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

Estratégia Paralela Exata para o Alinhamento Múltiplo de Sequências Biológicas Utilizando Unidades de Processamento Gráfico (GPU)

Daniel Sundfeld Lima

Dissertação apresentada como requisito parcial
para conclusão do Mestrado em Informática

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

Brasília
2012

Universidade de Brasília — UnB
Instituto de Ciências Exatas
Departamento de Ciência da Computação
Mestrado em Informática

Coordenadora: Prof.ª Dr.ª Mylene Christine Queiroz de Farias

Banca examinadora composta por:

Prof.ª Dr.ª Alba Cristina M. A. de Melo (Orientadora) — UnB
Prof. Dr. Edson Norberto Cáceres — UFMS
Prof. Dr. Pedro de Azevedo Berger — UnB

CIP — Catalogação Internacional na Publicação

Sundfeld Lima, Daniel.

Estratégia Paralela Exata para o Alinhamento Múltiplo de Sequências Biológicas Utilizando Unidades de Processamento Gráfico (GPU) / Daniel Sundfeld Lima. Brasília : UnB, 2012.
73 p. : il. ; 29,5 cm.

Dissertação (Mestrado) — Universidade de Brasília, Brasília, 2012.

1. Bioinformática, 2. Alinhamento Múltiplo de Sequências,
3. Computação de Alto Desempenho, 4. GPU

CDU 004.4

Endereço: Universidade de Brasília
Campus Universitário Darcy Ribeiro — Asa Norte
CEP 70910-900
Brasília–DF — Brasil

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

Estratégia Paralela Exata para o Alinhamento Múltiplo de Sequências Biológicas Utilizando Unidades de Processamento Gráfico (GPU)

Daniel Sundfeld Lima

Dissertação apresentada como requisito parcial
para conclusão do Mestrado em Informática

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

Prof. Dr. Edson Norberto Cáceres
UFMS

Prof. Dr. Pedro de Azevedo Berger
UnB

Prof.ª Dr.ª Mylene Christine Queiroz de Farias
Coordenadora do Mestrado em Informática

Brasília, 28 de agosto de 2012

Dedicatória

Dedico este trabalho à minha família, vocês são minha base e meu apoio. Vocês me apoiaram nesta jornada e me incentivaram todos os momentos necessários. Obrigado por tudo.

Agradecimentos

Agradeço a Deus, pelas oportunidades e desafios planejados para mim. Tu és causa primária de todas as coisas e nada se realiza sem Tua vontade.

Agradeço à minha esposa Elaine Cristine. Casamos entre a apresentação e conclusão deste trabalho e agora é a Sra. Elaine Sundfeld. Obrigado pelo apoio, troca de ideias, dicas e paciência que foram necessárias para que eu chegasse neste ponto. Você é a minha família, a base do nosso lar. Eu nunca conseguiria chegar onde estou sem teu amor e apoio incondicional.

À minha orientadora Alba, agradeço profundamente pelo tempo dedicado, os debates sobre as estratégias, o planejamento e as contínuas revisões e reuniões para que todo esse trabalho ficasse pronto. Sem sua vasta experiência e excelente orientação eu nunca teria chegado até aqui.

Aos meus pais, avós e minha família, eu agradeço. Vocês são os exemplos que sigo nesta vida, aqueles que me espelho para ter sucesso e busco continuamente ser motivo de orgulho para vocês.

Agradeço o Sr. Rodrigo Aranha que utilizou ferramentas de renderização gráficas para criação das imagens que esclareceram como as estratégias se comportam ao percorrer o espaço de busca. A criação de tais imagens utilizando ferramentas mais simples era tarefa árdua e nunca teria produzido um resultado tão bom. Agradeço, novamente, a Sra. Elaine Sundfeld que criou a animação baseada nessas imagens. A todos vocês, muito obrigado.

Resumo

O alinhamento múltiplo de sequências biológicas é um problema muito importante em Biologia Molecular, pois permite que sejam detectadas similaridades e diferenças entre um conjunto de sequências. Esse problema foi provado NP-Difícil e, por essa razão, geralmente algoritmos heurísticos são usados para resolvê-lo. No entanto, a obtenção da solução ótima é bastante desejada e, por essa razão, existem alguns algoritmos exatos que solucionam esse problema para um número reduzido de sequências. Dentre esses algoritmos, destaca-se o método exato Carrillo-Lipman, que permite reduzir o espaço de busca utilizando um limite inferior e superior. Mesmo com essa redução, o algoritmo com Carrillo-Lipman executa-se em tempo exponencial. Com o objetivo de acelerar a obtenção de resultados, plataformas computacionais de alto desempenho podem ser utilizadas para resolver o problema do alinhamento múltiplo. Dentre essas plataformas, destacam-se as Unidades de Processamento Gráfico (GPU) devido ao seu potencial para paralelismo massivo e baixo custo. O objetivo dessa dissertação de mestrado é propor e avaliar uma estratégia paralela para execução do algoritmo Carrillo-Lipman em GPU. A nossa estratégia permite a exploração do paralelismo em granularidade fina, onde o espaço de busca é percorrido por várias *threads* em um cubo tridimensional, divido em janelas de processamento que são diagonais projetadas em duas dimensões. Os resultados obtidos com a comparação de conjuntos de 3 sequências reais e sintéticas de diversos tamanhos mostram que *speedups* de até 8,60x podem ser atingidos com a nossa estratégia.

Palavras-chave: Bioinformática, Alinhamento Múltiplo de Sequências, Computação de Alto Desempenho, GPU

Abstract

Multiple Sequence Alignment is a very important problem in Molecular Biology since it is able to detect similarities and differences in a set of sequences. This problem has been proven NP-Hard and, for this reason, heuristic algorithms are usually used to solve it. Nevertheless, obtaining the optimal solution is highly desirable and there are indeed some exact algorithms that solve this problem for a reduced number of sequences. Carrillo-Lipman is a well-known exact algorithm for the Multiple Sequence Alignment problem that is able to reduce the search space by using inferior and superior bounds. Even with this reduction, the Carrillo-Lipman algorithm executes in exponential time. High Performance Computing (HPC) Platforms can be used in order to produce results faster. Among the existing HPC platforms, GPUs (Graphics Processing Units) are receiving a lot of attention due to their massive parallelism and low cost. The goal of this MsC dissertation is to propose and evaluate a parallel strategy to execute the Carrillo-Lipman algorithm in GPU. Our strategy explores parallelism at fine granularity, where the search space is a tridimensional cube, divided on processing windows with bidimensional diagonals, explored by multiple threads. The results obtained when comparing several sets of 3 real and synthetic sequences show that speedups of 8.60x can be obtained with our strategy.

Keywords: Bioinformatics, Multiple Sequence Alignment, High Performance Computing, GPU

Sumário

1 Introdução **1**
1.1 Motivação . . . 1
1.2 Contribuições . . . 2
1.3 Estrutura do documento . . . 3

2 Alinhamento Múltiplo de Sequências Biológicas **4**
2.1 Sequências e Alinhamentos . . . 4
2.1.1 Caminhos . . . 5
2.2 Sistemas de escore . . . 6
2.2.1 *Sum-of-pairs*(SP) . . . 6
2.2.2 *Weighted sum-of-pairs* (WSP) . . . 6
2.3 Algoritmo Exato Simples: Processamento em todo o espaço de busca . . . 7
2.4 *Wavefront* tradicional e *Wavefront* multidimensional . . . 9
2.5 Método Carrillo-Lipman . . . 10
2.5.1 Carrillo-Lipman *Bound* . . . 11
2.5.2 Simplificação das fórmulas . . . 12
2.5.3 MSA 1.0 . . . 12
2.5.4 MSA 2.0 . . . 13
2.6 Algoritmos Heurísticos . . . 15
2.6.1 Alinhamento Progressivo . . . 15
2.6.2 Alinhamento Iterativo . . . 16
2.6.3 Outros Métodos . . . 17

3 Plataformas de Alto Desempenho **19**
3.1 Clusters de Computadores . . . 20
3.2 FPGAs . . . 20
3.3 GPUs . . . 21
3.3.1 Arquitetura CUDA . . . 22
3.3.2 Modelo de Programação em CUDA . . . 23
3.3.3 Arquitetura Fermi . . . 25

4 Alinhamento Múltiplo de Sequências em Plataformas de Alto Desempenho **27**
4.1 Propostas de Alinhamento Heurístico . . . 27
4.1.1 MT-ClustalW . . . 27
4.1.2 ClustalW-MPI . . . 28
4.1.3 ClustalW em FPGA . . . 29

4.1.4 GPU-ClustalW . . . 30
4.1.5 MSA-CUDA . . . 31
4.1.6 G-MSA . . . 32
4.2 Propostas de Alinhamento Exato . . . 33
4.2.1 MSA exato com MPI . . . 33
4.2.2 MSA exato em FPGA . . . 34
4.3 Quadro Comparativo . . . 35

5 Projeto Carrillo-Lipman em GPU para Alinhamento Múltiplo Exato 37
5.1 Considerações Iniciais . . . 37
5.2 Estratégia de Granularidade Grossa . . . 38
5.2.1 Visão Geral . . . 38
5.2.2 Cálculo do wavefront uma thread por célula . . . 39
5.2.3 Percorrendo o espaço de busca . . . 41
5.2.4 Utilizando o limite de Carrillo-Lipman . . . 42
5.2.5 Estruturas em Memória . . . 43
5.2.6 Pseudocódigo em GPU . . . 43
5.2.7 Análise da abordagem . . . 44
5.3 Estratégia de Granularidade Fina . . . 46
5.3.1 Visão Geral . . . 46
5.3.2 Cálculo do wavefront um bloco por célula . . . 47
5.3.3 Escolha do Menor Valor . . . 49
5.3.4 Percorrendo o espaço de busca . . . 50
5.3.5 Utilizando o limite de Carrillo-Lipman . . . 52
5.3.6 Estruturas em Memória . . . 52
5.3.7 Pseudocódigo em GPU . . . 53
5.3.8 Análise da Abordagem . . . 54

6 Resultados experimentais 56
6.1 Ambiente de testes . . . 56
6.2 Sequências selecionadas . . . 56
6.3 Dados Experimentais e Análise dos resultados . . . 58
6.3.1 Tempo de execução total das estratégias . . . 58
6.3.2 Comparação com o MSA 2.0 . . . 58
6.3.3 Tempos de invocação de kernels em GPU . . . 60

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

Referências 64

I Sequências Sintéticas 68

Lista de Figuras

2.1 Alinhamento de três sequências [5]. 5
2.2 Alinhamento par-a-par projetado. 5
2.3 Escore de um alinhamento múltiplo com a função SP [16]. 6
2.4 Escores de um alinhamento múltiplo com a função WSP. 7
2.5 Algoritmo exato para o alinhamento múltiplo de três sequências [16]. . . . 8
2.6 Passos executados no wavefront tradicional 10
2.7 *Wavefront* tridimensional 10

3.1 Arquitetura CUDA [40]. 22
3.2 Modelo de programação CUDA [43]. 23
3.3 Hierarquia da memória em CUDA [43]. 24
3.4 Arquitetura Fermi [41]. 25

4.1 Código otimizado do MT-ClustalW e fluxograma[6]. 28
4.2 Exemplo de árvore produzida na segunda fase do MSA-CUDA [28]. 32
4.3 Programação dinâmica em três dimensões [30]. 35

5.1 Resumo da estratégia de granularidade grossa. 38
5.2 Alterações no algoritmo para a estratégia grossa. 40
5.3 Primeiros passos para percorrer o espaço de busca utilizando a estratégia de granularidade grossa. 41
5.4 Passos finais para percorrer o espaço de busca utilizando a estratégia de granularidade grossa. 42
5.5 Código executado em GPU pela estratégia grossa. 45
5.6 Resumo da estratégia de granularidade fina. 46
5.7 Alterações no algoritmo para a estratégia fina. 48
5.8 Inicialização do array *local_scores*. 49
5.9 Pesquisa do menor valor 50
5.10 Primeiros passos da estratégia de granularidade fina. 51
5.11 Últimos passos da estratégia de granularidade fina. 52
5.12 Kernel msa executado em GPU pela estratégia fina. 55

Lista de Tabelas

2.1 Matriz de substituição PAM 250 [7]. 7

4.1 Resultados Experimentais do MSA exato com MPI [19] 33
4.2 Resultados Experimentais do MSA exato em FPGA [30]. 35
4.3 Comparação entre as estratégias paralelas heurísticas em plataformas de alto desempenho 36
4.4 Comparação entre as estratégias paralelas exatas em plataformas de alto desempenho. . 36

5.1 Índices do *wavefront* pré-calculado, e seu tamanho, pelo número de *threads* por bloco para 3 sequências. 39
5.2 Índices do *wavefront* pré-calculado, e seu tamanho, pelo número de *threads* por bloco para 4 sequências. 39
5.3 Estruturas e memória utilizada para k sequências de tamanho n na estratégia de granularidade grossa. 44
5.4 Valores das variáveis neigh e a fórmula de custo utilizada. 47
5.5 Estruturas e memória utilizada para k sequências de tamanho n na estratégia de granularidade fina. 53

6.1 Configuração da estação de trabalho (CPU, GPU, Software) utilizada nos testes. . . . 56
6.2 Sequências utilizadas nos testes. 57
6.3 Comparação das abordagens 58
6.4 Tempos de execução do MSA 2.0 e do Carrillo-Lipman em GPU 59
6.5 *Overhead* de invocação de *kernels* em GPU. 60

Capítulo 1

Introdução

1.1 Motivação

A Bioinformática é uma área multidisciplinar de pesquisa que visa o desenvolvimento de ferramentas e algoritmos para auxiliar os biólogos na análise de dados, com o objetivo de determinar a função ou estrutura das sequências biológicas e inferir informação sobre sua evolução nos organismos. A Bioinformática auxilia bastante a execução de pesquisa em Biologia Molecular. Nesse contexto, o crescimento de projetos que envolvem sequenciamento genético está próximo do exponencial [23]. Além disso, com o advento de técnicas de sequenciamento de última geração, os bancos de dados genômicos também apresentam crescimento exponencial, tornando-se cada vez maiores.

Por isso, existe uma quantidade enorme de sequências precisam ser comparadas entre si, sendo necessário desenvolver ferramentas cada vez mais rápidas e eficientes. Neste aspecto, o tempo necessário total para a comparação e a precisão do resultado obtido são fatores determinantes.

A comparação de sequências biológicas pode ocorrer entre duas sequências (alinhamento em pares) ou entre mais de duas sequências (alinhamento múltiplo). Utilizar mais de duas sequências pode facilitar a visualização das diferenças e semelhanças entre os conjuntos de sequências como um todo. Uma alternativa ao alinhamento múltiplo seria a execução de várias comparações entre duas sequências. Porém, ao alinhar as sequências em pares, as diferenças e semelhanças globais podem não ser claras [9]. Por outro lado, alinhando-se múltiplas sequências, essas diferenças e semelhanças podem se tornar claras, ou, até mesmo, óbvias [34]. De posse de um alinhamento múltiplo, os biólogos podem então entender, por exemplo, como a evolução atuou, descobrindo a diferença genética entre as espécies.

Obter o alinhamento múltiplo ótimo de um conjunto de sequências é um problema de alta complexidade computacional, que foi provado NP-Difícil [57]. Por isso, muitas soluções heurísticas para o problema do alinhamento múltiplo foram propostas. Essas soluções podem ser classificadas em diversos tipos, sendo que os mais representativos são o alinhamento progressivo e o alinhamento iterativo [34]. No alinhamento múltiplo progressivo, as sequências de maior similaridade são alinhadas e então outras sequências são adicionadas ao alinhamento. O alinhamento múltiplo iterativo permite o realinhamento de sequências, buscando minimizar o impacto causado por erros nos alinhamentos iniciais

das sequências, reduzindo a probabilidade de propagá-los para o alinhamento múltiplo final.

No entanto, as soluções heurísticas não garantem que o alinhamento múltiplo ótimo das sequências seja obtido, retornando geralmente um resultado aproximado. Para a obtenção do alinhamento múltiplo ótimo, existe um algoritmo simples que é uma generalização do algoritmo ótimo para obtenção do alinhamento par a par [34]. No entanto, esse algoritmo possui complexidade exponencial, percorre todo espaço de busca e não é usado.

Carrillo e Lipman [5] fizeram uma importante contribuição na área de alinhamentos múltiplos ótimos ao demonstrarem que não é necessário percorrer todo espaço de busca para se obter o alinhamento ótimo. Eles mostraram que, de posse de um alinhamento heurístico, é possível obter-se um limite superior para o alinhamento ótimo e um limite inferior pode ser obtido pelo alinhamento em pares de todas as sequências. Desta forma, células que não estejam incluídas nestes limites não contribuem para o alinhamento múltiplo ótimo e podem ser descartadas, diminuindo-se o espaço de busca e obtendo-se mesmo assim o resultado ótimo. Mesmo com essa redução a complexidade de tempo de Carrillo--Lipman ainda é exponencial.

Para acelerar a obtenção de resultados em algoritmos de Bioinformática, plataformas de computação de alto desempenho são normalmente utilizadas. Dentre essas plataformas, destacam-se as unidades de processamento gráfico (*Graphics Processing Units* - GPU). As GPUs oferecem paralelismo massivo ao combinar múltiplos *cores* com capacidade de execução de operações vetoriais em cada *core*. Atualmente, existem várias GPUs capazes de executar 1 trilhão de operações de ponto flutuante por segundo (Flops), com desempenho comparável a *clusters* de multicores [28].

Muitas soluções heurísticas para o problema do alinhamento múltiplo de sequências foram propostas para arquiteturas de alto desempenho, incluindo contribuições em *Clusters* [6, 25], FPGA [44] e GPU [28, 27, 4]. Por outro lado, existem poucas soluções para o alinhamento múltiplo exato de sequências em plataformas de alto desempenho. São encontradas duas soluções para o alinhamento ótimo, sendo que uma executa-se em *Cluster* [19] e a outra em FPGA [30]. A nosso conhecimento, não existe até agora estratégia para o alinhamento múltiplo exato em GPU.

1.2 Contribuições

O objetivo dessa dissertação de mestrado é então propor e avaliar uma estratégia para encontrar o alinhamento múltiplo ótimo utilizando GPU. O problema do alinhamento ótimo é proporcional ao tamanho das sequências e exponencial em relação ao número de sequências. Sendo assim, ele possui um alto paralelismo que pode ser aproveitado pelas GPUs. Para obtenção do alinhamento ótimo em GPU escolhemos o algoritmo Carrillo-Lipman [5]. A estratégia paralela proposta nessa dissertação explora o espaço de busca em granularidade fina, onde múltiplas *threads* em GPU contribuem para o cálculo de uma célula no espaço de busca. O espaço de busca é percorrido no formato cúbico tridimensional, dividido em janelas de processamento de diagonais projetadas em duas dimensões. Propusemos essa solução por causa de seu alto paralelismo potencial.

Como contribuição secundária dessa dissertação, propusemos e avaliamos uma estratégia alternativa de granularidade grossa, onde uma única *thread* calcula uma célula no espaço de busca. Como será detalhado no Capítulo 6, essa estratégia possui um de-

sempenho inferior à estratégia de granularidade fina. Ao final, uma comparação entre as estratégias e fatores que influenciam no desempenho da estratégia fina em GPU são avaliados.

1.3 Estrutura do documento

O restante dessa dissertação está organizado como se segue. O Capítulo 2 apresenta o problema do alinhamento múltiplo de sequências biológicas, discorrendo sobre os principais algoritmos e técnicas utilizadas. O Capítulo 3 revisa as arquiteturas de alto desempenho mais conhecidas, apresentando suas principais características. O Capítulo 4 apresenta e discute alguns trabalhos de Alinhamento Múltiplo de Sequências em arquiteturas de alto desempenho. A estratégia proposta nessa dissertação para o alinhamento múltiplo de sequências em GPU é detalhada no Capítulo 5. O Capítulo 6 apresenta os resultados experimentais. O Capítulo 7 conclui o trabalho e apresenta sugestões de trabalhos futuros. Finalmente, o Anexo I apresenta as sequências sintéticas utilizadas neste trabalho.

Capítulo 2

Alinhamento Múltiplo de Sequências Biológicas

Um dos problemas mais básicos em Bioinformática consiste da comparação de sequências biológicas [34]. O resultado dessa operação é um escore que representa a similaridade entre as sequências e, adicionalmente, um alinhamento que representa claramente as semelhanças e diferenças entre as sequências.

Do ponto de vista matemático, a comparação entre sequências biológicas configura-se como um problema de reconhecimento aproximado de padrões [9]. Podem ser comparadas duas sequências (alinhamento em pares) ou mais de duas sequências (alinhamento múltiplo).

Neste capítulo será detalhado o problema do alinhamento múltiplo de sequências.

2.1 Sequências e Alinhamentos

As sequências biológicas são cadeias ordenadas de DNA, RNA ou proteínas. Sequências de DNA e RNA são compostas por nucleotídeos, representados pelo alfabeto $\Sigma = \{A, T, G, C\}$ e $\Sigma = \{A, U, G, C\}$, respectivamente. As proteínas são sequências de aminoácidos do alfabeto $\Sigma = \{A, C, D, E, F, G, H, I, K, L, M, N, P, Q, R, S, T, V, W, Y\}$.

Dado um alfabeto α de n caracteres, $\alpha = \{\alpha_1, ..., \alpha_n\}$, define-se uma sequência de k caracteres como um subconjunto do produto de caracteres:

$$\alpha^k = \prod_{i=1}^{k} \alpha_i$$

Também é definido um conjunto S de sequências como:

$$S = (\alpha_{n_1}, ..., \alpha_{n_k})$$

onde, para cada $j = 1, ..., k$, n_j é um número natural que satisfaz $1 \leq n_j \leq n$.

Considere também o alfabeto β obtido do alfabeto α adicionando-se um espaço em branco "$-$" (*gap*):

$$\beta = \{-\} \cup \{\alpha_1, ..., \alpha_n\}$$

Em β podemos definir um alinhamento múltiplo das sequências $S_1, ..., S_n$, formado por um outro conjunto, $\bar{S}_1, ..., \bar{S}_n$, de tal forma que cada sequência $\bar{S}_i$ é obtida a partir de S_i, inserindo-se *gaps* em posições onde alguma das outras sequências possui um caractere que não é branco [9].

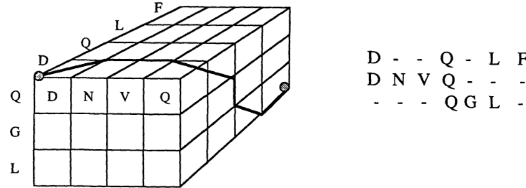

Figura 2.1: Alinhamento de três sequências [5].

Cada alinhamento de sequências codifica um diferente conjunto de inserções, deleções e substituições, de maneira que as semelhanças e diferenças sejam destacadas. A Figura 2.1 representa um alinhamento das sequências $S_1 = DQLF, S_2 = DNVQ$, e $S_3 = QGL$.

2.1.1 Caminhos

Ao conjunto de n sequências $S_1, ..., S_n$ de comprimento $k_1, ..., k_n$, associa-se um arranjo $L(S_1, .., S_n)$ em um espaço n-dimensional. Este espaço é obtido através do produto cartesiano das n strings, e também será chamado de espaço de busca. O vértice que corresponde ao primeiro caractere de todas as sequências é chamado origem. Analogamente, define-se o final como sendo o vértice que corresponde ao último caractere de todas as sequências.

Um caminho $\gamma(S_1, ..., S_n)$ entre as sequências $\bar{S}_1, ..., \bar{S}_n$ é uma linha "quebrada" que junta o início ao final sendo que, a cada passo, ele sempre se aproxima do final [9]. Um caminho corresponde a um, e apenas um, alinhamento de sequências. Na Figura 2.1, o lado esquerdo representa o início do caminho associado ao alinhamento que termina no lado direito da figura.

Um caminho pode ser projetado em relação a duas sequências. No exemplo da Figura 2.2, a projeção do alinhamento para as sequências S_1 e S_2 seria o alinhamento mostrado na figura. Denotamos a projeção de um alinhamento $\gamma(S_1, ..., S_n)$ nas sequências i e j como sendo $p_{ij}(\gamma(S_1, ..., S_n))$ [5].

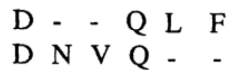

Figura 2.2: Alinhamento par-a-par projetado.

Define-se $M(\gamma)$ como uma medida de escores, que representa a similaridade das sequências $S_1, ..., S_n$ de acordo com o alinhamento associado ao caminho γ. Assim, podemos definir os alinhamentos ótimos como sendo os alinhamentos que possuem o menor escore entre todos [9]. Cada medida $M(\gamma)$ possui pelo menos um caminho $\gamma^*(S_1, ..., S_n)$ de forma que M possua o menor valor entre todos alinhamentos. Nesse caso, o escore representa o custo de se transformar uma sequência em outra.

2.2 Sistemas de escore

2.2.1 *Sum-of-pairs*(SP)

Um desafio para o Alinhamento Múltiplo de Sequências (*Multiple Sequence Alignment - MSA*) é computar o escore do alinhamento e não existe um consenso sobre uma função objetivo que seja bem aceita [34].

