parttree e no CUDA-Parttree, no Ambiente 680.

ID	Parttree (s)	CUDA Parttree (s)	Speedup total
bac_luciferase	15,41	15,78	0,98x
Peptidase_M24	10,19	12,03	0,85x
fn3	27,14	26,32	1,03x
Cyclodex_gly_tran	33,71	30,81	1,09x
mdd	54,47	48,38	1,13x
HATPase_c	121,29	102,18	1,19x

A Tabela 6.6 apresenta os tempos de execução das seis sequências no Ambiente 680. Os tempos de execução do Parttree original se mantém similares aos tempos obtidos no ambiente 980 pois a CPU possui a mesma configuração (Tabelas 6.1 e 6.2). Já os tempos de execução dos conjuntos no CUDA-Parttree foram maiores pois a GPU GTX 680 tem menos capacidade computacional que a GPU GTX 980 Ti. Assim, o *speedup* foi reduzido para uma média de 1,04x, com o maior obtendo *speedup* de 1,19x para o conjunto *HATPase_c*.

6.3.2 Speedup da computação

Os resultados obtidos no experimento descrito na Seção 6.3.1 ficaram aquém do esperado. Portanto, foi feita a medição somente do tempo de execução da computação da matriz de distâncias tanto no CUDA-Parttree como no Parttree original, que consiste na construção dos mapas de referência e a contagem de 6mers em comum entre as sequências base e as sequências de referência. No Parttree original, isso consiste da função *calculateSimilarity*. Já no CUDA-Parttree, essas etapas estão inclusas na fase de Computação em GPU (Seção 5.4, Algoritmo 9).

A Tabela 6.7 apresenta os resultados dessa medição para os seis conjuntos de teste no Ambiente 980. Os resultados obtidos indicaram que a computação da matriz de distâncias é realizada até 6,20x mais rápida, no caso do conjunto *bac_luciferase*, e o *speedup* médio foi de 5,79x. O pior *speedup* obtido foi de 5,40x.

Tabela 6.7: Tempos de execução da computação da matriz de distâncias no Ambiente 980.

ID	Parttree(s)	CUDA Parttree (s)	Speedup
bac_luciferase	15,47	2,49	6,20x
Peptidase_M24	10,29	1,79	5,74x
fn3	27,17	4,76	5,71x
Cyclodex_gly_tran	33,94	5,57	6,10x
mdd	54,02	9,62	5,62x
HATPase_c	121,56	22,53	5,40x

Tabela 6.8: Tempos de execução da computação da matriz de distâncias no Ambiente 680.

ID	Parttree(s)	CUDA Parttree(s)	Speedup total
bac_luciferase	15,41	12,37	1,25x
Peptidase_M24	10,19	8,70	1,17x
fn3	27,14	18,42	1,47x
Cyclodex_gly_tran	33,71	23,94	1,41x
mdd	54,47	34,11	1,60x
HATPase_c	121,29	75,79	1,60x

A Tabela 6.8 apresenta os resultados da medição para as sequências no ambiente 680. Nesse ambiente, foi obtido um *speedup* médio de $1{,}42x$ e um *speedup* máximo de $1{,}60x$.

Como houve um discrepância entre o *speedup* obtido no CUDA-Parttree (Tabelas 6.5 e 6.6) e os *speedup* do cálculo da rotina em GPU (Tabelas 6.7 e 6.8), foi decidido realizar o *profiling* do CUDA-Parttree para identificar o tempo de cada etapa.

6.3.3 *Profiling* do CUDA-Parttree

Para determinar o tempo gasto em cada rotina do CUDA-Parttree, foi feito um *breakdown* da execução, onde foram mensurados os tempos de cada etapa do CUDA-Parttree. As medições foram feitas sobre a etapas: (a) *sequence_setup*; (b) criação e envio dos mapas de referência (Calc/Send refMaps); (c) cálculo em GPU (GPU calc); (d) retorno da matriz para a CPU (distMtx retrieve); e (e) retorno do resultado para o Parttree (Parttree return). As medições foram feitas no Ambiente 980 e os tempos são apresentados em segundos. A etapa (a) corresponde à Conversão de dados - *Input* (Seção 5.3), as etapas (b), (c) e (d) fazem parte de fase de Computação em GPU (Seção 5.4) e a etapa (e) corresponde a fase de Recuperação de dados - *Output* (Seção 5.5).

