Alcides Carneiro de Araújo Neto

Comparação Sequência-Família em GPU

Campo Grande – MS
2014

Alcides Carneiro de Araújo Neto

Comparação Sequência-Família em GPU

Dissertação de Mestrado apresentada à Faculdade de Computação da Universidade Federal de Mato Grosso do Sul – UFMS, como parte dos requisitos para obtenção do título de Mestre em Ciência da Computação.

Universidade Federal de Mato Grosso do Sul – UFMS

Faculdade de Computação – FACOM

Orientadora: Prof.ª Dr.ª Nahri Balesdent Moreano

Campo Grande – MS

2014

Resumo

Durante as últimas décadas o volume de informações biológicas em algumas bases de dados cresceu em um ritmo quase exponencial. Ferramentas como o HMMER podem encontrar sequências biológicas homólogas a uma família de sequências modelada estatisticamente por um *profile* HMM utilizando o algoritmo de Viterbi. Dada a complexidade quadrática desse algoritmo, esse procedimento pode consumir longos tempos de execução dependendo da quantidade de sequências, do tamanho do *profile* HMM e do hardware utilizado. Esse trabalho descreve o desenvolvimento de uma solução em GPU, de alto desempenho, para o problema de determinar se uma nova sequência biológica é homóloga a uma família de sequências conhecida. A solução implementada alcançou desempenho compatível ou superior ao HMMER.

Palavras-chaves: Alinhamento sequência-família, algoritmo de Viterbi, HMMER, GPU, CUDA, acelerador.

Abstract

Over the past few decades the amount of biological data in some databases grew up in an almost exponential rate. Tools such as HMMER use the Viterbi algorithm to find biological sequences that are homologue to a family of sequences represented by a statistical model called profile HMM. Due to the quadratic time complexity of the Viterbi algorithm, this search procedure can demand long execution times depending on database size, profile HMM length, and hardware used. The purpose of this project is to design a high performance GPU solution for the problem of finding out if a new biological sequence is homologue to a known family of sequences. The implemented solution reached a performance compatible or superior to HMMER.

Keywords: Sequence-profile alignment, Viterbi algorithm, HMMER, GPU, CUDA, accelerator.

Sumário

Lista de Figuras

Lista de Tabelas

Lista de Algoritmos

1 Introdução

Em Bioinformática, uma sequência biológica é modelada como uma cadeia finita de símbolos que carregam informação biológica. Por sua vez, uma família de sequências é um conjunto de sequências homólogas, isto é, as sequências pertencentes a uma família divergiram durante a evolução de tal forma que compartilham semelhanças estruturais e/ou funcionais [45].

Uma vez que, na natureza, novas sequências evoluem a partir de sequências preexistentes através de mutações, a análise computacional das mesmas é beneficiada disso. Muitas vezes é possível reconhecer semelhanças significativas entre uma nova sequência e uma sequência já conhecida sobre a qual sabe-se algo. Quando isso ocorre é possível transferir informações sobre a estrutura e/ou função da sequência já conhecida para a nova sequência. Assim, as duas sequências relacionadas são então ditas homólogas e a transferência de informações entre elas é feita por homologia [7]. Portanto, o problema da **comparação sequência-família**, isto é, determinar se uma nova sequência biológica é homóloga a uma família de sequências conhecida é de grande importância na Bioinformática [6].

1.1 Motivação

O volume de dados biológicos vem crescendo de maneira surpreendente. O Projeto Genoma Humano (*Human Genome Project*) [33], um programa de pesquisa internacional e colaborativo cujo objetivo era o mapeamento completo e a compreensão de todos os genes dos seres humanos, foi declarado completo em 2003, identificando cerca de 3 bilhões de símbolos do código genético humano. Outro exemplo é a base de dados de proteínas UniProtKB/TrEMBL [51], que apresentou um crescimento quase exponencial desde o ano 2000, praticamente dobrando de tamanho a cada dois anos. A Figura 1.1 mostra o crescimento do número de sequências dessa base de dados entre os anos de 1996 e 2013.

A análise de grandes bases de dados de sequências biológicas e famílias de sequências exige, dentre outras técnicas, uma que permita resolver o problema da comparação sequência-família. O HMMER [9, 19], uma solução em software para computadores de arquitetura convencional, é uma das principais ferramentas utilizadas para este fim e baseia-se em um importante algoritmo denominado algoritmo de Viterbi [14].

Dada a complexidade de tempo quadrática do algoritmo de Viterbi, a análise de grandes bases de sequências e famílias utilizando o HMMER pode demandar longos

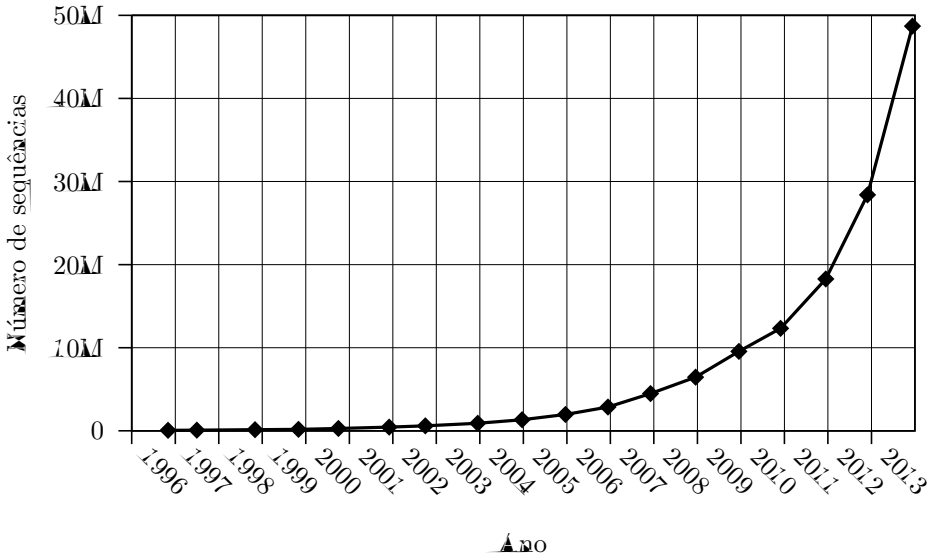


Figura 1.1: Crescimento da base de dados de proteínas UniProtKB/TrEMBL entre 1996 e 2013 [51]

tempos de execução. Com o rápido crescimento das bases de dados biológicos, esses tempos de execução tornam-se ainda mais críticos [6, 18, 42, 55, 56]. Em vista disso, surge a necessidade de soluções de alto desempenho, capazes de analisar grandes volumes de sequências e/ou famílias em pouco tempo, seja pela adoção de heurísticas ou pela exploração de paralelismo ou até mesmo por ambas [15].

Soluções paralelas baseadas em *clusters* e aceleradores em hardware implementados em FPGAs (*Field-Programmable Gate Arrays*) estão entre as propostas desenvolvidas que apresentaram *speedups* consideráveis se comparados com o HMMER. A ferramenta HMMER, a cada nova versão, também incorpora recursos com o objetivo de melhorar seu desempenho. Tais recursos podem permitir execução em *cluster*, execução *multithread* em um processador *multi-core* e exploração de paralelismo de dados utilizando extensões SIMD do conjunto de instruções de máquina (descritas na Seção 4.3).

Mais recentemente, devido aos resultados promissores obtidos em outras áreas utilizando-se GPUs (*Graphics Processing Units*), pesquisadores voltaram suas atenções para o uso das mesmas como plataforma de computação paralela. Assim, algumas soluções paralelas baseadas em GPU foram propostas para o problema da comparação sequência-família, com o objetivo de obter um bom desempenho.

A vantagem da utilização de GPUs está em sua arquitetura *many-core* que disponibiliza ao desenvolvedor uma grande quantidade de *cores* que continua a aumentar segundo a Lei de Moore [34]. Assim, problemas que podem ser modelados de forma a tirar vantagem desse crescente aumento na quantidade de *cores* possuem potencial para ganho de desempenho quando executados explorando paralelismo nesse tipo de plataforma.

1.2 Objetivos do Trabalho

A proposta desse trabalho é a construção de uma solução em GPU para o problema da comparação sequência-família, isto é, determinar se uma nova sequência biológica é homóloga a uma família de sequências conhecida. O objetivo é obter uma solução de alto desempenho que permita a análise de grandes bases de dados biológicos em pouco tempo, utilizando como plataforma de execução uma GPU.

Para tal, é necessário estudar os algoritmos utilizados para resolver o problema da comparação sequência-família, suas dependências de dados e como é possível explorar paralelismo na execução desses algoritmos. As abordagens das soluções já desenvolvidas nos variados dispositivos de alto desempenho devem ser estudas e uma possível adaptação das mesmas para GPU deve ser analisada.

Também é necessário conhecer a plataforma de computação paralela utilizada, no caso uma GPU, isto é, suas características arquiteturais e seu modelo de programação. Pretende-se compreender como tirar o máximo de proveito dos recursos dessa plataforma e minimizar as desvantagens no uso da mesma, através de boas práticas de programação e otimização de código, com o objetivo de melhorar o desempenho da solução.

Por fim, pretende-se realizar uma avaliação experimental de desempenho da solução desenvolvida em GPU, comparando-a com o HMMER e outros trabalhos. O objetivo é avaliar as otimizações aplicadas na solução em GPU e concluir a viabilidade de utilizá-las como plataforma de computação paralela para resolver o problema da comparação sequência-família e problemas similares da área de Bioinformática.

1.3 Organização do Texto

O restante desse texto está organizado em seis capítulos. O Capítulo 2 descreve a utilização das GPUs como plataforma de computação paralela, sua arquitetura, hierarquia de memórias, o modelo de programação CUDA e os mecanismos de sincronização deste modelo.

No Capítulo 3, conceitos básicos de Bioinformática necessários para a melhor compreensão do problema da comparação sequência-família, são introduzidos. O problema do alinhamento sequência-família é então formalizado. A ferramenta HMMER é descrita e os algoritmos de Viterbi, MSV (*Multiple Segment Viterbi*) e SSV (*Single Segment* Viterbi), utilizados para a comparação sequência-família, são apresentados e as dependências de dados presentes nos mesmos são examinadas com o intuito de encontrar possibilidades de exploração de paralelismo.

Em seguida, no Capítulo 4 são expostos trabalhos relacionados que propõem soluções paralelas para o problema da comparação sequência-família. São apresentadas soluções que utilizam extensões do conjunto de instruções da CPU, aceleradores em hardware implementados em FPGAs e soluções paralelas baseadas em *clusters* e GPUs, assim como seus resultados.

No Capítulo 5, a solução em GPU desenvolvida para o problema da comparação sequência-família é apresentada. As plataformas, ferramentas e dados utilizados para desenvolvimento, execução e experimentos são descritos. A abordagem adotada para a solução é analisada, detalhando-se os *kernels* desenvolvidos. A organização das estruturas de dados da solução e seu mapeamento na hierarquia de memórias da GPU são apresentados e o consequente impacto no desempenho da solução é analisado. Também são apresentadas as otimizações aplicadas na solução.

Os resultados obtidos na avaliação experimental são apresentados no Capítulo 6, sendo analisado o desempenho da solução em GPU e comparado com outras soluções. Por fim, o Capítulo 7 apresenta a conclusão deste trabalho, fazendo uma análise geral sobre o que foi desenvolvido. Outrossim, propostas de melhorias e trabalhos futuros são elicitadas.

2 Plataforma de Computação Paralela

Impulsionadas pela demanda do mercado por gráficos de alta definição em tempo real para aplicações computacionais, as GPUs evoluíram, durante os últimos 30 anos, de dispositivos simples com um único *pipeline* para dispositivos altamente paralelos formados por vários longos *pipelines*, capazes de processar imagens interativas complexas de cenas 3D. Paralelamente a isso, muitas das funcionalidades do hardware tornaram-se mais sofisticadas e programáveis pelo programador [25].

Em *pipelines* gráficos, alguns estágios fazem uma grande quantidade de operações aritméticas com números em ponto flutuante utilizando dados completamente independentes, tais como transformar posições dos vértices de um triângulo ou gerar as cores de *pixels*. Essa independência de dados, característica dominante em aplicações gráficas, é uma diferença fundamental nas considerações de projetos para GPUs.

As funções específicas executadas em alguns poucos estágios dos *pipelines* gráficos variam dependendo do algoritmo de renderização utilizado. Tal variação motivou o desenvolvimento de GPUs com estágios programáveis. Dois estágios destacaram-se, o *vertex shader* e o *pixel shader*. O primeiro mapeia as posições dos vértices de um triângulo na tela, alterando suas posições, cores ou orientação. Usualmente isso consiste em ler a posição de um vértice, que nada mais é que um conjunto de números em ponto flutuante, e calcular uma posição na tela que também é um conjunto de números em ponto flutuante. Já o segundo estágio define a cor de um *pixel* calculando as cores vermelho, verde, azul e alpha (RGBA), cada uma representada por um número em ponto flutuante, contribuindo assim para a renderização da imagem [25].

Uma vez que as operações ariméticas realizadas nesses dois estágios podiam ser controladas pelo programador, pesquisadores observaram que os dados usados como entrada poderiam na verdade ser quaisquer dados e não necessariamente cores ou posições, bastariam apenas estar no formato de cores ou posições. Em outras palavras, as GPUs poderiam ser "enganadas" para realizarem tarefas que não eram a de renderização de cenas através da formatação dos dados de entrada de maneira que elas pensassem que estariam renderizando uma cena [48].

Dado ao alto poder aritmético das GPUs, os resultados inciais desses experimentos foram bastante promissores. Entretanto, as limitações e dificuldades de se fazer isso pelo modelo de programação presente até então, as APIs gráficas, dificultavam o desenvolvimento. Esse desejo de usar as GPUs como um dispositivo de computação

paralela mais genérico motivou a NVIDIA [37], fabricante de GPUs, a desenvolver uma nova arquitetura gráfica unificada e também o modelo de programação CUDA (*Compute Unified Device Architecture*) [34].

Outro exemplo de modelo de programação paralela é o OpenCL (*Open Computing Language*) mantido pelo consórcio Khronos Group [23]. O OpenCL é um *framework* para desenvolvimento e execução de programas em plataformas heterogêneas, consistindo de CPUs, GPUs, FPGAs, entre outros.

Assim, as GPUs programáveis evoluíram para um processador *many-core*, *multithread* e altamente paralelo. Além disso, esse paralelismo continua a avançar segundo a Lei de Moore [34]. Portanto, o desafio tornou-se desenvolver programas que utilizem esse paralelismo de forma transparente e que ganhem desempenho de forma escalável, para tirar proveito do aumento no número de *cores* das GPUs que, até o momento, continua a crescer a cada nova geração [11].

2.1 Modelo de Programação CUDA

O modelo CUDA é uma extensão das linguagens de programação C e C++ [25]. O programador escreve programas sequenciais que invocam *kernels*. Esses, por sua vez, podem ser simples funções ou programas completos. Um *kernel* é executado em paralelo através de um conjunto de *threads* paralelas. Essas *threads* são organizadas em uma hierarquia, na forma de *grids* de blocos de *threads*, como pode ser visto na Figura 2.1. Dessa forma, define-se:

- *Thread*: unidade básica de processamento;
- Bloco: um conjunto de *threads* concorrentes que podem cooperar entre si, sincronizar-se através de uma barreira de sincronização e comunicar-se através da memória compartilhada pertencente ao bloco;
- *Grid*: um conjunto de blocos de *threads* que podem ser executados concorrentemente e assim explorar paralelismo, ou seja, não há dependências entre os mesmos.

A Figura 2.1 exemplifica o espaço de execução criado quando um determinado *kernel* é invocado. Um *grid* bidimensional é criado, com 3×2 blocos, cada bloco com 2×2 *threads*. Cada bloco e cada *thread* possuem uma identificação única. O modelo de programação CUDA permite a criação de *grids* unidimensionais ou bidimensionais e blocos unidimensionais, bidimensionais ou tridimensionais.

O requisito de independência entre os blocos de *threads* ajuda a evitar possíveis *deadlocks* e permite que os mesmos possam ser escalonados em qualquer ordem, em

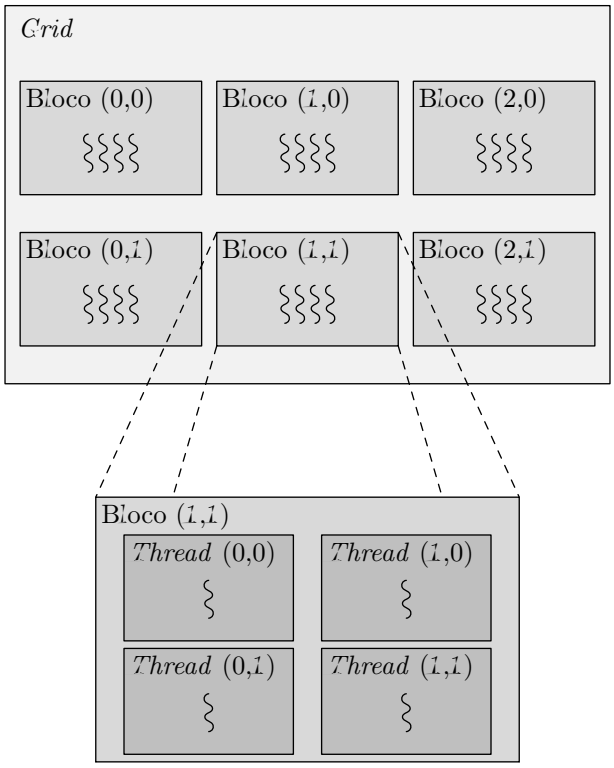


Figura 2.1: Organização das *threads* no modelo de programação CUDA: um *grid* com 2×3 blocos de 2×2 *threads*

qualquer quantidade de *cores*, favorecendo a escalabilidade. O modelo de programação CUDA também não estabelece restrições sobre a localização de cada *core*, potencialmente permitindo a portabilidade de forma transparente de programas escritos no modelo através de múltiplos dispositivos no futuro [11]. Portanto, um programa CUDA compilado pode ser executado em uma GPU com uma quantidade qualquer de *cores* e apenas o sistema, em tempo de execução, precisa saber a quantidade de *cores* físicos [34].

2.1.1 Programa Exemplo

O problema da soma de dois vetores com 102.400 elementos cada um, em paralelo, é usado para exemplificar uma solução desenvolvida usando o modelo de programação CUDA. A Figura 2.2 mostra o código CUDA que executa no computador *host* ao qual a CPU está acoplada (conforme será visto na Seção 2.2), enquanto a Figura 2.3 mostra o código do *kernel* executado na GPU.

Na Figura 2.2, o código que executa no *host* inicialmente aloca espaço para os vetores na memória RAM do *host* (linhas 8 a 10) e na memória global da GPU (linhas 13 a 16)[1]. Em seguida, os vetores são copiados da memória do *host* para a memória global

[1]A hierarquia de memórias da GPU será descrita na Subseção 2.2.1.

```
1  /* Função executada no host */
2  int main(int argc, char* argv[])
3  {
4     /* Tamanho dos vetores */
5     int n = 102400;
6
7     /* Aloca espaço para vetores na memória do host */
8     double *h_a = (double *) malloc(n * sizeof(double));
9     double *h_b = (double *) malloc(n * sizeof(double));
10    double *h_c = (double *) malloc(n * sizeof(double));
11
12    /* Aloca espaço para os vetores na memória global da GPU */
13    double *d_a, *d_b, *d_c;
14    cudaMalloc(&d_a, n * sizeof(double));
15    cudaMalloc(&d_b, n * sizeof(double));
16    cudaMalloc(&d_c, n * sizeof(double));
17
18    /* Inicializa vetores no host */
19    ...
20
21    /* Copia vetores do host para memória global da GPU */
22    cudaMemcpy(d_a, h_a, n * sizeof(double), cudaMemcpyHostToDevice);
23    cudaMemcpy(d_b, h_b, n * sizeof(double), cudaMemcpyHostToDevice);
24
25    /* Invoca kernel com 100 blocos de 1024 threads */
26    somaVetores<<<100, 1024>>>(d_a, d_b, d_c, n);
27
28    /* Copia resultados da memória global da GPU para o host */
29    cudaMemcpy(h_c, d_c, n * sizeof(double), cudaMemcpyDeviceToHost);
30
31    /* Imprime resultados */
32    ...
33
34    /* Libera espaço alocado para vetores na memória global da GPU */
35    cudaFree(d_a);
36    cudaFree(d_b);
37    cudaFree(d_c);
38
39    /* Libera espaço alocado para vetores na memória do host */
40    free(h_a);
41    free(h_b);
42    free(h_c);
43
44    return(0);
45 }
```

Figura 2.2: Programa em CUDA que executa no *host* para soma de dois vetores

da GPU (linhas 22 e 23). Essa transferência ocorre através do barramento PCI-Express, que conecta o *host* e a GPU.

O *host* então invoca um *kernel*, na linha 26, para a execução da soma dos dois vetores em paralelo na GPU. A GPU realiza o processamento e escreve o resultado em sua memória global. O *host* copia o vetor com o resultado da soma da memória global da GPU para a memória do *host* (linha 29). Por fim, os espaços alocados nas memórias da

GPU e do *host* são liberados.

Nesse exemplo o problema é modelado através de um *grid* unidimensional com 100 blocos de 1.024 *threads* cada um. Em outras palavras, o problema é dividido em 100 partições independentes que podem ser executadas concorrentemente. As *threads* de um mesmo bloco calculam os 1.024 elementos de cada partição, de forma concorrente às demais partições, visto que são operações independentes.

O *kernel* da Figura 2.3 é executado por cada uma das 102.400 *threads* do *grid* criado. Cada *thread* fica responsável por realizar a soma de um único par de elementos dos vetores de entrada. Cada *thread* determina a posição do vetor que ela deve acessar usando a identificação do bloco e da *thread* para compor o índice da posição no vetor (linha 6 no *kernel*).

```
1  /* Kernel: cada thread calcula um elemento do vetor 'c' */
2  __global__ void somaVetores(double *a, double *b, double *c, int n)
3  {
4     /* Calcula id da thread e
5        índice do elemento do vetor que será calculado pela thread */
6     int id = blockIdx.x * blockDim.x + threadIdx.x;
7
8     /* Verifica se limite do vetor não foi ultrapassado */
9     if (id < n)
10       c[id] = a[id] + b[id];
11 }
```

Figura 2.3: *Kernel* CUDA para soma de dois vetores

2.2 Arquitetura das GPUs CUDA

Atualmente, as GPU são acopladas ao *host* por um barramento PCI-Express [44] que em sua versão mais recente, a versão 3.0, pode alcançar velocidades de transferência de até 16 GB/s. Apesar de suas altas velocidades, o barramento PCI-Express possui alta latência, o que pode ser um gargalo na utilização de GPUs como dispositivos de computação paralela. Dessa forma, é necessário utilizar técnicas que mascarem essa latência e tornem possível o aproveitamento máximo da execução de tarefas na GPU [25].

A arquitetura desses dispositivos massivamente paralelos baseia-se na replicação de blocos físicos denominados SMs (*Streaming Multiprocessors*) que por sua vez possuem vários *cores* denominados SPs (*Streaming Processors*), isto é, SMs são blocos físicos *multithreaded* [4, 11, 25]. A Figura 2.4 mostra, de forma simplificada, a arquitetura de uma GPU, baseada no modelo de programação CUDA, com n SMs, cada SM com um determinado número de SPs. Os SPs de um SM podem ser divididos em grupos, onde todos os SPs de um mesmo grupo executam a mesma instrução, contudo, cada SP pode

operar dados diferentes segundo o modelo SIMD (*Single Instruction stream, Multiple Data streams*) de execução [11]. Pela taxonomia de Flynn [17], SIMD é a classe de arquitetura de computadores na qual a mesma operação é realizada simultaneamente em múltiplos conjuntos de dados.