No alinhamento ilustrado na Figura 2.2 temos *matches* (caracteres iguais) na primeira e quarta posições e *gaps* nas demais posições. Nesse alinhamento não existem *mismatches* (caracteres diferentes). Na Figura 2.3, é mostrado um alinhamento múltiplo que utiliza um esquema de escore 0 para cada *match* e 1 para cada *mismatch* ou *gap*. Caso duas sequências possuam *gaps* na mesma posição o escore considerado é 0.

Para calcular o escore total do alinhamento múltiplo, calcula-se o escore dos pares de sequências (*Sum-of-Pairs*). No exemplo, um par de sequências possui escore 4, os outros dois pares possuem escore 5 e a soma desses valores é o escore total do MSA (escore 14).

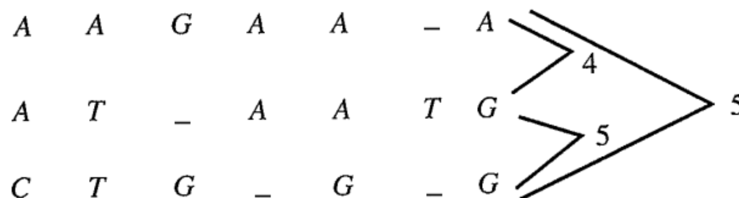

Figura 2.3: Escore de um alinhamento múltiplo com a função SP [16].

Pode-se definir a função *sum-of-pairs* conforme a Equação 2.1 [9]:

$$SP(m_i) = \sum_{k<l} s(m_i^k, m_i^l) \tag{2.1}$$

onde s é uma função que obtém o escore de uma posição dos alinhamentos e m_i^k e m_i^l representam os caracteres da coluna i das sequências k e l.

A função que obtém o escore pode associar um valor único a *matches*, *mismatches* e *gaps*, como ilustrado na Figura 2.3, ou pode utilizar matrizes de substituição. As matrizes de substituição são matrizes 4x4 ou 20x20 que contêm o escore para cada par possível de resíduos (*matches* ou *mismatches*). Esses escores são obtidos com dados biológicos estatisticamente relevantes. As matrizes de substituição mais comuns são a PAM [7] e a BLOSUM [20]. A Tabela 2.1 ilustra a matriz PAM250.

2.2.2 *Weighted sum-of-pairs* (WSP)

Um problema com o *sum-of-pairs* é que o escore de cada sequência é contabilizado como se fosse descendente de $N-1$ sequências ao invés de possuir um único ancestral [9]. Por isso, as mesmas diferenças entre as sequências são contadas múltiplas vezes e esse problema aumenta quando o número de sequências se torna maior.

Para a solução deste problema, um novo esquema de escore foi proposto. O *Weighted sum-of-pairs* [2] busca compensar parcialmente um efeito indesejado de contar múltiplas vezes alguns eventos evolucionários na soma SP.

	-	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

Tabela 2.1: Matriz de substituição PAM 250 [7].

A proximidade das sequências é detectada através da construção de uma árvore filogenética e, durante o cálculo, é especificado um peso aos pares de sequências alinhados. Cada um desses pesos é multiplicado pela pontuação do alinhamento dois-a-dois, onde a pontuação final é a soma dos alinhamentos ponderados [9].

Figura 2.4: Escores de um alinhamento múltiplo com a função WSP.

Considere $P(S_i, S_j)$ como sendo o peso das sequências S_i e S_j. A Figura 2.4 apresenta um exemplo de escore calculado com a função WSP, onde $P(S_1, S_2) = 3$, $P(S_2, S_3) = 2$, $P(S_1, S_3) = 2$. Com este esquema, é possível aumentar o escore de duas sequências que sejam muito próximas. O escore total do alinhamento da Figura 2.4 é 27.

2.3 Algoritmo Exato Simples: Processamento em todo o espaço de busca

O problema do alinhamento global múltiplo de sequências consiste em computar um alinhamento global múltiplo M com um escore mínimo [34]. A seguir, será apresentado o caso de três sequências, que pode ser generalizado para um número qualquer de sequências.

Considere S_1, S_2 e S_3, três sequências de tamanho n_1, n_2 e n_3, respectivamente, e seja $D(i, j, k)$ o escore ótimo soma dos pares (SP) para o alinhamento de $S_1[1..i], S_2[1..j]$ e $S_3[1..k]$. Consideremos também o escores para *match*, *mismatch* ou *gap* como sendo *smatch*, *smis* e *sspace*, respectivamente.

1: **for** $i = 1 \rightarrow n_1$ **do**
2: **for** $j = 1 \rightarrow n_2$ **do**
3: **for** $k = 1 \rightarrow n_3$ **do**
4: **if** $S_1(i) = S_2(j)$ **then**
5: $c_{ij} = smatch$
6: **else**
7: $c_{ij} = smis$
8: **end if**
9:
10: **if** $S_1(i) = S_3(k)$ **then**
11: $c_{ik} = smatch$
12: **else**
13: $c_{ik} = smis$
14: **end if**
15:
16: **if** $S_2(j) = S_3(k)$ **then**
17: $c_{jk} = smatch$
18: **else**
19: $c_{jk} = smis$
20: **end if**
21:
22: $d_1 = D(i-1, j-1, k-1) + c_{ij} + c_{ik} + c_{jk}$
23: $d_2 = D(i-1, j-1, k) + c_{ij} + sspace$
24: $d_3 = D(i-1, j, k-1) + c_{ik} + sspace$
25: $d_4 = D(i, j-1, k-1) + c_{jk} + sspace$
26: $d_5 = D(i-1, j, k) + 2 * sspace$
27: $d_6 = D(i, j-1, k) + 2 * sspace$
28: $d_7 = D(i, j, k-1) + 2 * sspace$
29:
30: $D(i, j, k) = Min[d_1, d_2, d_3, d_4, d_5, d_6, d_7]$
31: **end for**
32: **end for**
33: **end for**

Figura 2.5: Algoritmo exato para o alinhamento múltiplo de três sequências [16].

Para obter o escore do alinhamento múltiplo de três sequências, um cubo é utilizado e cada célula (i, j, k) que não está na fronteira (ou seja, todos os índices são diferentes de zero) possui sete vizinhos que precisam ser consultados para determinar $D(i, j, k)$.

Para computar os valores das fronteiras, considere $D'_{1,2}(i, j)$ como sendo o valor da soma de pares das subsequências $S_1[1..i]$ e $S_2[1..j]$. Definindo $D'_{1,3}(i, k)$ e $D'_{2,3}(j, k)$ por analogia, temos [16]:

$$D(i, j, 0) = D'_{1,2}(i, j) + (i + j) * sspace;$$
$$D(i, 0, k) = D'_{1,3}(i, k) + (i + k) * sspace;$$
$$D(0, j, k) = D'_{2,3}(j, k) + (j + k) * sspace;$$
$$D(0, 0, 0) = 0$$

A Figura 2.5 apresenta o algoritmo exato para o alinhamento de 3 sequências. Neste algoritmo, admite-se que a linha 0 e a coluna 0 são inicializadas com 0. Existem 3 laços "for", (linhas 1 a 3), indicando cada uma das três sequências. Inicialmente, são calculados os escores para *matches* e *mismatches* (linhas 4 a 20). Depois disso, 7 valores são calculados (linhas 22 a 28), que representam as 7 células vizinhas que devem ser consultadas. Após isso, $D(i, j, k)$ recebe o menor desses valores (linha 30).

O problema do alinhamento múltiplo de sequências foi resolvido neste caso utilizando programação dinâmica. No caso geral de alinhamento múltiplo de sequências, considerando k sequências de tamanho n, constrói-se uma matriz n-dimensional, assim o algoritmo exige tempo $\Theta(n^k)$ [16]. Esse problema foi provado NP-Difícil [57].

2.4 *Wavefront* tradicional e *Wavefront* multidimensional

Consideremos S_1 e S_2 duas sequências de tamanho n_1e n_2 respectivamente e seja $D(i, j)$ o escore do alinhamento em pares. Para calcular o alinhamento dessas duas sequências os escores são armazenados em uma matriz de tamanho $n_1 \times n_2$.

A Figura 2.6 ilustra o caso em que as sequências possuem 3 caracteres (tamanho 3). Em cinza são representadas as células em processamento, em preto estão as células já processadas e a cor branca representa células que ainda não foram processadas.

No começo da computação, apenas a primeira célula é calculada (Figura 2.6a). Em seguida, duas células podem ser computadas em paralelo, que estão presentes na próxima antidiagonal. Genericamente, para se calcular a célula $D(i, j)$ são necessários os valores de $D(i-1, j-1)$, $D(i-1, j)$ e $D(i, j-1)$. Desta forma o paralelismo máximo é atingido ao se calcular a antidiagonal principal da matriz (Figura 2.6c), e em seguida o paralelismo vai diminuindo (Figura 2.6d), até que apenas uma célula seja calculada no 5º e último passo (Figura 2.6e).

Para um alinhamento de três sequências, os escores são armazenados em um cubo, ilustrado na Figura 2.7. Nela são ilustrados o primeiro, segundo e quinto *wavefront*. No primeiro passo, existe apenas uma célula sendo calculada, $D(0, 0, 0)$. No passo seguinte, é possível calcular em paralelo os vizinhos da primeira célula, $D(1, 0, 0), D(0, 1, 0), D(0, 0, 1)$. No terceiro passo, pode-se calcular seis células em paralelo $D(2, 0, 0), D(1, 1, 0), D(0, 2, 0), D(1, 0, 1), D(0, 1, 1), D(0, 0, 2)$, e assim por diante.

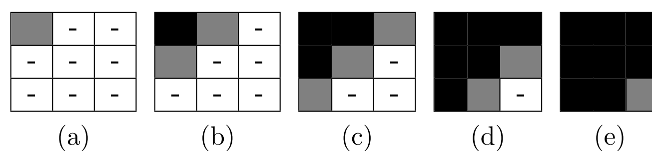

Figura 2.6: Passos executados no wavefront tradicional

Desta forma, vemos que o *wavefront* pode ser representando por um plano que atravessa o cubo formado pelas três sequências.

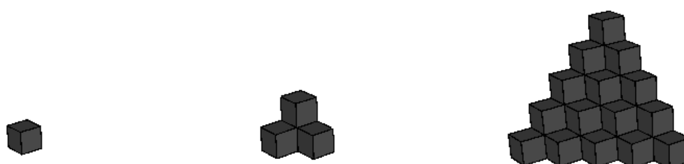

Figura 2.7: *Wavefront* tridimensional

Agora, considere o caso de um alinhamento múltiplo de n sequências. Para calcular o escore $D(a_1, a_2, ..., a_n)$, é necessário visitar todos os $2^n - 1$ vizinhos. Desta forma, vemos que, assim como o número de sequências cresce exponencialmente, o número de vizinhos de cada célula também cresce exponencialmente e conclui-se que, com o aumento do número de sequências, aumenta-se exponencialmente as células que podem ser calculadas em paralelo.

2.5 Método Carrillo-Lipman

H. Carrillo e D. Lipman [5] demonstraram que não é necessário percorrer todo o espaço de busca para encontrar o alinhamento múltiplo ótimo. Apesar de não implementar ou descrever um algoritmo, eles utilizaram propriedades dos caminhos para demonstrar que partes do espaço de busca podem ser descartadas e mesmo assim é possível obter o alinhamento múltiplo ótimo.

Baseado nos escores de um alinhamento heurístico e no alinhamento exato dos pares de sequências, um limite é encontrado e células que possuam um valor maior que este limite não podem fazer parte de um alinhamento ótimo. Neste caso, o caminho do alinhamento ótimo passa por alguma outra parte do espaço de busca e o valor da célula descartada não é relevante para o cálculo do escore do alinhamento múltiplo ótimo.

2.5.1 Carrillo-Lipman *Bound*

A seguir, são mostradas algumas observações demonstradas por Carrillo e Lipman [5] sobre o problema de se obter o alinhamento ótimo $\gamma^*(S_1, ..., S_n)$ onde $n > 2$.

Considere μ_{ij} uma medida de escore de duas sequências S_i e S_j. Nesse caso a medida M de um caminho N-dimensional γ é:

$$M(\gamma) = \sum_{i<j}^{N} \mu_{ij}(p_{ij}(\gamma)) \tag{2.2}$$

Onde p_{ij} é a projeção em γ do plano determinado pelas sequências S_i e S_j e $M(\gamma)$ é a medida de escore deste alinhamento.

Considere agora o alinhamento N-dimensional do caminho γ^e, que será chamado γ-estimado. Este é algum possível alinhamento das sequências, que não é necessariamente o ótimo. Por isso, temos as Inequações 2.3 e 2.4:

$$M(\gamma^e) - M(\gamma^*) \geq 0 \tag{2.3}$$

$$\sum_{i<j}^{N} \mu_{ij}(p_{ij}(\gamma^e)) - \sum_{i<j}^{N} \mu_{ij}(p_{ij}(\gamma^*)) \geq 0 \tag{2.4}$$

Denota-se γ_{ij}^* como sendo um caminho ótimo para a medida μ_{ij} das sequências S_i e S_j. Como $\mu_{ij}(p_{ij}(\gamma^*)) \leq \mu_{ij}(\gamma_{ij}^*)$, temos a Fórmula 2.5:

$$\sum_{i<j}^{N} \mu_{ij}(p_{ij}(\gamma^e)) - \sum_{i<j,(i,j)\neq(k,l)}^{N} \mu_{ij}(p_{ij}(\gamma^*)) \geq \mu_{kl}(p_{kl}(\gamma^*)) \tag{2.5}$$

Assim define-se U_{kl} conforme a Equação 2.6:

$$U_{kl} = \sum_{i<j}^{N} \mu_{ij}(p_{ij}(\gamma^e)) - \sum_{i<j,(i,j)\neq(k,l)}^{N} \mu_{ij}(p_{ij}(\gamma^*)) \tag{2.6}$$

Desta forma, U_{kl} é um limite superior para a medida das projeções de qualquer caminho ótimo N-dimensional, feitas no plano determinado pelas sequências S_k e S_j. Então, quando procuramos pelo alinhamento ótimo γ^*, apenas precisamos considerar os caminhos γ em $L(S_1, ..., S_n)$ que satisfazem $\mu_{kl}(p_{kl}(\gamma)) \leq U_{kl}$.

Define-se o conjunto P como sendo o conjunto composto pelos caminhos que satisfazem essa propriedade (Equação 2.7) :

$$P = \bigcap_{i<j}^{N} P_{ij} \tag{2.7}$$

Desta forma, para a obtenção do alinhamento ótimo, não é necessário aplicar o algoritmo de programação dinâmica em todo o espaço de busca $L(S_1, ..., S_n)$, mas é suficiente procurar na subregião P.

2.5.2 Simplificação das fórmulas

Em trabalhos posteriores [15] [30], as equações que demonstram o Carrillo-Lipman *bound* são mostradas em forma simplificada, com uma notação diferente e são apresentadas nesta seção.

Define-se L como sendo um limite inferior da soma de pares [15] (Equação 2.8) :

$$L = \sum_{i<j} d(S_i, S_j) \cdot scale(S_i, S_j) \tag{2.8}$$

Onde $scale(S_i, S_j)$ é o peso atribuído às sequências S_i e S_j e $d(S_i, S_j)$ é o custo do alinhamento ótimo em par das sequências. L é um limite inferior pois como o alinhamento em pares é ótimo, logo não existe um outro alinhamento com menor escore das sequências S_i e S_j. O escore do alinhamento múltiplo ótimo de todas as sequências é uma soma de pares, assim a soma dos pares ótimos das sequências deve ser menor ou igual ao escore do alinhamento múltiplo ótimo.

Considere um limite superior U, A^o o alinhamento ótimo, $c(A^o)$ como o custo do alinhamento múltiplo ótimo e $c(A^o_{i,j})$ o custo do alinhamento dos pares de sequências i e j no alinhamento A^o. Logo, temos a Inequação 2.9:

$$U - L \geq \sum_{i<j} [scale(S_i, S_j) \cdot (c(A^o_{i,j}) - d(S_i, S_j))] \tag{2.9}$$

Então, para quaisquer pares de sequências p e q, a Inequação 2.10 é válida:

$$U - L \geq scale(S_i, S_j) \cdot (c(A^o_{p,q}) - d(S_p, S_q)) \tag{2.10}$$

Rearranjando a inequação, obtemos a o Carrillo-Lipman *bound* [15] [30] (Inequação 2.11):

$$scale(S_i, S_j) \cdot c(A^o_{i,j}) \leq scale(S_i, S_j) \cdot d(S_i, S_j) + U - L \tag{2.11}$$

Considere um alinhamento heurístico A^h. Este alinhamento induz uma soma custo c na soma de pares i e j, assim $U - L$ pode ser obtido pela Equação 2.12:

$$U - L = \sum_{i<j} [c(A^h_{i,j}) - d(S_i, S_j)] \tag{2.12}$$

A Equação 2.12 mostra que os limites superiores e inferiores possuem um valor baseado na projeção de um alinhamento heurístico subtraído dos custos dos alinhamnetos em pares das sequências, garantindo assim um limite para que as células contribuam para encontrar o alinhamento ótimo.

2.5.3 MSA 1.0

D. Lipman e S. Altschul implementaram o MSA 1.0 [26], que é o primeiro programa que utiliza o Carrillo–Lipman *bound* para a redução do espaço de busca para a obtenção do alinhamento ótimo.

Os autores do MSA 1.0 propuseram em um outro trabalho o esquema de escore *weighted* SP [2] descrito na Seção 2.2.2. O MSA 1.0 implementa os dois esquemas de escore: SP e WSP. Para a criação de uma árvore filogenética, o MSA 1.0 implementou o método

neighbor-joining [49]. Utilizar SP ou *weighted SP* como medida de escore para o alinhamento é uma opção do usuário e utiliza-se a matriz PAM-250 (Tabela 2.1) para calcular o custo das substituições (*matches* e *mismatches*).

O método mais natural para encontrar custos de *gap* é utilizar uma analogia aos custos de substituição. Para alinhamentos SP, os custos de *gap* são a soma dos custos impostos nos pares de sequências. O MSA 1.0 implementa um sistema parecido com este, chamado de custo de *gap quasi-natural* [1]. O método é muito parecido com a pontuação natural de *gaps*, mas em raros casos, quando *gaps* consecutivos começam depois e terminam antes de *gaps* consecutivos numa segunda sequência, há um custo adicional de um *gap*.

Uma das observações feitas pelos autores é que o Carrillo-Lipman *bound* na prática é muito rigoroso, e muitas vezes não é necessário atingi-lo para poder desconsiderar partes do espaço de busca. A recomendação é que, caso sejam descartadas posições necessárias, o limite de busca seja aumentado e o programa seja executado novamente. Sendo assim, o MSA 1.0 permite que o usuário especifique $U - L$ para reduzir ainda mais o espaço de busca.

O programa foi implementado utilizando a linguagem de programação C e foi testado em computadores com sistema operacional UNIX-like. Os requisitos de memória e processamento são uma função do número de sequências, do seu comprimento e do tamanho dos limites superiores para os alinhamentos em pares. Na época de sua implementação, foi possível alinhar cinco sequências de diferentes famílias da superfamília das globinas utilizando um tempo inferior a dois minutos, sendo necessário menos de 1.3 Megabytes de memória.

2.5.4 MSA 2.0

Em uma versão posterior, Gupta *et. al* [15], melhoraram os requisitos de memória e tempo de execução do programa MSA 1.0. Esta nova versão é chamada de MSA 2.0.

Por padrão, o MSA 2.0 utiliza *weighted sum-of-pairs* e utiliza um método de árvore filogenética para calcular o peso das sequências [2].

A economia de recursos é feita pelo MSA 2.0 pois ele procura apenas entre alinhamentos que possuem um custo $\leq U$. A parte principal do MSA 2.0 considera que um valor finito U é conhecido e mais à frente mostramos as formas pelas quais ele pode ser obtido. Se U for muito pequeno, o programa não é capaz de encontrar alguma solução. Por esta razão, podem ser necessárias múltiplas execuções do programa incrementando o valor de U para se obter um alinhamento, ótimo ou não.

O motivo de não se utilizar um valor U arbitrário e muito grande é que este valor é diretamente proporcional à quantidade de memória e processamento exigidos pelo programa, e pode inviabilizar a finalização do programa.

Considere L o valor do limite inferior, calculado como descrito na Equação 2.8. L é a soma dos custos dos alinhamentos ótimos dos pares de sequências, independente do alinhamento das outras sequências. Sendo assim, ele é menor ou igual ao escore do alinhamento ótimo.

O valor $U - L$ é armazenado em uma variável δ. Existem três formas para se calcular δ e em todas elas U é calculado como $L + \delta$. Na primeira forma, o usuário especifica δ. Na segunda forma, o programa calcula um alinhamento múltiplo heurístico que não requer muita memória nem processamento. O alinhamento heurístico produz um custo de

alinhamento em pares. O valor $\epsilon_{i,j}$ é a diferença entre o custo deste alinhamento e o custo do alinhamento ótimo em pares $d(S_i, S_j)$. Sendo assim δ é calculado conforme a Equação 2.13:

$$\delta = \sum_{i<j}(min(\mathbf{maxepsilon}, \epsilon_{i,j}).scale(S_i, S_j). \quad (2.13)$$

O número **maxepsilon** é 50 por padrão, mas pode ser modificado em uma constante no código. A terceira forma é que o usuário pode passar para o programa um arquivo com valores para $\epsilon_{i,j}$, neste caso os valores são usados da mesma forma que os valores ϵ computados na segunda opção. Este método pode levar a valor U muito baixo.

MSA 2.0: implementação

A entrada do algoritmo MSA 2.0 são as sequências $S_1, S_2, ..., S_n$ de comprimento $k_1, ..., k_n$ e a diferença $\delta = U - L$. O maior valor $max(k_1, ..., k_n)$ é denotado apenas por k.

O algoritmo MSA 2.0 é organizado em duas fases. Na primeira, $O(n^2)$ comparações em pares são feitas para construir o conjunto $F_{i,j}$. Este conjunto contém todos os pontos que pertencem ao plano formado pelas sequências S_i e S_j que para algum caminho de $< 0, 0 >$ a $< k_i, k_j >$ possuem um valor pelo menos $d(S_i, S_j) + \epsilon_{i,j}$. Ao final desta fase, $F_{i,j}$ é determinado e os limites inferior $L = \sum_{i<j} d(S_i, S_j)$ e o superior $U = L + \delta$ podem ser calculados. Esta primeira fase não utiliza muito tempo ou memória.

Na segunda fase, um alinhamento que possui o menor escore é computado onde é garantido que as projeções do caminho ótimo estejam todas dentro de $F_{i,j}$.

Vértices e arestas são gerados dinamicamente e o algoritmo libera espaço quando um vértice não é mais necessário. Ao visitar um vértice, todos os vértices adjacentes são visitados e criados, caso ainda não tenham sido visitados por um outro caminho.

Para manter os vértices, é utilizada uma árvore de sufixos ordenada pelas coordenadas dos vértices existentes. Para organizar a pesquisa no espaço de busca, uma fila de prioridades com operações de inserir e extrair é utilizada para selecionar os vértices a serem percorridos.

Ao final, a função $TRACEBACK(t)$ retorna um alinhamento ótimo possível, percorrendo o caminho inverso, no sentido final até o início, examinando a distância e os pontos dos vértices predecessores.

O MSA 2.0 é um algoritmo proposto para obter o alinhamento múltiplo exato, mas o limite da variável δ igual a 50, por padrão, faz com que o mesmo não encontre o alinhamento ótimo em alguns conjuntos de sequências.

Gupta *et. al* [15] compararam o desempenho do MSA 1.0 com o MSA 2.0. Os programas foram testados em uma estação Sun Sparcstation 10 com 128 MB de memória RAM e compilados com flag de otimização. Os tempos foram medidos através do comando **time** e a quantidade de memória RAM foi medida através do comando **top**, sendo reportada a maior quantidade de memória utilizada durante toda a execução do programa.

Os alinhamentos produzidos pelo MSA 2.0 foram comparados com os resultados em McClure *et. al* [31]. Nessa comparação, cada *motif* é um bloco de uma a cinco colunas consecutivas alinhadas, que um bom alinhamento deve conter. Para cada conjunto de sequências, é reportada a soma dos *motifs* alinhados corretamente. Para a Globins A, que

possui 5 *motifs*, 4 deles foram alinhados corretamente pelo MSA 2.0 e, no quinto *motif*, 6 de 7 sequências foram alinhadas corretamente. Logo, o MSA 2.0 produziu um resultado que não era o desejado pelos biólogos, mas por se tratar de um alinhamento ótimo, sabemos que o erro não pode estar no alinhamento produzido, mas sim no conjunto formado pelo modelo utilizado, sistemas de escore e outras variáveis envolvidas na produção deste alinhamento.

2.6 Algoritmos Heurísticos

Como visto nas Seções 2.3 e 2.5, o alinhamento global múltiplo ótimo demanda muitos recursos computacionais e, por isso, métodos heurísticos foram criados para obter um alinhamento aproximado. O desafio desses métodos consiste em utilizar uma combinação apropriada de pesos de sequências, matrizes de escore e penalidade por *gaps* de forma que um bom alinhamento possa ser encontrado [34].

2.6.1 Alinhamento Progressivo

Algoritmos progressivos são baseados na ideia de construir algum alinhamento com um conjunto de sequências que possuem maior semelhança e então outras sequências são adicionadas ao alinhamento. O processo continua até que todas as sequências tenham sido consideradas [34].

A relação entre as sequências é geralmente estabelecida através de uma árvore filogenética, onde as sequências são comparadas em pares. Nessas árvores, as folhas são sequências que possuem maior similaridade.