A medição do tempo de execução em cada etapa do CUDA-Parttree (Tabela 6.9) revela que ele consegue um bom ganho no cálculo da matriz de distâncias (construção dos

Tabela 6.9: Tempos de execução de cada etapa do CUDA-Parttree no Ambiente 980.

ID	Sequence Setup	Calc & send RefMaps	GPU calc	distMtx Retrieve	Parttree return
	Input	Compute			Output
bac_luciferase	3,342	0,029	2,466	0,022	0,619
Peptidase_M24	3,319	0,024	1,769	0,020	0,506
fn3	5,792	0,086	4,670	0,177	2,739
Cyclodex_gly_tran	5,890	0,037	5,529	0,074	1,644
mdd	11,417	0,037	9,580	0,210	3,719
HATPase_c	16,531	0,120	22,411	10,936	18,230

mapas de referência e contagem de 6mers em comum entre as sequências) conseguindo *speedup* de até 6,20x no caso do conjunto *bac_luciferase*, mas que o impacto das fases de *Input* e *Output* reduz esse *speedup* para a média de 2,12x.

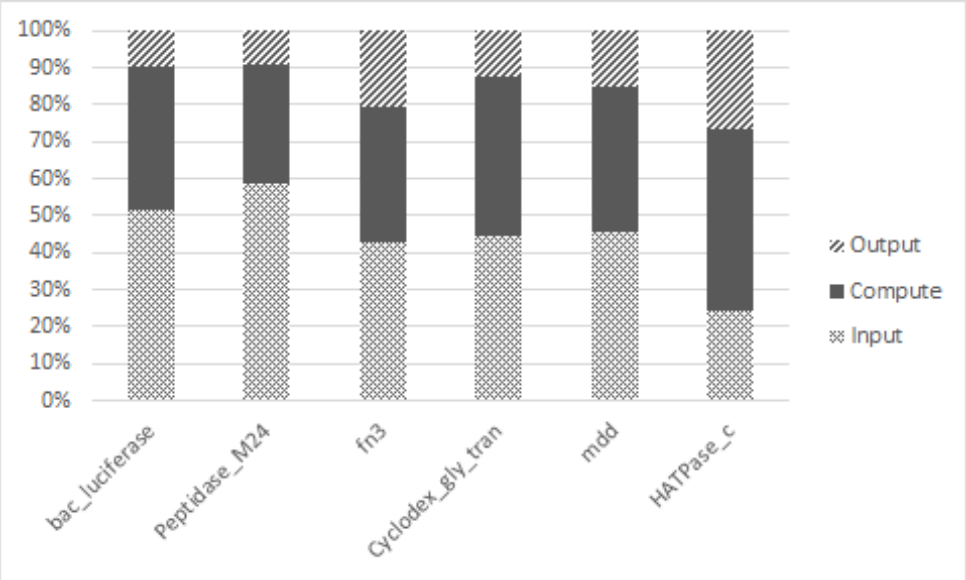

Figura 6.1: Proporção do tempo de execução das etapas do CUDA-Parttree.

A Tabela 6.9 e a Figura 6.1 mostram a distribuição do tempo de execução de cada parte do CUDA-Parttree, dividido em *input* (a), *compute* (b, c, e d) e *output* (e). Pode ser observado que o tempo da fase de computação (*compute*) é sempre inferior a soma dos tempos de *input* e *output*. Para os conjuntos menores (*bac_luciferase* e *Peptidase_24*), a fase de *input* requer mais tempo que as fases de *compute* e *output* somadas. Conforme os tamanho dos conjuntos aumenta, a proporção de tempo da fase de *output* tende a aumentar, chegando a ocupar 42,75% do tempo total no caso do maior conjunto (*HATPase_c*).

A Tabela 6.10 apresenta os tempos de cada etapa obtidos no Ambiente 680. Foi possível notar que os tempos das etapas (a, b, d e e) foram similares, e em alguns casos

Tabela 6.10: Tempos de execução de cada etapa do CUDA-Parttree no Ambiente 680.

ID	Sequence Setup	Calc & send RefMaps	GPU calc	distMtx Retrieve	Parttree return
	Input	Compute			Output
bac_luciferase	2,822	0,027	12,346	0,032	0,554
Peptidase_M24	2,791	0,025	8,678	0,030	0,506
fn3	5,100	0,063	18,355	0,156	2,648
Cyclodex_gly_tran	5,209	0,037	23,907	0,091	1,563
mdd	10,358	0,051	34,058	0,246	3,670
HATPase_c	14,838	0,068	75,722	2,878	8,562

até melhores quando comparados aos tempos obtidos no Ambiente 980, especialmente no caso do conjunto *HATPase_c*. Contudo, como era de se esperar, os tempos obtidos na etapa de computação em GPU (c) no Ambiente 680 são consideravelmente maiores.