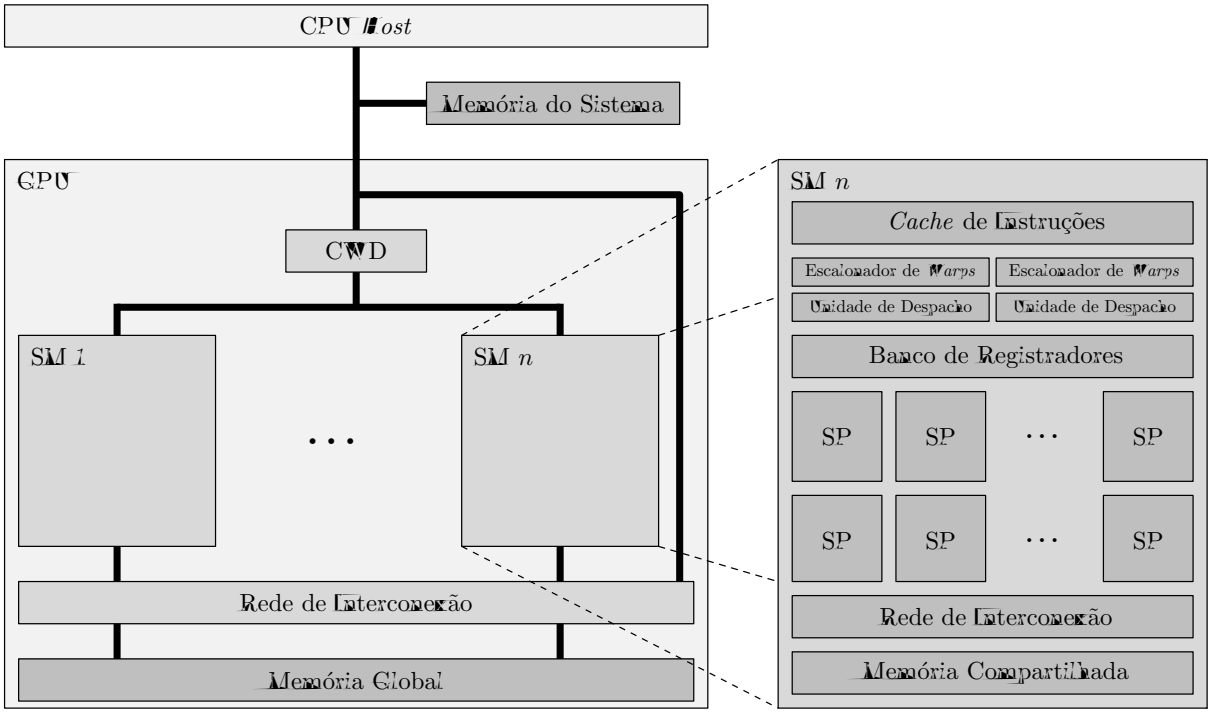


Figura 2.4: Arquitetura de uma GPU CUDA

Warp é a unidade básica para o escalonamento de trabalho dentro de um SM e é, na maioria das GPUs, um grupo de 32 *threads* de um bloco alocado a um grupo de 32 SPs de um SM, operando de forma SIMD. Como as *threads* de um *warp* executam em *lock-step*, o escalonador não precisa verificar por dependências no fluxo de instruções. Na Figura 2.4 é possível identificar dois escalonadores de *warps* e duas unidades de despacho, que permitem que dois *warps* sejam despachados e executados concorrentemente. Visto que cada SM é responsável pelo escalonamento dos seus recursos internos, a cada ciclo de *clock* os escalonadores de *warps* do SM decidem quais *warps* executarão em seguida [11].

Quando *threads* de um mesmo *warp* tomam diferentes caminhos de execução dentro de um *kernel* devido a desvios condicionais, criam-se as chamadas divergências. Esses caminhos são executados sequencialmente até que as *threads* juntem-se novamente em um mesmo caminho de execução. Portanto, trechos de código com divergências podem causar perda de desempenho, dado a redução do paralelismo na execução dos mesmos [48].

Quando um programa no modelo CUDA executando no *host* invoca um *kernel*, a unidade CWD (*Compute Work Distribution*) realiza o escalonamento dos blocos, isto é, ela enumera os blocos de *threads* do *grid* e inicia a distribuição deles aos SMs com

capacidade de execução disponíveis. As *threads* de um bloco executam concorrentemente em um SM. Ao fim da execução de um bloco, a unidade CWD atribui um novo bloco ao SM vago.

Portanto, a abstração em software de blocos de *threads* permite o mapeamento natural do *kernel* em uma GPU com um número arbitrário de SMs. Desse modo, blocos são um conjunto de *threads* que cooperam entre si, uma vez que apenas *threads* de um mesmo bloco podem compartilhar dados. Assim, essa abstração se torna uma expressão natural do paralelismo através do particionamento de um problema em múltiplas partições independentes que podem ser executadas concorrentemente. O escalonador pode dar a cada SM um ou mais blocos para serem executados, de acordo com a limitação do hardware da GPU [11].

Essa arquitetura é um fator chave na escalabilidade desses dispositivos, pois adicionando-se mais SMs em uma GPU pode-se executar mais tarefas ao mesmo tempo ou uma mesma tarefa mais rapidamente, se houver paralelismo suficiente para ser explorado nessa tarefa [4].

2.2.1 Hierarquia de Memórias

Uma GPU possui diferentes memórias, cada uma com características diferentes. Cada SM tem um barramento próprio de acesso à memória global da GPU e, consequentemente, à memória de textura e memória constante que são formas de endereçamento virtual dentro da memória global [4]. Cada bloco possui uma memória compartilhada, visível para todas as *threads* do bloco, e que tem o mesmo tempo de vida que o bloco. Por fim, as *threads* podem acessar dados de vários espaços de memória durante o seu tempo de execução. Cada *thread* possui uma memória local mapeada na memória global. Essa memória local é utilizada, no modelo CUDA, para as variáveis privadas da *thread* que não cabem nos registradores. Todas as *threads* de todos os blocos têm acesso à memória global.

A Figura 2.5 esboça a hierarquia de memórias de uma GPU segundo o modelo de programação CUDA e quais unidades possuem acesso a essas memórias. Todas as *threads* de todos os blocos têm acesso às memórias global, de textura e constante da GPU. Todas as *threads* do bloco 0 têm acesso à memória compartilhada do bloco 0, assim como as *threads* do bloco 1 têm acesso à memória compartilhada do bloco 1. Cada *thread* possui também acesso exclusivo à sua memória local.

Programas declaram variáveis na memória compartilhada e memória global do dispositivo através dos qualificadores ___*shared*___ e ___*device*___, respectivamente. Esses espaços de memória correspondem a memórias fisicamente separadas: a memória

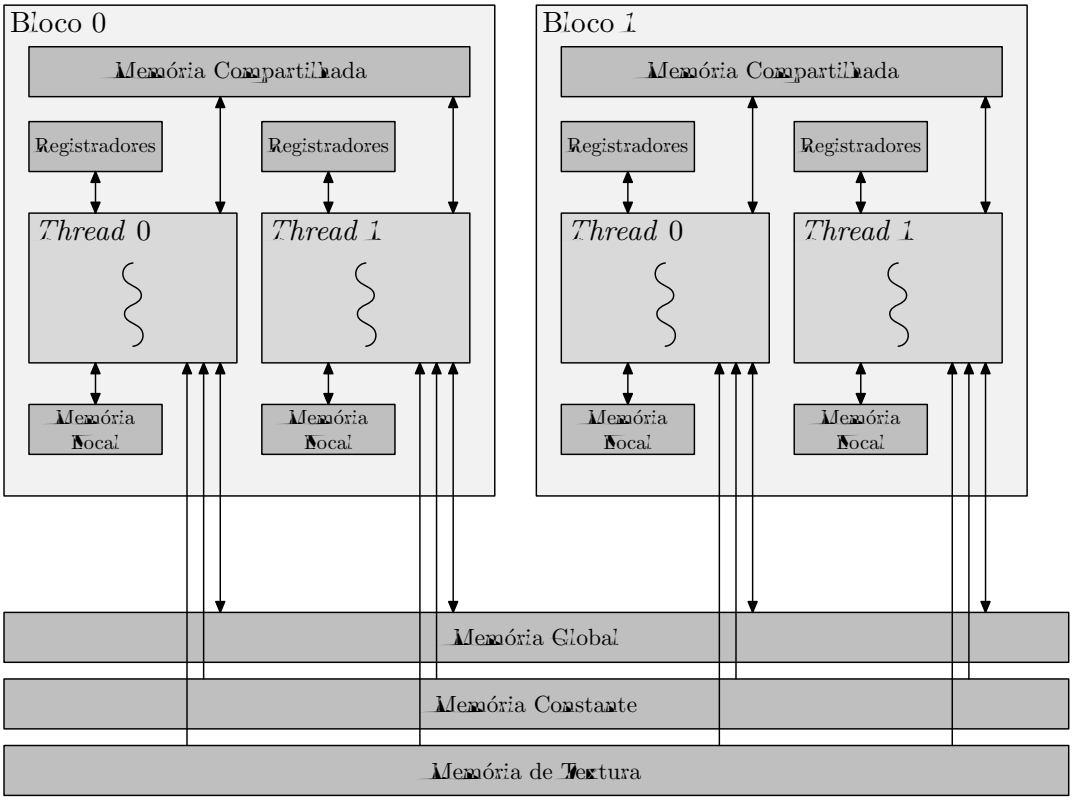


Figura 2.5: Hierarquia de memórias de uma GPU CUDA e unidades que possuem acesso às memórias

compartilhada por bloco possui baixa latência, enquanto a memória global possui uma latência maior [34]. A função da memória global é ser uma memória com alta largura de banda, embora com uma maior latência que a memória do *host* [25].

De maneira simplificada, a memória compartilhada da GPU desempenha um papel semelhante à *cache* L1 de uma CPU (embora tenha que ser programada manualmente pelo desenvolvedor), dada sua baixa latência e por estar próxima aos SPs. Portanto, ela pode prover um acesso rápido e compartilhamento de dados entre as *threads* de um mesmo bloco. Visto que essa memória tem o mesmo tempo de vida do seu bloco correspondente, o código do *kernel* normalmente inicializa dados em estruturas alocadas na memória compartilhada, faz cálculos usando essas estruturas e, ao final, copia os resultados da memória compartilhada para a memória global. Por sua vez, blocos de *threads* pertencentes a *grids* dependentes comunicam-se através da memória global, usando-a para ler dados de entrada e escrever os resultados [34].

2.2.1.1 Acesso Coalescido à Memória Global

Para alcançar um melhor desempenho durante leituras e escritas na memória global, *kernels* CUDA devem realizar transações coalescidas nessa memória. Acessos não

coalescidos à memória global geram mais de uma transação de memória, consumindo portanto um maior tempo de execução que os acessos coalescidos. As seguintes restrições devem ser respeitadas para que as leituras ou escritas realizadas pelas *threads* de um *warp* sejam coalescidas (considerando GPUs com *Compute Capability* até 1.1):

- As palavras acessadas pelas *threads* do *warp* devem ter ao menos 32 bits. Ler ou escrever bytes ou palavras de 16 bits são sempre transações não coalescidas;
- Os endereços acessados pelas *threads* do *warp* devem ser contíguos e crescentes, isto é, o *offset* entre eles deve ser positivo e baseado na identificação da *thread*;
- O endereço base do *warp*, isto é, o endereço acessado pela primeira *thread* do *warp* deve ser alinhado de acordo com a Tabela 2.1. Em outras palavras, o endereço base acessado deve ser múltiplo do valor na coluna Alinhamento da Tabela 2.1. Por exemplo, em uma transação em que as *threads* do *warp* acessam palavras de 32 bits, o endereço base deve ser múltiplo de 64 bytes.

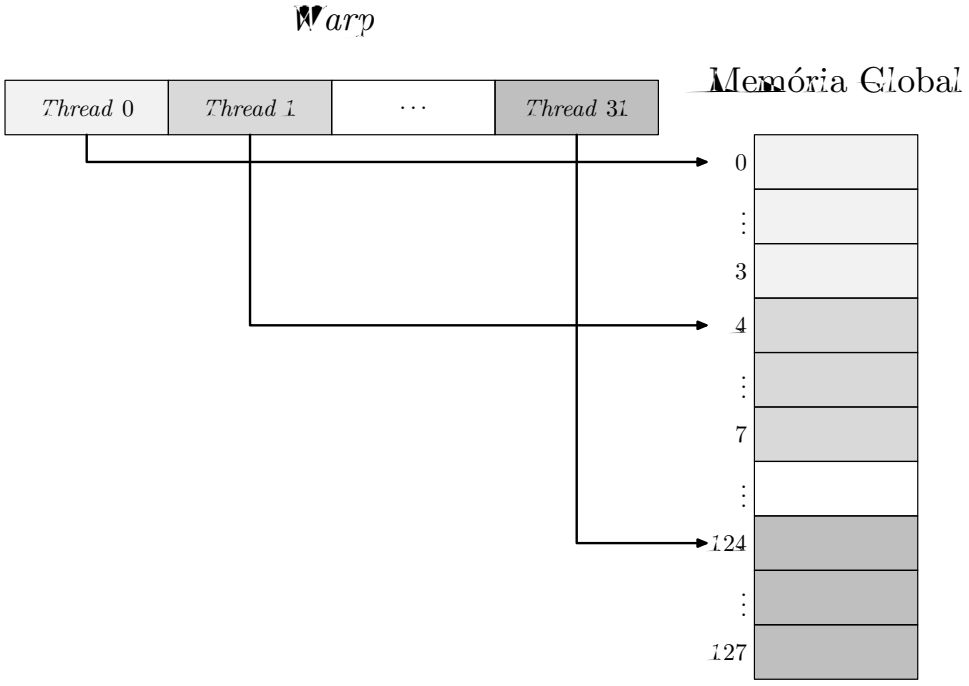
Tabela 2.1: Critério de alinhamento para transações coalescidas na memória global

Tamanho da palavra	Alinhamento
8 bits	Não coalescida
16 bits	Não coalescida
32 bits	64 bytes
64 bits	128 bytes
128 bits	256 bytes

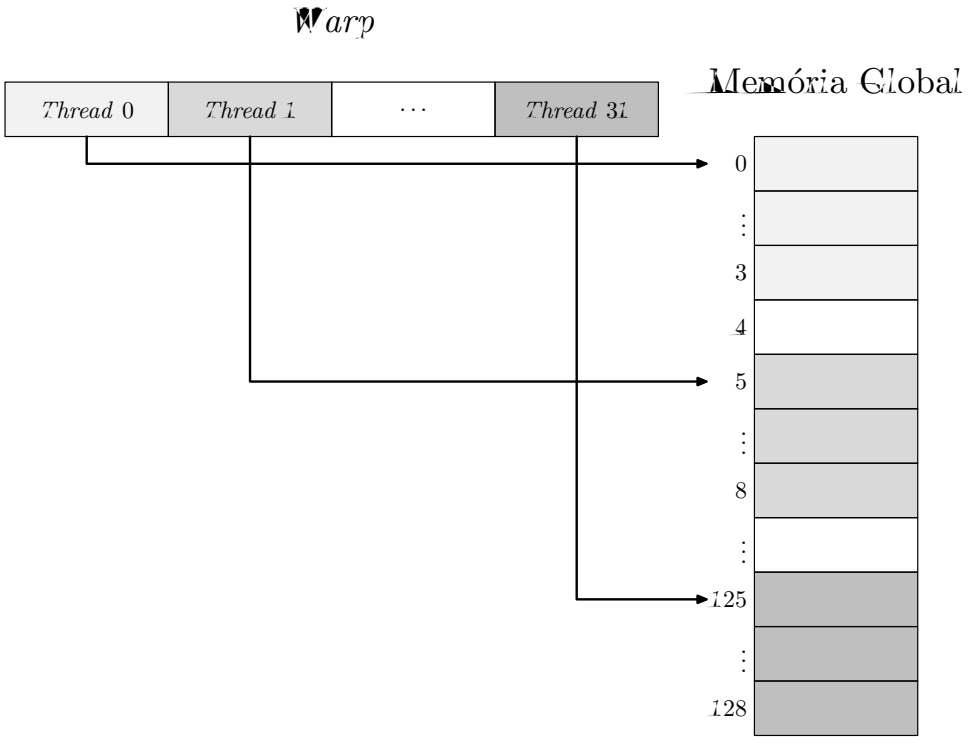
A Figura 2.6(a) mostra as *threads* de um *warp* acessando palavras de 32 bits na memória global. Os acessos geram uma transação coalescida pois todas as restrições são respeitadas: as palavras têm 32 bits os endereços acessados pelas *threads* são crescentes e contíguos e, por fim, o endereço base do *warp* está alinhado segundo a Tabela 2.1. Já no caso ilustrado na Figura 2.6(b), os acessos das *threads* não são coalescidos. Apesar de todas as demais restrições serem atendidas, os endereços acessados pelas *threads* não são contíguos. Há um intervalo entre o endereço acessado pela *thread* 0 e a *thread* 1.

2.3 Sincronização em CUDA

Um programa pode executar vários *grids* de forma independente ou dependente. *Grids* independentes podem executar concorrentemente se houver recursos de hardware suficientes. Por sua vez, *grids* dependentes executam sequencialmente, com uma barreira implícita inserida entre eles, garantindo assim que todos os blocos do primeiro *grid*



(a)



(b)

Figura 2.6: (a) Acessos coalescidos à memória global e (b) acessos não coalescidos à memória global

terminarão de executar antes de qualquer bloco do segundo *grid*, dependente, seja executado.

A execução em paralelo e o gerenciamento das *threads* são feitos de forma automática. Toda criação de *thread*, escalonamento e finalização são realizados pelo hardware para o programador [4].

2.3.1 Sincronização de *Threads*

As *threads* de um bloco são executadas concorrentemente e podem sincronizar-se em uma barreira através do comando *___syncthreads()*. Isto garante que nenhuma das *threads* participantes da barreira pode prosseguir até que todas atinjam a barreira, assegurando às mesmas a visualização de todas as escritas realizadas na memória. Dessa forma, *threads* em um bloco podem se comunicar através da escrita e leitura na memória compartilhada do bloco com uma barreira de sincronização para evitar condições de corrida [4].

2.3.2 *Streams*

No modelo CUDA, um *stream* é uma fila de operações que são executadas em uma ordem definida. Essas operações podem ser a invocação de *kernels*, transferências de dados entre *host* e GPU e o início ou parada de eventos. A ordem na qual as operações são adicionadas no *stream* é a ordem na qual elas serão executadas pela GPU. Assim, cada *stream* é um conjunto de operações que a GPU deve realizar em uma ordem definida. Uma operação de um *stream* só é iniciada após todas as operações anteriores daquele *stream* terem terminado. Entretanto, múltiplos *streams* podem ser executados em paralelo [48].

O uso de múltiplos *streams* pode tornar o código mais eficiente. Algumas GPUs mais novas permitem a execução em paralelo de *kernels*, de até duas transferências (uma do *host* para a GPU e outra da GPU para o *host*) e de *kernels* e transferências entre *host* e GPU. Sobrepor essas operações reduz o tempo total de execução do programa.

A Figura 2.7 contrasta a execução de um conjunto de operações pela GPU sem e com o uso de múltiplos *streams*. No programa executado na Figura 2.7(a), um único *stream* é utilizado, logo não há sobreposição na execução das operações. Na Figura 2.7(b), o programa executado utiliza dois *streams* independentes. Enquanto o *kernel* de um *stream* executa na GPU, uma transferência *host*-GPU do outro *stream* é realizada. Com a sobreposição na execução de operações dos dois *streams*, esta solução resulta em um tempo de execução menor que aquele obtido com um único *stream*.

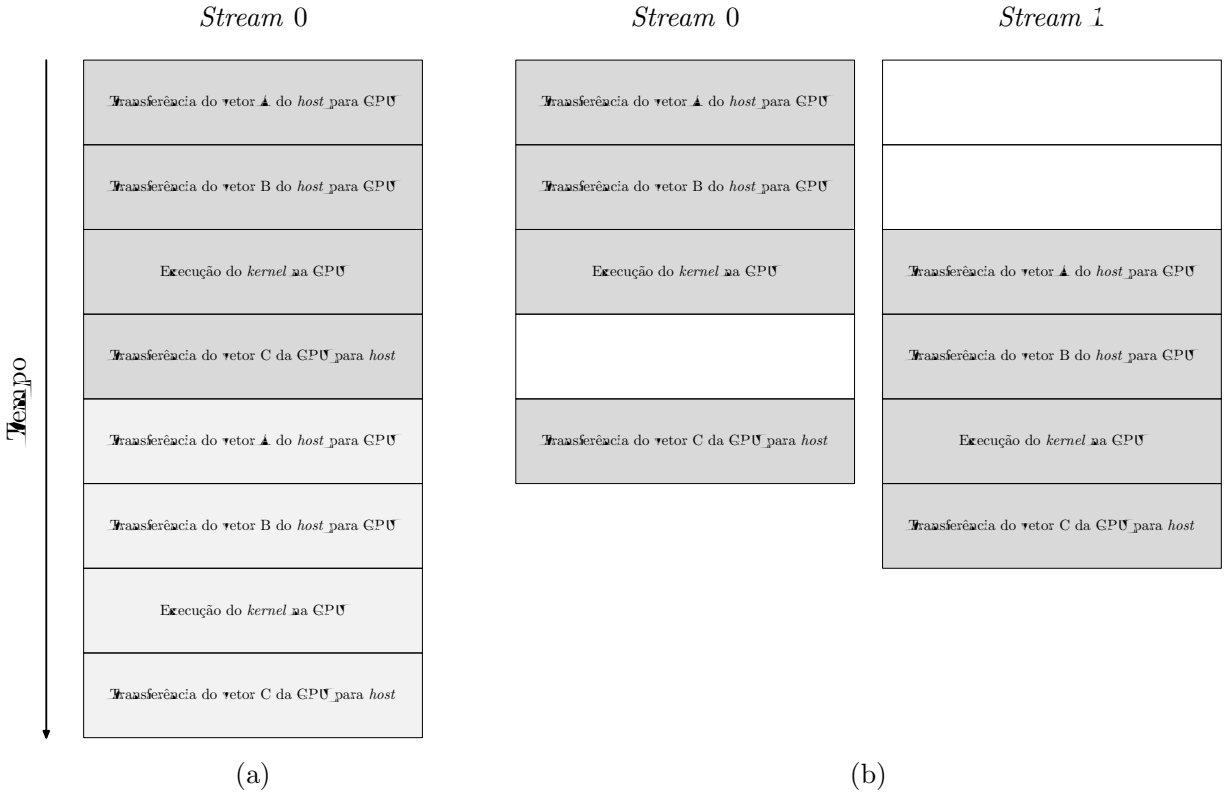


Figura 2.7: (a) Execução de programa com um único *stream*; (b) execução de programa com dois *streams*, sobrepondo operações e reduzindo o tempo total de execução

2.4 Escalabilidade do Modelo de Programação CUDA

O desempenho e a escalabilidade de um programa desenvolvido usando o modelo de programação CUDA depende do particionamento do trabalho realizado pelo programa em blocos de tamanho fixo durante a modelagem do problema. Esses blocos fazem o mapeamento entre o paralelismo do problema em questão e o hardware da GPU [11].

Visto que as *threads* de um bloco podem compartilhar a memória compartilhada e sincronizar-se através de barreiras, elas residirão no mesmo SM. Contudo, o número de blocos pode exceder a capacidade dos SMs de uma GPU, o que dá ao programador flexibilidade para paralelizar na granularidade que for mais conveniente. Isso permite a decomposição de problemas de forma intuitiva, uma vez que o número de blocos pode ser ditado pelo tamanho dos dados a serem processados e não pela quantidade de SMs no dispositivo. Portanto, um mesmo programa pode ser executado em GPUs com quantidades diferentes de SMs, assim possibilitando o ganho de desempenho na execução desse programa, sem modificações, em gerações de GPUs mais novas e com mais recursos de hardware [34].

Para gerenciar o escalonamento de blocos em SMs e fornecer escalabilidade, o modelo CUDA requer que os blocos executem de forma independente. Deve ser possível a execução dos blocos em qualquer ordem, em paralelo ou em série. Diferentes blocos não têm nenhum meio de comunicação direta, embora possam coordenar suas atividades com operações atômicas na memória global, visíveis a todas as *threads* [34]. Em suma, a abstração de blocos e a replicação de SMs no hardware trabalham em conjunto para fornecer escalabilidade de forma transparente [11].

3 Comparação Sequência-Família

Novas sequências biológicas evoluem a partir de sequências preexistentes, o que facilita a análise computacional das novas sequências. Muitas vezes é possível reconhecer uma semelhança significativa entre uma nova sequência e uma sequência biológica já conhecida e assim transferir informações sobre a estrutura e/ou função da sequência conhecida para a nova sequência. As duas sequências relacionadas são então ditas homólogas e a transferência de informações entre elas é feita por homologia [7].

À primeira vista, determinar se duas sequências biológicas são semelhantes não é muito diferente de determinar se duas cadeias de caracteres são semelhantes. Um conjunto de métodos para análise de sequências biológicas é, portanto, enraizado na Ciência da Computação, onde há uma extensa literatura sobre métodos de comparação de cadeias de caracteres [7].

3.1 Sequência Biológica e Família de Sequências

Uma sequência biológica pode ser definida como uma cadeia finita de símbolos que carregam informações biológicas. Estes símbolos pertencem a um alfabeto formado por um conjunto finito de símbolos [7]. Em geral, os alfabetos utilizados são [8]:

- $\Sigma = \{A, C, G, T\}$, para sequências de DNA, portanto composto por quatro símbolos que representam nucleotídeos;
- $\Sigma = \{A, C, G, U\}$, para sequências de RNA, portanto composto por quatro símbolos que representam nucleotídeos;
- $\Sigma = \{A, C, D, E, F, G, H, I, K, L, M, N, P, Q, R, S, T, V, W, Y\}$, para sequências de proteínas, portanto composto por vinte símbolos que representam aminoácidos.