Uma desvantagem desses métodos é a dependência de um alinhamento em pares inicial. Assim, se as sequências iniciais tiverem um bom alinhamento, poucos erros acontecerão. Mas, se isso não acontece, muitos erros iniciais se propagarão para os próximos alinhamentos, o que pode resultar em um alinhamento global final com vários erros, resultando em um escore bem menor do que o obtido pelo algoritmo exato.

ClustalW

Clustal [21] é um algoritmo bastante popular baseado em um método progressivo e utilizado desde 1988. Ele sofreu diversas mudanças ao longo dos anos buscando a melhoria do alinhamento múltiplo. ClustalW [55] é uma versão mais recente do Clustal, sendo que W significa *"weighting"*, representando a habilidade do programa de atribuir pesos para as sequências utilizadas.

O ClustalW se executa em 3 fases. Na fase 1, um alinhamento em pares de todas as sequências é realizado. Em seguida, na fase 2, os escores dos alinhamentos em pares são utilizados para a criação de uma árvore filogenética. Por último, na fase 3, o alinhamento múltiplo das sequências é realizado utilizando um algoritmo de programação dinâmica guiado pela árvore produzida na fase 2.

Dessa forma, as sequências mais semelhantes são alinhadas e em seguida as outras, ou grupos de sequências, são adicionadas e, guiados pelo alinhamento inicial, é produzido um alinhamento múltiplo global.

T-COFFEE

T-Coffee [37] é um programa de alinhamento progressivo que usa um sistema de pesos nas posições das sequências para gerar um alinhamento múltiplo que é mais consistente que o alinhamento em par de todas as sequências [34].

No T-Coffee, os dados são organizados em uma biblioteca primária e uma biblioteca estendida. A biblioteca primária armazena informações sobre todos os alinhamentos para-a-par das sequências de entrada. Logo existem $(n \times (n-1))/2$ elementos nessa biblioteca.

Cada elemento da biblioteca primária possui várias entradas com informações sobre o alinhamento global par-a-par e os 10 melhores alinhamentos locais sem sobreposição. A biblioteca primária é estendida da seguinte maneira. Cada entrada é comparada com todas as outras entradas e o resultado dessa comparação é um peso, que descreve o grau no qual essa entrada é consistente com as outras entradas. Tanto a biblioteca primária como a biblioteca estendida podem ser utilizadas para a obtenção do alinhamento múltiplo.

PILEUP

PILEUP [12] utiliza um método parecido com o ClustalW para obter um alinhamento múltiplo global. As sequências são alinhadas em pares usando o algoritmo exato de programação dinâmica Needleman - Wunsch [36] e, usando um método de médias aritméticas, é produzida uma árvore, utilizada para identificar as sequências e grupos que estão mais relacionados. É utilizado um sistema de escore e penalidades na inserção de *gaps*.

2.6.2 Alinhamento Iterativo

Os métodos iterativos buscam resolver o principal problema existente nos métodos progressivos, onde erros nos alinhamentos iniciais das sequências mais próximas possuem um grande peso e são propagados para o alinhamento múltiplo final. Buscando melhorar o alinhamento em modo geral, ou seja, melhorando o escore do alinhamento, os métodos iterativos buscam corrigir esse problema realinhando repetidamente os subgrupos de sequências e então alinhando esses subgrupos em um alinhamento global de todas as sequências.

SAGA

O SAGA [38] é um programa de alinhamento múltiplo baseado em algoritmos genéticos. Segundo os autores, possui a capacidade de encontrar alinhamentos com escores tão bons quanto os de outros métodos, como o ClustalW [34].

Sua ideia é gerar uma população inicial de possíveis alinhamentos baseados nas sequências dadas. A população inicial é analisada usando um esquema de escore. Depois, esses alinhamentos múltiplos iniciais são passados para uma nova geração de alinhamentos, sendo que metade dos alinhamentos iniciais que possuem maior escore são selecionados, aplicando-se regras que simulam a seleção natural e são sujeitos a mutações.

Em seguida, a próxima geração é criada, onde é gerada uma nova população de alinhamentos múltiplos e filhos são gerados a partir de dois pais. A população é novamente analisada, buscando-se manter alinhamentos múltiplos de melhor escore na população.

Esse processo normalmente é executado n vezes e, ao final, o alinhamento múltiplo de melhor escore é selecionado.

DIALIGN e DIALIGN-TX

O DIALIGN é um método iterativo que realiza o alinhamento múltiplo de sequências sem adicionar penalidade para *gaps* [33].

Este algoritmo é executado em três fases. Na primeira fase, todos os pares de alinhamentos DIALIGN são calculados, isto é, $n(n\text{-}1)/2$ cálculos, um para cada alinhamento, onde n é o número de sequências. Na segunda fase, as diagonais que compõem o alinhamento em pares são ordenadas pelo escore e grau de sobreposição com outras diagonais. Essa lista ordenada é usada para obter um Alinhamento Múltiplo com um algoritmo guloso, gerando um alinhamento A.

Na última fase, o alinhamento A é completado com um procedimento iterativo onde partes das sequências que ainda não foram alinhadas com A são realinhadas executando a fase 2 novamente, de forma que diagonais consistentes não alinhadas sejam incluídas em A. Esta fase é repetida até que nenhuma diagonal com peso positivo possa ser incluída em A.

Para melhorar a qualidade dos alinhamentos produzidos, o DIALIGN-TX [53] foi proposto. Como em todas as versões anteriores, a saída da primeira fase é um conjunto de diagonais de escore alto. Essas diagonais são usadas no DIALIGN-TX para construir uma árvore guia na fase 2. Na fase 3, dois métodos são usados para gerar 2 alinhamentos: um pelo método progressivo e o outro pelo método original do DIALIGN. Ambos alinhamentos são avaliados e o melhor é mantido.

PRRP

O programa PRRP [14] usa um método iterativo para produzir um alinhamento múltiplo. A partir de um alinhamento em pares inicial é criada uma árvore. A árvore é utilizada para gerar pesos e fazer alinhamentos.

Regiões que compõem o alinhamento são recalculadas em busca do melhor escore de alinhamento. Assim, o melhor esquema de escore encontrado é utilizado no próximo ciclo de cálculos para obter uma nova árvore, pesos e alinhamentos. O algoritmo continua a execução até que não seja possível incrementar o escore de alinhamento.

2.6.3 Outros Métodos

A maioria dos métodos para alinhamentos múltiplos normalmente determinam a semelhança entre todos os pares de sequências que devem ser comparados. Outros métodos, como a aproximação por grupos [34], utilizam um consenso entre cada grupo de sequências e esse consenso é utilizado para futuro alinhamento entre grupos. Exemplos de programas com essa abordagem são o PIMA [51] e o MULTAL [54]. Alguns desses métodos utilizam a distância de uma árvore filogenética para organizar as sequências e as duas mais próximas são alinhadas. O alinhamento consenso obtido é alinhado com outra sequência, conjunto ou outro consenso até se obter um alinhamento com todas sequências [34].

Cadeias de Markov escondidas também podem ser utilizadas para a obtenção do alinhamento múltiplo. As Cadeias de Markov são um modelo estatístico que considera todas

as possibilidades de combinação de *matches, mistaches* e *gaps* para gerar um alinhamento de um conjunto de sequências. Um modelo de uma família de sequências é produzido e um conjunto de sequências é utilizado para treinar o modelo. O resultado pode ser utilizado para obter uma alinhamento múltiplo ou utilizado para comparar em um banco de dados de sequências biológicas para encontrar outros membros da mesma família [24].

Capítulo 3

Plataformas de Alto Desempenho

A computação de alto desempenho é uma sub-área da Ciência da Computação que teve seu início na década de 1960, quando pela primeira vez cogitou-se quebrar um problema em sub-problemas menores, com o objetivo de resolvê-los em diversos elementos de processamento simultaneamente [8].

Na década de 1970, surgiram os supercomputadores vetoriais, que eram projetados para calcular rapidamente os elementos de um vetor ou matriz. Essas arquiteturas eram exemplos da categoria SIMD (*Single Instruction Multiple Data*) de Flynn [13] onde uma mesma instrução era executada ao mesmo tempo sobre diversos dados diferentes. Os supercomputadores vetoriais conseguiram ser uma ordem de magnitude mais rápidos do que as arquiteturas convencionais da época.

No início da década de 1980, tornou-se popular uma outra categora de máquinas, denominadas SMP (*Symmetric Multiprocessors*), que eram exemplos da categoria MIMD (*Multiple Instruction Multiple Data*) de Flynn, onde códigos diferentes podiam ser executados ao mesmo tempo em processadores distintos. Essas máquinas geralmente utilizavam o paradigma de memória compartilhada para a troca de dados entre os processadores.

No final da década de 1980, ficou claro que as máquinas SMP não eram escaláveis, ou seja, não era possível conectar um grande número de processadores às mesmas, pois os meios de comunicação (barramentos) rapidamente se tornavam saturados [8].

Para solucionar esse problema, foram criados sistemas de computação distribuída, onde computadores relativamente completos eram conectados através de uma rede de interconexão, de modo a criar um sistema integrado. Nos sistemas de computação distribuída, a troca de dados se dava através do paradigma de troca de mensagens.

Nessa categoria, inserem-se os *clusters* de computadores, que foram definidos como sendo sistemas de computação distribuída que utilizavam *hardware* e *software* disponíveis no mercado (*commodity components*) [48]. Os *clusters* de computadores rapidamente se disseminaram por possuírem um grande apelo comercial, que combinava baixo custo e relativa facilidade de programação.

Paralelamente à evolução dos *clusters*, arquiteturas específicas se disseminaram. Essas arquiteturas são geralmente otimizadas para execução de um tipo de aplicação, sendo chamadas de aceleradores. Exemplos típicos de aceleradores são FPGAs (*Field Programmable Gate Arrays*) e GPUs (*Graphics Processing Units*).

Atualmente, observa-se uma integração entre essas diversas categorias de máquinas. Por exemplo, é comum a existência de *clusters* de multicores (SMPs) onde um ou mais

multicores estão conectados a aceleradores (GPU, FPGA ou outros).

A seguir, são apresentadas as principais características dos *clusters* de computadores, dos FPGAs. Ao final deste capítulo as GPUs são apresentadas com maior detalhe, pois são o foco dessa dissertação.

3.1 Clusters de Computadores

Os *clusters* de computadores são um conjunto de computadores conectados através de *hardware* e *software* especializado, apresentando uma imagem única de sistema (*Single System Image*) ao usuários [56]. A imagem única de sistema permite que seja criada a ilusão de que os vários computadores que compõem o *cluster* são na verdade um único sistema.

Em um *cluster*, cada nó de processamento é um computador praticamente completo, com processador, memória e disco rígido, geralmente sem monitor, mouse ou teclado [22]. Os nós de processamento são interligados por uma rede de interconexão, geralmente de alta velocidade e disponível no mercado. Como vantagens do *cluster*, podemos citar:

- Potencial para escalabilidade, alta disponibilidade e alto desempenho;
- Relativa facilidade em adicionar/remover componentes;
- Relativa facilidade de programação, entre outros.

Atualmente, os *clusters* são geralmente programados com o padrão MPI [32] (*Message Passing Interface*), que permite a troca de mensagens síncronas e assíncronas entre os processos que se executam em computadores distintos. Além disso, oferece primitivas de *broadcast* "*one to all*" e "*all to all*" entre outras.

Apesar dos *clusters* SMPs (ou *clusters* de *multicores*) poderem ser programados unicamente com troca de mensagem, uma maneira mais eficiente de utilizá-los consiste em se usar um modelo de programação híbrido, onde os núcleos trocam valores por memória compartilhada (com OpenMP [46] ou POSIX pthreads (pthreads)) e os computadores trocam valores por troca de mensagens, com MPI.

3.2 FPGAs

Os FPGAs (*Field Programmable Gate Arrays*) [58] são circuitos integrados em larga escala que podem ter sua configuração interna modificada após a sua fabricação. Uma de suas características mais importantes são baixo o consumo de energia e um potencial alto *throughtput* de operações. A capacidade do FPGA é medida pela número de blocos lógicos configuráveis que ele possui ou pelo número de portas lógicas equivalentes.

Quase todos os FPGAs são compostos por blocos programáveis compostos. Esses blocos são compostos por registradores, elementos lógicos configuráveis e interconexões. Além disso existem outras características comuns aos FPGAS [58]:

- Elementos de Computação: compostos por alguns registradores e elementos lógicos de baixo nível.

- Tabelas de *Lookup*: Também chamadas de *Lookup Table* (LUT). Geralmente estão associadas a um ou mais registradores programáveis e podem implementar funções lógicas através de diferentes entradas. Os detalhes de implementação variam de acordo com a família e o fabricante.
- Memória: A memória pode ser global para todos elementos lógicos ou local para um determinado grupo de elementos.
- Recursos de Roteamento: Fazem a interligação entre os elementos de processamento, memória interna e outras estruturas do circuito. Possuem velocidades e níveis de flexibilidade bem diferentes.
- Entrada e Saída Configurável: A placa de FPGA deve se comunicar com um computador hospedeiro. Normalmente possuem circuitos chamados de blocos de entrada e saída cuja função é fazer o gerenciamento da comunicação.

Os FPGAs são geralmente programados com linguagens de descrição de *hardware* (*Hardware Descripto Languages* - HDL) que, ao serem compiladas, geram descrições de circuitos que são descarregadas no FPGA, para execução. As linguagens de descrição mais comuns são VHDL e Verilog.

3.3 GPUs

As Unidades de Processamento Gráfico (*Graphics Processing Unit* (GPU)) são microprocessadores com funções otimizadas para acelerar a renderização gráfica, executando em *hardware* as funções matemáticas mais comuns.

Na metade da década de 1990, iniciou-se uma forte demanda dos consumidores por aplicações que empregavam gráficos 3D. Principalmente na área de entretenimento eletrônico, as placas gráficas tornaram-se populares e progressivamente foram aumentando a complexidade e criando ambientes tridimensionais mais realistas. Nesta época três empresas, NVidia, ATI e 3dfx, começaram a lançar aceleradores gráficos a preços acessíveis, que se tornaram aceleradores gráficos populares [50].

As GPUs também são chamadas de *stream processors*, por causa da sua capacidade de executar paralelamente a mesma operação sobre vários elementos de dados, conceito chamado de SIMD (*Single Instruction Multiple Data*) na taxonomia de Flynn [13]. O alto *throughtput* de operações aritméticas chamou a atenção de desenvolvedores e pesquisadores que desejavam acelerar alguns problemas utilizando tal arquitetura.

Do ponto de vista da programação paralela, um marco importante nas tecnologias dos aceleradores gráficos ocorreu no ano de 2001. Neste ano foi lançado o primeiro chip industrial (NVidia GeForce 3) que implementava o recém-criado padrão da Microsoft DirectX 8.0, que exigia que o *hardware* tivesse um vertex programável e um estágio de *pixel shading* programável. Pela primeira vez, desenvolvedores tinham acesso a quais exatas funções seriam executadas nas GPUs de suas estações de trabalho [50].

Ao final da década de 1990, eram muito limitadas as formas de interação com as GPUs. Nessa época, todo programa deveria simular uma renderização tradicional de vídeo para poder executar em uma GPU. Desta forma, era muito complicado realizar operações com ponto flutuante, muito desejadas por aplicações científicas, e operações de *debug* eram praticamente impossíveis de serem realizadas.

De maneira a simplificar a programação das GPUs e permitir seu uso em áreas que não fossem o processamento de imagens, surgiu programação de propósito geral em GPUs (*General-purpose computing on graphics processing units* - GPGPU), que é o uso ambientes e ferramentas de programação paralela para executar códigos em unidades de processamento gráfico. Alguma linguagens foram criadas especificamente para este objetivo, dentre as quais podemos citar CUDA [43], que será detalhada na Seção 3.3.1.

3.3.1 Arquitetura CUDA

A arquitetura CUDA (*Compute Unified Device Architecture*) foi proposta pela NVidia em novembro 2006 [50] e é usada para programar suas placas. É composta por elementos em *hardware* de alto desempenho e *software*. A Figura 3.1 ilustra a arquitetura em *software*.

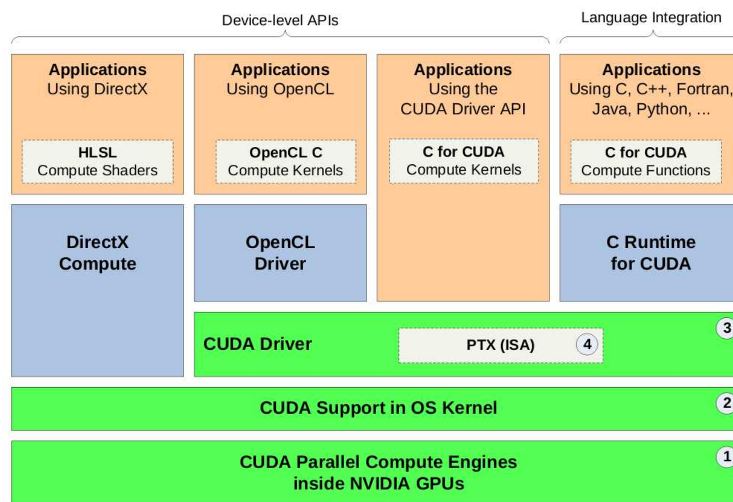

Figura 3.1: Arquitetura CUDA [40].

Na camada mais baixa da arquitetura CUDA, existem *engines* paralelas de computação que ficam presentes dentro das placas NVidia. Na camada imediatamente superior existe um *driver* do sistema operacional que é responsável pela inicialização do *hardware*, além da configuração e comunicação com a GPU. Acima desta camada, está um outro *driver* (CUDA Driver) que se executa em modo de usuário e provê uma interface de programação (API) de baixo nível. Dentro dessa API, existe o *Parallel Thread eXecution* (PTX) que define uma máquina virtual de baixo nível e um conjunto de instruções da arquitetura (ISA) para execução de *threads* paralelas de propósito geral [42]. Compiladores de alto nível como o CUDA geram instruções PTX que são otimizadas e traduzidas para um conjunto de instruções específicas da arquitetura disponível.

O nível acima da arquitetura ilustra as diversas possibilidades que estão disponíveis para se controlar a GPU, além de API CUDA: uma camada OpenCL [45], *framework* desenvolvido para a execução de aplicações paralelas em ambientes heterogêneos e algumas camadas para suporte CUDA em outras linguaguens, como C, Fortran, Python.

O ambiente de desenvolvimento de software provê um conjunto de ferramentas úteis para desenvolver aplicações na arquitetura CUDA. Podemos citar as bibliotecas que pro-

vêm soluções otimizadas para a arquitetura, como a transformada rápida de Fourrier, *C Runtime for CUDA*, que permite a executar funções padrão do C, além de ferramentas de compilação, *debug* e *Visual Profiler*.

3.3.2 Modelo de Programação em CUDA

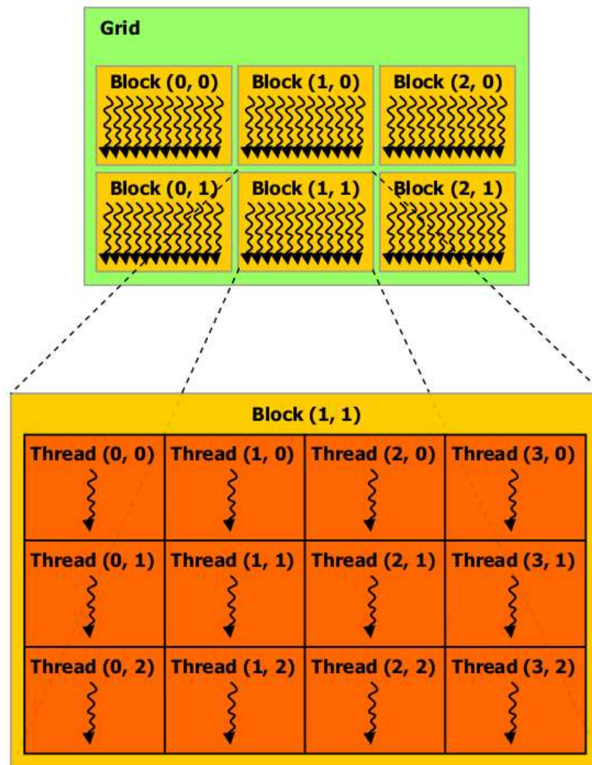

Figura 3.2: Modelo de programação CUDA [43].

Um programa em CUDA possui a capacidade de executar instruções na GPU. Ele pode ser dividido em duas partes: o código "*host*" que é executado na CPU, e o código "*device*", que se executa na placa de vídeo. Os *kernels* são a forma pela qual o código *host* é capaz de chamar o código *device*, sendo assim possível a interação entre eles. Toda função executada na placa gráfica está dentro de um *kernel*.

O modelo de programação massivamente paralela permite que várias instâncias sejam executadas em paralelo por diferentes *threads*. Cada *thread* possui um conjunto de registradores e uma memória local privada. Além disso, cada *thread* possui um identificador único dentro do seu bloco, que pode ser acessado através da variável local *built-in* $threadIdx$.

Conforme ilustrado na Figura 3.2, as *threads* podem ser agrupadas em blocos. *Threads* que estão no mesmo bloco podem se comunicar através de um tipo mais rápido de memória, e podem ser sincronizadas através de uma diretiva de barreira ($__syncthreads()$). Os blocos de *threads* podem possuir de uma a três dimensões.

Vários fatores influenciam a ordem na qual os blocos são executados, como o número de registradores que cada *thread* utiliza, e a quantidade de recursos [43]. Os blocos são

escalonados em vários multiprocessadores. À medida que a execução de um bloco é terminada, um novo bloco é alocado àquele processador até que todos os blocos tenham sido executados. Porém, a ordem de execução dos blocos não pode ser determinada. Sendo assim, é necessário que os blocos não possuam uma dependência de dados entre si.

Os blocos de *threads* são organizados *grids*. Em versões mais recentes do CUDA é possível criar *grids* de uma a três dimensões, assim como os blocos. Cada bloco possui um identificador único que pode ser acessado pelo *kernel* através da variável $blockIdx$. Ao contrário dos blocos, não existem chamadas de sincronia entre os *grids*. É esperado que a sincronia dos *grids* sejam feitas através de diferentes chamadas de *kernels*.

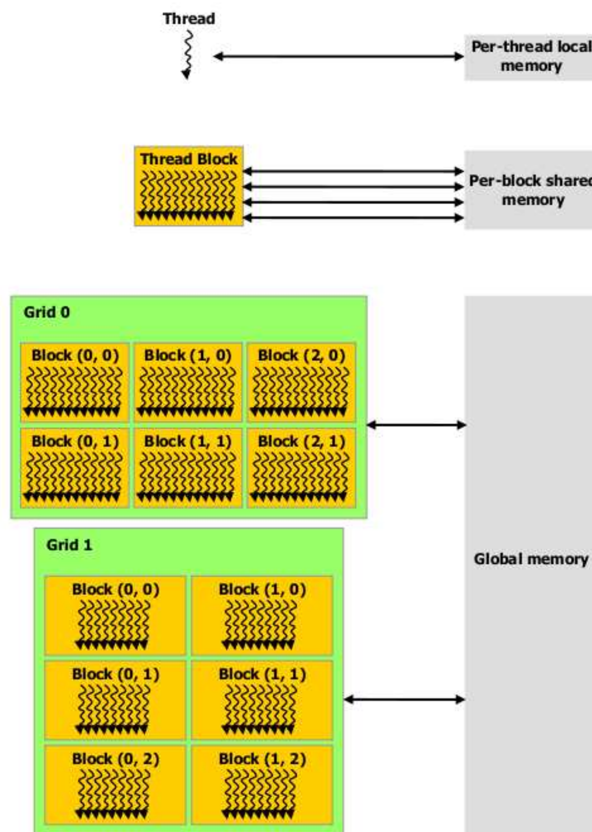

Figura 3.3: Hierarquia da memória em CUDA [43].

A arquitetura de memória em CUDA, exemplificada na Figura 3.3, ilustra os níveis de agrupamento das *threads* e a memória relacionada. Além das memórias mostradas figura, existem outros dois tipos de memória: a memória constante e a memória textura.

- **Local:** memória que apenas a *thread* pode acessar.
- **Compartilhada:** memória otimizada para uma rápida comunicação entre as *threads*. Permite a comunicação apenas entre *threads* do mesmo bloco.
- **Global:** memória que pode ser lida e escrita por todas as *threads*.

- **Textura:** permite acelerar o acesso a uma memória global de leitura (*read-only*). Para que este acesso seja acelerado, é necessário que as *threads* possuam algum padrão comum de acesso a memória (*coalesced*) [43].
- **Constante:** pequena memória rápida que pode ser lida por todas as *threads* (*read-only*).

3.3.3 Arquitetura Fermi

Atualmente, uma arquitetura de GPU muito utilizada é a arquitetura Fermi [41] disponível em diversas placas da NVidia. Lançada para substituir a série GF 200, possui diversas modificações e algumas serão detalhadas nesta seção.

Cada GPU da NVidia é composta por vários *Stream Processors* (SP), que são as menores unidades de execução da arquitetura. Um *Stream Multiprocessor* (SM) é um conjunto de SPs agrupados com outros elementos, como memória compartilhada, cache L1 e o *Special Function Units* (SFU), que são unidades responsáveis por processamento de operações matemáticas mais complexas. Nas placas com arquitetura Fermi, 32 SPs, 4 SFU e alguns outros elementos, são agrupados para formar 1 SM [39].