Como pode ser visto nas Tabelas 6.9 e 6.10, os tempos da etapa de *GPU calc* no Ambiente 980 foram de 3,38x a 5,01x menores do que no Ambiente 680. Dada essa diferença significante relacionada às configurações dos ambientes, foi decidido utilizar apenas o Ambiente 980 para os demais experimentos.

6.3.4 Uso de memória

Nesse experimento, foi medida a quantidade de memória usada em cada execução em GPU. A Tabela 6.11 apresenta para cada conjunto o seu ID, o número de sequências total, o número de sequências de referência encontradas e o tamanho da matriz de distâncias calculada (distMtx).

Tabela 6.11: Número de sequências base, sequências de referência e tamanho da matriz de distâncias.

ID	# sequências base	# sequências de referência	# de células da distMtx	Tamanho da distMtx(MB)
bac_luciferase	25.534	906	23.133.804	352,99
Peptidase_M24	25.574	835	21.354.290	325,84
fn3	49.584	2.278	112.952.352	1723,52
Cyclodex_gly_tran	50.280	1.291	64.911.480	990,47
mdd	104.692	1.446	151.384.632	2309,95
HATPase_c	151.443	2.281	345.441.483	5271,02

Essas medições são importantes pois o tamanho da matriz de distâncias calculada é o fator determinante no tempo de execução do algoritmo, como é possível observar nos conjuntos $fn3$ e $Cyclodex$, que possuem um número próximo de sequências, 49.584 e 50.280, respectivamente, mas o número de células que são calculadas no $fn3$ é 1,74x

maior. Essa diferença ressalta o impacto da similaridade das sequências do conjunto no tempo de execução do algoritmo.

6.3.5 Alinhamento de Sequências Sintéticas

Nos conjuntos de sequências reais escolhidos para a realização dos experimentos (Tabela 6.3) notou-se que, após a etapa de filtragem, o número de sequências de referência restantes (n') era consideravelmente inferior ao valor máximo definido (n), o que indica que os conjuntos possuem sequências muito similares entre si.

A fim de analisar o desempenho do CUDA-Parttree para conjuntos com sequências mais distantes entre si, foram usados quatro conjuntos de sequências sintéticas, gerados aleatoriamente, com 10.000, 25.000, 50.000 e 100.000 sequências (Tabela 6.4).

Tabela 6.12: Tempos de execução do *calculateSimilarity*, no Parttree e no CUDA-Parttree, no Ambiente 680.

ID	Parttree(s)	CUDA Parttree(s)	Speedup Total
Syn_10000	9,50	4,73	2,01x
Syn_25000	41,22	14,86	2,77x
Syn_50000	98,96	34,54	2,87x
Syn_100000	209,54	70,40	2,98x

Como pode ser visto na Tabela 6.12, o CUDA-Parttree obteve um *speedup* maior do que nos conjuntos obtidos do extHomFam (Tabelas 6.5 e 6.6), conseguindo um *speedup* de $2{,}98x$ no maior conjunto. Nos casos testados nenhuma sequência foi retirada na etapa de filtragem, o que significa que $n' = 5.000$ e a matriz de distâncias calculada para o último conjunto possuía 500 milhões de células (5.000×100.000).

Tabela 6.13: Tempos de execução de cada etapa do CUDA-Parttree no Ambiente 980 para as sequências sintéticas.

ID	Sequence Setup	Calc & send RefMaps	GPU calc	distMtx Retrieve	Parttree return
	Input	Compute			Output
Syn_10000	1,68	0,13	1,85	0,05	1,01
Syn_25000	3,23	0,13	8,30	0,12	3,09
Syn_50000	5,74	0,13	22,56	0,23	5,87
Syn_100000	10,90	0,12	46,88	0,69	11,80

A Tabela 6.13 apresenta o tempo de execução de cada etapa do CUDA-Parttree para as sequências sintéticas (Tabela 6.4). Para essas sequências, a maior parte do tempo é gasto na fase de computação, como pode ser visto na Figura 6.2, já que $n' = n$. Isso indica