Assim, uma sequência biológica pode ser formalmente definida como uma palavra S, tal que, $S \in \Sigma^+$.

Por sua vez, uma família de sequências é um conjunto finito de sequências homólogas. As sequências membros de uma família de sequências divergiram durante a evolução e compartilham semelhanças estruturais e/ou funcionais [45].

3.2 Alinhamento Múltiplo de Sequências e *Profile* HMM

O alinhamento múltiplo de sequências é o alinhamento de três ou mais sequências biológicas e consiste na inserção de *gaps* (representados por hifens '–') nas sequências com o objetivo de deixá-las com o mesmo tamanho e criar uma correspondência entre os símbolos das mesmas. Um alinhamento múltiplo pode ser representado por uma matriz, onde cada linha corresponde a uma sequência biológica e cada coluna tem os símbolos alinhados das sequências ou os *gaps* adicionados para alinhá-las [45]. A Figura 3.1 mostra o alinhamento múltiplo de cinco sequências de DNA.

$$\begin{bmatrix} A & A & G & A & A \\ A & T & - & A & A \\ C & T & G & - & G \\ C & C & - & A & G \\ C & C & G & - & G \end{bmatrix}$$

Figura 3.1: Alinhamento múltiplo de cinco sequências de DNA

Podem existir vários alinhamentos múltiplos distintos para um mesmo conjunto de sequências. De fato, várias funções e métodos para a obtenção do alinhamento múltiplo ótimo vêm sendo propostos na literatura. Por exemplo, para construir alinhamentos múltiplos de sequências biológicas é usual levar em consideração a relação evolucionária entre as sequências [45].

Dado o problema de determinar se uma nova sequência biológica é homóloga a uma família de sequências conhecida, é necessária uma forma de representar a família de sequências [7]. As variações em uma família de sequências podem ser descritas estatisticamente e essa é a base para a maioria dos métodos de análise de sequências biológicas atualmente [26]. Um *profile* é um modelo estatístico que mantém informações sobre uma família de sequências e descreve as variações entre os membros da família [31]. Krogh *et al.* [27] apresentam uma representação que usa Modelos Ocultos de Markov para descrever uma família de sequências e as relações entre seus membros.

Um *profile* HMM (*Hidden Markov Model* – Modelo Oculto de Markov) é um modelo estatístico baseado no alinhamento múltiplo das sequências de uma família e captura informações especificas de cada posição, representando o quão conservada cada coluna do alinhamento múltiplo está e quais símbolos são mais prováveis. Em outras palavras, um *profile* HMM estende o conceito de *profile* e modela uma família de sequências representando as similaridades entre seus membros na forma de estados discretos [31]. Uma

vantagem do uso de *profile* HMMs está no fato de que, quanto mais sequências da família forem usadas para construir o *profile* HMM, mais preciso ele se tornará [29].

3.3 Alinhamento Sequência-Família

Ao comparar quaisquer dois membros de uma família de sequências, as semelhanças fracas que abrangem toda a família são suscetíveis ao ofuscamento pelas semelhanças mais fortes entre os dois membros em particular. Para detectar semelhanças que abranjam uma família de sequências inteira é, portanto, aconselhável a utilização de outros métodos que não apenas comparações dos membros aos pares [45]. Nesse contexto, abordagens probabilísticas mais sofisticadas são usadas na área de Bioinformática.

Um *profile* HMM pode ser usado para procurar novos membros da família de sequências em uma base de dados de sequências biológicas [26]. A probabilidade de uma sequência biológica S ser homóloga a uma família de sequências modelada por um *profile* HMM λ é então determinada solucionando-se o problema da busca da melhor sequência de estados de λ para a observação da sequência S, ou seja, é o maior valor da probabilidade $P(S|\lambda)$. Esse valor mensura a similaridade entre S e a família de sequências modelada e é denominado *score*. Portanto, pode-se concluir que uma nova sequência é membro da família de sequências representada pelo *profile* HMM se o *score* obtido for significativo.

A arquitetura de *profile* HMM utilizada na versão 3 da ferramenta HMMER é *multi-hit local alignment* [9] e pode ser vista na Figura 3.2. Esse HMM é construído a partir do alinhamento múltiplo das sequências da família modelada e consiste em repetidos estados de *Match*, *Insert*, *Delete* e ainda estados *Start*, N, *Begin*, *Joining*, *End*, C e *Termination*. Os estados *Match*, *Insert*, *Delete* formam o núcleo do modelo e correspondem às colunas do alinhamento múltiplo. Os estados do *profile* HMM possuem as seguintes funções:

- Estado *Start* (S): estado inicial e mudo, isto é, que não emite símbolos, por onde começa o alinhamento da sequência sendo analisada com a família;
- Estado N: permite o descarte de símbolos do início da sequência sendo analisada;
- Estado *Begin* (B): estado mudo que permite a entrada no núcleo do modelo;
- Estado *Match* (M): corresponde a uma coluna do alinhamento múltiplo e permite que um símbolo da sequência sendo analisada seja emitido, isto é, alinhado com essa coluna;
- Estado *Insert* (I): permite a emissão de símbolos da sequência sendo analisada que não alinham com uma coluna do alinhamento múltiplo;

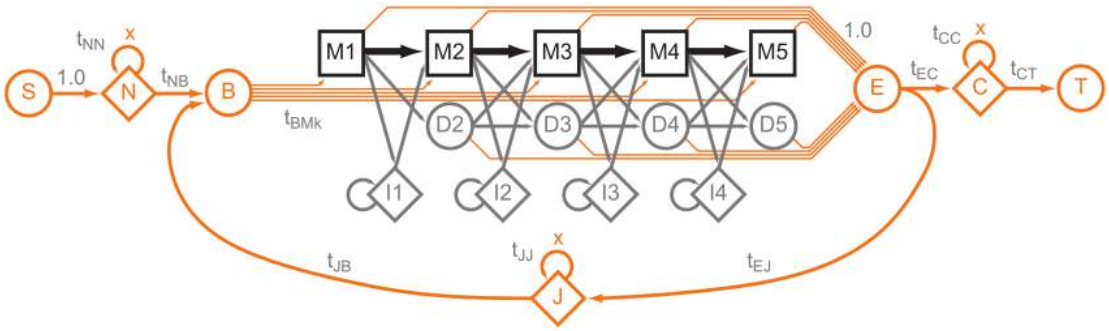


Figura 3.2: Arquitetura de *profile* HMM *multi-hit local alignment* usada pelo HMMER3 [9] no algoritmo de Viterbi

- Estado *Delete* (D): permite que a sequência sendo analisada atravesse colunas do alinhamento múltiplo sem emitir símbolos;
- Estado *Joining* (J): permite o alinhamento *multi-hit*, através de uma ligação do final do modelo com o seu início, denominada *feedback loop*, podendo descartar símbolos antes de alinhar um novo segmento da sequência sendo analisada com o alinhamento múltiplo da família;
- Estado *End* (E): estado mudo que permite a saída do núcleo do modelo;
- Estado C: permite o descarte de símbolos do final da sequência sendo analisada;
- Estado *Termination* (T): estado final e mudo que é atingido quando toda a sequência sendo analisada foi alinhada e nele é obtido o valor final do *score* do alinhamento da sequência com a família.

Um conjunto de estados M_j, I_j e D_j forma um nó do *profile* HMM e o número total de nós é considerado o tamanho do *profile* HMM. Outras arquiteturas de *profile* HMMs, como por exemplo a arquitetura Plan 7, também foram propostas [29, 31, 42].

O alinhamento de uma sequência com um *profile* HMM que representa uma família de sequências pode acontecer das seguintes formas:

- Alinhamento global: ocorre quando os símbolos da sequência começam a ser alinhados a partir do primeiro estado *Match* e terminam no último estado *Match*. Dada a sequência $S = CTGAG$, o alinhamento $(S) \rightarrow (N, -) \rightarrow (B) \rightarrow (M_1, C) \rightarrow (M_2, T) \rightarrow (M_3, G) \rightarrow (M_4, A) \rightarrow (M_5, G) \rightarrow (E) \rightarrow (C, -) \rightarrow (T)$ é um exemplo de alinhamento global de S com o *profile* HMM da Figura 3.2. Este alinhamento indica uma sequência de estados do *profile* HMM e o símbolo de S emitido em cada estado;

- Alinhamento local em relação à sequência: ocorre quando alguns símbolos da sequência são descartados antes do primeiro e/ou após o último estado *Match*. Dada a sequência $S = CTGAGTTT$, o alinhamento $(S) \rightarrow (N, -) \rightarrow (B) \rightarrow (M_1, C) \rightarrow (M_2, T) \rightarrow (M_3, G) \rightarrow (M_4, A) \rightarrow (M_5, G) \rightarrow (E) \rightarrow (C, T) \rightarrow (C, T) \rightarrow (C, T) \rightarrow (T)$ é um exemplo de alinhamento de S com o *profile* HMM da Figura 3.2, alinhamento esse local em relação à sequência S;

- Alinhamento local em relação ao *profile* HMM: ocorre quando os símbolos da sequência começam a ser alinhados a partir de um estado *Match* intermediário e/ou terminam de ser alinhados em um estado *Match* intermediário. Dada a sequência $S = CTGA$, o alinhamento $(S) \rightarrow (N, -) \rightarrow (B) \rightarrow (M_1, C) \rightarrow (M_2, T) \rightarrow (M_3, G) \rightarrow (M_4, A) \rightarrow (E) \rightarrow (C, -) \rightarrow (T)$ é um exemplo de alinhamento de S local em relação ao *profile* HMM da Figura 3.2;

- Alinhamento *multi-hit*: ocorre quando os símbolos da sequência são alinhados repetidas vezes com o núcleo do *profile* HMM. Dada a sequência $S = CTGAGCTGAG$, o alinhamento $(S) \rightarrow (N, -) \rightarrow (B) \rightarrow (M_1, C) \rightarrow (M_2, T) \rightarrow (M_3, G) \rightarrow (M_4, A) \rightarrow (M_5, G) \rightarrow (E) \rightarrow (J, -) \rightarrow (B) \rightarrow (M_1, C) \rightarrow (M_2, T) \rightarrow (M_3, G) \rightarrow (M_4, A) \rightarrow (M_5, G) \rightarrow (E) \rightarrow (C, -) \rightarrow (T)$ é um exemplo de alinhamento *multi-hit* de S com o *profile* HMM da Figura 3.2. Para obter um alinhamento *multi-hit*, é necessário que o HMM possua um estado *Joining* que permite que, após o último nó do HMM, se retorne para antes do primeiro nó.

3.4 Ferramenta HMMER

A construção e o uso da base de dados de família de sequências Pfam [13] estão relacionados à ferramenta HMMER [42]. HMMER [19] é um conjunto de programas para análise de sequências biológicas implementados em código aberto e vastamente adotado para análise de sequências em larga escala. Um de seus programas mais importante é denominado *hmmsearch*. Seu objetivo é encontrar sequências biológicas homólogas a uma família de sequências conhecida, utilizando um algoritmo denominado algoritmo de Viterbi. O programa tem como entrada um conjunto de sequências biológicas a serem investigadas e um *profile* HMM representando a família e calcula, para cada sequência, o *score* de similaridade da sequência em relação ao *profile* HMM e se o *score* for significativo, conclui-se que a sequência é homóloga à família. Dada a complexidade quadrática do algoritmo de Viterbi, esse procedimento pode demorar bastante, dependendo da quantidade de sequências, do tamanho do HMM e do hardware utilizado [42].

Nos *profile* HMMs provenientes da base Pfam todas as probabilidades são convertidas para valores logarítmicos. Dessa forma o cálculo do *score* é obtido realizando-se

a soma de valores e não mais a multiplicação, o que torna o processo menos custoso [8].

A partir de sua versão 3, o HMMER adota uma nova estratégia para cálculo do *score* de uma sequência: as sequências investigadas passam através de uma sequência de filtros, onde cada filtro calcula um *score* para a sequência utilizando um algoritmo diferente e dependendo do *score*, descarta a sequência ou encaminha-a para o próximo filtro. A ideia é que os filtros iniciais, que recebem muitas sequências, utilizem algoritmos mais rápidos. Portanto, mesmo que não classifiquem todas as sequências com precisão, esses filtros são capazes de descartar uma grande parte das sequências sendo investigadas, deixando passar para os próximos filtros apenas uma pequena parte, que será analisada por algoritmos mais lentos e precisos.

A Figura 3.3 mostra a sequência de filtros do HMMER3, que permite que o mesmo seja de 100 a 1000 vezes mais rápido que seu predecessor [9]. O primeiro filtro utiliza o algoritmo MSV (*Multiple Segment Viterbi*) [8] e consome 75% do tempo total de execução da solução, dado que todas as sequências investigadas passam por ele. Portanto, pela Lei de Amdahl [17], esforços para o ganho de desempenho devem ser focados nesse algoritmo. Como será visto nas próximas seções, o algoritmo empregado no filtro MSV é uma simplificação do algoritmo de Viterbi com a eliminação de várias dependências de dados, dependências essas que quando removidas possibilitam a exploração de paralelismo na execução do algoritmo.

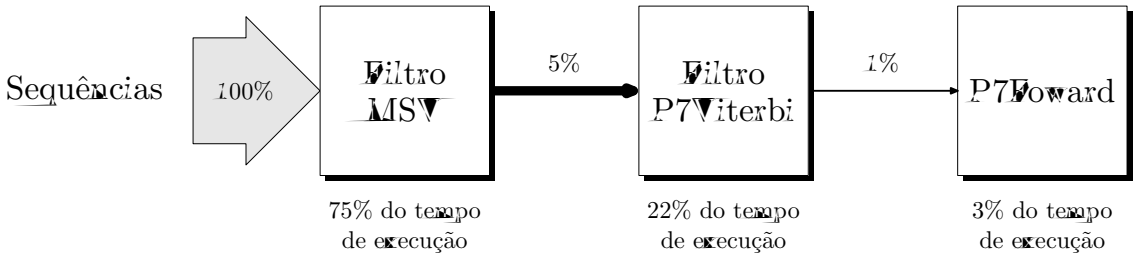


Figura 3.3: Sequência de filtros da ferramenta HMMER3 [1]

Posteriormente, em uma atualização menor que gerou sua versão 3.1*b* [10], o HMMER inseriu mais um filtro no início da sequência de filtros. Esse novo filtro utiliza o algoritmo SSV (*Single Segment* Viterbi). A adição desse novo filtro proporcionou resultados de desempenho até duas vezes mais rápidos que a sequência de filtros anterior [19].

3.5 Algoritmo de Viterbi

Ao encontrar um caminho através do *profile* HMM no qual a sequência sendo analisada se encaixa bem, isto é, ao alinhar a sequência com o *profile* HMM, obtém-se o

grau de similaridade entre a sequência e a família de sequências representada pelo *profile* HMM [26]. Por vezes é possível que haja mais de um alinhamento para uma mesma sequência [26, 31]. Por exemplo, a sequência $S = CCGA$ pode ser alinhada com o *profile* HMM da Figura 3.2 ao menos das seguintes duas maneiras:

- $(S) \rightarrow (N, -) \rightarrow (B) \rightarrow (M_1, C) \rightarrow (M_2, C) \rightarrow (M_3, G) \rightarrow (D_4, -) \rightarrow (M_5, A) \rightarrow (E) \rightarrow (C, -) \rightarrow (T)$;
- $(S) \rightarrow (N, -) \rightarrow (B) \rightarrow (M_1, C) \rightarrow (M_2, C) \rightarrow (D_3, -) \rightarrow (M_4, G) \rightarrow (M_5, A) \rightarrow (E) \rightarrow (C, -) \rightarrow (T)$.

Contudo, o objetivo é encontrar o melhor alinhamento da sequência com o *profile* HMM, isto é, o alinhamento com maior probabilidade. Embora haja um enorme número de possíveis alinhamentos, esse cálculo pode ser feito de maneira eficiente por um algoritmo de programação dinâmica [5] denominado algoritmo de Viterbi. O algoritmo encontra o melhor alinhamento e retorna sua probabilidade, denominada *score* [26, 31, 42].

A técnica de programação dinâmica utilizada no algoritmo de Viterbi otimiza o desempenho no processo de alinhamento mantendo o *score* parcial da sequência sendo analisada e assim minimizando o número de novos cálculos. Dados um *profile* HMM de Q nós e uma sequência S de comprimento L, três matrizes (de dimensão $L \times Q$) de programação dinâmica são utilizadas no algoritmo:

- Matriz de *scores* M: mantém os *scores* dos estados *Match* do *profile* HMM, de tal maneira que $M[i, j]$ contém o *score* do melhor alinhamento que emite os $i - 1$ símbolos iniciais de S e que alcança o estado M_j, responsável por emitir o i-ésimo símbolo de S;
- Matriz de *scores* I: mantém os *scores* dos estados *Insert* do *profile* HMM, de tal maneira que $I[i, j]$ contém o *score* do melhor alinhamento que emite os $i-1$ símbolos iniciais de S e que alcança o estado I_j, responsável por emitir o i-ésimo símbolo de S;
- Matriz de *scores* D: mantém os *scores* dos estados *Delete* do *profile* HMM, de tal maneira que $D[i, j]$ contém o *score* do melhor alinhamento que emite os i símbolos iniciais de S e que alcança o estado D_j. Diferentemente das outras matrizes, na matriz D, quando se calcula o *score* $D[i, j]$ o i-ésimo símbolo de S já foi emitido.

Além disso, cinco vetores (de tamanho L) de programação dinâmica, N, E, J, B e C são utilizados para o cálculos dos *scores*. Suas funções são:

- N, J e C: mantêm os *scores* dos estados N, *Joining* e C do *profile* HMM, de tal maneira que $N[i]$, $J[i]$ e $C[i]$ contêm os *scores* do melhor alinhamento que emite os

$i-1$ símbolos iniciais de S e que alcança os estados N, *Joining* e C, respectivamente, responsável por emitir o i-ésimo símbolo de S;

- B e E: mantêm os *scores* dos estados *Begin* e *End* do *profile* HMM, de tal maneira que $B[i]$ e $E[i]$ contêm os *scores* do melhor alinhamento que emite os i símbolos iniciais de S e que alcança os estados B e E, respectivamente.

O algoritmo de Viterbi aplicado a *profile* HMMs com arquitetura *multi-hit local alignment* é apresentado no Algoritmo 3.1 e encontra o *score* do melhor alinhamento de uma sequência S de comprimento L com um *profile* HMM de Q nós. Esse algoritmo possui complexidade de tempo $O(L \times Q)$, dados seus dois laços aninhados.

Algoritmo 3.1 Algoritmo de Viterbi

Entradas: *Profile* HMM de Q nós
Probabilidades P_{em} de emissão de símbolos por estados
Probabilidades P_{tr} de transição entre estados
Sequência S com L símbolos
Saída: *Score* do melhor alinhamento de S com o *profile* HMM

1: **for** $i \leftarrow 1$ **to** L **do**
2: **for** $j \leftarrow 1$ **to** Q **do**
3: $M[i,j] \leftarrow P_{em}(M_j, S_i) + max \begin{cases} M[i-1,j-1] + P_{tr}(M_{j-1}, M_j) \\ I[i-1,j-1] + P_{tr}(I_{j-1}, M_j) \\ D[i-1,j-1] + P_{tr}(D_{j-1}, M_j) \\ B[i-1] + P_{tr}(B, M_j) \end{cases}$
4: $I[i,j] \leftarrow P_{em}(I_j, S_i) + max \begin{cases} M[i-1,j] + P_{tr}(M_j, I_j) \\ I[i-1,j] + P_{tr}(I_j, I_j) \end{cases}$
5: $D[i,j] \leftarrow max \begin{cases} M[i,j-1] + P_{tr}(M_{j-1}, D_j) \\ D[i,j-1] + P_{tr}(D_{j-1}, D_j) \end{cases}$
6: **end for**
7: $N[i] \leftarrow N[i-1] + P_{tr}(N, N)$
8: $E[i] \leftarrow max_{1 \leq j \leq Q}(M[i,j] + P_{tr}(M_j, E))$
9: $J[i] \leftarrow max \begin{cases} J[i-1] + P_{tr}(J, J) \\ E[i] + P_{tr}(E, J) \end{cases}$
10: $B[i] \leftarrow max \begin{cases} N[i] + P_{tr}(N, B) \\ J[i] + P_{tr}(J, B) \end{cases}$
11: $C[i] \leftarrow max \begin{cases} C[i-1] + P_{tr}(C, C) \\ E[i] + P_{tr}(E, C) \end{cases}$
12: **end for**
13: **return** $C[L] + P_{tr}(C, T)$

Posto o intuito de encontrar formas de se explorar paralelismo, uma análise do algoritmo de Viterbi permite identificar dependências de dados entre os elementos das estruturas de dados. A Figura 3.4 mostra os vetores e matrizes do algoritmo de Viterbi, com as matrizes M, I e D mostradas sobrepostas e as dependências de dados representadas por setas, indicando quais elementos são necessários para o cálculo de outro elemento.

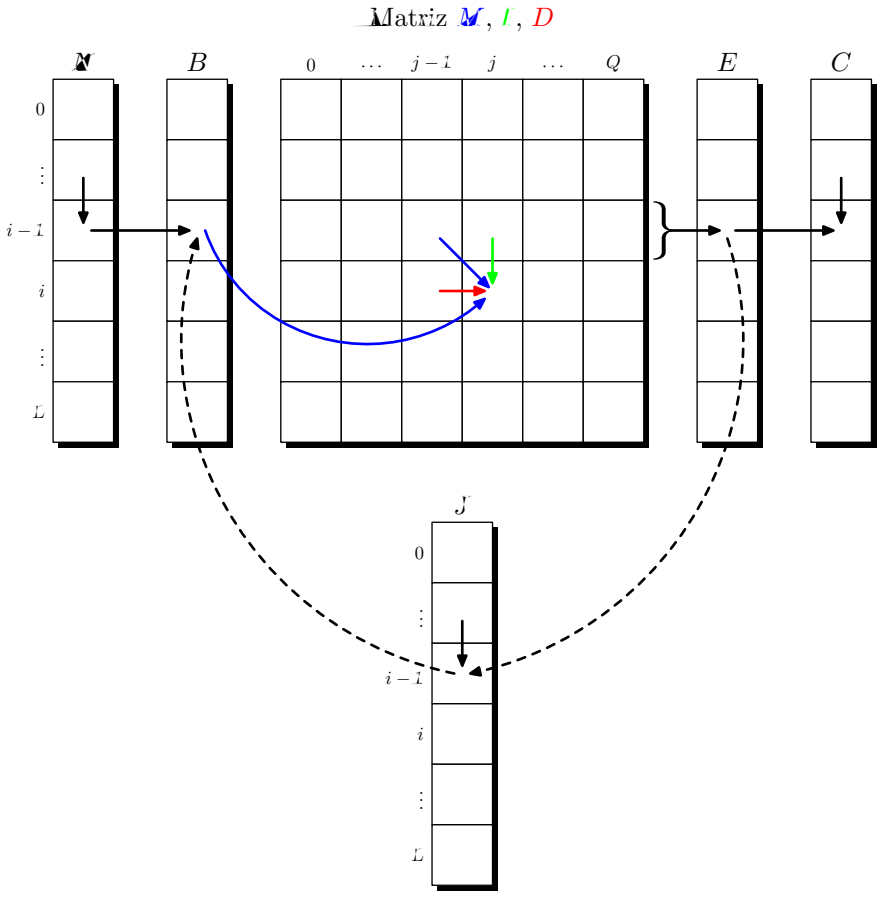


Figura 3.4: Dependências de dados do algoritmo de Viterbi

Por exemplo, a partir das linhas 3, 4 e 5 do Algoritmo 3.1, obtém-se as dependências de dados que impõem que, para o cálculo das células $M[i, j]$, $I[i, j]$ e $D[i, j]$, os valores das células $M[i-1, j-1]$, $I[i-1, j]$ e $D[i, j-1]$ são necessários, respectivamente. Essas dependências diretas impossibilitam a exploração de paralelismo no cálculo de todas as células de uma mesma linha das matrizes, ou das células de uma mesma coluna, ou das células de uma mesma diagonal.