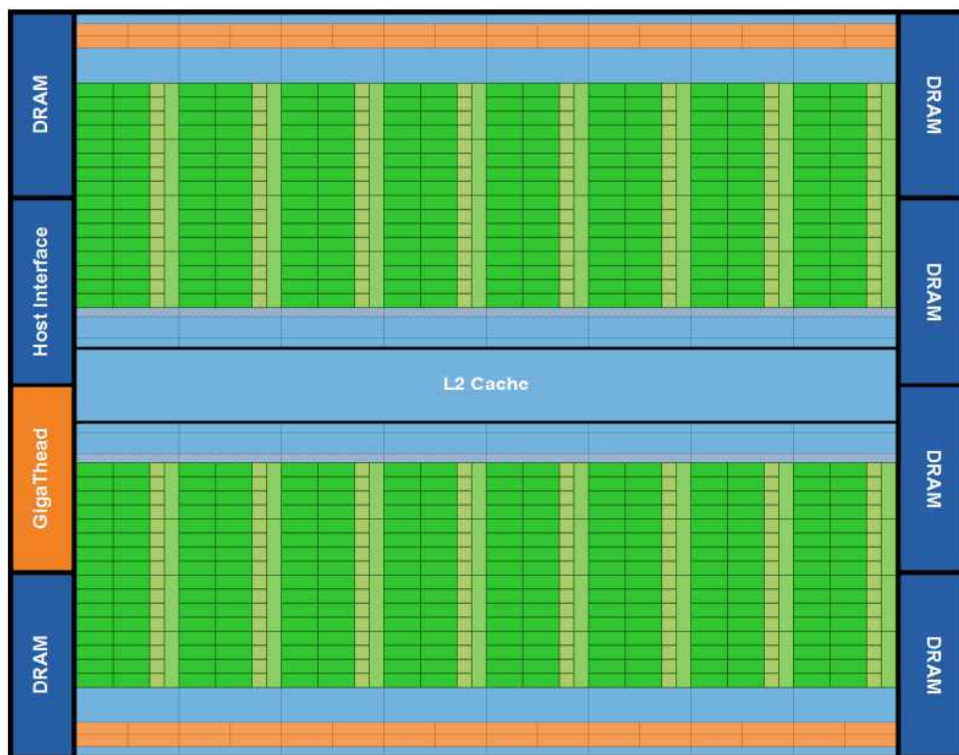

Figura 3.4: Arquitetura Fermi [41].

A Figura 3.4 representa a arquitetura Fermi, onde 16 SM de 32 núcleos são posicionados ao redor de um cache L2 comum. A porção mais clara e central da figura representa as unidades de execução. As porções médias nos extremos da figura representam as unidades de escalonamento (*scheduler* e *dispatch*) em conjunto com um cache L1, um registrador.

A arquitetura Fermi possui uma série de mudanças em relação ao seu predecessor, dentre elas [41]: suporte a até 512 núcleos; maior quantidade de memória compartilhada; 64KB de RAM em cada SM que são particionados entre o cache L1 e memória compartilhada; precisão dupla de operações de ponto flutuante; suporte a *error-correcting code* (ECC) para proteção contra erro de memória; troca de contexto mais rápida e otimizações em operações atômicas.

Capítulo 4

Alinhamento Múltiplo de Sequências em Plataformas de Alto Desempenho

Nesse capítulo, serão discutidas diversas propostas para execução de ferramentas de Alinhamento Múltiplo de Sequências em Plataformas de Alto Desempenho. Na Seção 4.1 serão apresentadas propostas para o alinhamento Múltiplo de Sequências heurístico em plataformas de alto desempenho e na Seção 4.2 serão apresentadas propostas para o alinhamento múltiplo exato em algumas dessas plataformas. Ao final do capítulo, serão apresentados e discutidos quadros comparativos das propostas.

4.1 Propostas de Alinhamento Heurístico

4.1.1 MT-ClustalW

K. Chaichoompu *et. al* [6] criaram uma versão do ClustalW (Seção 2.6.1) totalmente *multithreaded* chamada MT-ClustalW. Ela é baseada em uma outra implementação chamada ClustalW-SMP, criado por O. Duzlevski [10].

Utilizando uma ferramenta chamada *"Intel Thread Profiler"*, o ClustalW-SMP foi analisado e observou-se que apenas a fase 1 (alinhamento par a par) e a fase 2 (alinhamento progressivo) foram paralelizadas.

A otimização do MT-ClustalW foi realizada na fase 2. O código e fluxograma de execução propostos são apresentados na Figura 4.1. Nele, os dois loops internos foram movidos para funções paralelas. São criadas *thread_num* funções paralelas que recebem um sinal da função principal e aguardam a execução. Quando todas as *threads* estão ocupadas, a função principal aguarda que uma delas termine a execução.

O código foi implementando em C e utilizou a biblioteca *pthread* para obter paralelismo. Os testes foram realizados em uma estação com processador Intel Pentium D *Dual-Core* com 2.8GHz e 2GB de memória RAM. O sistema operacional utilizado foi o MS Windows XP PRO SP 2, e a máquina não possuía nenhuma outra aplicação instalada.

Os tempos de execução da fase 2 são comparados entre o MT-ClustalW e o ClustalW-SMP. Em todos os casos, o MT-ClustalW apresentou desempenho melhor do que o do ClustalW-SMP.

Na comparação dos métodos, o *speedup* do MT-ClustalW em relação ao Clustal-SMP é maior para sequências curtas, de até 200 aminoácidos. Isto acontece porque a fase pa-

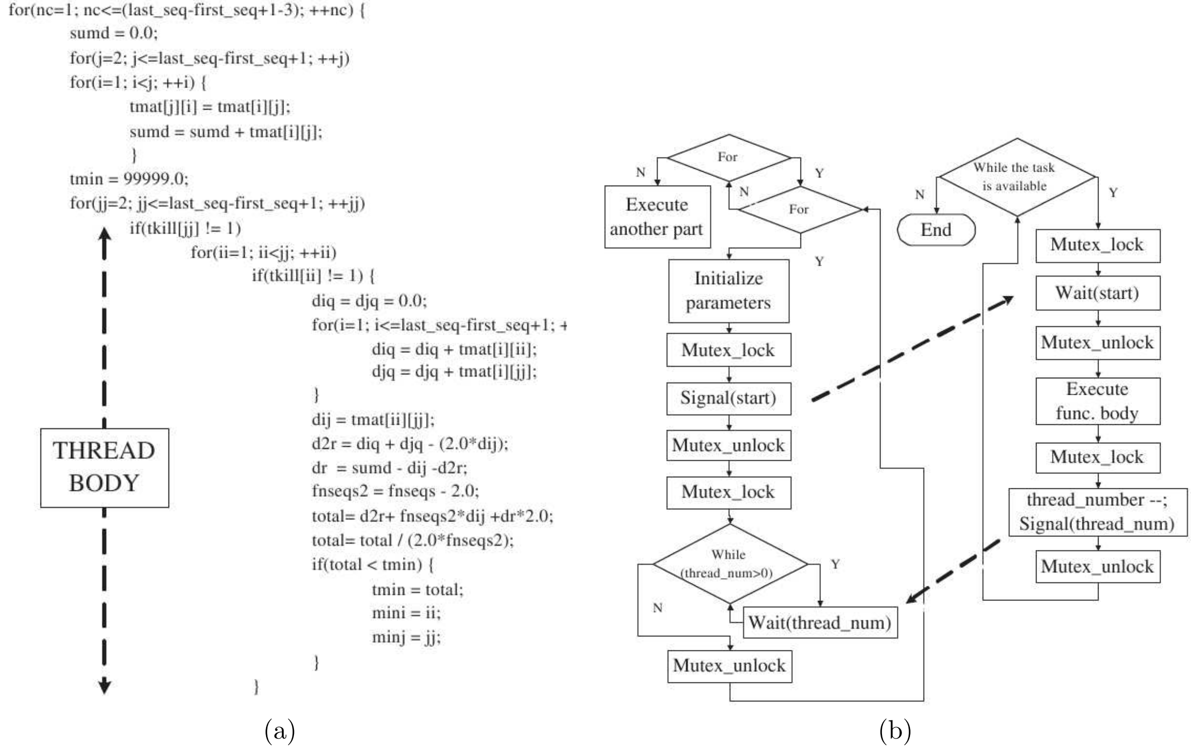

Figura 4.1: Código otimizado do MT-ClustalW e fluxograma[6].

ralelizada do algoritmo MT-ClustalW possui tempo de execução proporcional ao número de sequências. As outras fases são proporcionais ao número de sequências e seu comprimento. Assim, quando o comprimento das sequências aumenta, mais tempo é gasto em fases que são iguais em ambos algoritmos, desta forma o *speedup* diminui.

4.1.2 ClustalW-MPI

Em 2003, K. Li publicou um algoritmo que chamou de ClustalW-MPI [25], sendo muito referenciado na literatura ([44, 6]) e utilizado para comparação com implementações do ClustalW em outras arquiteturas de alto desempenho. Seu objetivo era utilizar um *cluster* de estações de trabalho (Seção 3.1) com uma arquitetura de memória distribuída em uma implementação que não exige nenhum *hardware* ou *software* proprietário.

A primeira etapa do algoritmo ClustalW é um alvo fácil para paralelização, pois possui uma granularidade grossa e as comparações par-a-par são independentes entre si. Já as duas últimas etapas são um desafio maior para paralelizar pois possuem uma razoável dependência entre os dados.

Para a primeira fase, é utilizada uma estratégia de escalonamento chamada *fixed-size chunking* (FSC), onde lotes de tarefas de um tamanho fixo são destinados para os processadores disponíveis.

Com os alinhamentos par-a-par calculados, uma árvore guia é produzida para servir de topologia para o alinhamento final (Seção 2.6.1). Algumas modificações foram realizadas no algoritmo para que o mesmo pudesse ser executado em $O(N^2)$, obtendo o mesmo resultado do algoritmo original. Porém, este estágio normalmente utiliza menos de 1% do tempo total de execução.

No último estágio, os nós externos da árvore são alinhados paralelamente. Esta abordagem obviamente depende da topologia da árvore obtida. No melhor caso, é possível obter um *speedup* de $N/logN$.

Nos testes, foi utilizado um *cluster* composto por oito processadores *dual-core*, Pentium III, 800 MHz conectados por Fast Ethernet. Foram alinhadas 500 sequências de proteínas de um tamanho médio de 1100 aminoácidos. Neste teste, cada processador recebe um lote de 80 alinhamentos em pares para realizar.

No artigo, são mostrados os tempos de execução e os *speedups* obtidos com o ClustalW-MPI para 500 sequências. O *speedup* do cálculo das distâncias é quase linear sendo 15,8x usando 16 processadores. Já para a fase 3 (alinhamento progressivo), é obtido um *speedup* de 4,3x utilizando 16 processadores.

4.1.3 ClustalW em FPGA

T. Oliver *et. al* [44] criaram uma abordagem paralela baseada em FPGA (Seção 3.2) para a execução do ClustalW (Seção 2.6.1).

Analisando as fases dos três estágios do ClustalW, os autores concluíram que mais de 90% do tempo total de execução foi gasto na primeira fase (alinhamentos em pares) e esta foi a fase adaptada para *hardware* reconfigurável.

O cálculo que utiliza programação dinâmica exibe uma alta regularidade e pode ser eficientemente mapeado em um *array* de pequenos elementos de processamento (PE). Cada caractere de uma sequência (sequência de busca) é atribuído para cada PE e os caracteres de uma segunda sequência (sequência *database*) percorrem os PEs. Desta forma, cada PE potencialmente calcula uma célula da matriz de programação dinâmica a cada ciclo de clock. Foi necessário particionar a sequência de busca, pois o comprimento de uma típica sequência de aminoácido é maior que a quantidade de PEs que a placa FPGA utilizada suporta.

Apesar da eficiência no cálculo das matrizes, é necessário obter o alinhamento das sequências, e a operação de *traceback* não pode ser tão eficientemente paralelizada em FPGA. Por isso, essa fase foi executada em CPU.

Nos testes, foi utilizada uma placa Xilinx Virtex II XC2V600, onde foi possível acomodar 92 PEs a um clock de 34MHz. A implementação conseguiu um desempenho de aproximadamente 1 GCUPS (Bilhões de Atualizações de Células por Segundo). No artigo, são comparados os resultados do desempenho do ClustalW em FPGA e a solução em *software* do ClustalW, executando em um Pentium IV de 3GHz e 1GB de memória RAM.

Os resultados mostram que, para a primeira fase, foi possível obter um *speedup* de 45 a 50 vezes.

Da mesma maneira que no ClustalW-MT, os melhores desempenhos são obtidos com sequências mais curtas. Observou-se também que o tempo de execução total foi acelerado em mais de dez vezes, enquanto o *speedup* da fase 1 foi ao menos 45x. Sendo assim, os autores concluíram que o tempo de execução da fase 1 (alinhamentos em pares) não é o que mais influencia no resultado final e a otimização da terceira fase é citada como trabalhos futuros.

4.1.4 GPU-ClustalW

GPU-ClustalW [27] é um exemplo de algoritmo que executa em GPUs que não possuem suporte à computação de propósito geral, ou seja, o ClustalW (Seção 2.6.1) foi re-escrito de forma a simular uma renderização de vídeo. Uma típica renderização segue uma ordem fixa de três estágios de processamento, chamada de *pipeline* gráfico.

O primeiro estágio, de processamento de vértices, transforma um conjunto de vértices tridimensionais em uma tela de vértices de duas dimensões. Na segunda fase, o rasterizador converte a representação geométrica dos vértices em coordenadas de uma tela. No final, o processador de fragmentos constrói uma cor para cada pixel, lendo os pixels de uma memória de textura. As GPUs permitem programar o processador de vértices e de fragmentos. Porém, os programas de fragmentos podem ser utilizados para implementar qualquer operação matemática em um ou mais vetores de entrada (texturas ou fragmentos) para computar a "cor" de um pixel.

Para calcular o alinhamento múltiplo, apenas a primeira fase do algoritmo ClustalW foi paralelizada, pois os autores concluíram que mais de 93% do tempo gasto no algoritmo é na primeira fase, conforme alguns testes de 200 a 1000 sequências.

Na implementação, é aproveitado o fato de que todos os membros de uma antidiagonal da matriz podem ser calculados em paralelo (*Wavefront* tradicional, Seção 2.4). Como a célula de posição (i, j) possui como dependência os vizinhos $(i-1, j)$ $(i, j-1)$ e $(i-1, j-1)$, sempre são mantidas em memória três antidiagonais em três *buffers* diferentes e é utilizado um método cíclico para sobrescrevê-los a cada iteração. As diagonais $k-1$ e $k-2$ são armazenadas como texturas de entrada e a diagonal k é o alvo a ser renderizado. Na iteração seguinte, k se torna $k-1$, $k-1$ se torna $k-2$ e $k-2$ se transforma em k.

Considerando um conjunto de n sequências, elas são ordenadas de acordo com o tamanho, de modo a calcular todas as $n \times (n-1)/2$ comparações em pares. Para melhor desempenho, as sequências são armazenadas em texturas 2D e múltiplas comparações em pares podem ser feitas ao mesmo tempo.

No algoritmo, os valores $H(i, j), H(i-1, j), H(i, j-1)$ são calculados para cada célula e armazenados em um canal-A de um pixel de cor RGBA em dois alvos de renderização separados. Cada pixel armazena uma conta matemática diferente para calcular o valor da célula usando Smith-Waterman [52] e considerando penalidades *affine-gap* [34].

O algoritmo proposto foi implementado com a linguagem de programação GLSL (*OpenGL Shading Language*) e foi testado com a placa de vídeo NVidia GeForce 7800 GTX, com 627 MHz, 512 MB de RAM a uma velocidade de 1.83 GHz, 8 processadores de vértices e 24 processadores de fragmentos. A estação onde os testes foram realizados possuía processador Intel Pentium4 3.0GHz, 1GB RAM, executando Windows XP.

Nos testes, foi possível obter um *speedup* de 10 vezes, comparado a uma solução em *software*.

Outras ferramentas de alinhamento mútiplo, como T-Coffee (Seção 2.6.1) e MUSCLE [11], também possuem fases que executam alinhamentos em pares e, de uma forma semelhante à apresentada no artigo, é proposto como trabalhos futuros a otimização dessas ferramentas.

4.1.5 MSA-CUDA

O MSA-CUDA [28] é um algoritmo que utiliza GPU para executar o ClustalW (Seção 2.6.1). Ele é construído utilizando CUDA (Seção 3.3.1). Nesta implementação, todas as três fases do algoritmo ClustalW foram paralelizadas e os resultados obtidos foram superiores aos obtidos com um *cluster* de 32 processadores.

Na primeira fase do algoritmo, utiliza-se Smith-Waterman [52] para encontrar o alinhamento ótimo de todos os pares de sequências que se deseja alinhar. Para obter o alinhamento é utilizado o algoritmo *traceback* de Myers-Miller [35] utilizando espaço linear. Nesta fase, a computação das distâncias dos pares de sequências é definida como uma tarefa. Foram testadas duas abordagens: paralelização *inter-task* e paralelização *intra-task* [28].

Na paralelização *inter-task*, cada tarefa é atribuída para exatamente uma *thread* e *dimBlocks*-tarefas são executadas em paralelo, dentro do bloco de *threads*. Na paralelização *intra-task*, cada tarefa é atribuída para um bloco de *threads*, que contruibuem para finalizar a tarefa em paralelo, utilizando as propriedades das antidiagonais da matriz. A otimização *inter-task* exige muitas tarefas para ser eficiente. Caso existam poucas tarefas (≤ 100), os autores acreditam que a paralelização *intra-task* é preferível.

Várias otimizações foram utilizadas para se atingir um bom resultado. Primeiramente, ordena-se todas as sequências pelo tamanho. Desta forma, a diferença do tempo de execução de duas *threads* adjacentes no bloco de *threads* é reduzida. Para se obter um melhor uso da largura de banda, todas as *threads* utilizam um acesso padronizado (*coalesced*) à memória [43].

Ambos métodos de paralelização utilizam memória constante para armazenar parâmetros de somente leitura, assim como a matriz de substituição (Seção 2.2). A matriz de substituição é carregada para a memória compartilhada ao ser acessada por uma *thread*, pois diferentes *threads* frequentemente acessam o mesmo dado da matriz de substituição. A memória de textura é utilizada para armazenar as sequências ordenadas. Estas são as otimizações apresentadas para a primeira fase do algoritmo.

Na segunda fase, a árvore guia é construída através do método *neighbor-joining* (NJ). O algoritmo utilizado para esta fase é apresentado em detalhes em um outro trabalho dos autores [29]. Esta fase pode ser dividida em duas subfases: reconstrução de uma árvore sem raiz e criação da raiz da árvore NJ com o cálculo do peso das sequências. Um bloco de *threads* é responsável por calcular as diferenças entre os filhos de cada nó da árvore e cada *thread* é responsável por calcular em um subconjunto separado de nós. Nesta etapa, a memória compartilhada é utilizada para armazenar resultados e a memória de textura é utilizada para armazenar a estrutura da árvore.

Na última fase é realizada uma série de alinhamentos segundo a estrutura da árvore. Obviamente, em uma árvore, todos os alinhamentos que estão no mesmo nível podem ser paralelizados. Porém, mesmo alinhamentos que não estão no mesmo nível podem ser paralelizados, como pode ser visto na Figura 4.2, onde os alinhamentos de mesmo padrão podem ser calculados em paralelo.

Para fazer a paralelização, cada possível alinhamento possui uma *flag* indicando se já foi ou não alinhado. Caso um alinhamento possua dois filhos que já foram alinhados, ele é marcado para ser alinhado na próxima iteração. Caso contrário, ele deverá aguardar até que os dois filhos sejam alinhados. O processo é repetido até que todos os nós tenham

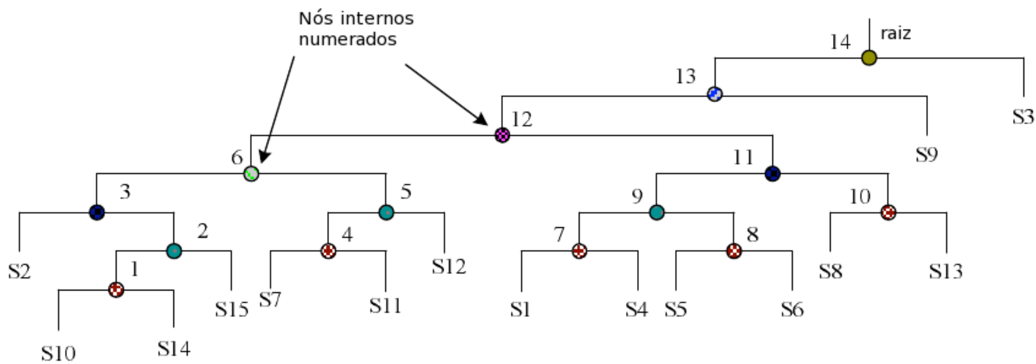

Figura 4.2: Exemplo de árvore produzida na segunda fase do MSA-CUDA [28].

sido alinhados. Todos os cálculos de comparação em pares dos alinhamentos que são produzidos são realizados na GPU.

Três algoritmos foram utilizados nos testes: ClustalW Sequencial, MSA-CUDA e ClustalW-MPI (Seção 4.1.2) executando-se em um *cluster* de 32 processadores.

O MSA-CUDA foi implementado em CUDA C e testado em uma estação com uma placa NVidia GeForce GTX 280 e 1 GB RAM e um processador AMD Opteron 248 2.2 GHz.

Nos resultados obtidos, a fase 1 do algoritmo claramente possuiu um melhor desempenho utilizando a abordagem *inter-task*. Por isso, se existir memória suficiente na GPU, o MSA-CUDA escolhe a otimização *inter-task* para esta fase. Os *speedups* são maiores em um número menor de sequências, mas de comprimento maior, pois neste caso o esforço computacional é maior.

Na fase dois, era esperado que o tamanho das sequências não influenciasse no desempenho e por isso os melhores *speedups* foram obtidos nessa fase com o aumento do número das sequências. Na terceira fase, existe uma grande divergência entre os *speedups* obtidos. Isto acontece pois a topologia da árvore guia e o tamanho das sequências interferem no desempenho. Normalmente, um número grande de sequências longas significa um melhor *speedup*.

Finalmente, os autores fazem a comparação entre o MSA-CUDA, o ClustalW-MPI e o ClustalW sequencial. Para um grande número de sequências, o desempenho do ClustalW-MPI cai muito pois a segunda fase exige grande esforço computacional e o ClustalW-MPI não paraleliza esta fase. Entretanto, mesmo no caso em que o ClustalW-MPI apresenta um bom desempenho em relação ao sequencial, uma única GPU utilizando MSA-CUDA é capaz de ultrapassar o desempenho do ClustalW-MPI em um *cluster* com 32 processadores, mostrando uma excelente relação de custo/desempenho.

4.1.6 G-MSA

G-MSA [4] é uma proposta de implementação do algoritmo T-Coffee (Seção 2.6.1) em GPU. Inicialmente, as sequências de entrada são recebidas pela CPU e a GPU é usada para gerar os alinhamentos globais de todos os possíveis pares de sequência, bem como os 10 melhores alinhamentos locais sem sobreposição de todos os possíveis pares de sequência. A seguir, a fase de construção da biblioteca primária (PL) também é executada em GPU.

Devido à grande necessidade de memória, essa fase é dividida em subproblemas, chamadas de janelas. Cada janela é processada por vez em GPU. A seguir, a biblioteca estendida (ES) também é gerada na GPU, usando uma estratégia que otimiza o uso de memória.

O algoritmo proposto foi implementado em C e CUDA e testado em 2 placas NVidia GTX-480. Os resultados foram obtidos com 25 a 500 sequências, cujo o tamanho variava de 100 a 420 aminoácidos. O G-MSA foi comparado com versões sequenciais do T-Coffee, Clustal-W e ClustalW-MPI (Seção 4.1.2), entre outros. Quando comparado ao T-Coffee sequencial, o G-MSA obteve um *speedup* de 193x, usando uma única GPU e fazendo o alinhamento múltiplo com a biblioteca estendida. O G-MSA, usando somente a biblioteca primária, conseguiu ser 10x mais rápido que o ClustalW sequencial.

4.2 Propostas de Alinhamento Exato

4.2.1 MSA exato com MPI

Helal *et. al* [17–19] apresentam estratégias em MPI para calcular o alinhamento múltiplo exato, que reduzem o espaço de busca sem usar o Carrillo-Lipman (Seção 2.5). Em [19], Helal *et. al* utilizam uma abordagem mestre/escravo, o espaço é dividido em ondas de computação que contém partições equidistantes de um mesmo ponto de origem. Essas ondas são pré-processadas para acelerar a computação. O processo mestre é responsável por consumir recursos e escalonar as partições nos processadores. Há um custo de comunicação dentro e através das ondas de processamento. Na abordagem distribuída [17, 18], não existe um processador mestre e o escalonamento de partições se dá através da interação entre processadores vizinhos.

O algoritmo proposto em [19] foi implementado em C e MPI, podendo ser executado em ambientes com até 64 processadores. Foi utilizado um SunFire X2200 com 2 processadores AMD Opteron *quad core* de 2.3 GHz, 512 Kb L2 cache e 2 MB L3 em cada processador. A estação estava equipada com 8GB RAM e 8 cores foram utilizados.