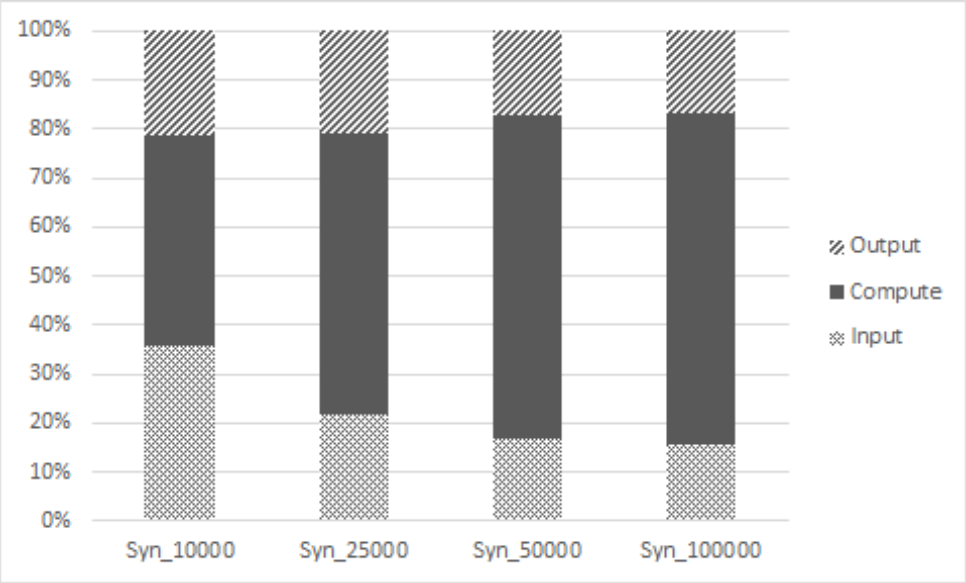

Figura 6.2: Proporção do tempo de execução das etapas do CUDA-Parttree para as sequências sintéticas.

que, quanto mais próximo for n' de n, melhor será o desempenho do CUDA-Parttree em relação ao Parttree.

Tabela 6.14: Tempos de execução da computação da matriz de distâncias, no Parttree e no CUDA-Parttree, das sequências sintéticas .

ID	Parttree(s)	CUDA Parttree(s)	Speedup Total
Syn_10000	9,50	1,98	4,79x
Syn_25000	41,22	8,43	4,89x
Syn_50000	98,96	22,69	4,36x
Syn_100000	209,54	47,00	4,46x

Como pode ser visto na Tabela 6.14, na comparação do tempo de execução do cálculo da matriz de distâncias, o CUDA-Parttree obteve um *speedup* de até 4,89x, reduzindo o tempo de 41,22s para 8,43s para o conjunto *Syn_25000*.

6.4 Comparação com o FAMSA

O FAMSA (Seção 4.5) foi o algoritmo de alinhamento múltiplo heurístico de sequências para conjunto com dezenas de milhares de sequências que mais se destacou no estado da arte. O autor apresentou alinhamentos mais rápidos e acurados que os demais algoritmos comparados, incluindo o Parttree em seus experimentos.

Para avalizar o desempenho do CUDA-Parttree foi decidido comparar os tempos totais de execução do FAMSA, Parttree e CUDA-Parttree, usando os mesmos parâmetros utilizados nos experimentos do FAMSA, no Ambiente 980 com as 6 sequências reais (Tabela 6.3). Como os resultados do Parttree e do CUDA-Parttree foram muito semelhantes, apenas os resultados do CUDA-Parttree são apresentados.

Tabela 6.15: Tempos de execução totais do FAMSA e do CUDA-Parttree, no Ambiente 980 para os conjuntos de sequências reais.

ID	FAMSA(s)	CUDA Parttree(s)	Speedup total
bac_luciferase	55,96	70,67	0,79x
Peptidase_M24	56,67	60,33	0,94x
fn3	91,35	118,33	0,77x
Cyclodex_gly_tran	182,80	428,33	0,43x
mdd	330,20	326,67	1,01x
HATPase_c	707,78	528,00	1,34x

Como pode ser visto na Tabela 6.15, o FAMSA de fato obteve melhor resultado nos conjuntos menores de sequências. Contudo, para os maiores conjuntos (*mdd* e *HATPase_c*), o CUDA-Parttree conseguiu um *speedup* de até 1,34x em relação ao FAMSA. Nesse caso, o tempo foi reduzido de 11 minutos e 46 segundos para 8 minutos e 48 segundos.