As dependências de dados representadas pelas setas tracejadas na Figura 3.4 são oriundas das linhas 9 e 10 do Algoritmo 3.1 e estão associadas ao estado *Joining* que liga o final do núcleo do modelo ao seu início (*feedback loop*). Essa ligação cria uma cadeia de dependências $M[i-1, 1 \ldots Q] \rightarrow E[i-1] \rightarrow J[i-1] \rightarrow B[i-1] \rightarrow M[i, j]$ que impõe que, para o cálculo da célula $M[i, j]$, os valores das células $M[i-1, 1], \ldots, M[i-1, Q]$ sejam necessários. Essa dependência indireta impossibilita a exploração de paralelismo no cálculo de todas as células de uma mesma anti-diagonal das matrizes.

Em conjunto, as dependências de dados impossibilitam a exploração de paralelismo no cálculo de quaisquer duas células das matrizes de programação dinâmica. Contudo, algumas soluções que contornam esse impedimento (por exemplo, não permitindo

alinhamentos *multi-hit*, isto é, que usem o *feedback loop*) e com isso conseguem explorar paralelismo no cálculo das células de uma mesma anti-diagonal foram apresentadas na literatura [3, 29, 50].

3.6 Algoritmo MSV

O algoritmo MSV (*Multiple Segment* Viterbi) é uma simplificação do algoritmo de Viterbi e se aplica a *profile* HMMs com a arquitetura *multi-hit ungapped local alignment*, mostrada na Figura 3.5. Essa arquitetura é obtida a partir da arquitetura *multi-hit local alignment* apresentada na Figura 3.2 e usada pelo algoritmo de Viterbi, simplesmente removendo os estados I e D e considerando as transições $M \rightarrow M$ como tendo probabilidade 1 [9].

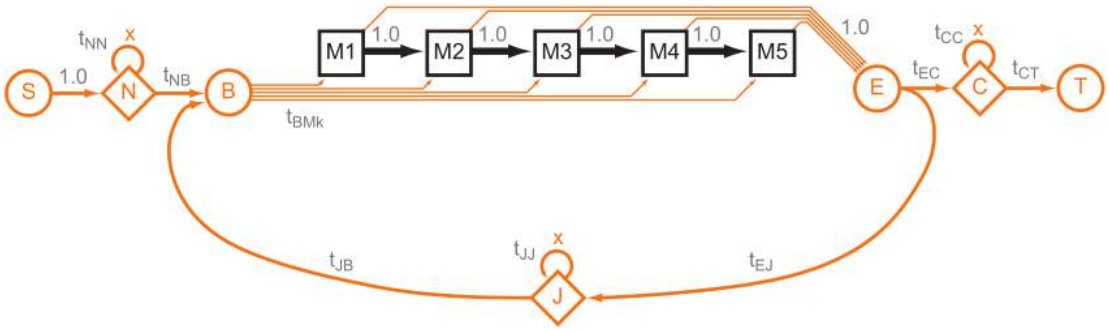


Figura 3.5: Arquitetura de *profile* HMM *multi-hit ungapped local alignment* usada pelo HMMER3 no algoritmo MSV [9]

O Algoritmo 3.2 apresenta o algoritmo MSV que encontra o *score* do melhor alinhamento de uma sequência S de comprimento L com um *profile* HMM de Q nós. Sua complexidade de tempo é $O(L \times Q)$.

Novamente, posto o intuito de encontrar formas de se explorar paralelismo, uma análise do algoritmo MSV permite identificar as dependências de dados, mostradas na Figura 3.6. Entretanto, nesse caso há a possibilidade de se explorar paralelismo no cálculo das células de uma mesma linha da matriz M.

Algoritmo 3.2 Algoritmo MSV

Entradas: *Profile* HMM de Q nós
Probabilidades P_{em} de emissão de símbolos por estados
Probabilidades P_{tr} de transição entre estados
Sequência S com L símbolos

Saída: *Score* do melhor alinhamento de S com o *profile* HMM

1: **for** $i \leftarrow 1$ **to** L **do**
2: **for** $j \leftarrow 1$ **to** Q **do**
3: $M[i,j] \leftarrow P_{em}(M_j, S_i) + max \begin{cases} M[i-1, j-1] \\ B[i-1] + P_{tr}(B, M_j) \end{cases}$
4: **end for**
5: $N[i] \leftarrow N[i-1] + P_{tr}(N, N)$
6: $E[i] \leftarrow max_{1 \leq j \leq Q}(M[i,j])$
7: $J[i] \leftarrow max \begin{cases} J[i-1] + P_{tr}(J, J) \\ E[i] + P_{tr}(E, J) \end{cases}$
8: $B[i] \leftarrow max \begin{cases} N[i] + P_{tr}(N, B) \\ J[i] + P_{tr}(J, B) \end{cases}$
9: $C[i] \leftarrow max \begin{cases} C[i-1] + P_{tr}(C, C) \\ E[i] + P_{tr}(E, C) \end{cases}$
10: **end for**
11: **return** $C[L] + P_{tr}(C, T)$

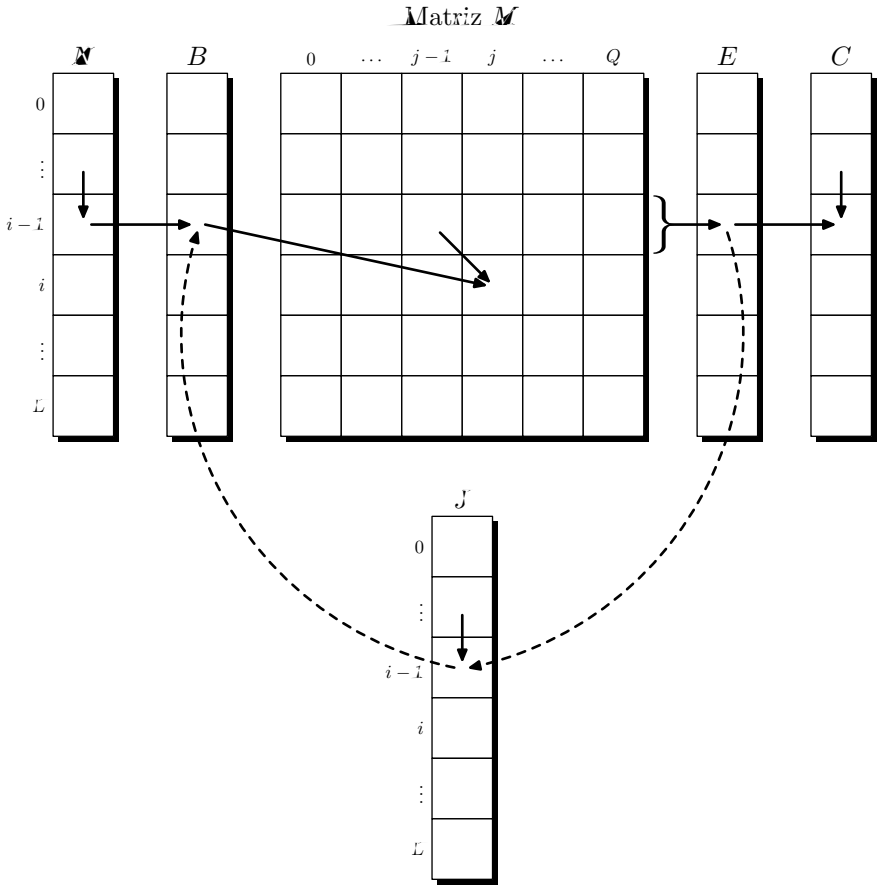


Figura 3.6: Dependências de dados do algoritmo MSV

3.7 Algoritmo SSV

O algoritmo SSV (*Simple Segment* Viterbi) é uma simplificação do algoritmo MSV, através da remoção do estado *Joining*. A abordagem de remoção do estado *Joining* já foi utilizada anteriormente em vários trabalhos visando ganho de desempenho ao custo de perda de precisão, como será visto no Capítulo 4.

O Algoritmo 3.3 apresenta o algoritmo SSV para encontrar o *score* do melhor alinhamento de uma sequência S de comprimento L com um *profile* HMM de Q nós. Sua complexidade de tempo é $O(L \times Q)$. Este algoritmo foi deduzido a partir do código fonte do próprio HMMER3.1b e ainda não foi apresentado em uma documentação formal pelos autores.

Algoritmo 3.3 Algoritmo SSV

Entradas: *Profile* HMM de Q nós
Probabilidades P_{em} de emissão de símbolos por estados
Probabilidades P_{tr} de transição entre estados
Sequência S com L símbolos
Saída: *Score* do melhor alinhamento de S com o *profile* HMM

1: **for** $i \leftarrow 1$ **to** L **do**
2: **for** $j \leftarrow 1$ **to** Q **do**
3: $M[i,j] \leftarrow P_{em}(M_j, S_i) + M[i-1, j-1]$
4: **end for**
5: $E[i] \leftarrow max \begin{cases} E[i-1] \\ max_{1 \leq j \leq Q}(M[i,j]) \end{cases}$
6: **end for**
7: **return** $E[L]$

Novamente, posto o intuito de encontrar formas de se explorar paralelismo, uma análise do algoritmo SSV permite identificar as dependências de dados, mostradas na Figura 3.7, notavelmente reduzidas desde o algoritmo de Viterbi original. Portanto, pode-se explorar paralelismo no cálculo das células de uma mesma linha também no algoritmo SSV.

3.8 Obtenção do Alinhamento Ótimo

Os algoritmos de Viterbi, MSV e SSV, da forma como foram apresentados, apenas calculam o *score* do alinhamento ótimo entre a sequência sendo investigada e uma família de sequências conhecida. Entretanto, deseja-se também obter o alinhamento ótimo propriamente dito entre a sequência e a família. Esse alinhamento ótimo é a sequência de estados do *profile* HMM que tem maior probabilidade de emitir os símbolos da sequência biológica sendo analisada, probabilidade essa correspondente ao *score* ótimo.

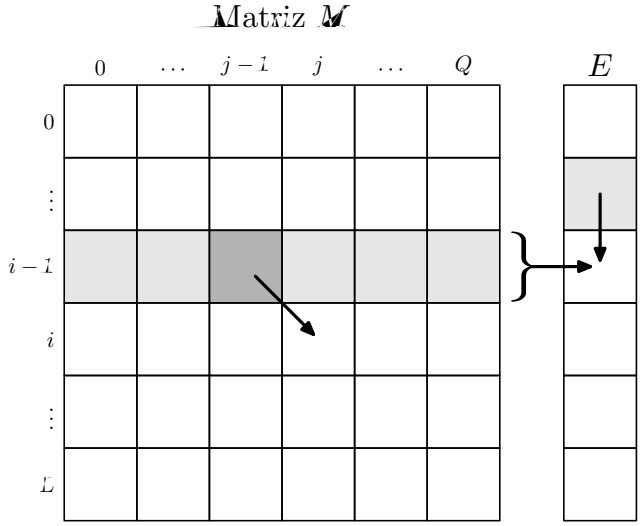


Figura 3.7: Dependências de dados do algoritmo SSV

O algoritmo para obter o alinhamento ótimo, denominado *Traceback*, necessita que as matrizes de programação dinâmica tenham sido preenchidas previamente pelos algoritmos de Viterbi, MSV ou SSV. O alinhamento ótimo é determinado a partir do seu fim, ou seja, a partir da célula que contém o *score* final que é então definida como célula corrente. A célula anterior é determinada através do maior *score* parcial dentre as que possuem transição para a célula corrente e, por sua vez, é definida como célula corrente. Esse passo se repete até que todos os estados do alinhamento sejam determinados.

O algoritmo de *Traceback* para obter o alinhamento ótimo MSV é apresentado no Algoritmo 3.4. A Figura 3.8 exemplifica um alinhamento ótimo MSV indicado na figura através de um caminho nas estruturas de dados do algoritmo. Os círculos escuros representam alinhamentos dos símbolos da sequência com os estados *Match* do *profile* HMM, os círculos claros representam o descarte de símbolos e os círculos vazios representam estados mudos (que não emitem símbolos da sequência) ou transições.

Algoritmo 3.4 Algoritmo de *Traceback* para obtenção do alinhamento ótimo MSV

Entradas: Vetores de *scores* N, B, E, J, C
Matriz de *scores* M
Saída: Sequência de estados do *profile* HMM com maior probabilidade de emitir os símbolos da sequência S

1: $i \leftarrow L - 1$
2: $current \leftarrow getPreviousState(C[L])$
3: **repeat**
4: $\quad path[i] \leftarrow current$
5: $\quad current \leftarrow getPreviousState(current)$
6: $\quad i \leftarrow i - 1$
7: **until** $current \neq N[0]$
8: **return** $path$

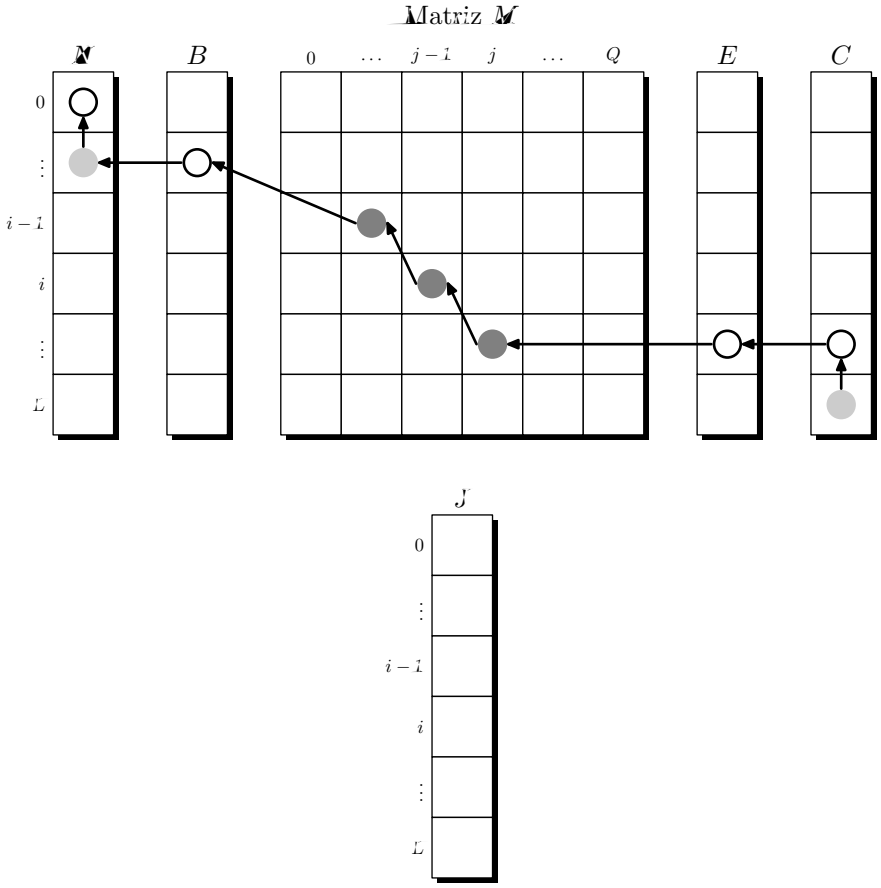


Figura 3.8: Exemplo de alinhamento ótimo MSV obtido pelo algoritmo de *Traceback*

4 Trabalhos Relacionados

Visto que os algoritmos SSV e MSV são simplificações do algoritmo de Viterbi e dada a maior quantidade de trabalhos abordando formas de se explorar o paralelismo no algoritmo de Viterbi em contraste aos recém introduzidos SSV e MSV, a revisão bibliográfica se estendeu a formas de se explorar paralelismo e consequentemente de melhorar o desempenho nesses três algoritmos.

Diversas soluções que tentam ganhar desempenho em comparação à solução em software implementada pelo HMMER são encontradas na literatura. Esses trabalhos vão desde soluções que exploram paralelismo de tarefas, como é o caso da análise de várias sequências simultaneamente utilizando *clusters* [22, 53, 54], a soluções que exploraram paralelismo de dados, como o cálculo de células de uma mesma anti-diagonal ou de uma mesma linha em paralelo utilizando dispositivos como FPGAs [1, 3, 29, 30, 42, 43, 49, 50] ou GPUs [6, 15, 18, 28, 46, 52, 56].

O paralelismo de tarefas é obtido de maneira simples visto que a comparação de uma sequência com o *profile* HMM é independente da comparação das demais. Dessa forma, em processadores multi-*core* ou em *clusters* com múltiplos nós, cada *core*/nó pode executar uma instância do algoritmo e analisar uma sequência distinta, como pode ser visto na Figura 4.1.

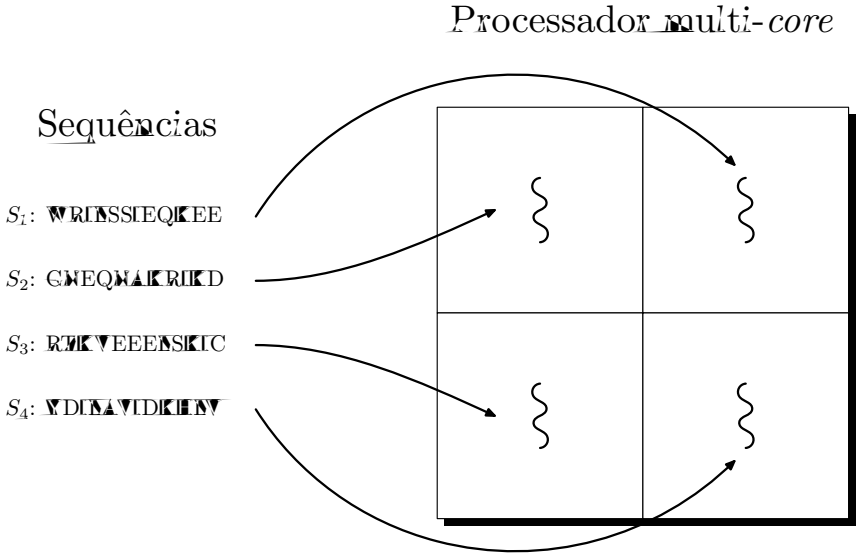


Figura 4.1: Exploração do paralelismo de tarefas na análise de um conjunto de sequências

A ferramenta HMMER oferece uma implementação usando a biblioteca MPI para execução em *cluster*, explorando o paralelismo de tarefas. Na operação *hmmsearch* do HMMER3, o paralelismo de tarefas é explorado através do processamento de múltiplas

sequências simultaneamente, em processadores multi-*core*, onde cada *core* é encarregado de executar uma instância do algoritmo MSV e analisar uma sequência.

4.1 Soluções em FPGA

A exploração do paralelismo entre células de uma mesma anti-diagonal das matrizes M, I e D no algoritmo de Viterbi é possível com a remoção do estado *Joining* que provoca a dependência de dados indireta do *feedback loop*, anteriormente mostrada na Seção 3.5. Diversos trabalhos explorando paralelismo de anti-diagonal neste algoritmo podem ser encontrados na literatura, como por exemplo [3, 49, 50].

Em alguns casos, como nas soluções apresentadas em [1, 49, 50], há também enfoque na otimização do uso da memória e dos recursos finitos da FPGA. Com uma melhor utilização dos recursos, mais PEs (*Processing Elements*) podem ser criados e consequentemente um maior grau de paralelismo é alcançado.

4.1.1 Soluções com Perda de Precisão

A remoção do estado *Joining* do *profile* HMM permite a exploração do paralelismo entre células de uma mesma anti-diagonal, porém faz com que alinhamentos *multi-hit* da sequência sendo investigada com a família não sejam mais possíveis. Como consequência, pode haver uma perda de precisão no *score* produzido com a utilização deste *profile* HMM modificado.

Em geral, soluções que apresentam perda de precisão são utilizadas como filtros. O objetivo é descartar rapidamente uma grande parte das sequências fornecidas como entrada e que possuem *score* insignificante, separando apenas sequências com maior potencial de pertencerem à família. Essas sequências com maior potencial têm então os *scores* calculados de forma precisa por uma segunda solução.

Alguns trabalhos como [3, 43] simplesmente ignoram a existência da dependência de dados do *feedback loop* através da remoção do estado *Joining*, e fornecem uma solução aproximada, com perda de precisão, para uma pequena fração das sequências, aquelas que têm um alinhamento *multi-hit* como seu alinhamento sequência-família ótimo.

Essa abordagem proporciona ganhos de desempenho através da exploração do paralelismo de dados na anti-diagonal, contudo a perda de precisão pode ocasionar a geração de falsos negativos. Em outras palavras, sequências que utilizam o *feedback loop* podem receber *scores* baixos, dada a ausência do estado *Joining* na solução, o que resulta no descarte errôneo das mesmas.

Os desempenhos de [3, 43] chegaram a 31,416 GCUPS e 5,3 GCUPS, respectivamente. A medida de desempenho CUPS (*Cell Updates Per Second*) representa quantas células das matrizes de programação dinâmica são atualizadas por segundo pela solução [43].

A solução apresentada por Maddimsetty *et al.* [30] tenta compensar a perda de precisão ocasionada pela remoção do estado *Joining* com a duplicação do núcleo do *profile* HMM e a inserção de um estado *Joining* entre as duas partes, denominado *Linker* (*L*), como mostrado na Figura 4.2.

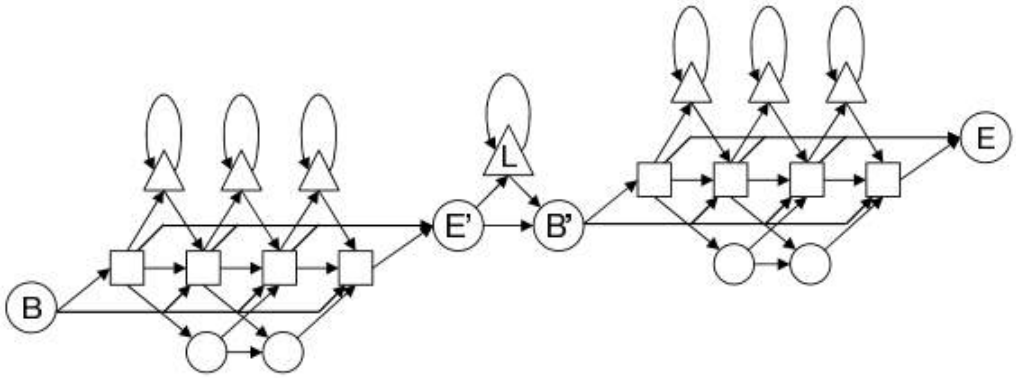


Figura 4.2: Arquitetura de *profile* HMM com duplicação do núcleo do modelo, visando minimizar a perda de precisão causada pela remoção do estado *Joining* [30]

Com isso há a possibilidade de uma única passagem da sequência pelo estado *Linker*, isto é, de um alinhamento de duas partes da sequência com o núcleo do *profile* HMM original. Todavia essa solução não elimina totalmente os falsos negativos, apenas os reduz, dado que a duplicação do núcleo do modelo não trata todos os alinhamentos *multi-hit*. Os autores tratam isso aumentando a tolerância do valor do *score* mínimo para a sequência não ser descartada, isto é, reduzindo a severidade do filtro. Contudo tal estratégia também tem efeitos colaterais, como a redução do número sequências que são descartadas, o que está diretamente associado ao desempenho da solução como um filtro. Os autores avaliaram um projeto com 50 PEs executando a uma frequência de *clock* de 200 MHz e obtiveram um desempenho estimado de 5 a 20 GCUPS.

4.1.2 Soluções sem Perda de Precisão

A utilização do *feedback loop* foi mensurada através de experimentos realizados por Takagi & Maruyama [50] que concluíram que, em média, o alinhamento *multi-hit* ocorre apenas 0,01% das vezes. Dada a rara utilização do *feedback loop*, alguns trabalhos como Sun *et al.* [49] e Takagi & Maruyama [50] apresentam soluções especulativas.

Nessa abordagem supõem-se que o *feedback loop* não existe e dessa forma os cálculos

são executados explorando paralelismo de dados na anti-diagonal. Porém, ao final de parte da execução, há uma verificação de se o alinhamento sequência-família utilizando o *feedback loop* forneceria um alinhamento com *score* superior. No caso da existência de um *score* superior utilizando o *feedback loop*, um *rollback* é realizado e o *score* é então recalculado, dessa vez considerando a existência do *feedback loop*, portanto uma execução sem exploração de paralelismo de dados.

Há perda de desempenho com os *rollbacks* e recálculos sem paralelismo de dados, contudo, isso ocorre tão raramente que, segundo os autores, não chega a reduzir o desempenho das soluções de forma expressiva, além de preservar a precisão do algoritmo de Viterbi. Assim, as soluções que adotaram essa abordagem não são apenas filtros e sim soluções completas que fornecem a solução exata. Os desempenhos das implementações de Sun *et al.* [49] e Takagi & Maruyama [50] atingiram *speedups* de 110,072 e 363, respectivamente.

Por fim, a solução de Abbas & Derrien [1] foi a única solução utilizando FPGA aplicada ao algoritmo MSV encontrada durante a revisão bibliográfica. A Figura 4.3 mostra essa solução e é possível identificar o caminho crítico do circuito no cálculo do máximo dentre as células de uma linha i da matriz M, isto é, para se calcular a célula $E[i]$ do vetor E (célula essa representada na figura por X_i).