Seqs.	Tamanho	Espaço de busca (%)	Tempo total (s)	SP
2	20	51,0	0,07	5
2	20	75,0	0,10	5
2	20	99,0	0,13	5
2	20	100	0,11	5
3	31	8,21	0,24	10
3	31	20,36	1,05	5
3	31	83,59	1,91	5
3	31	99,94	4,61	5
5	41	0,14	115,37	-41
5	41	0,97	916,71	-116
5	41	3,39	8381,24	-219
5	41	8,35	50321,35	-219
5	41	16,62	214324,39	-219

Tabela 4.1: Resultados Experimentais do MSA exato com MPI [19]

Como pode ser visto na Tabela 4.1, o número de sequências comparadas é muito pequeno (até 5) e seu tamanho máximo também é pequeno (até 41). Como o algoritmo implementado é exato e, portanto, executa-se em tempo exponencial, pode ser notada a grande diferença entre os tempos de comparação de três sequências e os tempos de

comparação de cinco sequências. Os testes realizados aumentam gradativamente o espaço de busca e mostram a porcentagem deste que foi necessária para encontrar o alinhamento ótimo.

4.2.2 MSA exato em FPGA

Utilizando FPGA (Seção 3.2), Yamaguchi *et. al* [30] implementaram o algoritmo Carrillo-Lipman (Seção 2.5) para o alinhamento múltiplo exato. Sua solução consiste em utilizar o FPGA para calcular o escore visitando todos os vizinhos de uma célula do espaço de busca e utilizar dados calculados em *software* para auxiliar na redução do espaço de busca.

No circuito implementado, a programação dinâmica N-dimensional é realizada repetindo-a em duas dimensões e variando as outras dimensões. Considere X, Y, Z e T os tamanhos de quatro sequências nos eixos x, y, z e t respectivamente. Sejam W_x, W_y, W_z e W_t as partes de uma sequência que pode ser processada continuamente, sem ser necessária qualquer entrada ou saída de dados da placa. A Figura 4.3 mostra como a programação dinâmica em três dimensões é realizada através de repetições de uma programação dinâmica de duas dimensões. A área em processamento é chamada de janela de varredura [30].

O total de ciclos pode ser aproximado por [30]:

4 dimensões:

$$max(W_x + W_y, 24^2/8 \times 2) \times \frac{XYZT}{W_xW_y} + max(24^2/2, W_xW_y \times 2/5) \times \frac{XYZT}{W_xW_yW_z}$$

5 dimensões:

$$max(W_x + W_y, 24^2/16 \times 2) \times \frac{XYZTU}{W_xW_y} + max(24^4/2, W_xW_y \times 2/5) \times \frac{XYZTU}{W_xW_yW_z} + max(24^2/2, W_x, W_y, W_z \times 2/5) \times \frac{XYZTU}{W_xW_yW_zW_t}$$

Para implementar o método de Carrillo-Lipman, seja P, assim como mostrado na Seção 2.5, o conjunto do espaço de busca reduzido. O limite do espaço de busca que deve ser calculado para se obter o alinhamento é calculado em CPU e carregado em bancos externos de memória na placa FPGA. Antes de iniciar o cálculo de alguma janela de varredura, o algoritmo decide se a janela de varredura atual deve ou não ser calculada. Desta forma, o espaço de busca é reduzido.

Dois circuitos foram implementados, um para alinhamento de quatro sequências e outro para alinhamento de cinco sequências com frequências de 36MHz e 31MHz, respectivamente. A frequência baixa de operação é causada pelos seletores para escolher entre $2^n - 1$ candidatos.

A Tabela 4.2 compara o desempenho do MSA em FPGA com MSA 2.0 (Seção 2.5.4), executado em um processador Intel Pentium4 3 GHz com 2GB de memória. O *speedup* obtido é influenciado pela similaridade das sequências comparadas. Quando as sequências são muito semelhantes, a solução em *software* funciona muito bem, pois grande parte do espaço de busca não é calculado. Por outro lado, grandes blocos de processamento são sempre executados em FPGA, resultando em um desempenho inferior neste caso.

Por não ser sempre superior, é aconselhável utilizar a solução em FPGA em paralelo com a *software*, pois algumas soluções podem ser melhores sem utilizar FPGA. Para problemas de baixa similaridade foi possível obter *speedups* de até 50 vezes quando comparada

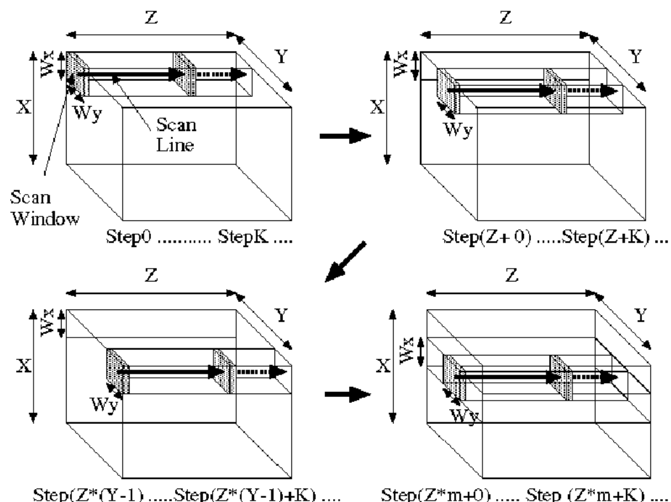

Figura 4.3: Programação dinâmica em três dimensões [30].

problema	Tamanho das seqs.	Tempo de execução (s)		*Speedup*
		MSA 2.0	FPGA	
1ajsA	364, 373, 358, 387	115,0	2,33	49
1bgl	993, 962, 950, 938	956,7	19,6	49
1gowA	469, 451, 476, 481	124,4	4,46	28
1ac5	483, 450, 421, 425	68,0	4,10	17
arp	380, 395, 418, 398, 394	597,9	304	2,0
1pkm	443, 439, 434, 449	3,36	1,89	1,8
glg	458, 475, 486, 484, 438	15,4	208	0,07

Tabela 4.2: Resultados Experimentais do MSA exato em FPGA [30].

à solução em *software*. Como trabalhos futuros, é proposto melhorar a baixa frequência dos circuitos.

4.3 Quadro Comparativo

Nesta seção são comparadas as implementações do alinhamento múltiplo de sequências descritas nas Seções 4.1 e 4.2, Nessa comparação são considerados o número de sequências, o tempo obtido (se disponível), o ano do trabalho e o *speedup* (quando existir).

A Tabela 4.3 compara as implementações de algoritmos heurísticos em plataformas de alto desempenho. Como pode ser visto, o ClustalW foi o algoritmo mais adaptado para plataformas de alto desempenho. Quanto ao tipo de plataforma usada, observamos que as implementações mais recentes são feitas em GPU e que, para o ClustalW, as implementações mais recentes conseguem comparar até 8000 sequências em GPU. Os melhores *speedups* são também obtidos em GPUs, sendo que alguns são comparáveis a um *cluster* de 32 cores.

Ainda na Tabela 4.3, uma implementação que utiliza GPU [27] teve um *speedup* inferior quando comparado ao uso de *hardware* reconfigurável [44], mas alguns anos depois, com a evolução das GPUs e, possivelmente, uma implementação melhor, o *speedup* médio obtido foi maior [28]. Apesar de que os *speedups* citados são os que foram apresentados nos

artigos e servem apenas como uma noção da aceleração, não deve ser feita a comparação direta entre eles. Os *speedups* variam de acordo com a forma utilizada para medir tempo, com as sequências utilizadas, com o tamanho delas e a quantidade de sequências. Em cada trabalho, a forma de medir o tempo muitas vezes não era citada e os conjuntos de sequências utilizados eram diferentes.

Ref.	Algortimo	Tipo	Ano	Seqs. (Tamanho)	Tempo (s)	*Speedup*
[6]	MT-ClustalW	SMP (8*threads*)	2006	200 (200) 800 (800) 500 (500)	22 7550 2194,5	1,18 1,13 1,13
[25]	ClustalW-MPI	Cluster (16 cores)	2003	500 (1000)	13000	9
[44]	ClustalW em FPGA	FPGA	2005	200 (412) 1000 (446) 600 (448,4)	14,7 399,5 181,28	13,3 11,8 12,08
[27]	GPU ClustalW	GPU	2006	1000 (446) 600 (436,4)	680,7 311,92	6,9 6,92
[28]	MSA-CUDA (ClustalW)	GPU	2009	400 (856) 8000 (73) 3233 (392,8)	N/A	32,29 10,38 22,3
[4]	G-MSA (T-Coffee)	GPU	2012	25 (100) 500 (420)	5 (PL) e 5 (ES) 9 (PL) e 700 (ES)	10 193

Tabela 4.3: Comparação entre as estratégias paralelas heurísticas em plataformas de alto desempenho

A Tabela 4.4 apresenta as implementações de algoritmos exatos de alinhamento múltiplo de sequências em plataformas de alto desempenho. Como pode ser visto, só encontramos duas propostas de implementação do algoritmo exato. Essas duas propostas comparam poucas sequências (até 5) e possuem um tempo de execução expressivo. A nosso conhecimento, não existe implementação em GPU do algoritmo MSA exato.

As arquiteturas de *hardware* reconfigurável apresentam um *speedup* considerável, tanto que a melhor ferramenta conhecida para o algoritmo ótimo utiliza FPGA [30].

Ref.	Tipo	Ano	Seqs. (Tamanho)	Tempo (s)	*Speedup*
[19]	*MSA exato com MPI. Cluster (8 cores)*	2009	2 (20) 5 (41) 3,28 (30,78)	0,11 8381,24 702,9	N/A
[30]	MSA exato em FPGA	2007	4 (370,5) 4 (960,75) 5 (397) 5 (467,8) 4,28 (635)	2,33 19,6 304 208 77,8	49 49 2,0 0,07 21

Tabela 4.4: Comparação entre as estratégias paralelas exatas em plataformas de alto desempenho.

Capítulo 5

Projeto Carrillo-Lipman em GPU para Alinhamento Múltiplo Exato

5.1 Considerações Iniciais

Conforme visto na Seção 2.3, o problema do alinhamento múltiplo de sequências é NP-Difícil, pois o número de vizinhos que precisam ser visitados para calcular o valor da célula corrente aumenta exponencialmente com o número de sequências. Carrillo-Lipman (Seção 2.5) propuseram limites inferior e superior que permitem que algumas células não sejam calculadas, reduzindo assim o espaço de busca. Mesmo com essa redução, o espaço de busca continua exponencial e sua exploração em tempo hábil necessita de plataformas de computação alto desempenho.

As GPUs (Seção 3.3) oferecem alto desempenho através da exploração de processamento massivamente paralelo, que permite a execução simultânea de milhares de *threads*.

Para o projeto da estratégia paralela para execução do Carrillo-Lipman decidimos então utilizar GPUs. Para a adaptação do Carrillo-Lipman a GPUs, consideramos duas estratégias de organização de *threads*, que chamamos de abordagem de granularidade grossa e abordagem de granularidade fina.

Na abordagem de granularidade grossa, cada *thread* calcula o valor de uma célula no espaço de busca. Para percorrer o espaço de busca, os índices das *threads* são pré-calculados e armazenados em um *array* chamado de *wavefront* pré-calculado. Porém, nesta estratégia de granularidade grossa, detectou-se que o número de células calculadas em paralelo pode diminuir ao se percorrer o espaço de busca, reduzindo assim o desempenho. Sendo assim, foi necessário projetar uma outra estratégia.

Pensando no crescimento exponencial dos vizinhos, uma estratégia de granularidade fina foi elaborada. Nela, cada *thread* calcula o valor de um vizinho e um bloco de *threads* calcula o valor de uma célula do espaço de busca. Para percorrer o espaço de busca, utiliza-se um comportamento semelhante ao trabalho em FPGA [30] para o alinhamento múltiplo, descrito na Seção 4.2.2, porém o comportamento é ligeiramente alterado para não se utilizar uma janela de processamento, mas sim diagonais de um plano do espaço de busca.

Neste capítulo são descritas as abordagens de granularidade grossa e granularidade fina, destacando-se as vantagens e os desafios encontrados para cada uma dessas estratégias.

5.2 Estratégia de Granularidade Grossa

5.2.1 Visão Geral

A estratégia de granularidade grossa utiliza o *wavefront* multidimensional (Seção 2.4, onde cada *thread* da GPU calcula uma célula da matriz de programação dinâmica. A Figura 5.1 apresenta uma visão geral dessa estratégia.

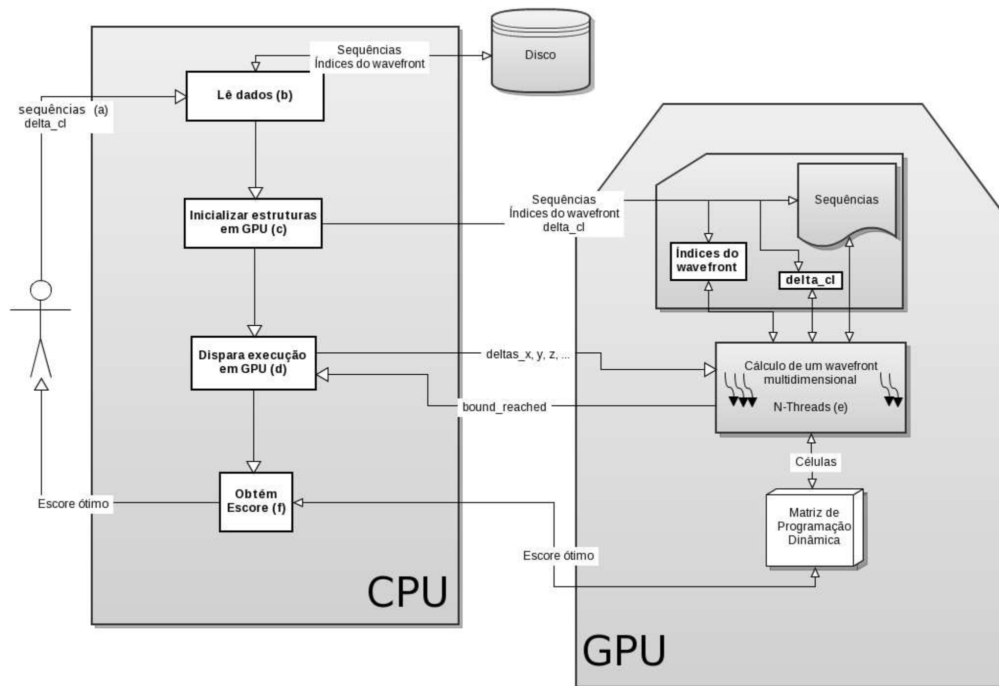

Figura 5.1: Resumo da estratégia de granularidade grossa.

Inicialmente, o usuário solicita a execução informando as sequências e o limite do Carrillo-Lipman (delta_cl) (Figura 5.1a). O programa, então, executando em CPU, lê as sequências e os índices do *wavefront* pré-calculado (Figura 5.1b). Esses índices foram calculados previamente e visam acelerar o cálculo em GPU. A seguir, é executado o módulo de inicialização da execução em GPU (Figura 5.1c) que copia as sequências, os índices pré-calculados e o delta_cl para GPU.

O cálculo da matriz de programação dinâmica é então feito de forma iterativa da seguinte maneira. Para cada *wavefront*, um kernel em GPU é invocado (Figura 5.1d), tendo os deltas do espaço de busca como parâmetro. Esses deltas indicam os deslocamentos do *wavefront* multidimensional (Seção 2.4) em relação à origem do espaço de busca e determinam as células que serão calculadas nessa iteração. O cálculo propriamente dito é feito por N *threads* em GPU, onde cada *thread* calcula uma célula da matriz de programação dinâmica (Figura 5.1e). Para esse cálculo, células anteriormente calculadas da matriz de programação dinâmica são lidas. Caso não haja *threads* suficientes para calcular um *wavefront* inteiro, diferentes partes do mesmo *wavefront* podem ser calculadas em diversas iterações. Ao terminar o cálculo, uma ou mais *threads* podem concluir que o limite Carrillo-Lipman foi atingido. Se esse for o caso, a variável bound_reached é setada para 1 e é usada pela CPU para descartar a execução de *wavefronts* que não são necessários.

Ao final do cálculo da matriz de programação dinâmica, o escore ótimo é copiado para CPU (Figura 5.1f). Finalmente, o escore ótimo é retornado para o usuário.

5.2.2 Cálculo do wavefront uma thread por célula

A Figura 5.2 ilustra as modificações do algoritmo exato para a estratégia grossa, onde cada *thread* calcula o valor de uma célula do espaço de busca.

O retângulo das linhas 1 a 3 destaca um trecho de código que foi paralelizado. Algumas das células percorridas são calculadas em paralelo, obedecendo a dependência de dados do *wavefront* multidimensional (Seção 2.4). O retângulo dass linhas 4 a 30 destaca o código que é executado em GPU. Inicialmente, cada *thread* calcula os valores de *match* e *mismatch* das sequências comparadas (linhas 4 a 20). Posteriormente, todos os valores dos vizinhos são consultados e os possíveis valores que a célula pode assumir são calculados e armazenados localmente em cada *thread* (linhas 22 a 28). Ao final, o menor dentre esses valores é escolhido e armazenado como o valor da célula que foi calculada (linha 30).

O objetivo é que algumas células percorridas das linhas 1 a 3 da Figura 5.2 sejam calculadas em paralelo. Para isso, definimos que um bloco de *threads* irá calcular um *wavefront* multidimensional. O tamanho deste *wavefront* depende de dois fatores: o número de sequências e o número máximo de *threads* em um bloco suportadas pelo *hardware.*

Optamos por utilizar um programa auxiliar para calcular, em tempo de compilação, os índices necessários à execução de cada *wavefront* e armazená-los no que chamamos de "*wavefront* pré-calculado.", para acelerar a computação.

	1 thread/bloco	3 *threads*/bloco	6 *threads*/bloco	512 *threads*/bloco	1024 *threads*/bloco
	(0,0,0)	(0,0,1) (0,1,0) (1,0,0)	(0,0,2) (0,1,1) (0,2,0) (1,0,1) (1,1,0) (2,0,0)	(0,0,30) (0,1,29) (0,2,28) ... (29,1,0) (30,0,0)	(0,0,43) (0,1,42) (0,2,41) ... (42,1,0) (43,0,0)
Número efetivo de threads:	**1**	**3**	**6**	**496**	**990**

Tabela 5.1: Índices do *wavefront* pré-calculado, e seu tamanho, pelo número de *threads* por bloco para 3 sequências.

	1 thread/bloco	4 *threads*/bloco	10 *threads*/bloco	512 *threads*/bloco	1024 *threads*/bloco
	(0,0,0,0)	(0,0,0,1) (0,0,1,0) (0,1,0,0) (1,0,0,0)	(0,0,0,2) (0,0,1,1) (0,0,2,0) (0,1,0,1) (0,1,1,0) (0,2,0,0) (1,0,0,1) (1,0,1,0) (1,1,0,0) (2,0,0,0)	(0,0,0,12) (0,0,1,11) (0,0,2,10) (0,0,3,9) (0,0,4,8) (0,0,5,7) (0,0,6,6) ... (11,1,0,0) (12,0,0,0)	(0,0,0,16) (0,0,1,15) (0,0,2,14) (0,0,3,13) (0,0,4,12) (0,0,5,11) (0,0,6,10) ... (15,1,0,0) (16,0,0,0)
Número efetivo de threads:	**1**	**4**	**10**	**455**	**969**

Tabela 5.2: Índices do *wavefront* pré-calculado, e seu tamanho, pelo número de *threads* por bloco para 4 sequências.

```
1: for i = 1 → n1 do
2:    for j = 1 → n2 do
3:       for k = 1 → n3 do
4:          if S1(i) = S2(j) then
5:             c_ij = smatch
6:          else
7:             c_ij = smis
8:          end if
9:
10:         if S1(i) = S3(k) then
11:            c_ik = smatch
12:         else
13:            c_ik = smis
14:         end if
15:
16:         if S2(j) = S3(k) then
17:            c_jk = smatch
18:         else
19:            c_jk = smis
20:         end if
21:
22:         d1 = D(i − 1, j − 1, k − 1) + c_ij + c_ik + c_jk
23:         d2 = D(i − 1, j − 1, k) + c_ij + sspace
24:         d3 = D(i − 1, j, k − 1) + c_ik + sspace
25:         d4 = D(i, j − 1, k − 1) + c_jk + sspace
26:         d5 = D(i − 1, j, k) + 2 ∗ sspace
27:         d6 = D(i, j − 1, k) + 2 ∗ sspace
28:         d7 = D(i, j, k − 1) + 2 ∗ sspace
29:
30:         D(i, j, k) = Min[d1, d2, d3, d4, d5, d6, d7]
31:      end for
32:   end for
33: end for
```

Figura 5.2: Alterações no algoritmo para a estratégia grossa.

As Tabelas 5.1 e 5.2 mostram os números de células que podem ser calculadas em paralelo em um wavefront multidimensional de três e quatro sequências, respectivamente, de acordo com a capacidade do *hardware*. Neste caso, consideramos um máximo de 1024 *threads* por bloco. As cinco colunas mostram o *wavefront* pré-calculado, dependendo do número de *threads* por bloco. Ao percorrer o espaço de busca os valores do *wavefront* pré-calculado são somados aos deslocamentos em relação à origem (variáveis delta).

Os valores apresentados nas Tabelas 5.1 e 5.2 são facilmente calculados por um programa auxiliar. Este *array* produzido é chamado de *wavefront* pré-calculado, sendo armazenado em memória local da GPU e as *threads* o utilizam para descobrir qual célula dentro de um *wavefront* elas devem calcular.

Desta forma, é evitado recálculo de células, ou seja, duas *threads* diferentes não calculam a mesma célula e são reduzidos os cálculos para se gerar os esses índices em tempo de execução.

5.2.3 Percorrendo o espaço de busca

Nesta seção, iremos explicar como funciona o esquema de percorrer o espaço de busca como se o limite Carrillo-Lipman não fosse utilizado, ou seja, quando todas as células do espaço de busca são calculadas.

O algoritmo é dividido em três fases. A primeira fase consiste no cálculo das primeiras células da matriz de programação até um limite de número de *threads* (T) determinado pelo *hardware*. A segunda fase consiste em se calcular T células paralelamente e a terceira fase é o calculo das células restantes, com um número de *threads* menor que T.

Na primeira fase, começa-se calculando o primeiro *wavefront* (célula (0,0,0)) na Figura 5.3, e se continua calculando *wavefronts* completos enquanto apenas um bloco conseguir calcular todas as células disponíveis nos *wavefronts*. Desta forma, na primeira iteração, a primeira coluna da Tabela 5.1 é calculada, na segunda iteração, a segunda coluna é calculada, e continua-se com este crescimento enquanto existir um número de células que o *hardware* suporte. Até esse limite, todas as células são calculadas na próxima iteração do algoritmo.

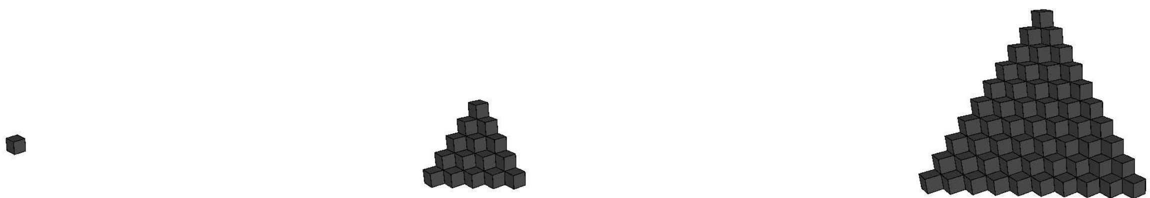

Figura 5.3: Primeiros passos para percorrer o espaço de busca utilizando a estratégia de granularidade grossa.

Como normalmente existe um número maior de células do que o número de *threads* suportado pelo *hardware*, alguns *wavefronts* não podem ser inteiramente calculados em um único bloco de *threads*. Nesse caso, partes do mesmo *wavefront* são calculadas desviando-se

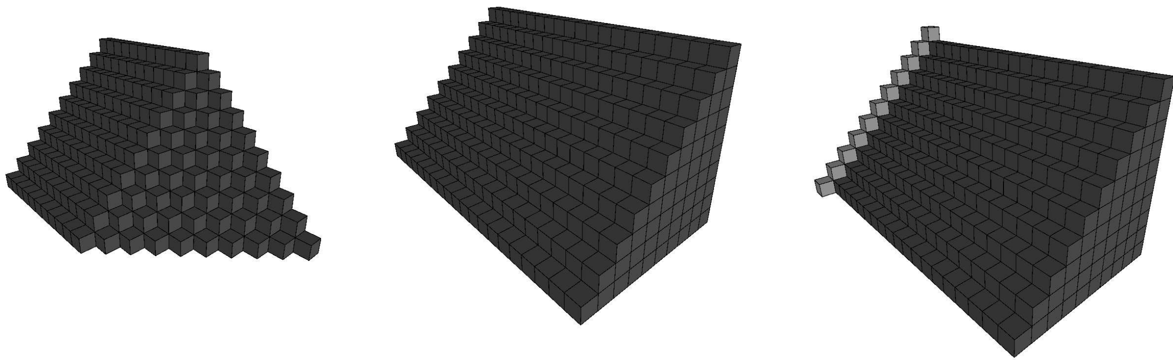

Figura 5.4: Passos finais para percorrer o espaço de busca utilizando a estratégia de granularidade grossa.

em uma das dimensões, conforme a Figura a 5.4. São utilizados contadores delta (delta_x, delta_y e delta_z) que armazenam o desvio do *wavefront* que está sendo calculado em relação à origem. No começo desta fase, todos os contadores delta possuem o valor 0 e então delta_x é aumentado em 1 e o *kernel* é invocado, com os novos parâmetros, enquanto delta_x for menor que o comprimento da primeira sequência.