6.5 Considerações Finais

Os experimentos mostram que o CUDA-Parttree é capaz de calcular a matriz de distâncias em bem menos tempo que o Parttree original. Contudo, o ganho está relacionado aos conjuntos de sequências utilizados e ao *hardware* utilizado.

Usando-se a GPU GTX 980 Ti, conseguiu-se um *speedup* máximo de 2,58x reduzindo-se o tempo de execução de 33,94s para 13,18s. Como esse *speedup*, apesar de bom, ficou abaixo do esperado, fizemos o *breakdown* da execução do CUDA-Parttree. Com o *breakdown* ficou claro que o cálculo de 6mers em GPU está rápido, tendo sido reduzido de 33,94s para 5,57s no conjunto *Cyclodex_gly_tran* (*speedup* de 6,10x). No entanto, as etapas de *sequence_setup*(*input*) e a conversão para o formato original do Parttree (*output*) tem um impacto negativo alto no *speedup*. Observamos também que, para os conjuntos testados, apesar de definirmos $n = 5.000$, o número de sequências de referência (n') utilizado fica bem abaixo disso, o que impacta também no *speedup*.

O teste com sequências sintéticas ($n = n' = 5.000$) mostra que atingimos *speedups* de até 2,98x (100.000 sequências sintéticas) na execução do cálculo da área 3 da Figura 5.2, reduzindo o tempo de execução de 209,54s para 47,00s.

A comparação com o FAMSA mostrou que, para os maiores conjuntos de sequências testados, o CUDA-Parttree foi mais rápido no nosso ambiente de testes, conseguindo um *speedup* de até 1,34x.

Capítulo 7

Conclusão e Trabalhos Futuros

7.1 Conclusão

Nesta dissertação foi proposto e avaliado o CUDA-Parttree, uma estratégia paralela em GPU para alinhamento múltiplo heurístico de dezenas de milhares de sequências com o Parttree (Seção 2.4.3).

O CUDA-Parttree executa a contagem de 6mers em comum entre as sequências de referência e as sequências base em GPU, que é utilizada para calcular a matriz de distâncias do Parttree. A utilização de uma nova estrutura de dados para representar as sequências bem como o uso de *paddings* permitiram reduzir o tempo ocioso das *threads* da GPU e melhoraram o desempenho do algoritmo (Seção 5.3).

Os resultados do experimentos realizados mostraram que o CUDA-Parttree foi capaz de calcular a matriz de distâncias em bem menos tempo que o Parttree (Seção 6.3.1), mas que o *speedup* de computação foi impactado negativamente pelas transformações necessárias para adaptar as sequências ao cálculo em GPU e para transferir a matriz calculada para o formato usado nas demais etapas do Parttree (Seção 6.3.3).

Usando uma GPU GTX 980 Ti para alinhar o conjunto *Cyclodex_gly_tran* (50.280 sequências), o CUDA-Parttree conseguiu um *speedup* de 6,10x, quando comparado ao Parttree, reduzindo o tempo de execução de 33,94s para 5,57s. Ao se considerar as etapas de conversão de dados, cópia para GPU, cópia para a CPU e retorno de dados para o Parttree, o *speedup* para esses conjunto foi de 2,58x, que ainda é um ganho muito bom.

Na mesma GPU, no alinhamento do conjunto de sequências sintéticas *Syn_100000* (100.000 sequências) o CUDA-Parttree conseguiu um *speedup* na computação da matriz de distâncias de 4,46x em relação ao Parttree, reduzindo o tempo de 209,54s para 47,00s. Considerando as demais etapas do CUDA-PArttree, o *speedup* obtido para esse conjunto foi de 2,98x.

7.2 Trabalhos Futuros

Como Trabalhos futuros sugerimos:

- **Executar o CUDA-Parttree em um GPUs mais modernas, especialmente da arquitetura Volta:** As GPUs da Arquitetura Volta possuem muito mais CUDA *cores* que as GPUs utilizadas nos nossos testes, possuem maior velocidade de transferência de dados entre CPU e GPU, e vice-versa, e principalmente possuem uma memória global HBM2 que permite o acesso muito mais rápido que as memórias GDDR5. Acreditamos que obteremos *speedups* bem maiores com a arquitetura Volta.
- **Utilizar técnicas de movimentação assíncrona de dados entre CPU e GPU:** Cada bloco da GPU é responsável pelo cálculo do escore entre um conjunto de sequências e todas as sequências de referência, logo, eles são independentes dos demais blocos. Assim, a computação em GPU de um bloco pode iniciar assim que as sequências de um bloco estiverem na memória da GPU e a linha calculada na matriz de distâncias pode ser retornada à CPU assim que a GPU encerra o cálculo do bloco. Acreditamos que essa estratégia permitiria reduzir bastante o tempo de execução do algoritmo.
- **Utilizar *threads* em CPU para acelerar a conversão de sequências e o retorno para o Parttree:** A conversão das sequências de um vetor de inteiros para um vetor de *structs* pode ser paralelizada usando *threads* em CPU para reduzir o tempo gasto. Além disso, também é possível usar *threads* em CPU para paralelizar a transferência da matriz de distâncias para o formato do Parttree.
- **Adaptar o CUDA-Parttre para o alinhamento de sequências de DNA/RNA:** O alinhamento de DNA/RNA é feito de forma muito parecida a do alinhamento de proteínas, alterando o comprimento do *kmer* utilizado para 4 e fazendo as adaptações necessárias.
- **Aumentar a escalabilidade do algoritmo:** O uso do CUDA-Parttree é limitado pelo tamanho da memória global das GPUs. Contudo, como não existe dependência entre as sequências, é possível adaptar o CUDA-Parttree para conjuntos maiores utilizando múltiplas invocações de *kernel*.
- **Adaptar a contagem de 6mers em comum para uso em outras aplicações:** A contagem de 6mers pode ser utilizada em outras aplicações de Bioinformática, além de outros algoritmos de alinhamento múltiplo, também são usadas na montagem de sequências (*sequence assembly*).

Referências

[1] Mount, David: *Bioinformatics: Sequence and Genome Analysis*. Cold Spring Harbor Laboratory Press, 2nd edição, 2013, ISBN 0879697121. 1, 8, 9, 18

[2] Wang, Lusheng e Tao Jiang: *On the complexity of multiple sequence alignment*. Journal of computational biology, 1(4):337–348, 1994. 1, 9, 11

[3] Durbin, Richard, Sean R Eddy, Anders Krogh e Graeme Mitchison: *Biological sequence analysis: probabilistic models of proteins and nucleic acids*. Cambridge university press, 1998. 1, 4, 5, 6, 7, 10, 11, 12, 13, 17, 32, 53

[4] Deorowicz, Sebastian, Agnieszka Debudaj-Grabysz e Adam Gudyś: *FAMSA: Fast and accurate multiple sequence alignment of huge protein families*. Scientific reports, 6, 2016. 1, 34

[5] Katoh, Kazutaka e Hiroyuki Toh: *PartTree: an algorithm to build an approximate tree from a large number of unaligned sequences*. Bioinformatics, 23(3):372–374, 2006. 1, 19, 20, 35, 37

[6] Sievers, Fabian, Andreas Wilm, David Dineen, Toby J Gibson, Kevin Karplus, Weizhong Li, Rodrigo Lopez, Hamish McWilliam, Michael Remmert, Johannes Söding *et al.*: *Fast, scalable generation of high-quality protein multiple sequence alignments using Clustal Omega*. Molecular systems biology, 7(1):539, 2011. 1, 35

[7] Hung, Che Lun, Yu Shiang Lin, Chun Yuan Lin, Yeh Ching Chung e Yi Fang Chung: *CUDA ClustalW: An efficient parallel algorithm for progressive multiple sequence alignment on Multi-GPUs*. Computational biology and chemistry, 58:62–68, 2015. 1, 29, 30

[8] Katoh, Kazutaka, Kazuharu Misawa, Kei ichi Kuma e Takashi Miyata: *MAFFT: a novel method for rapid multiple sequence alignment based on fast Fourier transform*. Nucleic acids research, 30(14):3059–3066, 2002. 2, 14, 15, 16, 17, 19

[9] Mi, Huaiyu, Anushya Muruganujan e Paul D Thomas: *Panther in 2013: modeling the evolution of gene function, and other gene attributes, in the context of phylogenetic trees*. Nucleic acids research, 41(D1):D377–D386, 2012. 2

[10] Nam-phuong, D Nguyen, Siavash Mirarab, Keerthana Kumar e Tandy Warnow: *Ultra-large alignments using phylogeny-aware profiles*. Genome biology, 16(1):124, 2015. 2

[11] Lamnidis, Thiseas C, Kerttu Majander, Choongwon Jeong, Elina Salmela, Anna Wessman, Vyacheslav Moiseyev, Valery Khartanovich, Oleg Balanovsky, Matthias Ongyerth, Antje Weihmann *et al.*: *Ancient fennoscandian genomes reveal origin and spread of siberian ancestry in europe*. Nature communications, 9(1):5018, 2018. 2