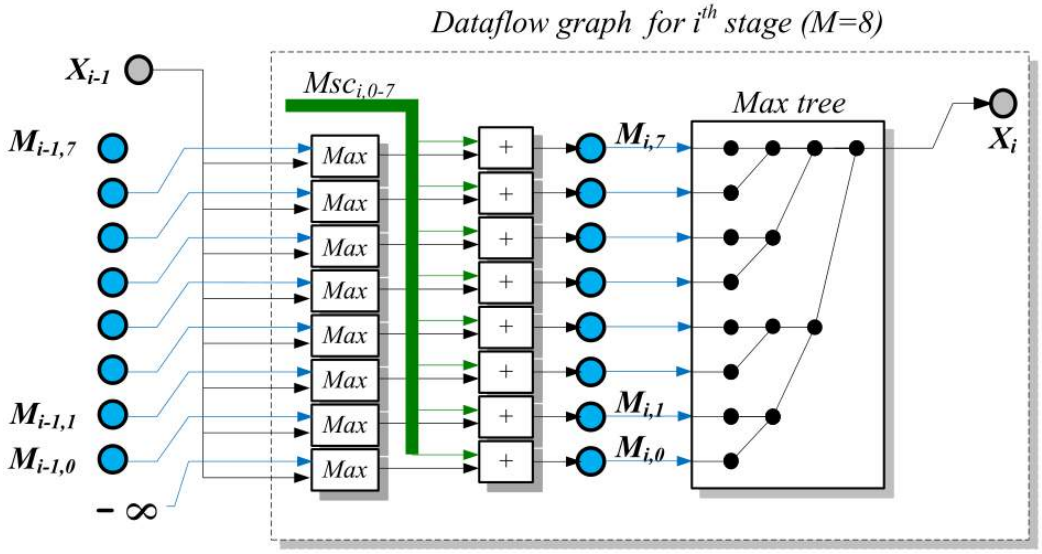


Figura 4.3: Solução em FPGA para algoritmo MSV com caminho crítico na árvore de máximos [1]

O problema de encontrar o máximo de n números precisa de ao menos $\log_2(n)$ passos de comparação, realizados pela estrutura em árvore esboçada em *Max tree* na Figura 4.3. Para um *profile* HMM demasiadamente grande tem-se uma árvore com altura muito grande, ou seja, um *profile* HMM grande resulta em um caminho crítico longo e em consequência disso, a solução sofre com frequências de *clock* baixas. Para contornar isso,

os autores utilizam a técnica de C-*slowing* [1]. Esta técnica realiza a inserção de blocos de registradores entre os níveis da árvore. Dessa forma o caminho crítico é quebrado e taxas de *clock* mais altas podem ser atingidas.

Contudo, a inserção desses registradores cria uma espécie de *pipeline* que não pode ficar vazio, enquanto se espera a conclusão dos cálculos da linha i da matriz, para se calcular a linha $i + 1$. Em vista disso, sequências distintas são intercaladas de forma a manter esse *pipeline* cheio e trabalhando, uma vez que o processamento de sequências distintas não carregam dependências de dados entre si, e com isso essa perda de desempenho é evitada. Dessa forma, a técnica de C-*slowing* associada à intercalação de sequências melhora o *throughput* da solução.

Vale observar também que os recursos de uma FPGA são finitos e dessa forma a representação de um *profile* HMM demasiadamente grande diretamente torna-se impossível. Em [3], a técnica de *tiling* [32] utilizando uma fila FIFO é aplicada para se modelar *profile* HMMs que ultrapassam os recursos disponíveis na FPGA. Nessa solução apenas uma quantidade limitada de nós do *profile* HMM é mantida ativa e representada, os outros nós são mantidos em memória e carregados quando necessários.

Já na solução proposta em Abbas & Derrien [1], o problema dos recursos finitos é contornado utilizando a técnica de *tiling* de forma um pouco diferente. Um *profile* HMM de tamanho M é dividido em P partições de modo que apenas P células de uma linha da matriz são calculadas em paralelo. Essa técnica exige M/P passos para o cálculo de uma linha completa da matriz, ou seja, há perda de desempenho, porém menos recursos da FPGA são necessários para calcular apenas uma porção da linha e com isso *profile* HMMs de tamanhos maiores pode ser utilizados. A Figura 4.4 mostra um exemplo de *tiling* com duas partições.

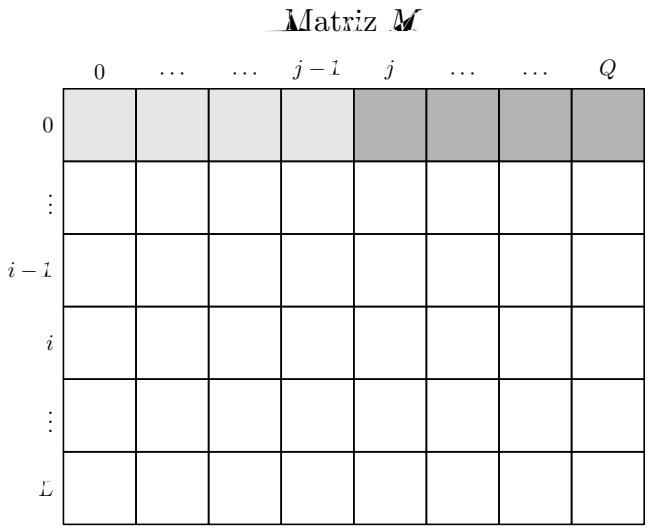


Figura 4.4: Técnica de *tiling* dividindo as linhas da matriz M em duas partições

Diversas técnicas que permitem contornar as limitações de hardware, tanto para um conjunto de entradas grande quanto para o caminho crítico da solução, são mostradas em Abbas & Derrien [1] e o seu desempenho chegou a até 69 GCUPS.

4.2 Soluções em GPU

Soluções em GPU desfrutam do hardware massivamente paralelo dessas plataformas. O modelo SIMT (*Single Instruction, Multiple Threads*) [25] de programação das GPUs é algo híbrido entre os modelos SIMD e SMT (*Simultaneous Multithreading*), o que permite aos pesquisadores o desenvolvimento de implementações das mais diversas, que vão desde a exploração de paralelismo de dados e/ou tarefas na plataforma até a exploração de paralelismo com computação heterogênea através da cooperação do processador.

4.2.1 Soluções para o Algoritmo de Viterbi

Como já explicado anteriormente, o algoritmo de Viterbi possui muitas dependências intrínsecas, dependências essas que impossibilitam a exploração de paralelismo de dados. Portanto, em geral, as soluções propostas para ele baseadas em GPU, assim como as soluções baseadas em *cluster*, exploram o paralelismo de tarefas, isto é, processam várias sequências simultaneamente. Esse tipo de paralelismo é facilmente explorado no algoritmo de Viterbi visto que a análise de uma sequência é independente das demais, o que possibilita a execução de várias instâncias do algoritmo para o processamento de várias sequências simultaneamente.

4.2.1.1 Soluções com Perda de Precisão

ClawHMMER [18] foi a primeira implementação da operação *hmmsearch* do HMMER2 usando GPU. Lançado antes da existência dos modelos CUDA ou OpenCL, a implementação foi feita em Brook, uma linguagem de programação desenvolvida na Universidade de Stanford que permite usar GPUs para computação de propósito geral e que não teve continuidade no seu desenvolvimento.

A implementação focou em como executar o algoritmo de Viterbi em GPUs e no uso de estratégias para melhorar o desempenho do mesmo nessas plataformas. Para isso, uma série de otimizações foram realizadas como o balanceamento de carga entre as *threads*, a aplicação da técnica de *loop unrolling* em um laço do algoritmo, o armazenamento das probabilidades na memória de textura visando tirar proveito da *cache* disponível para a

mesma, a transferência de *batches* de sequências de forma assíncrona com a execução na GPU, entre outras.

Outra otimização está na prévia ordenação das sequências de acordo com seu comprimento, de forma que *batches* processadas pela GPU tenham sequências de comprimento aproximado, beneficiando o balanceamento das cargas de trabalho atribuídas a cada *thread*.

Dada as limitações de memória das GPUs, ClawHMMER é incapaz de manter todas as matrizes de programação dinâmica necessárias para realizar o *traceback* nessa plataforma. Cabe então ao *host* realizar essa tarefa caso o *score* da sequência seja significativo, sendo a solução, portanto, um filtro em GPU.

A implementação foi testada em diversas GPUs, obtendo melhores resultados em uma GPU ATI R520 onde foram alcançados *speedups* entre 19,62 e 36,97, comparados com a versão 2.3.2 do HMMER executado em um Pentium 4 Xeon de 2,8GHz.

Du *et al.* [6] apresentam uma implementação em GPU com o modelo CUDA que segue a mesma ideia de vários trabalhos em FPGA que removem o estado *Joining* para com isso explorar paralelismo no cálculo das células de uma mesma anti-diagonal das matrizes M, I e D de programação dinâmica. Basicamente a implementação segue a seguinte abordagem: se a matriz de programação dinâmica pode ser carregada completamente na GPU, ela é carregada e a sequência é processada. Caso contrário, a matriz é dividida em *tiles* de forma que cada *tile* caiba completamente na memória da GPU e então os *tiles* são calculados um por vez ou em paralelo.

Utilizando um processador Intel Dual Core de 2,83GHz, uma GPU NVIDIA GeForce 9800 GTX e dados sintéticos de entrada, a solução produziu *speedups* entre 1,79 e 72,21 em relação a uma implementação sequencial desenvolvida pelos autores.

4.2.1.2 Soluções sem Perda de Precisão

Walters *et al.* [52] também implementaram a operação *hmmsearch* do HMMER2 em GPU usando o modelo CUDA. A implementação explora paralelismo de tarefas, processando várias sequências simultaneamente, com cada *thread* operando em uma única sequência.

As sequências também são ordenadas por comprimento, de forma que *threads* de um mesmo bloco recebem sequências de comprimento aproximado e assim fiquem ociosas o mínimo de tempo possível. Essa ordenação prévia das sequências proporcionou um *speedup* de 7 em comparação à execução sem a ordenação das mesmas. Estratégias como a utilização da memória de textura para armazenar o *profile* HMM e o *batch* de sequências, além da aplicação da técnica de *loop unrolling* em um laço do código também foram

aplicadas. Essa última estratégia proporcionou um *speedup* de 1,8 quando utilizado um fator de 2 para desenrolar o laço mais interno do algoritmo de Viterbi.

Também incapaz de realizar o *traceback* na GPU, a implementação é um filtro, onde os menos de 2% de sequências que resultam em acerto têm o *traceback* realizado no *host*. Os *speedups* obtidos em uma GPU NVIDIA GeForce 8800 GTX Ultra foram de 19 à 38,6, em relação a um AMD Athlon 275, dependendo do tamanho do *profile* HMM utilizado.

Outra solução em GPU usando o modelo CUDA que difere das demais por apresentar uma abordagem cooperativa entre o processador *host* e a GPU é denominada CuHMMER [56]. A implementação foca em uma estrutura de dados capaz de gerar o balanceamento de carga entre *host* e GPU de forma eficiente e na criação de um código multi-*thread* responsável por dividir a computação entre o *host* e a GPU. Uma *thread* principal faz *fork* para duas novas *threads*, uma responsável pelo *host* e outra pela GPU. Essas *threads* monitoram os dados e, quando necessário, o *host* ou a GPU são invocados.

CuHMMER também é um filtro e quando executado em uma GPU NVIDIA GeForce GTX 8800 acoplada a um processador Intel Core 2 Duo E7200, obteve *speedups* entre 13 e 45, comparados com o HMMER 2.3.2 executado em um AMD Athlon 64 X2 3800+.

Por sua vez, Ganesan *et al.* [15], apresentam uma implementação do algoritmo de Viterbi também em GPU com o modelo CUDA, porém capaz de explorar paralelismo de tarefas e de dados simultaneamente. Para lidar com dependências intrínsecas do algoritmo de Viterbi e tornar possível uma solução híbrida, este algoritmo é remodelado e uma série de operações matemáticas são realizadas para transpor as dependências de dados que impossibilitam a exploração de paralelismo de dados.

A ideia utilizada para resolver as dependências de dados que impedem o cálculo em paralelo de células da mesma linha é ilustrada na Figura 4.5, que representa a linha i de uma matriz de programação dinâmica. Os elementos de índice 0, k e $2k$ dessa linha são denominados âncoras, e as posições entre duas âncoras formam uma partição. A relação de recorrência das matrizes são desenvolvidas até que os elementos de uma partição dependam apenas do valor da âncora que inicia a partição.

O processamento de uma linha das matrizes é realizado da seguinte maneira: as âncoras são calculadas sequencialmente, dado que cada âncora depende apenas do valor da âncora anterior. Em seguida, os demais elementos da linha são calculados em paralelo por diferentes *threads*, uma vez que dependem apenas da âncora anterior, que já está calculada. Sucessivas linhas são calculadas sequencialmente, para que as demais dependências de dados do algoritmo de Viterbi sejam respeitadas.

Diferentemente das outras soluções, a implementação apresentada usa um conjunto

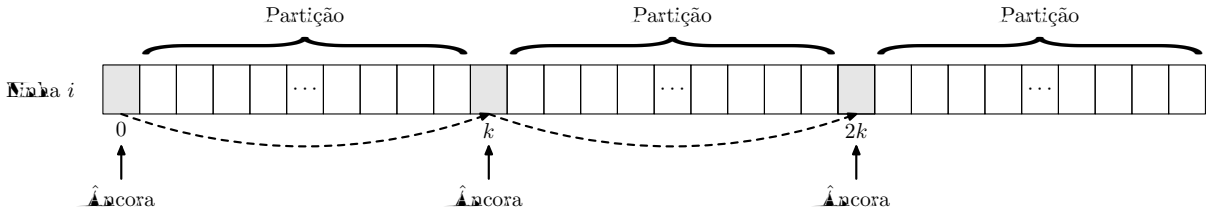


Figura 4.5: Determinação de âncoras e partições de uma linha da matriz de programação dinâmica para exploração de paralelismo de dados

de *threads* para analisar uma sequência e várias sequências são analisadas em paralelo. Essa abordagem de um conjunto de *threads* processando uma única sequência anula a necessidade de uma ordenação prévia das sequências como em outras soluções. O tempo gasto nesse pré-ordenamento é fixo visto que a base de dados ordenada pode ser reutilizada inúmeras vezes. Contudo, o ritmo frequente de atualizações das bases de dados de sequências biológicas torna uma solução que não necessita desse processamento prévio mais interessante.

Dispondo de um *host* com quatro GPUs NVIDIA Tesla C1060 acopladas, a implementação alcançou um *speedup* de mais de 100 em relação à versão serial em um processador AMD Opteron de 2,33GHz. A partir dos resultados, os autores concluíram que o tempo de execução da solução cresce linearmente com o aumento do tamanho do *profile* HMM e reduz linearmente com o aumento do número de GPUs.

Por fim, Ferraz & Moreano [12] desenvolveram uma solução que explora paralelismo de tarefas através da investigação de sequências distintas em paralelo (cada *thread* executa uma instância do algoritmo de Viterbi). A solução foi construída através de sucessivas otimizações que proporcionaram ganho de desempenho como *loop unrolling*, acesso coalescido à memória, redução nos acessos à memória, entre outras e que associadas proporcionaram um ganho de desempenho ainda maior. Utilizando OpenCL no desenvolvimento dessa solução os *speedups* alcançados em uma GPU GeForce GTX 460 acoplada a um processador AMD Athlon II X3 ficaram entre 40,35 e 102,83, quando comparados com o HMMER2 executado neste mesmo processador.

4.2.2 Soluções para o Algoritmo MSV

Quirem *et al.* [46] foram os primeiros a implementar o algoritmo MSV em GPU e para tal usaram o modelo de programação CUDA. Cientes da possibilidade de exploração de paralelismo de tarefas e de dados simultaneamente na execução do algoritmo MSV, os autores atribuíram o processamento de uma sequência a um bloco de *threads*. Dessa forma, um bloco de *threads* calcula células de uma mesma linha da matriz de programação dinâmica em paralelo, explorando paralelismo de dados, e calcula uma linha da matriz por

vez. Como a análise de uma sequência independe das demais, várias sequências podem ser analisadas simultaneamente através da execução de vários blocos em paralelo pela GPU, explorando paralelismo de tarefas.

A implementação, segundo os próprios autores, carece de otimizações como a utilização da memória constante e de textura. Contudo, otimizações como a transferência de dados e a execução assíncrona do *kernel*, além do uso de memória não-paginada foram implementadas. Essa última otimização proporcionou *speedup* de 1,2 em relação à versão sem memória não-paginada.

Os autores observaram que o *speedup* cresce exponencialmente quando o número *threads* sendo usadas é dobrado, contudo não foram usadas mais de 16.384 *threads* dada as limitações da GPU. Também foi observado que em alguns casos o *host* é mais rápido e, portanto, é importante selecionar qual plataforma de execução usar em cada caso. Nos melhores casos a solução proporcionou *speedups* entre 10 e 15, quando executada em uma GPU NVIDIA Tesla C1060.

Li *et al.* [28] também apresentam uma solução para o algoritmo MSV em GPU usando o modelo CUDA. Nessa solução houve um esforço em usar de forma eficiente a hierarquia de memórias da plataforma para obter ganho de desempenho. Foi constatado que os acessos à memória da GPU eram o gargalo da execução e deveriam ser minimizados. Para isso, a solução realiza o máximo de acessos coalescidos possível, trazendo múltiplos dados da memória de uma única vez, e mantém os dados mais utilizados em registradores, que possuem tempo de acesso mais curto.

Além disso a solução converte os símbolos das sequências para dígitos para facilitar o seu processamento, ordena as sequências para balancear as cargas de trabalho nas *threads*, implementa transferência assíncrona e mantém as probabilidades na memória de textura.

O laço externo do algoritmo MSV depende do laço interno, isto é, as linhas da matriz de programação dinâmica devem ser calculadas sequencialmente. Contudo, uma abordagem especulativa para o algoritmo MSV é adotada, onde o laço mais externo é desenrolado por um fator de 2. Segundo os autores, o valor do *score* $B[i]$ só muda em 1% dos casos, o que faz com que a especulação em geral tenha sucesso. Contudo, quando ela falha o recálculo da sequência é feito no *host*.

Apenas o algoritmo MSV é implementado na GPU. Os outros dois algoritmos da sequência de filtros do HMMER3 são implementados no *host*, utilizando os múltiplos *cores* disponíveis neste, de forma que um *core* é responsável pela comunicação com a GPU enquanto os outros executam os próximos algoritmos da sequência de filtros.

Executada em um processador Intel Xeon E5506 com uma GPU NVIDIA Tesla C2050 acoplada, esta solução alcançou *speedup* de até 6,5 em relação ao HMMER3 com

SSE executando em um único *core*.

4.2.3 Outros Trabalhos

Visto que a transferência de dados entre o *host* e a GPU possui uma alta latência, isso pode se tornar um gargalo na execução. Se esse tempo de transferências somado ao tempo de execução na GPU ultrapassar o tempo de execução no *host*, é melhor que essa tarefa seja executada no *host*. Assim, o trabalho em Ahmed *et al.* [2] investiga os pontos do HMMER3 onde é viável a execução na GPU.

Sua análise se restringiu ao *jackhammer*, outro programa do pacote de programas que formam o HMMER3. A execução em GPU dos módulos que se mostraram viáveis nesta investigação proporcionou um *speedup* médio de 2,1. Algo semelhante foi realizado por Abbas & Derrien [1] para o programa *hmmsearch*, quando constatou-se que o algoritmo MSV era responsável por 75% do tempo de execução do programa no HMMER3. Portanto, os resultados dessas investigações reforçam a ideia de potencial de ganho de desempenho na execução dos algoritmos MSV e SSV em GPU.

4.3 Soluções que Utilizam Extensões do Conjunto de Instruções

Muitas aplicações de multimídia realizam a mesma operação em vetores de dados de um byte ou dois bytes. Portanto, particionando a propagação do sinal de *carry* de um circuito somador de 128 bits de largura por exemplo, um processador pode realizar simultaneamente operações em pequenos vetores cujos operandos possuem largura de poucos bytes. Atualmente vários processadores adotam esta abordagem e oferecem extensões de seu conjunto de instruções de máquina, através de instruções do tipo SIMD que são capazes de explorar o paralelismo de dados de forma limitada.

Na operação *hmmsearch* do HMMER3, o paralelismo de dados é explorado utilizando extensões SIMD do conjunto de instruções de máquina. Uma destas extensões é descrita a seguir.

4.3.1 *Streaming* SIMD *Extensions*

SSE (*Streaming* SIMD *Extensions*) [21] é uma extensão do conjunto de instruções de máquina, introduzida pela Intel em 1999 e presente na maioria dos processadores Intel e AMD atuais. A extensão SSE adiciona novos registradores de 128 bits e instruções que permitem utilizar esses registradores para operações do tipo SIMD.

Cada um dos registradores de 128 bits pode armazenar mais de um dado ao mesmo tempo para que múltiplos dados sejam operados simultaneamente. Assim, as instruções oferecidas pelo SSE2 (versão do SSE utilizada pelo HMMER3) podem operar sobre:

- dois números em ponto flutuante de 64 bits (precisão dupla), como mostrado na Figura 4.6(a);
- dois números inteiros de 64 bits, como mostrado na Figura 4.6(a);
- quatro números em ponto flutuante de 32 bits (precisão simples), como mostrado na Figura 4.6(b);
- quatro números inteiros de 32 bits, como mostrado na Figura 4.6(b);
- oito números inteiros de 16 bits, como mostrado na Figura 4.6(c);
- 16 números inteiros de 8 bits, como mostrado na Figura 4.6(d).

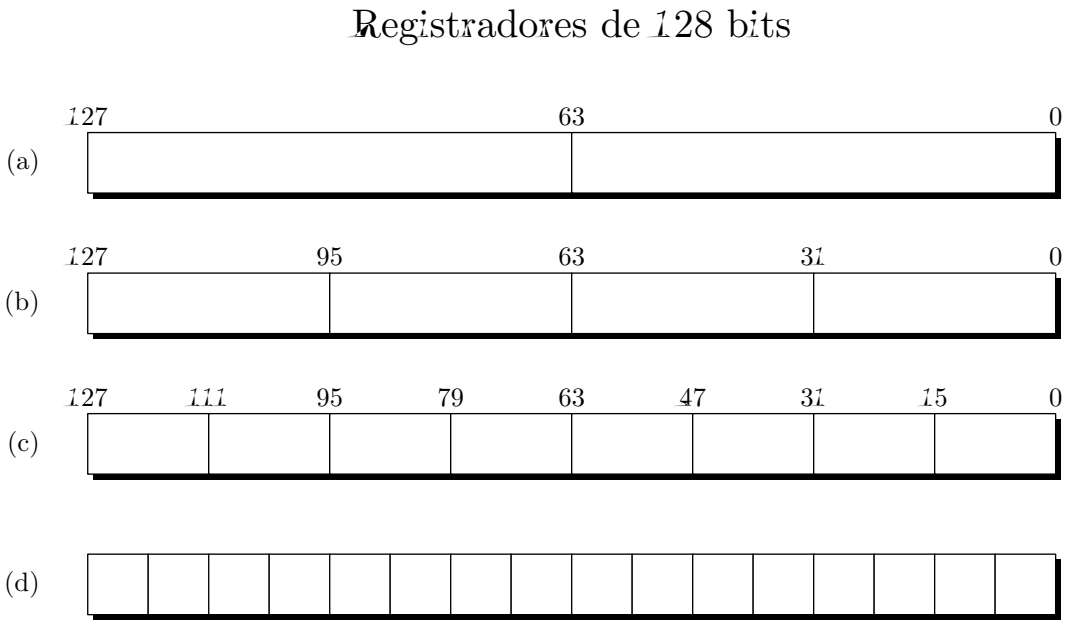


Figura 4.6: Registradores de 128 bits do SSE2 sendo usados para armazenar: (a) dois dados de 64 bits; (b) quatro dados de 32 bits; (c) oito dados de 16 bits; (d) 16 dados de 8 bits

A Figura 4.7 exemplifica uma operação de soma do tipo SIMD. Nesse exemplo dois vetores de 128 bits, X e Y, armazenam quatro números inteiros de 32 bits cada. A operação é feita por uma única instrução especial que realiza simultaneamente a soma dos quatro números. Portanto, para o processador parece que apenas uma instrução foi executada, contudo quatro operações foram realizadas, o que proporciona ganho de desempenho.