Durante a terceira fase, última etapa da Figura 5.4, o valor de delta_x volta a ser novamente a zero e o valor de delta_y é aumentado em 1. Inicia-se então um ciclo parecido com o da segunda fase, mas calculando as células que ainda não foram calculadas. Quando delta_x possuir o tamanho da primeira sequência e delta_y possuir o tamanho da segunda sequência, seus valores voltam a ser zero e delta_z é incrementado em 1.

Este ciclo é repetido até que delta_x, delta_y e delta_z possuam o tamanho da primeira, segunda e terceira sequência, respectivamente. Nesta hora, a última célula possui o escore do alinhamento múltiplo exato.

5.2.4 Utilizando o limite de Carrillo-Lipman

Na nossa estratégia de granularidade grossa, o limite de Carrillo-Lipman para se descartar células será obtido através da execução do programa MSA 2.0 (Seção 2.5.4) e é utilizado pela nossa estratégia para eliminar espaço de busca que não influencie no escore do alinhamento ótimo. Este valor é recebido como argumento (Figura 5.1(a)) e é passado para a GPU em uma variável chamada delta_cl (δ).

Antes de calcular um *wavefront*, uma variável, *bound_unreached*, é inicializada com 0. Ao final da execução de cada *kernel*, caso alguma célula calculada pelo *wavefront* possua o valor menor que δ, o valor *bound_unreached* é alterado para 1.

Desta forma, se todas as células do *wavefront* calculado não influenciarem para o cálculo do alinhamento ótimo, *bound_unreached* possui valor 0 ao final da execução e alguns *wavefronts* seguintes podem ser descartados. Nesta hora, o valor de delta_x é incrementado imediatamente para o tamanho da primeira sequência e as células que não foram calculadas são inicializadas com infinito. Assim, ao serem lidas, o seu valor será o maior possível e não será escolhido entre os menores escores. Desta forma, o escore exato do alinhamento múltiplo de sequências é obtido.

5.2.5 Estruturas em Memória

Nesta seção mostraremos as principais estruturas em memória, os tipos de memórias utilizadas e o tamanho de cada uma para alinhamento múltiplo exato de três sequências de tamanho n. As memórias utilizadas são da GPU NVidia e suas características foram descritas na Seção 3.3.

- **Sequências:** as sequências são armazenadas na memória constante. Cada aminoácido das sequências ocupa um byte em memória. Também são armazenados três contadores com o tamanho das sequências.
- **Matriz de programação dinâmica:** A matriz de programação dinâmica é uma matriz n^k. A princípio, é armazenado todo o espaço de busca. A descrição de uma melhora para a redução do uso de memória é feita na Seção 5.2.7. Este é o fator limitante do tamanho das sequências da abordagem. Por seu tamanho, a matriz de programação dinâmica é armazenada em memória global.
- **Deltas:** para que o *kernel* identifique qual *wavefront* está sendo calculado, ele recebe como argumento três variáveis: delta_x, delta_y e delta_z, que são os desvios em relação à primeira célula calculada pelo algoritmo. Essas variáveis são armazenadas em memória local.
- **Índices do *wavefront* pré-calculado:** O *wavefront* pré-calculado foi explicado na Seção 5.2.2 e é armazenado em memória global. O tamanho M do array pode variar de acordo com o *hardware* utilizado, podendo chegar até 1024 [43].
- **Escores dos vizinhos e menor_escore:** Em um alinhamento de três sequências, cada célula do espaço de busca possui sete vizinhos que são consultados para determinar o escore da célula que está sendo calculada. O primeiro vizinho calculado possui seu valor armazenado na variável menor_escore. Os vizinhos subsequentes tem o valor armazenado em uma variável temporária. Depois de calcular um vizinho, o seu valor é comparado com menor_escore, caso seja menor, este valor é atribuído a menor_escore. Todos os escores são armazenados em memória local.
- **Limite do Carrillo-Lipman:** Duas variáveis são utilizadas para descartar células que não precisem ser calculadas. A primeira é recebida pelo algoritmo como parâmetro e deve ser obtida utilizando o programa MSA 2.0 (Seção 2.5.4), que executa em CPU. Chamada de delta_cl (δ), é equivalente a variável delta do MSA 2.0. A segunda variável *bound_unreached* possui um valor 0 antes da execução do *kernel*, e tem seu valor alterado caso alguma célula do *wavefront* não possa ser descartada. Esses valores são armazenados em memória global.

A Tabela 5.3 apresenta um resumo do tamanho e tipo de memórias utilizadas para cada estrutura.

5.2.6 Pseudocódigo em GPU

A figura 5.5 representa o código executado em GPU pela estratégia grossa. Durante o cálculo do *wavefront* pré-calculado, também é definido WV_SIZE, que é o número de

Estrutura	Memória	Tipo
Sequências	$k \times n$ *bytes*	Constante
Matriz de programação dinâmica	$n \times n \times n$ *integer*	Global
Deltas	k *integer*	Local
Índices do *wavefront*	$M \times k$ *integer*	Local
Escore do vizinho	1 *integer*	Local
Menor escore	1 *integer*	Local
Limite do Carrillo-Lipman	2 *integer*	Global

Tabela 5.3: Estruturas e memória utilizada para k sequências de tamanho n na estratégia de granularidade grossa.

células presentes no *wavefront* e é utilizado para inicializar a matriz *array_index* que armazena o *wavefront* pré-calculado.

Quando o *kernel* é invocado, o procedimento *init_wv_pre_cal_array*() o inicializa de acordo com a Tabela 5.1 (linha 1). Cada *thread* calcula os índices x, y e z da matriz de programação dinâmica que será calculado por ela. Esses índices são obtidos acessando o *wavefront* pré-calculado e aplicando o deslocamento em relação à origem (*delta_x*, *delta_y*, *delta_z*) recebidos como argumento da CPU (linhas 3 a 5). Então os custos de *match* e *mismatch* de cada sequência é calculado (linhas 7 a 9). Na linha 11 à linha 17, os valores de todos os vizinhos da células são obtidos e os custos de *match* e *mismatch* são aplicados a cada um, assim decide-se todos os possíveis valores que uma célula pode possuir. Ao final (linha 19) o menor valor dentre todos os possíveis é escolhido como o valor da célula calculada pela *thread*.

5.2.7 Análise da abordagem

Uma das vantagens da estratégia de granularidade grossa é que o pré-cálculo do *wavefront* garante que um grande número de *threads* seja utilizado até atingir o limite do *hardware* durante a segunda fase do algoritmo. A manutenção do formato de um *wavefront* ao percorrer o espaço de busca traz uma característica muito desejável, que é saber em quantas iterações são necessárias até que uma célula não seja mais acessada. Por exemplo, para três sequências, no primeiro *wavefront*, a célula calculada é a $(0, 0, 0)$. Logo em seguida são calculadas as células $(0, 0, 1), (0, 1, 0)$ e $(1, 0, 0)$ no segundo *wavefront*. Em seguida, são calculadas em paralelo as células $(0, 1, 1), (1, 0, 1), (1, 1, 0)$, dentre outras, durante o terceiro *wavefront*. Neste momento, temos todos os vizinhos necessários para se calcular a célula $(1, 1, 1)$ no quarto *wavefront* e após este cálculo, sabemos que as células do primeiro *wavefront* não são mais necessárias e podem ser liberadas.

Tratando os casos das células que estão na borda do *wavefront* como exceção, todas as outras células possuem um tempo de vida igual e fixo: os vizinhos que precisam dos valores que foram calculados nos três *wavefronts* anteriores para três sequências, quatro para quatro sequências e assim por diante. Assim, esta abordagem permite uma gerência muito simples e eficiente de memória.

Na segunda fase do algoritmo, existe um paralelismo muito alto. Porém, na terceira fase do algoritmo, a quantidade de células que podem ser calculadas paralelamente não é tão alta e esta queda de performance é a principal desvantagem desta abordagem.

Conforme pode ser visto nas Figuras 5.3 e 5.4, percorrer o espaço de busca formando um *wavefront* permite obter um alto paralelismo apenas nas primeira e segunda fases.

$__global__$ *void msa* (*char* seq_1[ ], *char* seq_2[ ], *char* seq_3[ ], *int matriz_pd*[ ][ ][ ], *int delta_x*, *int delta_y*, *int delta_z*, *int* δ, *int* $*bound_unreached$)

1: $int\ array_index[WV_SIZE][3] = init_wv_pre_calc();$
2:
3: $int\ x = array_index[threadIdx.x][0] + delta_x;$
4: $int\ y = array_index[threadIdx.x][1] + delta_y;$
5: $int\ z = array_index[threadIdx.x][2] + delta_z;$
6:
7: $c_{xy} = cost(seq_1, seq_2, x, y);$
8: $c_{yz} = cost(seq_2, seq_3, y, z);$
9: $c_{xz} = cost(seq_1, seq_3, x, z);$
10:
11: $d_1 = matriz_pd[x-1][y-1][z-1] + c_{xy} + c_{yz} + c_{xz};$
12: $d_2 = matriz_pd[x-1][y-1][z] + c_{xy} + GAP;$
13: $d_3 = matriz_pd[x-1][y][z-1] + c_{xz} + GAP;$
14: $d_4 = matriz_pd[x][y-1][z-1] + c_{yz} + GAP;$
15: $d_5 = matriz_pd[x-1][y][z] + GAP;$
16: $d_6 = matriz_pd[x][y-1][z] + GAP;$
17: $d_7 = matriz_pd[x][y][z-1] + GAP;$
18:
19: $matriz_pd[x][y][z] = MIN(d_1, d_2, d_3, d_4, d_5, d_6, d_7);$
20:
21: **if** $(matriz_pd[x][y][z] < \delta)$ **then**
22: $\quad *bound_unreached = 1;$
23: **end if**

Figura 5.5: Código executado em GPU pela estratégia grossa.

Como o tamanho do *wavefront* é limitado pelo *hardware*, algumas células não são calculadas na segunda fase. Na terceira fase, procura-se calcular em paralelo o maior número de células possível, mas o número de céulas é muito inferior quando comparados à segunda fase, conforme visto nas células mais claras da Figura 5.4, e resultam em subutilização do *hardware* e em uma queda significativa de desempenho.

5.3 Estratégia de Granularidade Fina

5.3.1 Visão Geral

Na estratégia de granularidade fina, optamos por aumentar o paralelismo potencial, utilizando múltiplos blocos e múltiplas *threads* dentro de cada bloco. Nessa estratégia, várias *threads* calculam uma célula da matriz de programação dinâmica. A Figura 5.6 apresenta uma visão geral dessa estratégia.

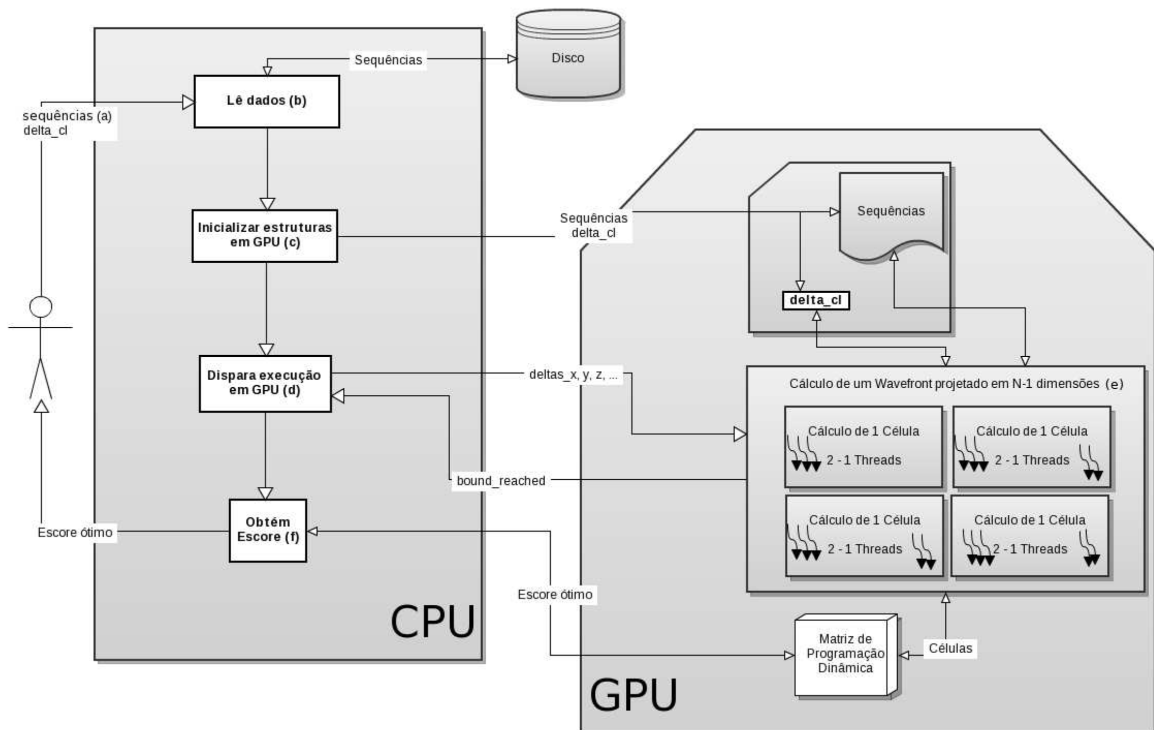

Figura 5.6: Resumo da estratégia de granularidade fina.

Os mesmos parâmetros de entrada da estratégia grossa são utilizados, ou seja, o usuário solicita a execução informando as sequências e o limite do Carrillo-Lipman (delta_cl) (Figura 5.6a). O programa, então, executando em CPU, lê as sequências (Figura 5.6b) e em seguida, as sequências e o delta_cl são copiados para GPU (Figura 5.6c).

O cálculo da matriz de programação dinâmica é então feito de forma iterativa como descrito a seguir. Os *wavefronts* multidimensionais são projetados e, para cada um, um kernel em GPU é invocado (Figura 5.6d), tendo os deltas do espaço de busca como parâmetro.

O cálculo de cada célula propriamente dito é feito por $2^n - 1$ *threads* em GPU, onde cada *thread* calcula o escore de um vizinho de uma célula da matriz de programação dinâmica (Figura 5.6e). O número de blocos varia de acordo com qual *wavefront* está sendo calculado. Para calcular o escore, células da matriz de programação dinâmica são lidas. Durante este processo, as *threads* no interior de um mesmo bloco comparam seus valores e o escore é escolhido ao se comparar todas. Novamente, uma ou mais *threads* podem concluir que o limite Carrillo-Lipman foi atingido. Se esse for o caso, a variável bound_reached é setada para 1 e é usada pela CPU para descartar a execução de *wavefronts* que não são necessários. Ao final do cálculo da matriz de programação dinâmica, o escore ótimo é copiado para CPU (Figura 5.6f). Finalmente, o escore ótimo é retornado para o usuário.

5.3.2 Cálculo do wavefront um bloco por célula

A Figura 5.7 mostra as modificações feitas no algoritmo exato para a estratégia fina. Nela, um bloco de *threads* é responsável por calcular o valor de uma célula do espaço de busca.

Assim como a estratégia de granularidade grossa (Seção 5.2), os *loops "for"* mais externos são paralelizados, destacados pelo retângulo das linhas 1 a 3. O restante do código é executado em um *kernel* em GPU, destacado pelo retângulo das linhas 4 a 30. A estratégia de granularidade fina paraleliza também as consultas aos vizinhos da célula (retângulo tracejado das linhas 22 a 28). As variáveis d_1 a d_7 representam os possíveis valores que a célula que está sendo calculada pode assumir, sendo que o menor deles deve ser escolhido. Mas, esses valores são independentes, podendo ser calculados em paralelo. Além disso, o número de variáveis é exponencial em relação ao número de sequências, pois cada célula possui $2^n - 1$ vizinhos.

$threadIdx.x$	$neigh_z$	$neigh_y$	$neigh_x$	Fórmula de custo
1	0	0	1	$D(x-1, y, z) + 2 \times GAP$
2	0	1	0	$D(x, y-1, z) + 2 \times GAP$
3	0	1	1	$D(x-1, y-1, z) + c_{xy} + GAP$
4	1	0	0	$D(x, y, z-1) + 2 \times GAP$
5	1	0	1	$D(x-1, y, z-1) + c_{xz} + GAP$
6	1	1	0	$D(x, y-1, z-1) + c_{yz} + GAP$
7	1	1	1	$D(x-1, y-1, z-1) + c_{xy} + c_{xz} + c_{yz}$

Tabela 5.4: Valores das variáveis neigh e a fórmula de custo utilizada.

Nesta abordagem, os valores de *match* e *mistmatch* (calculados nas linhas 4 a 20) são armazenados em variáveis locais chamadas c_{xy}, c_{xz} e c_{yz}. É criado um bloco com 7 *threads*, que possuem um identificador $threadIdx.x$ de 1 a 7. Seja $D(x, y, z)$ a função para se obter o custo do vizinho (x, y, z). Para escolher qual a dimensão x do vizinho que a *thread* deve acessar, é feita uma operação bit a bit com o último bit do $threadIdx.x$. Desta forma, as *threads* $1, 3, 5$ e 7 visitam os vizinhos $x - 1$ e as *threads* $2, 4, 6$ visitam o vizinho x. As outras dimensões y e z são escolhidas de forma semelhante, mas utilizando o segundo e primeiro bit do $threadIdx.x$, respectivamente. Os índices dos vizinhos acessados por cada *thread* (variável neigh) estão resumidos na Tabela 5.4.

Os índices são utilizados para calcular qual célula $D(x, y, z)$ deve ser acessada, mas para se obter o valor final deve-se somar os custos de GAP e, em alguns casos, o valor de

```
1: for i = 1 → n1 do
2:   for j = 1 → n2 do
3:     for k = 1 → n3 do
4:       if S1(i) = S2(j) then
5:         c_ij = smatch
6:       else
7:         c_ij = smis
8:       end if
9:
10:      if S1(i) = S3(k) then
11:        c_ik = smatch
12:      else
13:        c_ik = smis
14:      end if
15:
16:      if S2(j) = S3(k) then
17:        c_jk = smatch
18:      else
19:        c_jk = smis
20:      end if
21:
22:      d1 = D(i − 1, j − 1, k − 1) + c_ij + c_ik + c_jk
23:      d2 = D(i − 1, j − 1, k) + c_ij + sspace
24:      d3 = D(i − 1, j, k − 1) + c_ik + sspace
25:      d4 = D(i, j − 1, k − 1) + c_jk + sspace
26:      d5 = D(i − 1, j, k) + 2 * sspace
27:      d6 = D(i, j − 1, k) + 2 * sspace
28:      d7 = D(i, j, k − 1) + 2 * sspace
29:
30:      D(i, j, k) = Min[d1, d2, d3, d4, d5, d6, d7]
31:    end for
32:  end for
33: end for
```

Figura 5.7: Alterações no algoritmo para a estratégia fina.

```
__device__ __host__ void init_local_scores(local_scores, c_xy, c_xz, c_yz)
{
  local_scores[7][3] = {
        {GAP, GAP, 0},
        {GAP, GAP, 0},
        {c_xy, GAP, 0},
        {GAP, GAP, 0},
        {c_xz, GAP, 0},
        {c_yz, GAP, 0},
        {c_xy, c_xz, c_yz},
  };
}
```

Figura 5.8: Inicialização do array *local_scores*.

custo de *match* ou *mismatch* apropriado. Esse custo pode ser visto na quinta coluna da Tabela 5.4, onde todos os vizinhos possuem um custo a ser somado no retorno da função $D(x, y, z)$. Para o caso de três sequências, é utilizado um *array* auxiliar, *local_scores*[7][3], que pode ser generalizado. Esse *array* é inicializado na função *init_local_scores* com os valores que devem ser somados a $D(x, y, z)$ de acordo com o *threadIdx.x* (Figura 5.8).

Sendo assim, para o caso de 3 sequências, o valor *cell_score* calculado pela *thread* pode ser obtido da seguinte maneira:

$$\begin{aligned} cell_score = D(x &- neigh_x, y - neigh_y, z - neigh_z) + \\ &local_scores[threadIdx.x][0] + \\ &local_scores[threadIdx.x][1] + \\ &local_scores[threadIdx.x][2]; \end{aligned}$$

Nessa fórmula, pode-se ver que apenas um vizinho é consultado por cada *thread*. Seu valor é obtido e armazenado posteriormente em um array interno de resultados (*escores*). Depois disso, as *threads* devem cooperar para decidir o resultado final da célula.

5.3.3 Escolha do Menor Valor

Em um bloco de *threads*, cada uma das $2^n - 1$ *threads* possui um valor, mas o valor da célula que está sendo calculado deve ser o menor entre eles. Para se obter este menor valor, cada thread armazena o seu valor em um *array*:

$$escores[threadIdx.x] = cell_score$$

Após isto, são feitas $log(n)$ iterações, para se escolher o menor valor. É utilizada uma comparação em árvore binária conforme o ilustrado na Figura 5.9. O algoritmo começa a executar das folhas e cada iteração representa um nível da árvore.

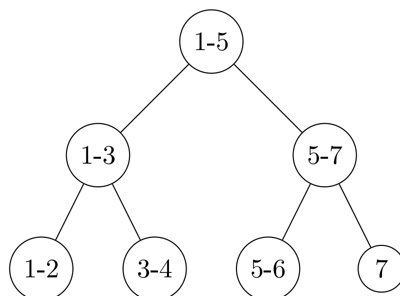

Figura 5.9: Pesquisa do menor valor

Cada nó da árvore representa uma comparação feita por uma *thread*. Além disso, após cada comparação, o menor valor é armazenado na menor das posições comparadas. Por exemplo, 1-5 quer dizer que o valor da primeira célula do array *escores*[] foi comparado com o quinto escore e o menor desses valores foi armazenado na primeira posição.

Para o caso de três sequências, cada bloco de *threads* possui 7 *threads* que calcularam os valores de cada um dos vizinhos. Quando inicia-se a escolha do menor valor, apenas metade dessas *threads* são necessárias para fazer as comparações. Após isso, o número de *threads* necessárias cai pela metade, onde a comparação dos *arrays* é feitas entre outras células e esse processo continua até chegar na raiz da árvore.

Ao se calcular o valor da raiz, o menor valor se encontra na primeira posição do *array* *escores*. Este é o menor valor de todos os vizinhos e é armazenado como o valor final da célula calculada pelo bloco (retângulo tracejado, linha 30, da Figura 5.7).

5.3.4 Percorrendo o espaço de busca

Para a estratégia de granularidade fina, o formato utilizado para se percorrer o espaço de busca foi alterado e agora não segue o formato de um *wavefront*. Da mesma maneira que a estratégia de MSA exato em FPGA (Seção 4.2.2), optamos aqui por percorrer todas as células em uma dimensão, depois todas as células em outra dimensão e assim por diante.

Para simplificar a explicação, inicialmente o algoritmo que utiliza apenas um bloco de *threads* será explicado, e logo em seguida explicaremos como foram adicionados múltiplos blocos.

Utilização de um bloco de threads

Da mesma maneira que a estratégia de granularidade grossa, são utilizadas variáveis delta, que armazenam o deslocamento em relação à origem para percorrer todo o espaço de busca. As dimensões x, y, z, são relacionadas à primeira, segunda e terceira sequência, respectivamente. Na primeira iteração do algoritmo, a célula $(0, 0, 0)$ é calculada. As variáveis delta_x, delta_y e delta_z possuem valor 0, nesta etapa. Então, delta_x é aumentado em 1 e o *kernel* é invocado, até que delta_x tenha o tamanho da primeira sequência. Depois, delta_x é zerado e delta_y é aumentado em 1 e inicia-se o ciclo novamente. Quando delta_y atinge o tamanho da segunda sequência e delta_x atinge o tamanho da primeira sequência, delta_z é acrescido de 1. O programa termina quando

delta_z, delta_y e delta_x possuem o tamanho das sequências. Ao término, o escore do alinhamento ótimo está armazenado na última célula calculada.

Utilização de vários blocos de threads

Na abordagem com mais de um bloco, a variável delta_y foi eliminada do loop. Apenas delta_x e delta_z são incrementadas. Mas, para garantir que todas as células necessárias sejam calculadas, delta_y é obtido dentro do *kernel* através do identificador do número do bloco corrente, *blockIdx.x*. Assim, ao iniciar o *kernel*, os valores delta_x e delta_y são calculados da seguinte maneira:

$delta_x = delta_x - blockIdx.x$
$delta_y = blockIdx.x$

Também é necessário verificar se delta_x ou delta_y excedem o tamanho da primeira ou da segunda sequência, caso o *kernel* seja invocado com uma quantidade muito grande de blocos.

Com esta alteração, estamos limitando o tamanho máximo que a segunda sequência pode ter. Em outras palavras, o *hardware* deve ser capaz de criar uma quantidade de blocos que seja maior que o tamanho desta segunda sequência. Porém, é bem realista assumirmos isto, pois o limite para o número de blocos é atualmente 65.535 [50], um número muito maior que o esperado para um alinhamento múltiplo exato.