[12] Ghosh, Priyanka, Sriram Krishnamoorthy e Ananth Kalyanaraman: *Pakman: Scalable assembly of large genomes on distributed memory machines*. bioRxiv, página 523068, 2019. 2

[13] Gudys, Adam: *extHomFam benchmark*, 2016. `https://doi.org/10.7910/DVN/BO2SVW`. 2, 35, 51

[14] Gusfield, Dan: *Algorithms on strings, trees and sequences: computer science and computational biology*. Cambridge university press, 1997. 5, 6, 8, 12

[15] Dayhoff, Margaret O e Robert M Schwartz: *A model of evolutionary change in proteins*. Em *In Atlas of protein sequence and structure*, 1978. 6, 7

[16] Henikoff, Steven e Jorja G Henikoff: *Amino acid substitution matrices from protein blocks*. Proceedings of the National Academy of Sciences, 89(22):10915–10919, 1992. 6, 13

[17] Altschul, Stephen F, Raymond J Carroll e David J Lipman: *Weights for data related by a tree*. Journal of molecular biology, 207(4):647–653, 1989. 7

[18] Carrillo, Humberto e David Lipman: *The multiple sequence alignment problem in biology*. SIAM Journal on Applied Mathematics, 48(5):1073–1082, 1988. 9, 10

[19] Spouge, John L: *Speeding up dynamic programming algorithms for finding optimal lattice paths*. SIAM Journal on Applied Mathematics, 49(5):1552–1566, 1989. 10

[20] Hart, Peter E, Nils J Nilsson e Bertram Raphael: *A formal basis for the heuristic determination of minimum cost paths*. IEEE transactions on Systems Science and Cybernetics, 4(2):100–107, 1968. 10

[21] Feng, Da Fei e Russell F Doolittle: *Progressive sequence alignment as a prerequisitetto correct phylogenetic trees*. Journal of molecular evolution, 25(4):351–360, 1987. 11, 13

[22] Higgins, Desmond G e Paul M Sharp: *CLUSTAL: a package for performing multiple sequence alignment on a microcomputer*. Gene, 73(1):237–244, 1988. 11, 13

[23] 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. 12

[24] Smith, Temple F, Michael S Waterman *et al.*: *Identification of common molecular subsequences*. Journal of molecular biology, 147(1):195–197, 1981. 12, 19

[25] Hirschberg, Daniel S: *Algorithms for the longest common subsequence problem*. Journal of the ACM (JACM), 24(4):664–675, 1977. 12, 29

[26] Sneath, Peter HA, Robert R Sokal *et al.*: *Numerical taxonomy. The principles and practice of numerical classification.* 1973. 12, 13

[27] Thompson, Julie D, Desmond G Higgins e Toby J Gibson: *CLUSTAL W: improving the sensitivity of progressive multiple sequence alignment through sequence weighting, position-specific gap penalties and weight matrix choice.* Nucleic acids research, 22(22):4673–4680, 1994. 13

[28] Saitou, Naruya e Masatoshi Nei: *The neighbor-joining method: a new method for reconstructing phylogenetic trees.* Molecular biology and evolution, 4(4):406–425, 1987. 13

[29] Press, William H, Saul A Teukolsky, William T Vetterling e Brian P Flannery: *Numerical recipes in C*, volume 2. Cambridge university press Cambridge, 1996. 14

[30] Jones, David T, William R Taylor e Janet M Thornton: *The rapid generation of mutation data matrices from protein sequences.* Bioinformatics, 8(3):275–282, 1992. 17

[31] Liu, Yongchao, Bertil Schmidt e Douglas L Maskell: *MSAProbs: multiple sequence alignment based on pair hidden Markov models and partition function posterior probabilities.* Bioinformatics, 26(16):1958–1964, 2010. 17

[32] Barton, Geoffrey J e Michael JE Sternberg: *A strategy for the rapid multiple alignment of protein sequences: confidence levels from tertiary structure comparisons.* Journal of molecular biology, 198(2):327–337, 1987. 18

[33] Hirosawa, Makoto, Yasushi Totoki, Masaki Hoshida e Masato Ishikawa: *Comprehensive study on iterative algorithms of multiple sequence alignment.* Bioinformatics, 11(1):13–18, 1995. 19