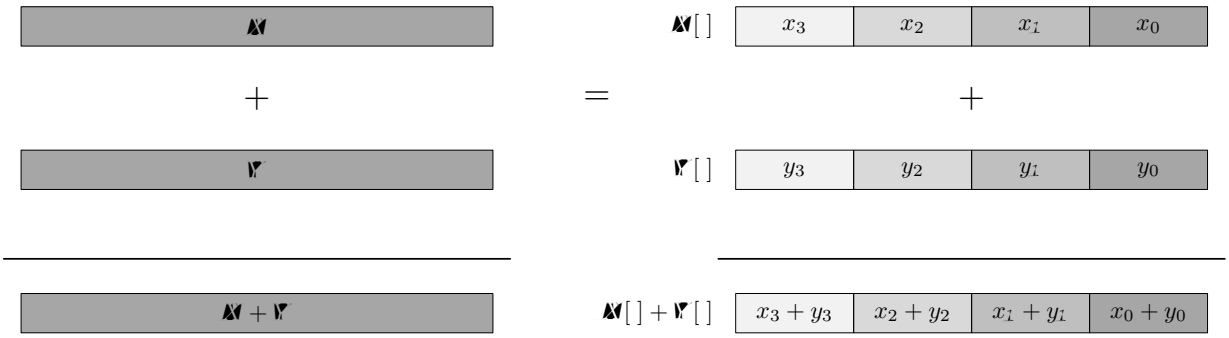


Figura 4.7: Exemplo de uma operação de soma do tipo SIMD

4.3.2 Paralelismo de Dados no MSV

Na operação hmmsearch do HMMER3, o paralelismo de dados é explorado utilizando a extensão SSE2 dos processadores Intel ou AMD e a extensão Altivec/VMX dos processadores PowerPC.

O ganho de desempenho do algoritmo MSV em relação ao algoritmo de Viterbi se deve à possibilidade de exploração de paralelismo de dados em sua execução, o que era impossível no algoritmo de Viterbi tradicional. A remoção de algumas das dependências de dados intrínsecas do algoritmo de Viterbi possibilitaram a construção de um algoritmo mais simples onde células de uma mesma linha da matriz M de programação dinâmica podem ser calculadas simultaneamente e as linhas consecutivas dessa matriz devem ser calculas uma por vez.

Esse paralelismo de dados pode ser explorado com instruções do tipo SIMD como a extensão SSE2. Visto que o SSE2 possibilita a execução de uma mesma operação em até 16 dados distintos simultaneamente, o algoritmo MSV usando SSE2 consegue calcular até 16 células de uma mesma linha da matriz M em paralelo. Assim, a matriz M tem suas linhas divididas em porções de até 16 células e cada porção tem suas células calculadas em paralelo utilizando instruções SSE2. Essa mudança permite reduzir a quantidade de iterações do laço interno do algoritmo por um fator de 16.

A Figura 4.8 mostra um exemplo da matriz M com linhas de tamanho 48, onde cada linha está dividida em três porções de 16 células cada. Originalmente, o laço interno do algoritmo MSV (linhas 2 a 4 do Algoritmo 3.2) precisaria de 48 iterações para calcular todas as células de uma linha. Utilizando instruções SSE2, apenas três iterações são necessárias, pois em cada interação do laço até 16 células são calculadas.

Para que esta abordagem seja possível, os valores dos *scores* calculados pela implementação do algoritmo MSV usando instruções SSE2 do HMMER3 são aproximados para a precisão reduzida de 8 bits, permitindo uma faixa de *scores* de 0 a 255 [9].

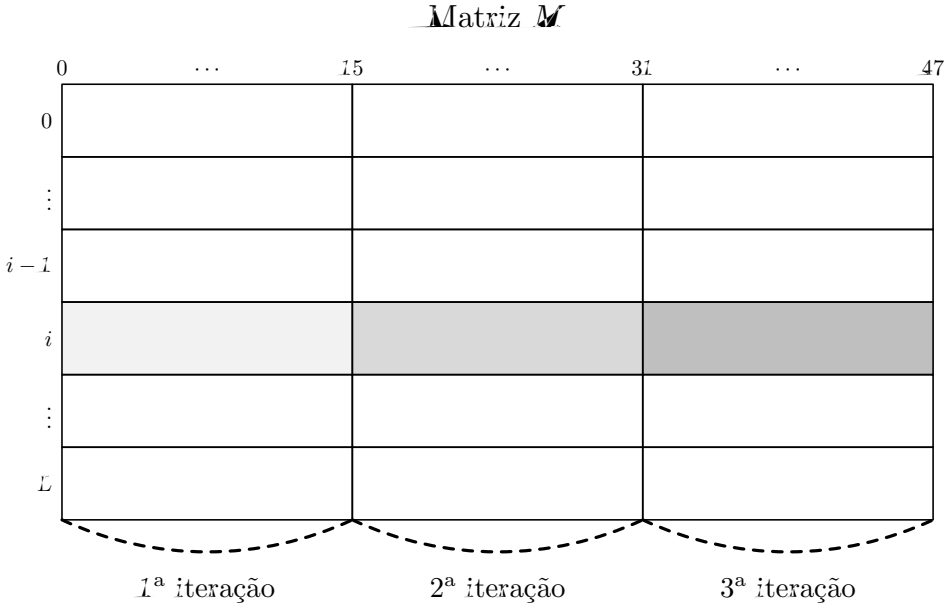


Figura 4.8: Linhas da matriz M divididas em porções de até 16 células, para exploração de paralelismo de dados com instruções SIMD no algoritmo MSV

4.4 Considerações Finais

A Tabela 4.1 resume as soluções encontradas durante a revisão bibliográfica, elencando a plataforma utilizada, o algoritmo implementado, as formas de paralelismo exploradas na implementação e o desempenho alcançado.

Tabela 4.1: Trabalhos relacionados: plataforma utilizada, algoritmo implementado, formas de paralelismo exploradas e desempenho alcançado

Solução	Ano	Plataforma utilizada	Algoritmo implementado	Paralelismo de tarefas	Paralelismo de dados	*Speedup* ou CUPS
Horn *et al.* [18]	2005	GPU	Viterbi	•		19,62 a 36,97
Maddimsetty *et al.* [30]	2006	FPGA	Viterbi		•	20 GCUPS
Oliver *et al.* [43]	2006	FPGA	Viterbi		•	5,3 GCUPS
Sun *et al.* [49]	2009	FPGA	Viterbi		•	110,072
Takagi & Maruyama [50]	2009	FPGA	Viterbi		•	363
Walters *et al.* [52]	2009	GPU	Viterbi	•		19 a 38,6
Abbas & Derrien [1]	2010	FPGA	MSV	•	•	69
Du *et al.* [6]	2010	GPU	Viterbi	•	•	1,97 a 72,21
Ganesan *et al.* [15]	2010	4 GPUs	Viterbi	•	•	100
Yao *et al.* [56]	2010	*host* + GPU	Viterbi	•		13 a 45
Quirem *et al.* [46]	2011	GPU	MSV	•	•	10 a 15
Li *et al.* [28]	2012	GPU	MSV	•		6,5
Ferraz & Moreano [12]	2013	GPU	Viterbi	•		40,35 a 102,83

A implementação e avaliação das soluções foram realizadas em hardwares distintos, portanto, possuem medidas de desempenho com referenciais distintos, o que dificulta a sua

comparação. De qualquer maneira, analisando os trabalhos relacionados e os resultados da Tabela 4.1, fica claro que, em geral, apesar da maior dificuldade de se explorar paralelismo de dados, as soluções que o fazem têm desempenho superior às soluções que exploram apenas paralelismo de tarefas. E as soluções que associam a exploração de ambas as formas de paralelismo obtêm resultados ainda melhores.

A exploração do paralelismo de tarefas pode ser realizada de forma simples em *clusters* e ocorre através da execução de várias instâncias da solução nos diferentes nós do *cluster*. Entretanto, construir ou comprar um *cluster* de computadores para o desenvolvimento de soluções paralelas para problemas pode ser custoso em termos financeiros [24].

O paralelismo de dados é explorado de forma eficiente pelas implementações em FPGA. No entanto, este dispositivo apresenta modelos de programação e fluxos de desenvolvimento mais complexos, quando comparados aos das outras plataformas, o que dificulta a sua utilização e torna o tempo de desenvolvimento utilizando-o mais longo.

O paralelismo de tarefas pode ser explorado em processadores de arquitetura convencional, também através da execução de várias instâncias da solução, porém desta vez nos diferentes *cores* do processador. O paralelismo de dados pode ser explorado em processadores convencionais através de instruções SIMD. Entretanto, os processadores atuais possuem entre 1 e 8 *cores* e as instruções SIMD operam vetores de poucos elementos (no máximo 16, no caso do SSE2), o que coloca limitações na exploração de ambas as formas de paralelismo.

Para exploração do paralelismo de tarefas, as GPUs possuem até 16 SMs (dependendo do modelo) que podem executar instâncias da solução. Já para exploração do paralelismo de dados, cada SM de uma GPU possui até 192 SPs (dependendo do modelo) que podem realizar uma mesma operação sobre dados distintos [40]. Portanto, as GPUs apresentam um maior potencial para exploração de paralelismo híbrido que os processadores convencionais.

Neste contexto, as GPUs apresentam um grande potencial de prover ganhos de desempenho, aliado a um baixo custo e facilidade de programação, o que motiva a utilização das mesmas para o desenvolvimento de soluções paralelas para problemas.

5 Solução em GPU para Comparação Sequência-Família

Uma solução integrada *host*-GPU foi desenvolvida para o problema de determinar se uma nova sequência biológica é homóloga a uma família de sequências conhecida. A solução recebe como entrada um conjunto de sequências a serem investigadas e um *profile* HMM representando uma família de sequências e calcula o *score* do alinhamento ótimo de cada sequência com a família. O modelo de programação CUDA foi usado para o desenvolvimento da solução.

5.1 Sequência de Filtros

A Figura 5.1 mostra a sequência de filtros implementada na solução desenvolvida. As caixas representam os algoritmos implementados e as setas os caminhos que as sequências podem percorrer, a partir do início da sequência de filtros. As sequências são analisadas por um ou mais algoritmos, até o seu descarte ou a sua aceitação, com a obtenção do seu *score* final.

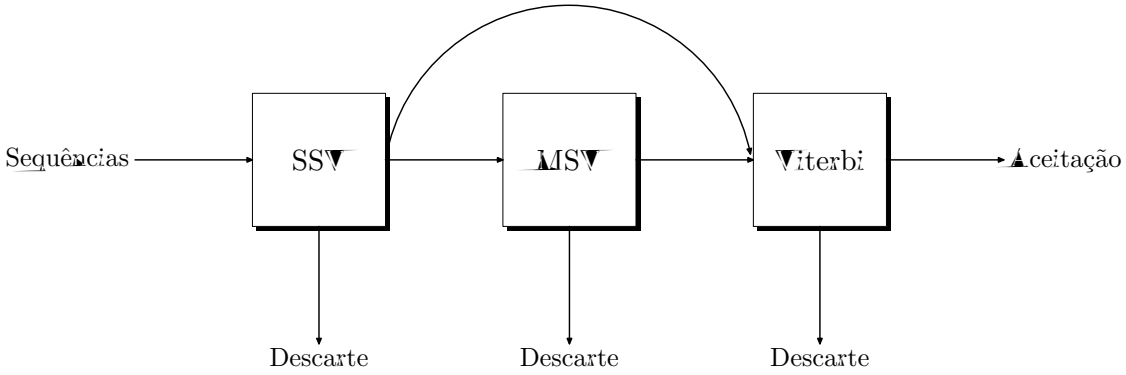


Figura 5.1: Sequência de filtros da solução desenvolvida

Os filtros SSV e MSV foram implementados para executar na GPU, através de dois *kernels* CUDA, enquanto o algoritmo de Viterbi foi implementado para executar no *host*. O controle da sequência de filtros foi implementado no *host*. Após a análise de um conjunto de sequências por um algoritmo da sequência de filtros, o *host* decide, baseado nos *scores* parciais calculados, quais sequências são descartadas e quais serão analisadas por um próximo algoritmo, mais lento, porém mais preciso.

5.2 Modelagem dos *Kernels*

Os *kernels* SSV e MSV foram modelados de forma a explorar paralelismo híbrido, ou seja, explorar simultaneamente paralelismo de tarefas e paralelismo de dados.

Visto que não há uma garantia da ordem de escalonamento dos blocos de um *grid* e não há mecanismos simples para sincronizar os mesmos de forma que cooperem (se comuniquem) uns com os outros na solução de um problema, o que é conhecido como requisito de independência entre os blocos, a exploração de paralelismo de dados entre os blocos para o problema em questão torna-se bastante difícil. Dada as dependências de dados de células anteriores para o cálculo de uma célula corrente, não há como particionar o problema de forma que os blocos explorem paralelismo de dados e não quebrem o requisito de independência entre os mesmos. Em outras palavras, para o cálculo de uma célula corrente é necessário assegurar que as células anteriores e necessárias para o cálculo da célula corrente tenham sido calculadas. Para tal, é necessário uma garantia na ordem do escalonamento dos blocos ou um método simples para sincronizar a execução dos mesmos, ambos inexistentes.

Contudo, *threads* pertencentes a um mesmo bloco podem cooperar entre si e sincronizar-se através de barreiras de sincronização, fornecendo assim o controle necessário para garantir a correta execução do algoritmo. Portanto, nos *kernels* SSV e MSV, a modelagem usada para exploração de paralelismo híbrido é a exploração de paralelismo de tarefas pelos vários blocos, que analisam sequências distintas simultaneamente. Cada bloco analisa uma sequência distinta e isso é possível visto que a análise de cada sequência é independente das demais. A exploração do paralelismo de dados ocorre através das várias *threads* de cada bloco, que calculam várias células da matriz de programação dinâmica paralelamente. Essa abordagem é ilustrada na Figura 5.2.

5.3 *Kernel* MSV

O código do *kernel* é apresentado na Figura 5.3. Por simplicidade e facilidade de compreensão, alguns detalhes foram omitidos.

No algoritmo MSV, apresentado no Algoritmo 3.2 na Seção 3.6, cada instância do algoritmo analisa uma sequência. Aqui, quando o *kernel* é invocado e um *grid* é criado, cada bloco do *grid* analisa uma sequência, portanto, múltiplas sequências são analisadas simultaneamente, o que garante a exploração do paralelismo de tarefas.

Já o paralelismo de dados pode ser observado, por exemplo, no cálculo da linha i da matriz M, na linha 29 do *kernel*. No Algoritmo 3.2, este cálculo é realizado através do

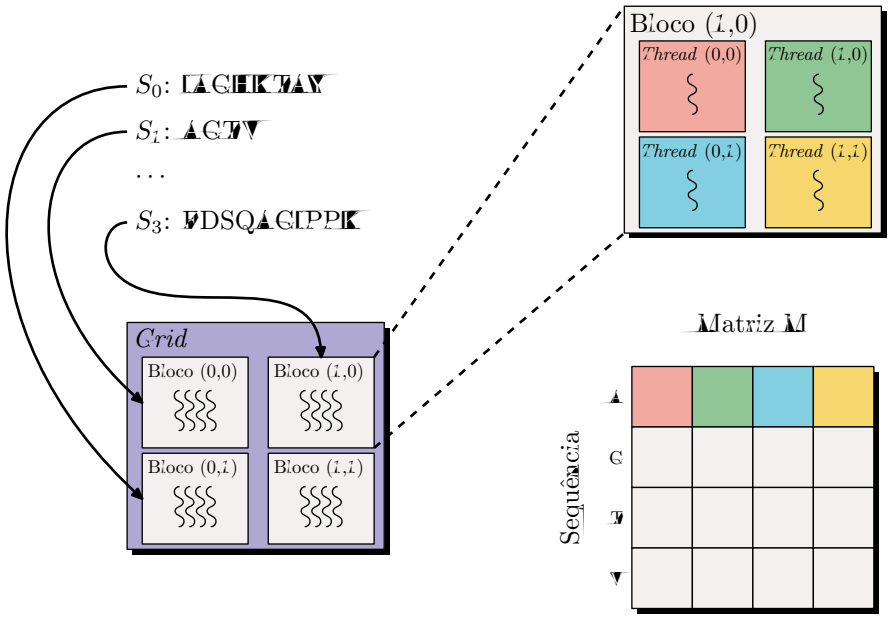


Figura 5.2: Exploração de paralelismo de tarefas (análise de sequências distintas por diferentes blocos) e exploração de paralelismo de dados (cálculo de células da matriz de programação dinâmica por *threads*)

laço das linhas 2 a 4. No *kernel*, ao contrário do laço presente no algoritmo, há apenas a linha 29 que é executada por todas as *threads* do bloco, porém com índices diferentes, pois cada *thread* possui um *tid* distinto. Assim, as várias *threads* do bloco executam a mesma operação para um conjunto diferente de dados, de forma que cada uma calcule uma célula da linha da matriz M. Portanto, várias células são calculadas simultaneamente, o que, por sua vez, garante a exploração do paralelismo de dados.

Barreiras de sincronização são necessárias em cinco pontos do *kernel*. As duas primeiras barreiras, nas linhas 20 e 30, garantem que a leitura dos dados escritos ocorra sem problemas, isto é, sem condições de corrida. As duas próximas barreiras, nas linhas 34 e 39, estão relacionadas ao cálculo do máximo de n números. Primeiramente há uma barreira para garantir que a escrita dos valores calculados por todas as *threads* nas células da matriz de programação dinâmica M já foi concretizada. Então, já com a linha atual da matriz M preenchida, o máximo pode ser encontrado. Visto que, a cada passo dos $\log_2(n)$ passos, apenas metade dos números são selecionados e escritos na primeira metade do vetor E para o processamento no próximo passo, é preciso uma sincronização ao final de cada passo para se evitar condições de corrida. Por fim, há uma barreira na linha 50, após o cálculo dos vetores que representam os estados especiais do HMM. O excesso de barreiras de sincronização prejudica o desempenho, uma vez que *threads* que alcançam a barreira ficam ociosas esperando as que ainda não alcançaram. Entretanto, em determinados trechos de código as sincronizações são necessárias para evitar condições de corrida.

```
1  __global__ void kernelMSV(float *emiss, char *seq, float *scores)
2  {
3     /* emiss  : probabilidades de emissão do estado Match */
4     /* seq    : sequência sendo investigada */
5     /* scores : estrutura de scores na memória global */
6     unsigned int tid = threadIdx.x; /* Id da thread */
7     unsigned int bid = blockIdx.x;  /* Id do bloco */
8
9     /* Aloca estruturas de scores na memória compartilhada */
10    __shared__ float N, B, M[2][Q + 1], E[Q], J, C;
11
12    /* Inicializa estruturas de scores */
13    if (tid == 0)
14    {
15       N = 0;
16       B = tnb_tjb_tct;
17       M[1][0] = J = C = -INFINITY;
18    }
19    M[0][tid] = -INFINITY;
20    __syncthreads();
21
22    /* Para cada símbolo da sequência */
23    for (int i = 0; i < L; i++)
24    {
25       int prev_row =  i      % 2;
26       int curr_row = (i + 1) % 2;
27
28       /* Calcula linha i da matriz M */
29       M[curr_row][tid+1] = emiss[seq[i]+tid] + max(M[prev_row][tid], B + tbmj);
30       __syncthreads();
31
32       /* Encontra o máximo da linha i da matriz M */
33       E[tid] = M[current_row][tid+1];
34       __syncthreads();
35       for (int offset = Q / 2; offset > 0; offset /= 2)
36       {
37          if ((tid < offset) && (tid + offset < Q))
38             E[tid] = max(E[tid], E[tid+offset]);
39          __syncthreads();
40       }
41
42       /* Calcula posição i dos vetores J, C, N e B */
43       if (tid == 0)
44       {
45          J  = max(J + tnn_tjj_tcc, E[0] + tej_tec);
46          C  = max(C + tnn_tjj_tcc, E[0] + tej_tec);
47          N += tnn_tjj_tcc;
48          B  = max(N + tnb_tjb_tct, J    + tnb_tjb_tct);
49       }
50       __syncthreads();
51    }
52
53    /* Escreve score final na memória global */
54    if (tid == 0) scores[bid] = C;
55 }
```

Figura 5.3: Código CUDA do *kernel* MSV

Tanto no cálculo dos vetores (linhas 45 a 48) quanto na escrita do *score* na memória global (linha 54), há divergência na execução do *kernel*, pois apenas a *thread* 0 realiza as operações. Divergências como essas também podem implicar em perda de desempenho, pois representam perda de paralelismo na execução do código. No caso em questão apenas a *thread* 0 realiza as operações, enquanto todas as outras não contribuem no avanço na solução. Assim, esses trechos de código são executados sequencialmente e representam um contrassenso à ideia na qual GPUs se baseiam para o ganho de desempenho, que é ter várias *threads* trabalhando simultaneamente para o avanço na solução.

5.4 *Kernel* SSV

Seguindo a ideia de exploração de paralelismo híbrido usada no *kernel* MSV, o *kernel* SSV apresenta uma abordagem especulativa eliminando o estado *Joining*. Esta abordagem remove a dependência existente e que outrora foi definida como *feedback loop*. Dessa forma, o algoritmo é simplificado, tornando-o suscetível ao ganho de desempenho ao custo da perda de precisão.

A utilização do *feedback loop* foi mensurada através de experimentos realizados por Takagi & Maruyama [50] que concluíram que, em média, o alinhamento *multi-hit* ocorre apenas 0,01% das vezes. Portanto, a abordagem especulativa apresenta resultados corretos 99,99% das vezes e, nas poucas vezes que isso não ocorre, essas sequências são selecionadas para a análise pelo filtro MSV que é mais preciso. O código do *kernel* SSV é apresentado na Figura 5.4. Novamente, por simplicidade e facilidade de compreensão, alguns detalhes foram omitidos.

Comparando ambos os *kernels*, é possível observar que a abordagem especulativa propicia uma simplificação no final do laço que percorre os símbolos da sequência. Sem o estado *Joining*, a divergência presente na linha 43 do *kernel* MSV da Figura 5.3 é eliminada, assim como a sincronização da linha 50 que também é removida.

Outra simplificação está nas operações necessárias para encontrar o máximo de n números. No *kernel* MSV são realizados $\log_2(n)$ passos, a cada iteração do laço que percorre os símbolos da sequência, nas linhas 35 a 40 da Figura 5.3. No *kernel* SSV, esses passos são realizados apenas uma vez, fora do laço que percorre os símbolos da sequência, nas linhas 35 a 40 da Figura 5.4. Isto representa uma redução significativa de $L-1$ vezes no número de reduções executadas. Além disso, as sincronizações (linha 39 da Figura 5.3 e linha 39 da Figura 5.4) necessárias a cada passo dos $\log_2(n)$ passos para encontrar o máximo de n números também são reduzidas em $L-1$ vezes no *kernel* SSV.

Estas simplificações combinadas explicam o melhor desempenho do *kernel* SSV em relação ao *kernel* MSV, como será visto na Seção 6.11.

```
1  __global__ void kernelSSV(float *emiss, char *seq, float *scores)
2  {
3     /* emiss  : probabilidades de emissão do estado Match */
4     /* seq    : sequência sendo investigada */
5     /* scores : estrutura de scores na memória global */
6     unsigned int tid = threadIdx.x; /* Id da thread */
7     unsigned int bid = blockIdx.x;  /* Id do bloco */
8
9     /* Aloca estruturas de scores na memória compartilhada */
10    __shared__ float M[2][Q + 1], E[Q];
11
12    /* Inicializa estruturas de scores */
13    if (tid == 0)
14    {
15       M[1][0] = -INFINITY;
16    }
17    M[0][tid] = E[tid] = -INFINITY;
18    __syncthreads();
19
20    /* Para cada símbolo da sequência */
21    for (int i = 0; i < L; i++)
22    {
23       int prev_row =  i      % 2;
24       int curr_row = (i + 1) % 2;
25
26       /* Calcula linha i da matriz M */
27       M[curr_row][tid + 1] = emiss[seq[i] + tid] + M[prev_row][tid];
28
29       /* Encontra os máximos entre a linha i da matriz M e o vetor E */
30       E[tid] = max(E[tid], M[curr_row][tid + 1]);
31       __syncthreads();
32    }
33
34    /* Encontra o máximo do vetor E */
35    for (int offset = Q / 2; offset > 0; offset /= 2)
36    {
37       if ((tid < offset) && (tid + offset < Q))
38          E[tid] = max(E[tid], E[tid + offset]);
39       __syncthreads();
40    }
41
42    /* Escreve score final na memória global */
43    if (tid == 0)
44       scores[bid] = E[0];
45 }
```

Figura 5.4: Código CUDA do *kernel* SSV

5.5 Estruturas de Dados

A execução dos *kernels* SSV e MSV na GPU exige que as probabilidades de emissão dos estados *Match* e as sequências a serem investigadas estejam na memória do dispositivo. Para isso, ambas as informações são copiadas para a memória da GPU antes que o primeiro

kernel seja invocado.