Externamente ao *kernel*, o loop foi alterado para possuir apenas delta_x e delta_z. O valor de delta_y é calculado pelo índice do bloco dentro do *kernel*, sendo invocados delta_x +1 blocos a cada execução. O valor delta_x precisa variar até o comprimento da primeira sequência adicionado ao comprimento da segunda sequência.

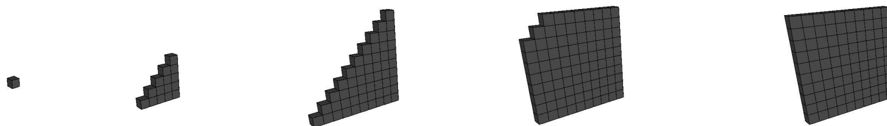

Figura 5.10: Primeiros passos da estratégia de granularidade fina.

As Figuras 5.10 e 5.11 ilustram como o espaço de busca é percorrido na estratégia de granularidade fina multibloco. Inicialmente calcula-se apenas uma célula, neste momento, delta_x e delta_z são zero. Nas próximas iterações, delta_x é aumentado e o valor de delta_z é mantido em zero. Esse processo é repetido até que se calculem todas as células com z igual a zero (Figura 5.10). Neste momento delta_z é incrementado em 1 e delta_x é zerado, e o processo se repete até que se calculem todas as células do espaço de busca (Figura 5.11).

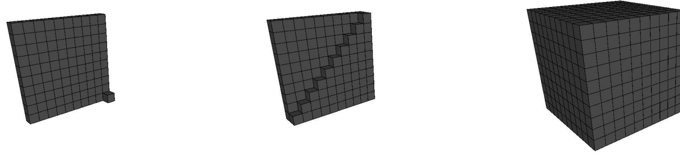

Figura 5.11: Últimos passos da estratégia de granularidade fina.

5.3.5 Utilizando o limite de Carrillo-Lipman

Em relação à estratégia apresentada na Seção 5.2.4, o descarte de células foi muito pouco alterado.

Existe uma variável global delta_cl (δ), que limita o valor que uma célula pode assumir. O valor *bound_unreached* é inicializado como 0 antes de cada execução e, caso alguma célula precise que os vizinhos sejam calculados, o seu valor é alterado para 1. Caso nenhuma célula influencie no cálculo do alinhamento ótimo, as células seguintes são inicializadas com o maior valor possível, sem serem calculadas.

Diferente da abordagem anterior, onde um bloco de células calculava um *wavefront*, nesta abordagem um bloco de *threads* calcula o valor de uma célula e vários blocos podem ser calculados em paralelo pois respeitam a dependência de um *wavefront* (Seção 2.4).

O limite de Carrillo-Lipman é utilizado da seguinte forma. Se todas as células calculadas dentro do bloco não contribuírem para o alinhamento múltiplo ótimo, as próximas células não podem contribuir para encontrar o alinhamento múltiplo exato. Desta forma, os valores de delta_x e delta_z são zerados e o espaço de busca é reduzido.

5.3.6 Estruturas em Memória

- **Sequências:** as sequências são armazenadas na memória constante. Cada aminoácido das sequências ocupa um *byte* em memória. Também são armazenados três contadores com o tamanho das sequências para controlar o acesso.
- **Matriz de programação dinâmica:** A matriz de programação dinâmica é uma matriz n^k, onde n é o número de sequências e k é o número de sequências. Nessa estratégia, é armazenado todo o espaço de busca. Por seu tamanho, a matriz de programação dinâmica é armazenada em memória global.
- **Deltas:** Para que o *kernel* identifique qual o *wavefront* está sendo calculado, ele recebe como argumento duas variáveis: delta_x, e delta_z, que significam os desvios em relação à primeira célula calculada pelo algoritmo. Essas variáveis são armazenadas em memória local.
- **Escores dos vizinhos e menor_escore:** Em um alinhamento de três sequências, cada célula do espaço de busca possui sete vizinhos que são consultados para determinar o escore da célula que está sendo calculada. Cada *thread* armazena o valor em um *array* de memória compartilhada e ao final da execução do *kernel*, o valor escolhido para a célula corrente é o menor valor dentre os calculados.

- **Limite do Carrillo-Lipman:** Duas variáveis são utilizadas para descartar células que não precisem ser calculadas. A primeira é recebida pelo algoritmo como parâmetro, chamada de delta_cl (δ). A segunda variável, *bound_unreached*, possui um valor 0 antes da execução do *kernel*, e tem seu valor alterado caso alguma célula do wavefront não possa ser descartada. Esse valores são armazenados em memória global.
- c_{xy}, c_{xz}, c_{yz}**:** Custos de *match* e *mismatch*, armazenados em memória local.
- **neighs:** K variáveis $neigh_x$, $neigh_y$, e $neigh_z$, ..., ilustradas na Tabela 5.4. São aplicadas às variáveis deltas para se calcular qual vizinho a *thread* corrente irá visitar, armazenados em memória local.
- *local_escores*: *array* que possui os valores de *GAP*, e/ou *match/mismatch* das sequências, que deve ser aplicado ao escore final do vizinho que a *thread* era responsável por calcular. Armazenado em memória local.

Estrutura	**Memória**	**Tipo**
Sequências	$k \times n$ *bytes*	Constante
Matriz de programação dinâmica	n^k *integer*	Global
Deltas	2 *integer*	Local
Escores dos vizinhos	$2^k - 1$ *integer*	Compartilhada
Limite do Carrillo-Lipman	2 *integer*	Global
c_{xy}, c_{xz}, c_{yz}	k *integer*	Local
Neighs	k *integer*	Local
local_scores	$k \times (2^k - 1)$ *integer*	Local

Tabela 5.5: Estruturas e memória utilizada para k sequências de tamanho n na estratégia de granularidade fina.

A Tabela 5.5 apresenta o uso de memórias da GPU pela estratégia de granularidade fina.

5.3.7 Pseudocódigo em GPU

A figura 5.12 apresenta o pseudocódigo executado em GPU pela estratégia de granularidade fina. Os parâmetros recebidos são os mesmos que a estratégia de granularidade grossa.

Inicialmente (linhas 5 a 7), são definidos os valores x, y e z da célula de programação dinâmica que será calculada. Esses valores são baseado no deslocamento em relação à origem, (*delta_x*, *delta_y* e *delta_z*), recebidos como argumento da CPU, e o blockIdx.x que é o identificador do bloco de cada *thread*. Desta forma, as *threads* que pertencem ao mesmo bloco calculam a mesma célula.

Os valores *neigh_x*, *neigh_y* e *neigh_z* são decididos nas linhas 9 a 11. Essas variáveis assumem valor 0 ou 1, de acordo com a Tabela 5.4, são diferentes para todas as *threads* e serão utilizados para decidir qual vizinho a *thread* atual deve visitar.

Os valores de custo de *match* e *mismatch* das sequências é calculado nas linhas 13 a 15 e são armazenados em três variáveis locais. Esses custos são utilizados no procedimento *init_local_scores*, que irá armazenar quais custos de *match*, *mismatch* ou *gap* devem ser aplicados ao escore do vizinho que a *thread* está visitando (linhas 17 e 18).

Os escores são armazenados em um array arr, responsável por armazenar todos os escores dos vizinhos. Depois que todas as *threads* visitaram os vizinhos e calcularam $arr[threadIdx.x]$, é necessário aguardar que todas as outras *threads* façam o seu cálculo, por isso é utilizada uma chamada de sincronia $__syncthreads()$.

Nas linhas 21 a 30 é escolhido o menor valor entre as sequências, seguindo uma busca binária como mostrada na Figura 5.9. Mais uma vez a chamada de sincronia entre as *threads* na linha 29 é necessária, pois uma *thread* só pode executar um passo da comparação entre as células depois que todas as outras executaram a comparação anterior.

Para o caso de três sequências, o laço é executado três vezes e ao final do processo, está presente na posição arr[0]. Esse valor é armazenado como o menor valor que a célula pode assumir e é armazenado na matriz de programação dinâmica. Nas linhas 33 e 34 o valor da célula é comparado com o limite Carrillo-Lipman e o valor de *bound_unreached* é alterado caso a célula calculada seja inferior ao limite δ.

5.3.8 Análise da Abordagem

Com a abordagem apresentada na Seção 5.2, observamos que percorrer o espaço de busca em forma de um *wavefront* pode levar a uma diferença grande entre o número de células que são calculadas em paralelo durante as diferentes fases do algoritmo.

Observando a queda no número células que podem ser calculadas em paralelo, foi aumentado o paralelismo, calculando-se em paralelo o valor de cada *threads*. Isso ocasiona uma melhora no desempenho, que consideramos a principal vantagem desta abordagem.

Foi observada a queda de paralelismo com o uso da estratégia grossa. Como pode ser visto na Figura 5.4, o número de células que podem ser calculadas em paralelo na fase 3 é bem inferior aos da fase 2 dessa estratégia. Na abordagem de granularidade fina, todo o espaço de busca é percorrido baseado apenas em diagonais do cubo. Desta forma, não há uma queda abrupta da quantidade de células que podem ser calculadas em paralelo.

Percorrer o espaço de busca dividindo o espaço de busca em janelas de processamento é uma abordagem conhecida na literatura que teve bons resultados (Seção 4.2.2). Para adaptar essa estratégia à arquiteturas massivamente paralelas, observou-se que é possível, em alguns casos, ter uma quantidade menor de *threads* do que o máximo permitido pela arquitetura. Portanto, foi decidido elevar o paralelismo e decidiu-se paralelizar, também, a consulta aos vizinhos, optando-se por fazer um bloco de *threads* calcular o valor de uma célula. Desta forma, cada bloco possuirá $2^n - 1$ *threads*, onde n é o número de sequências. Além disso, vários blocos serão utilizados ao mesmo tempo, atingindo alto paralelismo potencial.

Porém, esta abordagem possui uma desvantagem, pois requer uma gerência de memória mais complexa, já que não existe uma quantidade fixa de iterações entre o momento em que uma célula é calculada e o momento em que o espaço por ela utilizado possa ser liberado. Outra desvantagem é o grande uso do espaço em memória, necessário para armazenar toda a matriz, que configura-se como o seu principal fator limitante.

```
__global__ void msa (char seq1[ ], char seq2[ ], char seq3[ ], int matriz_pd[ ][ ][ ], int
delta_x, int delta_y, int delta_z, int δ, int *bound_unreached)
 1: __shared__ int arr[8];
 2: int c_xy, c_xz, c_yz;
 3: local_scores[7][3];
 4:
 5: int x = delta_x − blockIdx.x;
 6: int y = delta_y + blockIdx.x;
 7: int z = delta_z;
 8:
 9: int neigh_x = x − ((threadIdx.x & 1));
10: int neigh_y = y − ((threadIdx.x & 2) >> 1);
11: int neigh_z = z − ((threadIdx.x & 4) >> 2);
12:
13: c_xy = cost(seq1, seq2, x, y);
14: c_yz = cost(seq2, seq3, y, z);
15: c_xz = cost(seq1, seq3, x, z);
16:
17: init_local_scores(local_scores, c_xy, c_xz, c_yz);
18: cell_score = D(x −neigh_x, y − neigh_y, z − neigh_z)+
                      local_scores[threadIdx.x][0]+
                      local_scores[threadIdx.x][1]+
                      local_scores[threadIdx.x][2];

19: arr[threadIdx.x] = cell_score;
20: __syncthreads();
21: for (int i = 2; i <= 8; i = i * 2) do
22:    if (threadIdx.x % i) ! = 0 then
23:       elem1 = threadIdx.x;
24:       elem2 = threadIdx.x + (i/2)
25:       if (arr[elem1] > arr[elem2]) then
26:          arr[elem1] = arr[elem2];
27:       end if
28:    end if
29:    __syncthreads();
30: end for
31: matriz_pd[x][y][z] = arr[0];
32:
33: if (matriz_pd[x][y][z] < δ) then
34:    *bound_unreached = 1;
35: end if
```

Figura 5.12: Kernel msa executado em GPU pela estratégia fina.

Capítulo 6

Resultados experimentais

Neste capítulo, serão descritos os testes realizados com a nossa implementação do Carrillo-Lipman em GPU (Capítulo 5). Inicilamente é feita a comparação das abordagens de granularidade fina (Seção 5.3) e granularidade grossa (Seção 5.2). Em seguida, apresentamos a comparação entre os tempos de execução do programa MSA 2.0 em CPU (Seção 2.5.4) e a nossa implementação do Carrillo-Lipman em GPU (Capítulo 5). Ao final do capítulo, são apresentados e discutidos *overhead* de invocação de *kernels* pelo nosso programa em GPU.

6.1 Ambiente de testes

As estratégias de granularidade grossa e fina para execução do Carrillo-Lipman em GPU (Capítulo 5) foram implementadas em CUDA C e testadas em GPU. Os testes foram realizados em uma única estação de trabalho dedicada para experimentos em CUDA e disponível no LAboratório de sistemas Integrados e COncorrentes (LAICO) da Universidade de Brasília. A Tabela 6.1 apresenta a configuração da máquina utilizada nos testes, incluindo os dados da CPU, da GPU, do Sistema Operacional, do CUDA *Toolkit* e dos compiladores utilizados nos testes.

Máquina (CPU)	GPU	Software
Processador Intel Core i5, 3.30Ghz 64KB de cache L1 6GB RAM 1 TB Disco	GeForce GTX 580 512 CUDA Cores 1.544GHz 1535 MB RAM	Sistema Operacional Ubuntu 11.10 Linux Kernel 3.0.0-15 CUDA Toolkit 4.1.21 nvcc release 4.1 gcc 4.5.4

Tabela 6.1: Configuração da estação de trabalho (CPU, GPU, Software) utilizada nos testes.

Para comparação das sequências, foi utilizada a matriz PAM 250 (Tabela 2.1) para obtenção dos custos de *match*/*mismatch* e o valor 12 para o *gap*.

6.2 Sequências selecionadas

Diversas sequências reais e algumas sequências sintéticas foram selecionadas para os experimentos. Essas sequências são mostradas na Tabela 6.2.

Na primeira coluna da tabela é mostrado o nome que utilizamos nesta dissertação para cada conjunto de sequências. A segunda coluna representa a base genômica da qual as sequências foram retiradas. Como pode ser visto, a maioria dos conjuntos de sequências foram retirados da base PFAM [47], sendo que duas sequências foram obtidas da base Balibase 2 [3]. Os quatro primeiros conjuntos de sequências sintéticas foram gerados de maneira que o MSA 2.0 (Seção 2.5.4) não fosse capaz de obter o alinhamento múltiplo com os parâmetros *default*. O último conjunto de sequências sintéticas contém 3 sequências praticamente iguais, com somente 1 caracter diferente em cada sequência. O conjunto de sequências *synthetic_easy* foi gerado com a inserção em uma sequência de caracteres formando um bloco distinto composto por 90 caracteres. O conjunto *synthetic_medium* foi gerado com a inserção de duas cópias desse mesmo bloco em uma das sequências. O conjunto *synthetic_hard1* foi gerado a partir do conjunto *synthetic_medium* inserindo-se adicionalmente dois blocos iguais de 30 caracteres ao final de uma das sequências. Por fim, o conjunto *synthetic_hard2* foi obtido a partir do último, inserindo-se um bloco de 30 caracteres ao final de uma das sequências. As sequências sintéticas utilizadas nos testes podem ser encontradas no Anexo I.

Finalmente, na última coluna da Tabela 6.2 são mostrados os tamanhos de cada sequência.

Nome	Referência	Seq	Tamanho
Seq15	PFAM	PF10550	15 15 14
Seq22	PFAM	PF08095	22 22 22
Seq35	PFAM	PF08117	34 35 35
Seq37	PFAM	PF08119	36 37 37
Seq41	PFAM	PF11045	41 41 41
Seq42	PFAM	PF03855	41 44 42
Seq44	PFAM	PF11406	44 44 21
Seq59	PFAM	PF08184	59 59 59
Seq71	PFAM	PF09629	68 71 71
Seq94	PFAM	PF07590	94 94 93
Seq106.1	PFAM	PF09241	106 106 106
Seq106.2	PFAM	PF11516	106 106 106
Seq107	PFAM	PF10353	107 107 107
Seq108	PFAM	PF09645	95 69 108
Seq110	PFAM	PF11513	106 111 110
Seq122	PFAM	PF06453	122 122 122
Seq143	PFAM	PF09155	143 143 143
Seq162	PFAM	PF03426	158 158 162
Seq164	PFAM	PF03866	163 164 164
Seq373	Balibase2	1pedA	350 326 373
Seq446	Balibase2	1ad3	423 441 446
Seq416	Sintética	synthetic_easy	231 416 363
Seq447.1	Sintética	synthetic_medium	231 447 363
Seq447.2	Sintética	synthetic_hard1	231 447 423
Seq453	Sintética	synthetic_hard2	231 446 453
Seq585	Sintética	synthetic_equal	585 585 585

Tabela 6.2: Sequências utilizadas nos testes.

6.3 Dados Experimentais e Análise dos resultados

6.3.1 Tempo de execução total das estratégias

Neste primeiro teste, foi realizada a comparação entre as estratégias apresentadas nas Seções 5.2 e 5.3. A estratégia de granularidade grossa, que possui apenas um bloco (Seção 5.2), é comparada com a estratégia de granularidade fina com apenas um bloco (Monobloco) e com múltiplos blocos (Multibloco) (Seção 5.3).

Nome	Granularidade Grossa		Granularidade fina Monobloco		Granularidade Fina Multibloco	
Seq15	1,09s	990 *threads*	0,12s	7 *threads*	0,05s	7 a 105 *threads*
Seq22	1,34s	990 *threads*	0,27s	7 *threads*	0,07s	7 a 154 *threads*
Seq42	3,61s	990 *threads*	1,68s	7 *threads*	0,10s	7 a 308 *threads*
Seq59	4,60s	990 *threads*	4,280s	7 *threads*	0,14s	7 a 413 *threads*
Seq110	16,51s	990 *threads*	27,25s	7 *threads*	0,35s	7 a 777 *threads*
Seq122	26,05s	990 *threads*	37,71s	7 *threads*	0,46s	7 a 854 *threads*
Seq143	29,84s	990 *threads*	1m0,39s	7 *threads*	0,47s	7 a 1001 *threads*
Seq162	41,07s	990 *threads*	1m25,20s	7 *threads*	0,74s	7 a 1106 *threads*
Seq373	5m33,99s	990 *threads*	14m22,95s	7 *threads*	4,04s	7 a 2282 *threads*
Seq416	4m16,52s	990 *threads*	16m11,83s	7 *threads*	3,69s	7 a 2912 *threads*
Seq446	8m55,44s	990 *threads*	29m30,42s	7 *threads*	6,62s	7 a 3087 *threads*
Seq447.2	5m10,95s	990 *threads*	> 30 minutos	7 *threads*	6,49s	7 a 3129 *threads*

Tabela 6.3: Comparação das abordagens

Conforme apresentado na Tabela 6.3, em sequências pequenas, a abordagem de granularidade fina apresenta tempo de execução melhor, ou equivalente, ao da abordagem de granularidade grossa, mesmo com uma quantidade muito inferior de *threads*.

Com o aumento do tamanho de sequências, o número de *threads* em paralelo começa a influenciar e o baixo número de *threads* da estratégia de granularidade fina monobloco começa a apresentar impacto negativo no desempenho. A quantidade de *threads* foi fixada na estratégia fina apenas para efeitos comparativos. Pode-se ver que com o uso de vários blocos na estratégia de granularidade fina, utilizando um maior número de *threads*, atinge-se o melhor desempenho em todos os casos.

Para sequências pequenas, a estratégia de granularidade grossa possui um tempo de inicialização maior pois é necessário carregar em memória um *array* pré-calculado, e isso impacta no desempenho.

Como a estratégia fina com múltiplos blocos possui claramente um desempenho superior ao das outras estratégias, ela foi a escolhida para ser comparada ao programa MSA 2.0 (Seção 2.5.4), executado em CPU.

6.3.2 Comparação com o MSA 2.0

Neste teste a estratégia de granularidade fina com múltiplos blocos é comparada com o MSA 2.0, que se executa em CPU (Tabela 6.4). O programa MSA 2.0 foi obtido do site www.ncbi.nim.nih/gov/CBBresearch/Schaffer/msa.html e os resultados obtidos estão na Tabela 6.4.

A primeira coluna da Tabela 6.4 representa o nome das sequências. A segunda coluna (δ) representa o valor que foi passado ao programa Carrillo-Lipman em GPU para que ele possa descartar células do espaço de busca, conforme explicado na Seção 5.3.5. A terceira coluna apresenta o tempo de execução do programa MSA 2.0. A quarta coluna apresenta

Nome	δ	MSA 2.0 (s)	Cuda MSA (FSS) (s)	Cuda MSA (CL) (s)
Seq15	15	0,001	0,051	0,051
Seq22	15	0,001	0,068	0,056
Seq35	15	0,002	0,088	0,081
Seq37	15	0,002	0,088	0,066
Seq41	15	0,002	0,098	0,083
Seq42	15	0,002	0,102	0,096
Seq44	15	0,002	0,085	0,069
Seq59	15	0,002	0,140	0,126
Seq71	15	0,003	0,176	0,156
Seq94	23	0,005	0,298	0,253
Seq106.1	15	0,005	0,333	0,292
Seq106.2	15	0,005	0,366	0,226
Seq107	15	0,005	0,339	0,303
Seq108	24	0,005	0,273	0,241
Seq110	15	0,005	0,353	0,313
Seq122	15	0,006	0,461	0,435
Seq143	15	0,007	0,473	0,423
Seq162	15	0,010	0,743	0,650
Seq164	15	0,010	0,904	0,815
Seq373	137	0,180	4,041	3,324
Seq416	502	1,652	3,693	2,611
Seq446	29	0,140	6,624	6,203
Seq447.1	185	23,507	3,837	3,011
Seq447.2	484	28,711	6,478	4,764
Seq453	387	31,078	6,948	3,612
Seq585	15	0,619	14,976	7,345

Tabela 6.4: Tempos de execução do MSA 2.0 e do Carrillo-Lipman em GPU

o tempo de execução de uma procura pelo alinhamento ótimo em todo espaço de busca (*Full Space Search*), executado em GPU. A última coluna apresenta o tempo de execução em GPU com o espaço de busca reduzido pelo parâmetro δ.

Observa-se pela tabela que, para as sequências de Seq15 a Seq 446 e Seq 585, o MSA 2.0 é capaz de obter muito rapidamente o alinhamento múltiplo ótimo com tempos inferiores aos tempos de execução em GPU. Isso ocorre porque essas sequências possuem uma similaridade muito alta, o que leva ao grande descarte de células, aumentando a eficiência do programa em MSA 2.0 em CPU. Porém, para as sequências Seq416, Seq447.1, Seq447.2 e Seq453, que possuem similaridade média ou baixa, o algoritmo MSA 2.0 não é capaz de obter o alinhamento na primeira execução com os parâmetros *default*. Ao se executar o programa para estas sequências, é apresentada a mensagem de que não é possível obter o alinhamento com os parâmetros passados. Então, foi necessário aumentar a variável δ, passando como argumento limites maiores. Nos nossos testes, foram fornecidos como parâmetros os valores 100, 100 e 1000. Na Tabela 6.4, o tempo da primeira execução é desconsiderado e são apresentados apenas os tempos da segunda execução do MSA 2.0 para essas sequências.

Para sequências de menor similaridade, nota-se que há um aumento do valor da variável δ. Consequentemente, o número de células descartadas diminui e o programa em CPU precisa de mais tempo para encontrar o alinhamento. Para estes casos, a abordagem em GPU apresenta *speedup* de 7,8x para as sequências de similaridade média (Seq447.1). Para as sequências de baixa similaridade (Seq447.2 e Seq453), os *speedups* obtidos foram 6,02x e 8,60x, respectivamente.

As estratégias propostas em GPU descartam células baseadas em diagonais do cubo do espaço de busca utilizando o limite Carrillo-Lipman. Como várias células são processadas

em paralelo, caso apenas uma das células contribua para o alinhamento ótimo e o restante das células não contribua, precisa-se continuar a busca e outras células que não contribuem para o alinhamento ótimo serão calculadas. Caso nenhum argumento seja passado ao programa em GPU, é realizada uma "*Full Space Search*". Neste caso, o alinhamento ótimo é obtido calculando-se todas as células da matriz de programação dinâmica. Sendo assim, o tempo para se obter o alinhamento é maior, conforme pode ser visto comparando-se a quarta e quinta coluna da Tabela 6.4.

Com base nas sequências de testes utilizadas, verificamos que o programa MSA 2.0 encontra rapidamente alinhamentos múltiplos de 3 sequências quando as sequências são similares. No entanto, para sequências com baixa ou média similaridade, o tempo de execução do MSA 2.0 cresce muito. Ao se obter o parâmetro δ do MSA 2.0, é impresso na tela um alinhamento heurístico que é utilizado para calcular o Carrillo-Lipman *bound* e este alinhamento pode ser utilizado para medir a similaridade entre as sequências.

Para os casos de sequências de baixa similaridade, nosso Carrillo-Lipman em GPU é capaz de reduzir bastante o tempo de execução. Sendo assim, acreditamos que o MSA 2.0 deve ser utilizado para sequências similares e o Carrillo-Lipman em GPU deve ser usado para os outros casos.