[34] Notredame, Cédric, Desmond G Higgins e Jaap Heringa: *T-Coffee: A novel method for fast and accurate multiple sequence alignment.* Journal of molecular biology, 302(1):205–217, 2000. 19

[35] Katoh, Kazutaka e Hiroyuki Toh: *Recent developments in the MAFFT multiple sequence alignment program.* Briefings in bioinformatics, 9(4):286–298, 2008. 19

[36] Finn, Robert D, Jaina Mistry, Benjamin Schuster-Böckler, Sam Griffiths-Jones, Volker Hollich, Timo Lassmann, Simon Moxon, Mhairi Marshall, Ajay Khanna, Richard Durbin *et al.*: *Pfam: clans, web tools and services.* Nucleic acids research, 34(suppl_1):D247–D251, 2006. 20, 37

[37] Kirk, David B e W Hwu Wen-Mei: *Programming massively parallel processors: a hands-on approach.* Morgan kaufmann, 2016. 21, 22, 23, 26, 27

[38] Sutter, Herb e James Larus: *Software and the concurrency revolution.* Queue, 3(7):54–62, 2005. 21

[39] NVIDIA: *Nvidia tesla v100 gpu architecture.* https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf, 2017. 22, 24

[40] Sanders, Jason e Edward Kandrot: *CUDA by Example: An Introduction to General-Purpose GPU Programming, Portable Documents.* Addison-Wesley Professional, 2010. 22, 25

[41] Flynn, Michael J: *Some computer organizations and their effectiveness.* IEEE transactions on computers, 100(9):948–960, 1972. 23

[42] Wittenbrink, Craig M, Emmett Kilgariff e Arjun Prabhu: *Fermi gf100 gpu architecture.* IEEE Micro, 31(2):50–59, 2011.

[43] NVIDIA: *Nvidia geforce gtx 680.* https://www.nvidia.com/content/PDF/product-specifications/GeForce_GTX_680_Whitepaper_FINAL.pdf, 2012. 23, 24

[44] NVIDIA: *Nvidia geforce gtx 980*, 2014. 23, 25

[45] NVIDIA: *Nvidia tesla p100.* https://images.nvidia.com/content/pdf/tesla/whitepaper/pascal-architecture-whitepaper.pdf, 2016. 24

[46] Zhu, Xiangyuan, Kenli Li, Ahmad Salah, Lin Shi e Keqin Li: *Parallel implementation of MAFFT on CUDA-enabled graphics hardware.* IEEE/ACM Transactions on Computational Biology and Bioinformatics, 12(1):205–218, 2015. 30, 31

[47] Chen, Xi, Chen Wang, Shanjiang Tang, Ce Yu e Quan Zou: *CMSA: a heterogeneous CPU/GPU computing system for multiple similar RNA/DNA sequence alignment.* BMC bioinformatics, 18(1):315, 2017. 32

[48] Zou, Quan, Xiao Shan e Yi Jiang: *A novel center star multiple sequence alignment algorithm based on affine gap penalty and k-band.* Physics Procedia, 33:322–327, 2012. 33

[49] Zou, Quan, Qinghua Hu, Maozu Guo e Guohua Wang: *HAlign: Fast multiple similar DNA/RNA sequence alignment based on the centre star strategy.* Bioinformatics, 31(15):2475–2481, 2015. 33

[50] Lassmann, Timo e Erik LL Sonnhammer: *Kalign–an accurate and fast multiple sequence alignment algorithm.* BMC bioinformatics, 6(1):298, 2005. 33

[51] González-Domínguez, Jorge, Yongchao Liu, Juan Touriño e Bertil Schmidt: *MSAProbs-MPI: parallel multiple sequence aligner for distributed-memory systems.* Bioinformatics, 32(24):3826–3828, 2016. 33

[52] Sibson, Robin: *Slink: an optimally efficient algorithm for the single-link cluster method.* The computer journal, 16(1):30–34, 1973. 34

[53] Cole, James R, Benli Chai, Ryan J Farris, Qiong Wang, SA Kulam, Donna M McGarrell, George M Garrity e James M Tiedje: *The Ribosomal Database Project (RDP-II): sequences and tools for high-throughput rRNA analysis.* Nucleic acids research, 33(suppl_1):D294–D296, 2005. 37

[54] NVIDIA: *Cuda C Programming guide*, 2018. 45

[55] Katoh, Kazutaka: *Mafft algorithms.* `https://mafft.cbrc.jp/alignment/software/algorithms/algorithms.html`, 2019. 53