Nessa solução, apenas duas linhas da matriz de programação dinâmica M e apenas uma posição de cada vetor de *scores* são armazenados na memória compartilhada da GPU, minimizando assim a quantidade de memória utilizada por bloco. Reduzir a memória utilizada em cada bloco favorece a execução de um número maior de blocos, especialmente se o problema for *memory bound*, e é, portanto, uma boa prática de programação que favorece o aumento do grau de paralelismo.

Isso justifica minimizar a quantidade de dados que precisam ser mantidos e tentar alocar o máximo possível desses dados na memória compartilhada, que é mais rápida que a memória global, porém limitada a apenas alguns kilobytes por bloco. Contudo, o descarte das linhas anteriores da matriz de programação dinâmica M impossibilita o *traceback*, que, se necessário, é então realizado pelo *host*.

5.5.1 Probabilidades de Emissão

As probabilidades de emissão dos estados *Match* são dados que dependem do tamanho do alfabeto e do número de nós do *profile* HMM. Além disso, por não sofrerem mudanças durante a execução da solução, elas só precisam ser carregadas para a GPU uma única vez antes da invocação do primeiro *kernel*.

Uma vez que o alfabeto pode possuir até 29 símbolos e o maior *profile* HMM possui 2207 nós, armazenar as probabilidades de emissão, para esse pior caso, em valores de 32 bis requer 250,02 KB de espaço. Apesar de possuir no máximo alguns kilobytes, as probabilidades são acessadas com uma alta frequência durante a execução do código. Portanto, é fundamental garantir acesso rápido e eficiente a essa pequena porção dados.

5.5.2 Sequências

Dado o grande volume de dados dos conjuntos de sequências a serem investigadas, que em geral varia de alguns megabytes a alguns gigabytes, a memória global da GPU é o único lugar capaz de armazenar esses dados. Todavia, esses conjuntos de dados são muitas vezes maiores do que o tamanho da memória global. Para contornar essa limitação o conjunto de sequências é particionado em subconjuntos de tamanho menor que a memória global, denominados *batches*.

Assim, um *batch* é transferido para a memória global da GPU, investigado pela GPU e seus resultados são transferidos para a memória do *host*. Esses passos se repetem até que todos os *batches* tenham sido investigados, sendo o processamento de diferentes *batches* realizado sequencialmente.

As sequências armazenadas na memória global são acessíveis a todos os blocos. Cada bloco, baseado em seu identificador único, acessa uma única sequência, investiga a mesma e escreve seu *score* parcial também na memória global, visto que esse resultado precisa ser transferido posteriormente para o *host*.

O número de sequências em um *batch* é definido empiricamente em 65.535, como será visto na Seção 6.12.

5.6 Otimizações

A solução desenvolvida nesse trabalho apresenta uma série de otimizações que visam o ganho de desempenho. Por vezes, essas otimizações são referentes à arquitetura da GPU e, em geral, são fatores chaves para o sucesso da modelagem do problema na plataforma em questão. Algumas das otimizações, como, por exemplo, o acesso coalescido à memória global da GPU, são tidas como triviais em qualquer solução com o intuito de ganho de desempenho, outras, porém, são inerentes ao problema sendo resolvido.

5.6.1 Sequência de Filtros

Na abordagem adotada, todos os *batches* de sequências passam primeiro por um algoritmo, para então serem analisados pelo próximo. Visto que a maior parte das sequências de um *batch* são descartadas por um algoritmo no início da sequência de filtros, a construção de um novo *batch* com número de sequências suficiente para o próximo algoritmo requer as poucas sequências não descartadas de vários *batches* investigados.

Por exemplo, considere uma sequência de filtros como a da Figura 5.1, onde vários *batches* são processados inicialmente pelo algoritmo SSV e em média 95% das sequências contidas neles são descartadas. São necessários em torno de 20 *batches* ($20 \times 5\% = 100\%$) para a construção de um novo *batch* com número de sequências semelhante aos anteriores, contendo apenas as sequências não descartadas dos 20 *batches*. Esse novo *batch* é então investigado pelo algoritmo MSV, filtro subsequente na sequência de filtros.

Uma outra abordagem envolveria um *batch* sendo investigado por todos os filtros da sequência de filtros em GPU antes que um novo *batch* seja investigado. Apesar do benefício de não se enviar novamente as sequências não descartadas para a GPU, essa abordagem traz uma série de empecilhos, como, por exemplo, o modo de se ocupar a GPU enquanto os *scores* estão sendo julgados pelo *host* como relevantes ou irrelevantes. Na abordagem utilizada isso é feito por um dos *cores* do processador enquanto um novo *batch* está sendo investigado pela GPU. Outro detalhe envolve como indicar a cada bloco qual sequência não descartada deve ser investigada uma vez que isso não depende mais

do identificador único de cada bloco.

Em suma, a segunda abordagem é possível, contudo a complexidade adicionada não compensa os pequenos tempos gastos com as retransferências de sequências não descartadas para a análise de filtros mais precisos. De fato, essas transferências podem ser otimizadas e sobrepostas com a execução de *kernels* tornando o tempo gasto com suas realizações imperceptível no tempo total de execução da solução.

5.6.2 Probabilidades de Emissão

Uma otimização presente no próprio algoritmo SSV é a junção do valor do estado *Begin* nas probabilidades de emissão. Com a remoção do estado *Joining* o valor de B pode ser tratado como uma constante e com operações matemáticas ser incluído nas probabilidades de emissão evitando operações de máximo no algoritmo (linha 29 da Figura 5.3), o que reduz as operações realizadas e deixa o algoritmo mais rápido.

As probabilidades de emissão dos estados *Match* podem ser organizadas como uma matriz de 29 linhas e Q colunas. Contudo, essa estrutura pode gerar acessos não coalescidos. Um acesso não coalescido é realizado através de mais de uma transação de memória, ocasionando desperdício de banda e, consequentemente, perda de desempenho. Uma matriz com 29 linhas e uma quantidade de colunas que garanta que o endereço base acessado seja múltiplo do valor na coluna Alinhamento da Tabela 2.1, gera acessos coalescidos.

Portanto, em casos onde o valor Q não gera acessos coalescidos, o procedimento é aumentar a quantidade de colunas da matriz que armazena as probabilidades para um valor o mais próximo possível de Q e que forneça acesso coalescido. Essa técnica de adicionar células sem uso a uma estrutura de dados com o propósito de prover alinhamento nos acessos é denominada *padding* [4]. Assim, uma vez que as probabilidades são acessadas com uma alta frequência durante a execução do código, é primordial garantir acesso coalescido e uso eficiente da memória *cache* da memória global para esse pequeno conjunto de dados.

5.6.3 Sequências

Uma otimização realizada durante o carregamento das sequências na memória global da GPU é o armazenamento de cada uma delas como uma sequência de números positivos, em vez de uma sequência de caracteres (correspondentes aos símbolos do alfabeto). Estes números funcionam como índices para a leitura da probabilidade de emissão dos símbolos da sequência na memória. Portanto, o acesso da probabilidade de emissão do i-ésimo símbolo de uma sequência pode ser feito da forma *emiss[seq[i]][tid]*,

onde o índice aponta para a linha da matriz de probabilidades referente ao caractere da sequência e cada *thread* acessa uma célula da linha referente ao estado.

5.6.4 Memória Não-paginada

Umas das recomendações para ganho de desempenho nas transferências de dados entre *host* e GPU é que é possível aumentar a velocidade das mesmas usando memória não-paginada (*pinned memory*) [48].

Alocações de memória no *host* são paginàveis por padrão e as GPUs não podem acessar dados diretamente de regiões paginàveis da memória do *host*. Dessa forma, quando uma transferência de dados de uma região paginável da memória do *host* para a GPU é requisitada, o *driver* do CUDA primeiramente aloca uma região temporária na memória não-paginada do *host*, para onde ele copia os dados da memória paginável do *host*. Então, a partir desta região de memória não-paginada, a transferência para a GPU é realizada.

Uma comparação entre transferências de dados usando memória paginável e memória não-paginada pode ser vista na Figura 5.5. Como pode ser observado, a memória não-paginada funciona com uma área para as transferências entre *host* e GPU. Alocando as estruturas de dados diretamente nessa área de memória, evita-se a transferência dos dados entre as áreas de memória paginável e não-paginada, reduzindo assim o tempo total da transferência.

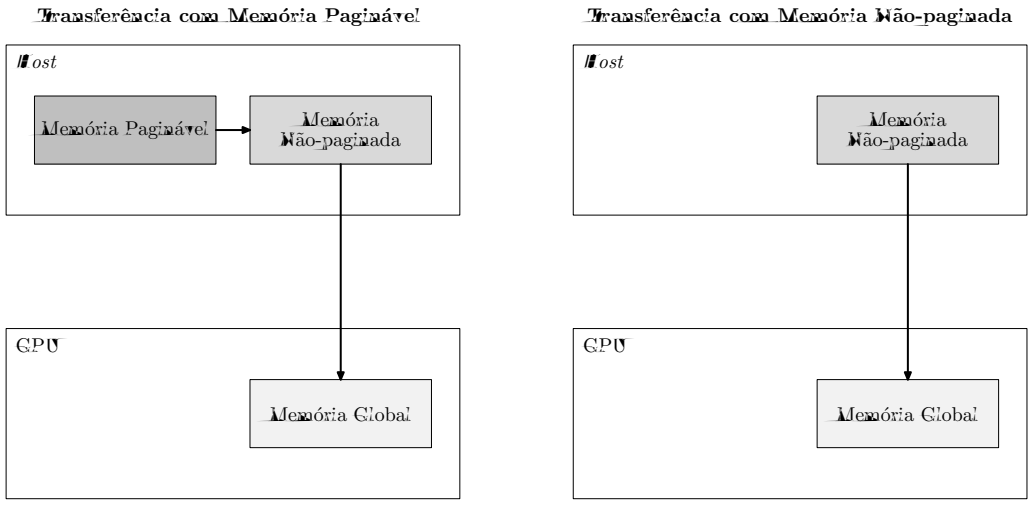


Figura 5.5: Comparação entre transferências de dados usando memória paginável e memória não-paginada

Para usar memória não-paginada no *host*, basta substituir o comando *malloc()* da linguagem C pelo comando *cudaHostAlloc()* do modelo de programação CUDA. Para liberar um bloco de memória alocado nesta região de memória, basta substituir o comando *free()* do C pelo comando *cudaFreeHost()* do CUDA [48]. Portanto, o uso de memória

não-paginada requer poucas mudanças no código, sendo viável mesmo que para pequenos ganhos de desempenho, dada sua simplicidade de uso e foi aplicada a todas as estruturadas de dados utilizadas na solução.

5.6.5 Árvore de Máximos e Redução

Como dito anteriormente, o problema de encontrar o máximo de n números em paralelo é implementado em $\log_2(n)$ passos de comparação. Isso foi feito de forma ótima através de uma árvore de máximos onde, a cada passo, metade dos valores é descartada, e ao final de $\log_2(n)$ passos, resta apenas o valor máximo. As comparações realizadas em um mesmo passo são feitas em paralelo por diferentes *threads*.

Uma árvore de máximos para um vetor de oito elementos pode ser vista na Figura 5.6, onde são necessários três passos. No primeiro passo são realizadas quatro comparações, no segundo, duas, e, por fim, no último passo, apenas uma comparação é realizada.

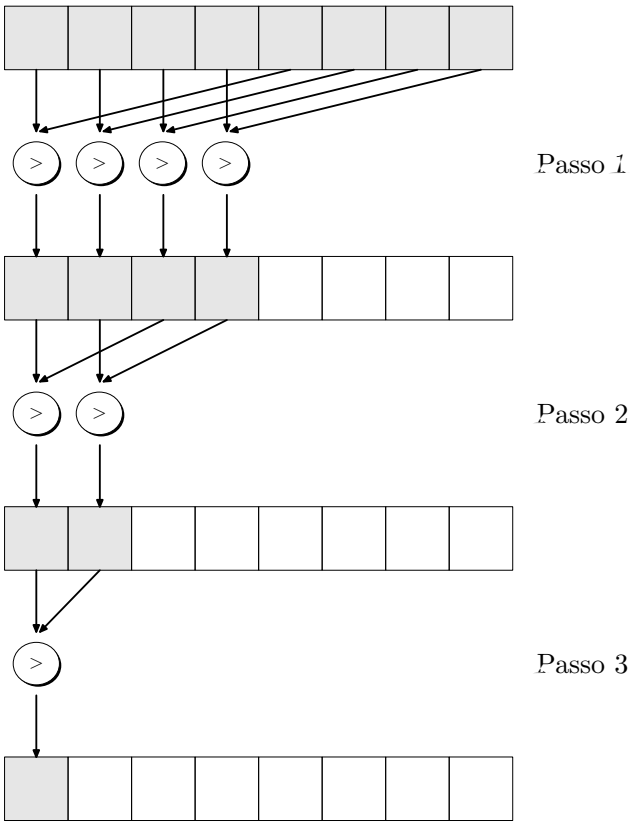


Figura 5.6: Árvore de máximos para um vetor de tamanho 8

A implementação da árvore de máximos no *kernel* MSV encontra-se nas linhas 35 a 40 da Figura 5.3. A cada passo, a primeira metade do vetor é comparada com a segunda, descartando metade dos números que não são máximos e escrevendo os números

que são máximos dessas comparações na primeira metade do vetor. Essas operações são realizadas na linha 38 do *kernel*. Após a escrita, a sincronização da linha 39 é necessária para que, no próximo passo, a primeira metade do vetor tenha apenas números que podem ser o máximo, ou seja, que todos os resultados das comparações tenham sido escritos na primeira metade do vetor evitando condições de corrida.

Contudo, essa é uma abordagem simples e ingênua para o problema da redução, que, definido de forma simples, reduz uma sequência de dados a um único valor usando uma operação binária associativa. Harris [16] apresenta otimizações para reduções em paralelo usando GPU que alcançam *speedups* de até 30,04 em relação à versão sem otimizações e com divergências no código.

Primeiramente, é observado que é possível aplicar a técnica de *loop unrolling* [4], desenrolando as interações do laço referentes ao último *warp* e removendo suas sincronizações. O último *warp* (32 *threads*) é responsável por encontrar o máximo dentre os 64 valores presentes no começo do vetor. Uma vez que as *threads* dentro de um *warp* executam de forma síncrona, não são necessárias as sincronizações entre os seis passos ($\log_2 64$) que realizam a redução desses 64 valores. Como sincronizações representam perda de desempenho, a remoção das mesmas acarreta em uma solução com melhor desempenho.

Outra observação é que as outras iterações também podem ser desenroladas, porém mantendo as sincronizações. A Figura 5.7 apresenta a implementação otimizada da árvore de máximos com a aplicação das estratégias descritas acima. Este código realiza o cálculo do máximo da linha i da matriz M do *kernel* MSV, para *profile* HMMs de até 1024 nós, substituindo assim o laço das linhas 35 a 40 da Figura 5.3.

Conhecendo previamente os tamanhos possíveis dos *profile* HMMs, *templates* do C++ podem ser utilizados para a geração de múltiplos *kernels*, com as condições presentes nas linhas 2, 8, 14, 20, 29, 31, 33, 35, 37 e 39 da Figura 5.7 sendo resolvidas em tempo de compilação. Assim, evita-se a realização de comparações e desvios condicionais durante a execução.

Um exemplo do *kernel* MSV usando *templates* pode ser visto na Figura 5.8. A geração de múltiplos *kernels* é possível com o simples uso de um comando *switch* com as possibilidades desejadas. Na Figura 5.9 são criados 10 *kernels* para 10 tamanhos distintos de *profile* HMM. Cada *kernel* apresenta otimizações nas comparações e desvios condicionais que são realizadas em tempo de compilação para o tamanho do *profile* HMM informado. Um exemplo de *kernel* SSV otimizado para *profile* HMMs de tamanho 128 pode ser visto na Figura 5.10.

```
1  /* Encontra o máximo da linha i da matriz M */
2  if (Q >= 1024)
3  {
4     if ((tid < 512) && (tid + 512 < Q))
5        E[tid] = max(E[tid], E[tid + 512]);
6     __syncthreads();
7  }
8  if (Q >= 512)
9  {
10    if ((tid < 256) && (tid + 256 < Q))
11       E[tid] = max(E[tid], E[tid + 256]);
12    __syncthreads();
13 }
14 if (Q >= 256)
15 {
16    if ((tid < 128) && (tid + 128 < Q))
17       E[tid] = max(E[tid], E[tid + 128]);
18    __syncthreads();
19 }
20 if (Q >= 128)
21 {
22    if ((tid <  64) && (tid +  64 < Q))
23       E[tid] = max(E[tid], E[tid +  64]);
24    __syncthreads();
25 }
26 if (tid < 32)
27 {
28    volatile unsigned int *vE = E;
29    if (Q >= 64)
30       if (tid + 32 < Q) vE[tid] = max(vE[tid], vE[tid + 32]);
31    if (Q >= 32)
32       if (tid + 16 < Q) vE[tid] = max(vE[tid], vE[tid + 16]);
33    if (Q >= 16)
34       if (tid +  8 < Q) vE[tid] = max(vE[tid], vE[tid +  8]);
35    if (Q >=  8)
36       if (tid +  4 < Q) vE[tid] = max(vE[tid], vE[tid +  4]);
37    if (Q >=  4)
38       if (tid +  2 < Q) vE[tid] = max(vE[tid], vE[tid +  2]);
39    if (Q >=  2)
40       if (tid +  1 < Q) vE[tid] = max(vE[tid], vE[tid +  1]);
41 }
```

Figura 5.7: Código CUDA otimizado para cálculo do máximo da linha i (redução) da matriz M do *kernel* MSV

```
1  template <unsigned int Q>
2  __global__ void kernelMSV(float *emiss, char *seq, float *scores)
```

Figura 5.8: Exemplo de uso de *templates* no *kernel* MSV

5.6.6 *Loop Unrolling*

Laços, em geral, apresentam instruções que incrementam um ponteiro ou índice para o próximo elemento em um vetor (aritmética de ponteiros), bem como testes

```
1  switch (Q)
2  {
3     case 1024:
4        kernelMSV<1024><<<dimGrid, dimBlock>>>(emiss, seq, scores);
5        break;
6     case 512:
7        kernelMSV<512><<<dimGrid, dimBlock>>>(emiss, seq, scores);
8        break;
9     case 256:
10       kernelMSV<256><<<dimGrid, dimBlock>>>(emiss, seq, scores);
11       break;
12    case 128:
13       kernelMSV<128><<<dimGrid, dimBlock>>>(emiss, seq, scores);
14       break;
15    case 64:
16       kernelMSV<64><<<dimGrid, dimBlock>>>(emiss, seq, scores);
17       break;
18    case 32:
19       kernelMSV<32><<<dimGrid, dimBlock>>>(emiss, seq, scores);
20       break;
21    case 16:
22       kernelMSV<16><<<dimGrid, dimBlock>>>(emiss, seq, scores);
23       break;
24    case 8:
25       kernelMSV<8><<<dimGrid, dimBlock>>>(emiss, seq, scores);
26       break;
27    case 4:
28       kernelMSV<4><<<dimGrid, dimBlock>>>(emiss, seq, scores);
29       break;
30    case 2:
31       kernelMSV<2><<<dimGrid, dimBlock>>>(emiss, seq, scores);
32       break;
33 }
```

Figura 5.9: Código para a geração de múltiplos *kernels* MSV em tempo de compilação

condicionais de fim de laço. A técnica de *loop unrolling* consiste em reduzir/eliminar essas operações com aritmética de ponteiros e também testes condicionais de fim de laço. Se o compilador é capaz de pré-calcular os *offsets* de cada variável (vetor) individualmente, estes podem ser integrados nas instruções de código de máquina diretamente reduzindo assim as operações aritméticas necessárias em tempo de execução.

Entretanto, nem tudo são benefícios. Essa técnica também implica algumas desvantagens, são elas:

- Aumenta o tamanho do código de programa;
- Pode causar um aumento de *cache misses* durante a *instruction fetch*;
- Pode aumentar o uso dos registradores em uma única iteração para o armazenamento de variáveis temporárias.

O aumento no número de registradores usados pode ser um problema, uma vez que é comum *kernels* limitados pelo número de registradores por *thread*, ainda mais quando há um pouco mais de complexidade, como é o caso do *kernel* MSV. Entretanto, é possível testar e determinar o fator de *unrolling* que apresenta melhores resultados para a solução e até mesmo se é melhor não usar a técnica em um *kernel* específico.

5.6.7 Transferência *Host*-GPU e Múltiplos *Streams*

Para que a adoção de uma solução em GPU seja viável, o tempo gasto nas transferências de dados entre *host* e GPU somado ao tempo de execução dessa solução em GPU deve ser menor que o tempo de se executar uma solução no *host*. Essa análise de viabilidade coloca um peso importante no tempo gasto com as transferências de dados entre *host* e GPU, ao ponto de que uma gerência ineficiente dessas transferências pode tornar a solução em GPU inviável se comparada com uma solução no *host*.

Uma orientação para ganho de desempenho em transferências de dados entre *host* e GPU é juntar várias pequenas transferências em uma única transferência grande para minimizar *overheads* [41]. Assim, em um contexto de solução simples, o habitual é transferir o máximo possível de dados para a GPU, processar os mesmos e, por fim, recuperar os resultados. Esses passos podem ser repetidos até que todo o conjunto de dados tenha sido processado. Dessa forma, a cada iteração desta, o tempo gasto na transferência que abastece a GPU com dados para serem processados e também o tempo gasto recuperando os resultados incrementam o tempo total gasto pela solução.

Entretanto, dada a capacidade das GPUs mais modernas de realizar transferências concomitante com a execução de um *kernel* ou com a realização de outra transferência, o tempo total de execução da solução pode ser reduzindo realizando algumas dessas atividades simultaneamente. A Figura 2.7 exemplifica como a sobreposição dessas atividades pode reduzir o tempo total de execução de uma solução, através da utilização de múltiplos *streams*.

Como dito anteriormente muitas pequenas transferências resultam em *overhead*, porém transferências muito grandes podem resultar em um atraso inicial muito grande para o início do processamento na GPU. Portanto, é necessário experimentação para determinar o tamanho ideal dessas transferências para o problema tratado. O experimento para determinar o tamanho ideal para as transferências de dados *host*-GPU e os resultados obtidos são descritos no próximo capítulo. Com essa abordagem é possível mascarar a latência de transferência que é tida como um gargalo na execução.

Na solução desenvolvida, múltiplos *streams* são utilizadas para a realização da transferência de um *batch* de sequências do *host* para a memória global da GPU, enquanto

a mesma executa o *kernel* de algum filtro. Assim, a ideia é manter a GPU processando sequências o máximo de tempo possível, evitando paradas por falta de sequências para serem processadas em memória. Com isso, a GPU é mantida menos tempo ociosa garantindo menores tempos de execução para a solução.

5.6.8 *Tiling*

Os *kernels* apresentados na Figura 5.3 e Figura 5.4 são limitados a *profile* HMMs de até 1.024 nós. Essa limitação é consequência da modelagem do problema que usa cada *thread* para calcular uma célula de cada linha da matriz M e, consequentemente, esbarra no limite do hardware usado que é de 1.024 *threads* por bloco.

Para contornar essa limitação, técnicas como o *tiling*, que consiste na divisão do problema em porções menores e na resolução dessas porções para a obtenção da solução final, podem ser aplicadas. Como apresentando na Figura 4.4, essa técnica pode ser usada de forma que as linhas da matriz M sejam divididas em porções e que cada *thread* calcule células das porções, permitindo assim que mesmo com a limitação de 1.024 *threads* por bloco, *profile* HMMs longos possam ser resolvidos pela solução.