6.3.3 Tempos de invocação de kernels em GPU

Por observar, na Tabela 6.4, que o tempo de inicialização da GPU pode ser um fator que atrapalha o desempenho, decidimos, neste teste, medir o tempo gasto para invocar *kernels* em GPU.

A Tabela 6.5 mostra os tempos de invocação de *kernels* em GPU para os conjuntos de sequências. Durante o experimento, o código que existe dentro do *kernel* do Carrillo-Lipman em GPU é comentado, ou seja, o *kernel* não executa código nenhum. Ele apenas é invocado com os parâmetros de entrada e o restante do algoritmo em CPU executa normalmente, percorrendo o espaço de busca. Sendo assim, são feitas várias invocações nulas de *kernel*.

Desta forma, medimos o tempo que o código gasta ao apenas invocar todos os *kernels* necessários para obter o alinhamento ótimo na GPU, inicializando as variáveis necessárias e lendo o resultado da variável *bound_unreached*. O tempo total representa o tempo gasto em um *full space search* e a última coluna mostra o percentual de tempo de execução dos *kernels* nulos em relação ao tempo total.

Nome	Tempo de invocação (s)	Tempo total (s)	%
Seq15	0,050	0,051	98,03 %
Seq22	0,061	0,068	89,70 %
Seq42	0,080	0,102	78,43 %
Seq59	0,120	0,140	85,71 %
Seq110	0,292	0,353	82,71 %
Seq122	0,358	0,461	77,65 %
Seq143	0,397	0,473	83,93 %
Seq162	0,574	0,743	77,25 %
Seq373	2,730	4,041	67,55 %
Seq416	2,515	3,693	68,10 %
Seq446	4,220	6,624	63,70 %
Seq447.2	3,145	6,478	48,54 %

Tabela 6.5: *Overhead* de invocação de *kernels* em GPU.

Pode-se observar que o tempo de invocação representa um percentual considerável nas sequências menores, de 78,43% a 98,03%. Esse resultado é esperado, pois as GPUs possuem um *delay* fixo de tempo para inicializar os programas que serão executados [43]. Desta forma, o tempo fixo é grande para sequências pequenas, mas à medida que o tamanho das sequências aumenta, o tempo gasto na execução do *kernel* é maior, tornando o percentual de invocação menor.

Para sequências de baixa similaridade (Seq447.2), o parâmetro δ é alto (Tabela 6.4), indicando que um pedaço significativo do espaço de busca é percorrido. Nesse caso, há mais processamento em GPU e o custo de invocação dos *kernels* fica amortizado (48,54% do tempo total).

Capítulo 7

Conclusão e trabalhos futuros

7.1 Conclusão

Nesta dissertação, foi proposta e avaliada uma estratégia de paralelismo fino multibloco em GPU para a obtenção do alinhamento múltiplo ótimo. Adicionalmente, foi também proposta uma estratégia alternativa que foi descartada por possuir um desempenho inferior.

A primeira estratégia proposta possuía granularidade grossa (Seção 5.2) e era otimizada para seguir o formato do *wavefront* multidimensional (Seção 2.4). Porém, ao analisar o formato de percorrer o espaço de busca desta estratégia, observou-se uma queda no número de células que poderiam ser calculadas em paralelo (Seção 5.2.3).

Com isso, uma nova estratégia de granularidade fina (Seção 5.3) foi proposta, aumentando ainda mais o paralelismo, fazendo que um bloco de *threads* calcule o valor de uma célula, ao invés de apenas uma *thread* calcular esse mesmo valor. Além disso, vários blocos de *threads* são executados simultaneamente. Esta estratégia percorre o espaço de busca usando o formato de diagonais projetadas em duas dimensões.

Nos resultados experimentais (Capítulo 6), obtidos com sequências reais e sintéticas de diversos tamanhos na GPU GTX 580, verificou-se que a estratégia de granularidade fina possui um melhor desempenho para obter o alinhamento ótimo. Por isso, a estratégia de granularidade fina multibloco, foi chamada de Carrillo-Lipman em GPU, e foi usada para comparar com o tempo de execução do alinhamento múltiplo em CPU utilizando o programa MSA 2.0.

Constatou-se que para algumas sequências de baixa similaridade é necessário relaxar alguns parâmetros utilizados pelo programa MSA 2.0, sendo necessário uma nova execução do programa para obter o alinhamento. Nesses casos, o Carrillo-Lipman em GPU foi até 8,6 vezes mais rápido do que o MSA 2.0. Para sequências de alta similaridade, o programa MSA 2.0 reduz bastante o espaço de busca e nestes casos o seu desempenho é bem melhor do que o desempenho de nossa estratégia em GPU. Situação semelhante foi encontrada na literatura, onde o desempenho do programa em CPU foi superior ao do FPGA para sequências de alta similaridade (Seção 4.2.2).

Para analisar o tempo gasto com a invocação dos *kernels* do nosso Carrillo-Lipman em GPU, foram realizados testes e concluiu-se que as invocações são um fator importante no tempo de execução total. Para sequências pequenas, esse tempo representa até 98%

do tempo total. Já para sequências longas, representa no mínimo 48,54% do tempo total de execução, o que ainda é um tempo considerável.

7.2 Trabalhos Futuros

Como trabalhos futuros, sugerimos inicialmente melhorar a gerência de memória da estratégia de granularidade fina, permitindo o alinhamento de sequências com tamanho muito maiores que os obtidos atualmente. Além disso, pretendemos implementar a versão do Carrillo-Lipman em GPU para um maior número de sequências. Como o problema do alinhamento múltiplo de sequências é exponencial ao número de sequências, este número não pode crescer muito, mas, para uma quantidade maior de sequências, o número de células que podem ser calculadas em paralelo é cada vez maior. Sendo assim, acreditamos que esse alto paralelismo pode ser efetivamente explorado pelas GPUs, tornando possível a obtenção do alinhamento múltiplo ótimo de quatro a dez sequências de tamanho médio ou pequeno.

O Carrillo-Lipman em GPU obtém apenas o escore do alinhamento ótimo, não retornando o alinhamento em si. Para versões futuras, é desejável implementar esta funcionalidade, para que seja possível visualizar as regiões de semelhanças e diferenças entre as sequências e não apenas quantificar a diferença entre elas.

Além disso, é interessante mostrar teoricamente que a estratégia grossa possui pior desempenho que a estratégia fina. Com isso, pode-se analisar os melhores casos de desempenho da estratégia fina e da estratégia grossa para criar ao final uma estratégia mista que combina ambas estratégias procurando obter um melhor desempenho que ambas. Também é conhecido e citado por alguns autores que o limite Carrillo-Lipman não é o limite exato para o descarte de células. Sendo assim, acreta-se que ainda seja possível descartar mais células e mesmo assim obter-se um alinhamento ótimo. Demostrar qual seria esse limite é tarefa bastante complexa porém de grande valor para melhorar o tempo de execução dos algoritmos exatos de alinhamento múltiplo ótimo.

Finalmente, planeja-se reimplementar a estratégia de granularidade fina multibloco em outras arquiteturas paralelas, podendo ser da própria NVidia (por exemplo, Kepler) como de outros fabricantes, como AMD ou Intel. Para isso, será necessário migrar para outras plataformas de *software* como o OpenCL ou Many Integrated Core (MIC) Architecture.

Referências

[1] S. F. Altschul. Gap costs for multiple sequence alignment. *Journal of Theoretical Biology*, 138(3):297–309, 1989.

[2] S. F. Altschul, R. J. Carroll, and D. J. Lipman. Weights for data related by a tree. *Journal of Molecular Biology*, 207(4):647–653, 1989.

[3] Balibase. Website. `http://bips.u--strasbg.fr/fr/Products/Databases/BAliBASE2/align_index.html` (accessed August 1, 2012).

[4] J. Blazewicz, W. Frohmberg, M. Kierzynka, and P. Wojciechowski. G-MSA — A GPU-based, fast and accurate algorithm for multiple sequence alignment. *Journal of Parallel and Distributed Computing*, 2012. Early Access.

[5] H. Carrillo and D. Lipman. The multiple sequence alignment problem in biology. *SIAM Journal of Applied Mathematics*, 48:1073–1082, 1988.

[6] K. Chaichoompu, S. Kittitornkun, and S. Tongsima. MT-clustalW: multithreading multiple sequence alignment. In *20th International Conference on Parallel and Distributed Processing*, pages 254–254. IEEE Computer Society, 2006.

[7] M. O. Dayhoff, R. M. Schwartz, and B. C. Orcutt. A model of evolutionary change in proteins. *Atlas of Protein Sequence and Structure*, 5(suppl 3):345–351, 1978.

[8] J. Dongarra. Trends in High-Performance Computing. *IEEE Circuits & Devices Magazine*, pages 22–27, 2006.

[9] R. Durbin, S. R. Eddy, A. Krogh, and G. Mitchison. *Biological Sequence Analysis: Probabilistic Models of Proteins and Nucleic Acids*. Cambridge University Press, 1999.

[10] O. Duzlevski. Smp version of clustalw 1.82. available at `http://bioinfo.pbi.nrc.ca/clustalw-smp/` (accessed August 3), 2002.

[11] R. C. Edgar. Muscle: multiple sequence alignment with high accuracy and high throughput. *Nucleic Acids Research*, 32:1792–1797, 2004.

[12] D. F. Feng and R. F. Doolittle. Progressive sequence alignment as a prerequisite to correct phylogenetic trees. *Journal of Molecular Evolution*, 25(4):351–360, 1987.

[13] M. Flynn. Some Computer Organizations and Their Effectiveness. *IEEE Transactions on Computers*, C-21:948–960, 1972.

[14] O. Gotoh. Significant improvement in accuracy of multiple protein sequence alignments by iterative refinement as assessed by reference to structural alignments. *Journal of Molecular Biology*, 4(264):823–838, 1996.

[15] S. A. Gupta S., Kececioglu J. Improving the practical space and time efficiency of the shortest-paths approach to sum-of-pairs multiple sequence alignment. *Journal of Computational Biology*, 2(3):4590–472, 1995.

[16] D. Gusfield. *Algorithms on Strings, Trees and Sequences.* Cambridge University Press, 1997.

[17] M. Helal, H. El-Gindy, L. Mullin, and B. Gaëta. Parallelizing optimal multiple sequence alignment by dynamic programming. In *International Symposium on Parallel and Distributed Processing with Applications*, pages 669–674, 2008.

[18] M. Helal, L. Mullin, B. Gaëta, and H. El-Gindy. Multiple sequence alignment using massively parallel mathematics of arrays. In *International Conference on High Performance Computing, Networking and Communication Systems*, pages 120–127, 2007.

[19] M. Helal, L. Mullin, J. Potter, and V. Sintchenko. Search space reduction technique for distributed multiple sequence alignment. In *6th IFIP International Conference on Network and Parallel Computing*, pages 219–226. IEEE Computer Society, 2009.

[20] S. Henikoff and J. G. Henikoff. Amino acid substitution matrices from protein blocks. *National Academy of Sciences of the United States of America*, 89(22):10915–9, 1992.

[21] D. G. Higgins and P. M. Sharp. CLUSTAL: a package for performing multiple sequence alignment on a microcomputer. *Gene*, 73(1):237–244, 1988.

[22] K. Hwang and Z. Xu. *Scalable Parallel Computing: Technology, Architecture, Programming.* McGraw-Hill Science/Engineering/Math, 1st edition, 1998.

[23] A. Johnson and C. O'Donnell. An Open Access Database of Genome-wide Association Results. *BMC Medical Genetics*, 10(1):6, 2009.

[24] L. S. Johnson, S. R. Eddy, and E. Portugaly. Hidden Markov model speed heuristic and iterative HMM search procedure. *BMC Bioinformatics*, 11(1):431, 2010.

[25] K. Li. ClustalW-MPI: ClustalW analysis using distributed and parallel computing. *Bioinformatics*, 19:1585–1586, 2003.

[26] D. J. Lipman, S. F. Altschul, and J. D. Kececioglu. A tool for multiple sequence alignment. *National Academy of Sciences of the United States of America*, 86(12):4412–4415, 1989.

[27] W. Liu, B. Schmidt, G. Voss, and W. Müller-Wittig. GPU-ClustalW: Using Graphics Hardware to Accelerate Multiple Sequence Alignment. In *High Performance Computing*, volume 4297, chapter 37, pages 363–374. Springer Berlin Heidelberg, 2006.

[28] Y. Liu, B. Schmidt, and D. L. Maskell. MSA-CUDA: Multiple Sequence Alignment on Graphics Processing Units with CUDA. In *20th IEEE International Conference on Application-specific Systems, Architectures and Processors*, pages 121–128. IEEE Computer Society, 2009.

[29] Y. Liu, B. Schmidt, and D. L. Maskell. Parallel reconstruction of neighbor-joining trees for large multiple sequence alignments using cuda. In *2009 IEEE International Symposium on Parallel&Distributed Processing*, pages 1–8. IEEE Computer Society, 2009.

[30] S. Masuno, T. Maruyama, Y. Yamaguchi, and A. Konagaya. An FPGA implementation of multiple sequence alignment based on Carrillo-Lipman method. In *2007 International Conference on Field Programmable Logic and Applications*, pages 489–492, 2007.

[31] M. A. McClure, T. K. Vasi, and W. M. Fitch. Comparative analysis of multiple protein-sequence alignment methods. *Molecular Biology and Evolution*, 11(4):571–592, 1994.

[32] The Message Passing Interface Standard. available at `http://www-unix.mcs.anl.gov/mpi/` (accessed August 5, 2012).

[33] B. Morgenstern, K. Frech, A. Dress, and T. Werner. DIALIGN: finding local similarities by multiple sequence alignment. *Bioinformatics*, 14(3):290–294, 1998.

[34] D. W. Mount. *Bioinformatics: Sequence and Genome Analysis*. Cold Spring Harbor Laboratory Press, 1st edition, 2001.

[35] E. W. Myers and W. Miller. Optimal alignments in linear space. *Computer Applications in the Biosciences*, 4(1):11–17, 1988.

[36] S. Needleman and C. 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.

[37] C. Notredame. T-Coffee: a novel method for fast and accurate multiple sequence alignment. *Journal of Molecular Biology*, 302(1):205–217, 2000.

[38] C. Notredame and D. G. Higgins. SAGA: sequence alignment by genetic algorithm. *Nucleic Acids Research*, 24(8):1515–1524, 1996.

[39] NVidia's "Fermi" GPU architecture revealed. Website. `http://techreport.com/articles.x/17670/2` (accessed August 8).

[40] NVidia. NVidia CUDA Architecture, 2009.

[41] NVidia. NVIDIA Fermi Compute Architecture Whitepaper, 2011. available at `http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf` (accessed August 8, 2012).

[42] NVidia. PTX: Parallel Thread Execution, 2011. available at `http://developer.download.nvidia.com/compute/DevZone/docs/html/C/doc/ptx_isa_2.3.pdf` (accessed August 8, 2012).

[43] NVIDIA. NVIDIA CUDA Programming Guide 4.2, 2012.

[44] T. Oliver, B. Schmidt, D. Nathan, R. Clemens, and D. Maskell. Using reconfigurable hardware to accelerate multiple sequence alignment with clustalw. *Bioinformatics*, 21:3431–3432, 2005.

[45] Opencl. Website. `http://www.khronos.org/opencl/` (accessed August 5, 2012).

[46] The OpenMP® API specification for parallel programming, available at `http://www.openmp.org` (accessed august 5, 2012).

[47] Pfam. Website. `http://pfam.sanger.ac.uk/` (accessed August 1, 2012).

[48] G. F. Pfister. *In search of clusters*. Prentice-Hall, Inc., 2nd edition, 1998.

[49] N. Saitou and M. Nei. The neighbor-joining method: a new method for reconstructing phylogenetic trees. *Molecular Biology and Evolution*, 4(4):406–425, 1987.

[50] J. Sanders and E. Kandrot. *CUDA by Example: An Introduction to General-Purpose GPU Programming*. Addison-Wesley Professional, 1st edition, 2010.

[51] R. Smith and T. Smith. Pattern-induced multi-sequence alignment (pima) algorithm employing secondary structure-dependent gap penalties for use in comparative protein modelling. *Protein Engineering*, 5, 1992.

[52] T. F. Smith and M. S. Waterman. Identification of common molecular subsequences. *Journal of Molecular Biology*, 147(1):195–197, 1981.

[53] A. R. Subramanian, M. Kaufmann, and B. Morgenstern. DIALIGN-TX: greedy and progressive approaches for segment-based multiple sequence alignment. *Algorithms for Molecular Biology*, 3(1):6, 2008.

[54] W. Taylor. A flexible method to align large numbers of biological sequences. *Journal of Molecular Evolution*, 28:161–169, 1988.

[55] J. D. Thompson, D. G. Higgins, and T. 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.

[56] A. Vrenios. *Linux Cluster Architecture*. Sams Pub Co., 2002.

[57] L. Wang and T. Jiang. On the complexity of multiple sequence alignment. *Journal of Computational Biology*, 1(4):337–348, 1994.

[58] W. Wolf. *FPGA-Based System Design*. Prentice Hall PTR, 2004.

Anexo I

Sequências Sintéticas

synthetic_easy.fasta

```
>Sequence 1
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
VLSPADKTNVKAAWGKVGAHAGEYGAEALE
RMFLSFPTTKTYFPHFDLSHGSAQVKGHGK
KVADALTNAVAHVDDMPNALSALSDLHAHK
LRVDPVNFKLLSHCLLVTLAAHLPAEFTPA
VHASLDKFLASVSTVLTSKYR
>Sequence 2
AAAAAPPPPPPPPPPPPPPPPPPPPPPPPP
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
PPPPPPPPEKSAVTALPPPPPPPPPPPPPP
VHLTPEEKSAVTALWGKVNVDEVGGEALGR
LLVVYPWTQRFFESFGDLSTPDAVMGNPKV
KAHGKKVLGAFSDGLAHLDNLKGTFATLSE
LHCDKLHVDPENFRLLGNVLVCVLAHHFGK
EFTPPVQAAYQKVVAGVANALAHKYH
DDDDDDDDDDDDDDDDDDDDDDDDDDDDDD
DDDDDDDDDDDDDDDDDDDDDDDDDPPPPP
AAAAAAAAADDDDDAAADDDDAPPPPADDD
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
>Sequence 3
DDDDDDDDDDDDDDDDDDDDDDDDDDDDDD
DDDDDDDDDDDDDDDDDDDDDDDDDPPPPP
AAAAAAAAADDDDDAAADDDDAPPPPADDD
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
VASAPADKTNVKAAWGKVGAHAGEYAAAAA
GLSDGEWQLVLNVWGKVEADIPGHGQEVLI
RLFKGHPETLEKFDKFKHLKSEDEMKASED
LKKHGATVLTALGGILKKKGHHEAEIKPLA
QSHATKHKIPVKYLEFISECIIQVLQSKHP
GDFGADAQGAMNKALELFRKDMASNYKELG
FQG
```

synthetic_medium.fasta

```
>Sequence 1
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
VLSPADKTNVKAAWGKVGAHAGEYGAEALE
RMFLSFPTTKTYFPHFDLSHGSAQVKGHGK
KVADALTNAVAHVDDMPNALSALSDLHAHK
LRVDPVNFKLLSHCLLVTLAAHLPAEFTPA
VHASLDKFLASVSTVLTSKYR
>Sequence 2
AAAAAPPPPPPPPPPPPPPPPPPPPPPPPP
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
PPPPPPPPEKSAVTALPPPPPPPPPPPPPP
VHLTPEEKSAVTALWGKVNVDEVGGEALGR
LLVVYPWTQRFFESFGDLSTPDAVMGNPKV
KAHGKKVLGAFSDGLAHLDNLKGTFATLSE
LHCDKLHVDPENFRLLGNVLVCVLAHHFGK
EFTPPVQAAYQKVVAGVANALAHKYH
DDDDDDDDDDDDDDDDDDDDDDDDDDDDDD
DDDDDDDDDDDDDDDDDDDDDDDDDDPPPP
VLSPADKTNVKAAWGKVGAHAGEYGAEALE
RMFLSFPTTKTYFPHFDLSHGSAQVKGHGK
KVADALTNAVAHVDDMPNALSALSDLHAHK
AAAAAAAAADDDDDAAADDDDAPPPPADDD
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
>Sequence 3
DDDDDDDDDDDDDDDDDDDDDDDDDDDDDD
DDDDDDDDDDDDDDDDDDDDDDDDDPPPPP
AAAAAAAAADDDDDAAADDDDAPPPPADDD
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
VASAPADKTNVKAAWGKVGAHAGEYAAAAA
GLSDGEWQLVLNVWGKVEADIPGHGQEVLI
RLFKGHPETLEKFDKFKHLKSEDEMKASED
LKKHGATVLTALGGILKKKGHHEAEIKPLA
QSHATKHKIPVKYLEFISECIIQVLQSKHP
GDFGADAQGAMNKALELFRKDMASNYKELG
FQG
```

synthetic_hard1.fasta

```
>Sequence 1
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
VLSPADKTNVKAAWGKVGAHAGEYGAEALE
RMFLSFPTTKTYFPHFDLSHGSAQVKGHGK
KVADALTNAVAHVDDMPNALSALSDLHAHK
LRVDPVNFKLLSHCLLVTLAAHLPAEFTPA
VHASLDKFLASVSTVLTSKYR
>Sequence 2
AAAAAPPPPPPPPPPPPPPPPPPPPPPPPP
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
PPPPPPPPEKSAVTALPPPPPPPPPPPPPP
VHLTPEEKSAVTALWGKVNVDEVGGEALGR
LLVVYPWTQRFFESFGDLSTPDAVMGNPKV
KAHGKKVLGAFSDGLAHLDNLKGTFATLSE
LHCDKLHVDPENFRLLGNVLVCVLAHHFGK
EFTPPVQAAYQKVVAGVANALAHKYH
DDDDDDDDDDDDDDDDDDDDDDDDDDDDDD
DDDDDDDDDDDDDDDDDDDDDDDDDPPPPP
VLSPADKTNVKAAWGKVGAHAGEYGAEALE
RMFLSFPTTKTYFPHFDLSHGSAQVKGHGK
KVADALTNAVAHVDDMPNALSALSDLHAHK
AAAAAAAAADDDDDAAADDDDAPPPPADDD
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
>Sequence 3
DDDDDDDDDDDDDDDDDDDDDDDDDDDDDD
DDDDDDDDDDDDDDDDDDDDDDDDDPPPPP
AAAAAAAAADDDDDAAADDDDAPPPPADDD
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
VASAPADKTNVKAAWGKVGAHAGEYAAAAA
GLSDGEWQLVLNVWGKVEADIPGHGQEVLI
RLFKGHPETLEKFDKFKHLKSEDEMKASED
LKKHGATVLTALGGILKKKGHHEAEIKPLA
AAAAAPPPPPPPPPPPPPPPPPPPPPPPPP
QSHATKHKIPVKYLEFISECIIQVLQSKHP
AAAAAPPPPPPPPPPPPPPPPPPPPPPPPP
GDFGADAQGAMNKALELFRKDMASNYKELG
FQG
```

synthetic_hard2.fasta

```
>Sequence 1
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
VLSPADKTNVKAAWGKVGAHAGEYGAEALE
RMFLSFPTTKTYFPHFDLSHGSAQVKGHGK
KVADALTNAVAHVDDMPNALSALSDLHAHK
LRVDPVNFKLLSHCLLVTLAAHLPAEFTPA
VHASLDKFLASVSTVLTSKYR
>Sequence 2
AAAAAPPPPPPPPPPPPPPPPPPPPPPPPP
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
PPPPPPPPEKSAVTALPPPPPPPPPPPPPP
VHLTPEEKSAVTALWGKVNVDEVGGEALGR
LLVVYPWTQRFFESFGDLSTPDAVMGNPKV
KAHGKKVLGAFSDGLAHLDNLKGTFATLSE
LHCDKLHVDPENFRLLGNVLVCVLAHHFGK
EFTPPVQAAYQKVVAGVANALAHKYH
DDDDDDDDDDDDDDDDDDDDDDDDDDDDDD
DDDDDDDDDDDDDDDDDDDDDDDDDPPPPP
VLSPADKTNVKAAWGKVGAHAGEYGAEALE
RMFLSFPTTKTYFPHFDLSHGSAQVKGHGK
KVADALTNAVAHVDDMPNALSALSDLHAHK
AAAAAAAAADDDDDAAADDDDAPPPPADDD
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
>Sequence 3
DDDDDDDDDDDDDDDDDDDDDDDDDDDDDD
DDDDDDDDDDDDDDDDDDDDDDDDDPPPPP
AAAAAAAAADDDDDAAADDDDAPPPPADDD
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
VASAPADKTNVKAAWGKVGAHAGEYAAAAA
GLSDGEWQLVLNVWGKVEADIPGHGQEVLI
RLFKGHPETLEKFDKFKHLKSEDEMKASED
LKKHGATVLTALGGILKKKGHHEAEIKPLA
AAAAAPPPPPPPPPPPPPPPPPPPPPPPPP
QSHATKHKIPVKYLEFISECIIQVLQSKHP
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAPPPPPPPPPPPPPPPPPPPPPPPPP
GDFGADAQGAMNKALELFRKDMASNYKELG
FQG
```

synthetic_equal.fasta

```
>Sequence1
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAA
>Sequence2
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAABAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAA
>Sequence3
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
```

```
AAAAAAAAAAAAAAAACAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAA
```