Nessa abordagem, as *threads* calculam as células de uma porção em paralelo e cada porção é calculada de forma sequencial, isto é, uma porção após a outra. A princípio pode parecer ruim a ideia de resolver porções de forma sequencial, porém determinar um fator de *tiling* que garanta um número de *threads* capaz de resolver o problema e também de promover boas taxas de ocupação do dispositivo podem garantir ganho de desempenho até para *profile* HMMs menores que 1.024 nós.

```
1  __global__ void kernelSSV(float *emiss, char *seq, float *scores)
2  {
3     /* emiss  : probabilidades de emissão do estado Match */
4     /* seq    : sequência sendo investigada */
5     /* scores : estrutura de scores na memória global */
6     unsigned int tid = threadIdx.x; /* Id da thread */
7     unsigned int bid = blockIdx.x;  /* Id do bloco */
8
9     /* Aloca estruturas de scores na memória compartilhada */
10    __shared__ float M[2][Q + 1], E[Q];
11
12    /* Inicializa estruturas de scores */
13    if (tid == 0)
14       M[1][0] = -INFINITY;
15    M[0][tid] = E[tid] = -INFINITY;
16    __syncthreads();
17
18    /* Para cada símbolo da sequência */
19    for (int i = 0; i < L; i++)
20    {
21       int prev_row =  i      % 2;
22       int curr_row = (i + 1) % 2;
23
24       /* Calcula linha i da matriz M */
25       M[curr_row][tid + 1] = emiss[seq[i] + tid] + M[prev_row][tid];
26
27       /* Encontra os máximos entre a linha i da matriz M e o vetor E */
28       E[tid] = max(E[tid], M[curr_row][tid + 1]);
29       __syncthreads();
30    }
31
32    /* Encontra o máximo do vetor E */
33    if ((tid <  64) && (tid +  64 < Q))
34       E[tid] = max(E[tid], E[tid +  64]);
35    __syncthreads();
36    if (tid < 32)
37    {
38       volatile unsigned int *vE = E;
39       if (tid + 32 < Q)
40          vE[tid] = max(vE[tid], vE[tid + 32]);
41       if (tid + 16 < Q)
42          vE[tid] = max(vE[tid], vE[tid + 16]);
43       if (tid +  8 < Q)
44          vE[tid] = max(vE[tid], vE[tid +  8]);
45       if (tid +  4 < Q)
46          vE[tid] = max(vE[tid], vE[tid +  4]);
47       if (tid +  2 < Q)
48          vE[tid] = max(vE[tid], vE[tid +  2]);
49       if (tid +  1 < Q)
50          vE[tid] = max(vE[tid], vE[tid +  1]);
51    }
52
53    /* Escreve score final na memória global */
54    if (tid == 0) scores[bid] = E[0];
55 }
```

Figura 5.10: Exemplo de *kernel* SSV com otimização na redução para *profile* HMMs de tamanho 128

6 Resultados

Esse capítulo apresenta os resultados obtidos na avaliação de desempenho da solução integrada *host*-GPU desenvolvida para o problema de determinar se uma nova sequência biológica é homóloga a uma família de sequências conhecida. As diversas otimizações propostas também são analisadas.

6.1 Plataformas, Ferramentas e Dados Utilizados

A solução foi implementada e avaliada em um computador *host* com processador Intel Core i7-3770S que possui quatro *cores* com tecnologia *Hyper-Threading*, cada um com *clock* flutuante que pode chegar a até 3.9 GHz [20]. Além disso, o *host* possui 8 GB de memória RAM DDR3 de 1333 MHz em *Dual Channel*.

A GPU utilizada é a NVIDIA GeForce GTX 570 [38] com 1,25 GB de memória. Informações mais detalhadas, obtidas através do utilitário CUDA *Device Query* [35], podem ser vistas na Figura 6.1 (informações não relevantes para a solução desenvolvida foram omitidas por simplicidade).

```
Device 0: "GeForce GTX 570"
  CUDA Driver Version / Runtime Version          6.0 / 6.0
  CUDA Capability Major/Minor version number:    2.0
  Total amount of global memory:                 1280 MBytes
  (15) Multiprocessors x ( 32) CUDA Cores/MP:    480 CUDA Cores
  GPU Clock rate:                                1.59 GHz
  Memory Clock rate:                             1950 Mhz
  Memory Bus Width:                              320-bit
  L2 Cache Size:                                 655360 bytes
  Total amount of constant memory:               65536 bytes
  Total amount of shared memory per block:       49152 bytes
  Total number of registers available per block: 32768
  Warp size:                                     32
  Maximum number of threads per multiprocessor:  1536
  Maximum number of threads per block:           1024
  Maximum sizes of each dimension of a block:    1024 x 1024 x 64
  Maximum sizes of each dimension of a grid:     65535 x 65535 x 65535
  Maximum memory pitch:                          2147483647 bytes
  Concurrent copy and kernel execution:          Yes with 1 copy engine
  Integrated GPU sharing Host Memory:            No
  Support host page-locked memory mapping:       Yes
```

Figura 6.1: Resultados do CUDA *Device Query* na GPU usada nos experimentos

O sistema operacional instalado no *host* é o Ubuntu 13.04 64 bits com *kernel* Linux

3.13.0-24-generic e a versão utilizada do modelo de programação CUDA é a 6.0.37.

A ferramenta HMMER, utilizada para comparação com a solução desenvolvida, foi compilada usando o compilador GCC (GNU *Compiler Collection*) versão 4.8.2, com as *flags* de otimização presentes por padrão em seu *Makefile* (*-std=gnu99 -O3 -fomit-frame-pointer -malign-double -fstrict-aliasing -pthread -msse2*). O código CUDA da solução foi compilado usando o compilador G++ 4.8.2 com as *flags* de otimização *-O3 -fomit-frame-pointer*.

Dez *profile* HMMs foram utilizados nos experimentos e correspondem a famílias de sequências oriundas do projeto Pfam [13]. Essas famílias foram escolhidas segundo o tamanho dos *profile* HMMs, de tal forma que elas sejam manejáveis pela solução desenvolvida, dadas as limitações de memória da GPU utilizada, mantendo em torno de 256 de diferença entre seus tamanhos. A Tabela 6.1 apresenta as famílias de sequências selecionadas e o número de nós dos *profile* HMMs.

Tabela 6.1: Famílias de sequências da base de dados Pfam utilizadas

Família	Número de nós do *profile* HMM
Avian_gp85	256
CABIT	256
DUF530	512
PaRep2b	512
Flu_PB2	759
Totivirus_coat	759
ACR_tran	1.021
RdRP_5	1.271
Bac_GDH	1.528
AvrE	1.774
Média	864,80

O conjunto de sequências utilizado como entrada foi a base de dados UniProtKB/Swiss-Prot [51] completa, que possui 540.732 sequências, totalizando aproximadamente 183,19 MB. Estatísticas dessa base de dados podem ser vistas na Tabela 6.2.

Tabela 6.2: Estatísticas das sequências da base de dados UniProtKB/Swiss-Prot

Comprimento das sequências	
Mínimo	2
Máximo	35.213
Médio	355,24

As tabelas apresentadas a seguir mostram os tempos de execução do HMMER3,

do HMMER3.1b e da solução desenvolvida em GPU, assim como os *speedups* obtidos. O tempo de execução consiste no tempo gasto para análise de todas as sequências da base de dados UniProtKB/Swiss-Prot, para cada uma das famílias de sequências selecionadas previamente.

A ferramenta HMMER foi executada com quatro configurações distintas:

1 *core*: Utilizando um *core* do processador sem o uso de instruções SSE2;

4 *cores*: Utilizando quatro *cores* do processador sem o uso de instruções SSE2;

1 *core*/SSE2: Utilizando um *core* do processador com o uso de instruções SSE2;

4 *cores*/SSE2: Utilizando quatro *cores* do processador com o uso de instruções SSE2.

O tempo de execução do HMMER inclui apenas o tempo de execução da seguinte sequência de filtros: apenas o filtro MSV, no caso da versão 3, e os filtros SSV e MSV, no caso da versão 3.1*b*. O tempo de execução da solução integrada *host*-GPU desenvolvida considera todo o tempo de execução da sua sequência de filtros, e inclui o tempo de transferência de dados entre o *host* e GPU somado ao tempo gasto para investigação das sequências em GPU. A sequência de filtros utilizada na solução desenvolvida corresponde à sequência de filtros da versão do HMMER usada para comparação. Isto é, quando a solução em GPU é comparada com o HMMER3, a sequência de filtros implementada contém apenas o algoritmo MSV. Quando a solução em GPU é comparada com o HMMER3.1b, a sequência de filtros implementada contém os algoritmos SSV e MSV.

6.2 Solução Inicial sem Otimizações

Uma primeira solução foi desenvolvida buscando apenas desenvolver os algoritmos SSV e MSV em GPU de forma correta. Além disso, essa solução básica, sem otimizações, possibilita mensurar o quanto cada otimização representa de melhoria. Assim, as soluções seguintes são otimizações a partir dessa solução inicial, e as otimizações que resultarem em ganho desempenho serão associadas em uma solução final. Os *kernels* MSV e SSV das soluções podem ser vistos nas Figura 5.1 e Figura 5.3.

A Tabela 6.3 apresenta os tempos de execução da solução inicial em GPU e do HMMER3, para cada configuração sua, para a base de dados UniProtKB/Swiss-Prot completa e cada uma das famílias de sequências selecionadas. Na Tabela 6.4 os resultados apresentados referem-se ao HMMER3.1b.

A Tabela 6.5 apresenta os *speedups* da solução inicial em GPU em relação a cada configuração do HMMER3, enquanto na Tabela 6.6 a comparação é feita com o HMMER3.1b.

Tabela 6.3: Tempos de execução da solução inicial em GPU e do HMMER3

Família	Tempo de execução (ms)				
	HMMER3 1 *core*	HMMER3 4 *cores*	HMMER3 1 *core*/SSE2	HMMER3 4 *cores*/SSE2	Solução em GPU
Avian_gp85	41.240	15.750	1.340	490	2.233,67
CABIT	41.320	15.710	1.370	460	2.233,68
DUF530	81.980	44.400	2.590	770	5.023,62
PaRep2b	82.060	44.320	2.470	740	5.023,81
Flu_PB2	121.940	69.600	3.950	1.190	8.659,01
Totivirus_coat	122.010	69.780	3.860	1.120	8.659,01
ACR_tran	164.450	93.490	5.020	1.420	16.053,90
RdRP_5	205.900	116.320	6.240	1.870	-
Bac_GDH	247.000	140.100	7.270	2.080	-
AvrE	285.510	162.440	8.360	2.370	-

Tabela 6.4: Tempos de execução da solução inicial em GPU e do HMMER3.1b

Família	Tempo de execução (ms)				
	HMMER3.1b 1 *core*	HMMER3.1b 4 *cores*	HMMER3.1b 1 *core*/SSE2	HMMER3.1b 4 *cores*/SSE2	Solução em GPU
Avian_gp85	41.190	15.770	510	260	611,82
CABIT	41.210	15.560	490	270	611,84
DUF530	82.230	45.840	960	340	1.320,16
PaRep2b	82.130	46.590	960	340	1.320,18
Flu_PB2	121.580	69.540	1.420	450	2.188,73
Totivirus_coat	123.510	69.440	1.440	440	2.188,72
ACR_tran	164.240	93.440	1.930	590	3.785,47
RdRP_5	205.580	116.440	2.400	710	-
Bac_GDH	247.230	139.880	2.990	910	-
AvrE	287.850	162.670	3.500	1.020	-

Os campos sem valores indicam as famílias de tamanho maior que 1.024 das quais a solução inicial não é capaz de lidar. Os resultados tornam claro que um problema bem modelado em GPU, como é o caso da solução proposta que explora paralelismo de tarefas e de dados, proporciona *speedups* consideráveis mesmo que a solução em GPU ainda não apresente nenhuma das otimizações que serão apresentadas nas próximas seções.

6.3 Solução com Memória Não-paginada

A utilização de memória não-paginada, como visto na Figura 5.5, simplifica a transferência de dados entre *host* e GPU removendo a cópia de dados entre memória paginável e memória não-paginada.

Tabela 6.5: *Speedups* da solução inicial em GPU em comparação com o HMMER3

Família	*Speedup*			
	HMMER3 1 *core*	HMMER3 4 *cores*	HMMER3 1 *core*/SSE2	HMMER3 4 *cores*/SSE2
Avian_gp85	18,46	7,05	0,60	0,22
CABIT	18,50	7,03	0,61	0,21
DUF530	16,32	8,84	0,52	0,15
PaRep2b	16,33	8,82	0,49	0,15
Flu_PB2	14,08	8,04	0,46	0,14
Totivirus_coat	14,09	8,06	0,45	0,13
ACR_tran	10,24	5,82	0,31	0,09
RdRP_5	-	-	-	-
Bac_GDH	-	-	-	-
AvrE	-	-	-	-

Tabela 6.6: *Speedups* da solução inicial em GPU em comparação com o HMMER3.1b

Família	*Speedup*			
	HMMER3.1b 1 *core*	HMMER3.1b 4 *cores*	HMMER3.1b 1 *core*/SSE2	HMMER3.1b 4 *cores*/SSE2
Avian_gp85	67,32	25,78	0,83	0,42
CABIT	67,35	25,43	0,80	0,44
DUF530	62,29	34,72	0,73	0,26
PaRep2b	62,21	35,29	0,73	0,26
Flu_PB2	55,55	31,77	0,65	0,21
Totivirus_coat	56,43	31,73	0,66	0,20
ACR_tran	43,39	24,68	0,51	0,16
RdRP_5	-	-	-	-
Bac_GDH	-	-	-	-
AvrE	-	-	-	-

Observar apenas os tempos totais de execução dificulta a confirmação de ganho de desempenho uma vez que as transferências representam apenas uma pequena fração de tempo total de execução. Entretanto, a ferramenta *nvprof*, fornecida juntamente com o CUDA, permite analisar o perfil de execução de uma aplicação CUDA apresentando os tempos gastos com as transferências e as execuções de *kernels*. As Figuras 6.2 e 6.3 expõem o perfil de execução do *kernel* MSV com memória paginável e com memória não-paginada, respectivamente. O desempenho das transferências de dados entre *host* e GPU melhorou com essa otimização atingindo *speedup* de até 1,74.

Visto que os tempos gastos com transferências representam apenas uma pequena porcentagem do tempo total de execução da solução, pela Lei de Amdahl [17] o foco para melhorias deve ser dado ao *kernel* que detém mais 90% do tempo gasto pela solução.

```
==11575== NVPROF is profiling process 11575, command: ./myMSVFilter
==11575== Profiling application: ./myMSVFilter
==11575== Profiling result:
Time(%)      Time     Calls       Avg       Min       Max  Name
      %        ms                 ms        ms        ms
  97.47  612.4586         1  612.4586  612.4586  612.4586  void kernelMSV
   2.53  15.86967         4  3.967418  5.15e-03  15.78247  [CUDA memcpy HtoD]
   0.01  0.049091         1  0.049091  0.049091  0.049091  [CUDA memcpy DtoH]
```

Figura 6.2: Tempos gastos com transferências entre *host* e GPU e execução do *kernel* MSV com memória paginável

```
==11705== NVPROF is profiling process 11705, command: ./myMSVFilter
==11705== Profiling application: ./myMSVFilter
==11705== Profiling result:
Time(%)      Time     Calls       Avg       Min       Max  Name
      %        ms                 ms        ms        ms
  98.27  612.5054         1  612.5054  612.5054  612.5054  void kernelMSV
   1.72  10.72722         4  2.681804  6.98e-03  10.62049  [CUDA memcpy HtoD]
   0.01  0.041954         1  0.041954  0.041954  0.041954  [CUDA memcpy DtoH]
```

Figura 6.3: Tempos gastos com transferências entre *host* e GPU e execução do *kernel* MSV com memória não-paginada

Contudo, o uso de memória não-paginada requer poucas mudanças no código, sendo viável mesmo que para pequenos ganhos de desempenho, dada sua simplicidade de uso.

6.4 Solução com Árvore de Máximos e Redução Otimizada

Utilizando a redução otimizada apresentada na Figura 5.7, para cálculo do máximo de uma linha i da matriz de programação dinâmica M, onde técnicas de *loop unrolling* e resolução de comparações em tempo de compilação são realizadas, os resultados da solução melhoraram, alcançando *speedups* de até 1,18, como pode ser visto na Tabela 6.7.

É possível observar que a otimização proporciona melhores resultados no *kernel* MSV, enquanto no *kernel* SSV o ganho de desempenho é quase irrelevante. Isso é compreensível visto que no *kernel* MSV a redução encontra-se dentro do laço e, portanto, é realiza L vezes, uma para cada linha da matriz M, enquanto no *kernel* SSV a redução encontra-se fora do laço e, portanto, é realizada apenas uma vez.

Assim, apesar da ideia de criar vários *kernels* soar complexa, isso é facilmente alcançado utilizando *templates* do C++. Dessa forma, apenas um *kernel* é codificado como um *template* de código e o compilador, a partir desse *template*, gera dez *kernels*

Tabela 6.7: Tempos de execução e *speedups* da solução com redução otimizada em relação à solução inicial, para os *kernels* SSV e MSV

Família	*Kernel* SSV			*Kernel* MSV		
	Tempo de execução (ms)		*Speedup*	Tempo de execução (ms)		*Speedup*
	Solução inicial	Solução otimizada		Solução inicial	Solução otimizada	
Avian_gp85	611,82	610,83	1,00	2.233,67	1.931,10	1,16
CABIT	611,84	611,00	1,00	2.233,68	1.931,10	1,16
DUF530	1.320,16	1.317,98	1,00	5.023,62	4.270,96	1,18
PaRep2b	1.320,18	1.318,07	1,00	5.023,81	4.271,00	1,18
Flu_PB2	2.188,73	2.185,30	1,00	8.659,01	7.594,12	1,14
Totivirus_coat	2.188,72	2.185,32	1,00	8.659,01	7.594,11	1,14
ACR_tran	3.785,47	3.781,50	1,00	16.053,90	13.813,90	1,16
RdRP_5	-	-	-	-	-	-
Bac_GDH	-	-	-	-	-	-
AvrE	-	-	-	-	-	-

otimizados para tamanhos específicos de *profile* HMM.

6.5 Solução com *Scores* Representados com Números Inteiros

Uma suposição feita foi que lidar com números reais, como a solução inicial faz, pode resultar em operações aritméticas mais complexas e que, portanto, a substituição dos números reais por inteiros resultaria em operações aritméticas mais rápidas e consequentemente em ganho de desempenho.

Como pode ser visto na Tabela 6.8, a suposição foi confirmada, o desempenho da solução melhorou com a adoção de números inteiros, alcançando *speedup* de 1,03 no melhor caso.

6.6 Solução com *Scores* Representados com Números Naturais

Ainda trabalhando na suposição anterior, uma nova tentativa, dessa vez substituindo os números inteiros por números naturais, foi feita. A suposição é que lidar com palavras que podem conter valores tanto negativos quanto positivos pode resultar em operações aritméticas mais complexas e, portanto, a substituição dos números inteiros por naturais resultaria em operações aritméticas mais rápidas e ganho de desempenho.

Tabela 6.8: Tempos de execução e *speedups* da solução com números inteiros em relação à solução inicial, para os *kernels* SSV e MSV

Família	*Kernel* SSV			*Kernel* MSV		
	Tempo de execução (ms)		*Speedup*	Tempo de execução (ms)		*Speedup*
	Solução inicial	Solução otimizada		Solução inicial	Solução otimizada	
Avian_gp85	611,82	610,15	1,00	2.233,67	2.186,81	1,02
CABIT	611,84	610,16	1,00	2.233,68	2.186,82	1,02
DUF530	1.320,16	1.297,36	1,02	5.023,62	4.956,74	1,01
PaRep2b	1.320,18	1.297,38	1,02	5.023,81	4.956,78	1,01
Flu_PB2	2.188,73	2.152,78	1,02	8.659,01	8.552,44	1,01
Totivirus_coat	2.188,72	2.152,78	1,02	8.659,01	8.552,43	1,01
ACR_tran	3.785,47	3.729,63	1,01	16.053,90	15.649,90	1,03
RdRP_5	-	-	-	-	-	-
Bac_GDH	-	-	-	-	-	-
AvrE	-	-	-	-	-	-

Novamente a suposição se mostrou correta para o *kernel* SSV, porém os resultados pioraram na maioria dos casos de teste para o *kernel* MSV. Como pode ser visto na Tabela 6.9, a otimização proporcionou *speedups* de até 1,07 para o *kernel* SSV, em relação à solução inicial.

Tabela 6.9: Tempos de execução e *speedups* da solução com números naturais em relação à solução inicial, para os *kernels* SSV e MSV

Família	*Kernel* SSV			*Kernel* MSV		
	Tempo de execução (ms)		*Speedup*	Tempo de execução (ms)		*Speedup*
	Solução inicial	Solução otimizada		Solução inicial	Solução otimizada	
Avian_gp85	611,82	572,11	1,07	2.233,67	2.215,05	1,01
CABIT	611,84	572,11	1,07	2.233,68	2.215,08	1,01
DUF530	1.320,16	1.258,09	1,05	5.023,62	6.192,50	0,81
PaRep2b	1.320,18	1.258,10	1,05	5.023,81	6.192,55	0,81
Flu_PB2	2.188,73	2.124,77	1,03	8.659,01	13.668,20	0,63
Totivirus_coat	2.188,72	2.124,77	1,03	8.659,01	13.668,21	0,63
ACR_tran	3.785,47	3.644,51	1,04	16.053,90	15.854,70	1,01
RdRP_5	-	-	-	-	-	-
Bac_GDH	-	-	-	-	-	-
AvrE	-	-	-	-	-	-

Portanto, as operações em dados de 32 bits que a solução realiza são simples, não há multiplicações, por exemplo, tornando o ganho de desempenho com essas mudanças menos expressivo. O próprio compilador CUDA oferece otimizações com perda de precisão para multiplicações em 24 bits através da *flag* *–use_fast_math*, que fornecem ganho de

desempenho quando multiplicações são feitas em grande quantidade no código.

6.7 Solução com Probabilidades de Emissão na Memória Constante

A característica chave da memória constante, como o próprio nome diz, é ser uma memória de apenas leitura. Outro ponto importante é sua capacidade de fazer *broadcast* de até 32 bits de dados para todas as *threads* de um *warp* em apenas dois ciclos de *clock* [11]. Em outras palavras, se todas as *threads* de um *warp* realizarem leitura em um mesmo endereço, esse acesso pode ser otimizado com o dado sendo distribuído para todas as *threads* do *warp* por *broadcast*. Quando as *threads* não lêem de um mesmo endereço na memória constante, o acesso é sequencial consumindo vários ciclos de *clock* para a leitura que deve ser feita para cada endereço distinto.

Analisando os benefícios da memória constante, suas características e limitação de apenas 64 KB de armazenamento e 8 KB de *cache* no dispositivo usado, a única informação constante, pequena e com acesso ao mesmo endereço por todas as *threads* de um *warp* seria o comprimento das sequências. Visto que o comprimento das sequências são representados como números naturais que ocupam 32 bits, a memória constate conseguiria, portanto, armazenar até 16.384 números dada sua limitação de tamanho. Em outras palavras, a quantidade de sequências enviadas em cada *batch* ficaria limitada a 16.384 com o uso da memória constante.

Como pode ser visto na Tabela 6.10, essa otimização proporcionou *speedups* de até 1,03 no *kernel* SSV, porém proporcionou perda de desempenho na maioria dos casos no *kernel* MSV, em relação à solução inicial.

O uso da memória constante é muito benéfico em dispositivos mais antigos. Todavia, a partir do *Compute Capability* 2.0, a memória global apresenta *cache* L2 que também dispõe do mecanismo de *broadcast* [4], tornando assim a memória constante redundante e presente apenas para questões de compatibilidade com códigos legados. Dispositivos com ao menos *Compute Capability* 2.0, como é o caso da GPU sendo utilizada, permitem ao desenvolvedor acessar a memória global com a eficiência da memória constante. Para tal, o compilador precisa reconhecer dados com as seguintes características [11]:

- dados que residem na memória global;
- dados são apenas lidos pelo *kernel* (palavra reservada *const* da linguagem C);
- dados não devem depender do identificador de cada *thread*.

Tabela 6.10: Tempos de execução e *speedups* da solução com memória constante em relação à solução inicial, para os *kernels* SSV e MSV

Família	*Kernel* SSV			*Kernel* MSV		
	Tempo de execução (ms)		*Speedup*	Tempo de execução (ms)		*Speedup*
	Solução inicial	Solução otimizada		Solução inicial	Solução otimizada	
Avian_gp85	611,82	612,16	1,00	2.233,67	2.284,20	0,98
CABIT	611,84	612,18	1,00	2.233,68	2.284,21	0,98
DUF530	1.320,16	1.301,56	1,01	5.023,62	5.020,44	1,00
PaRep2b	1.320,18	1.301,60	1,01	5.023,81	5.020,52	1,00
Flu_PB2	2.188,73	2.170,92	1,01	8.659,01	8.725,04	0,99
Totivirus_coat	2.188,72	2.170,93	1,01	8.659,01	8.725,06	0,99
ACR_tran	3.785,47	3.691,20	1,03	16.053,90	16.022,64	1,00
RdRP_5	-	-	-	-	-	-
Bac_GDH	-	-	-	-	-	-
AvrE	-	-	-	-	-	-

Portanto, com os resultados não muito expressivos e a limitação na quantidade de sequências, o uso da memória constante foi descartado e o uso da palavra reservada *const* foi mantido, como na solução inicial.

6.8 Solução com Probabilidades de Emissão na Memória de Textura

Uma das características da memória de textura é sua *cache* otimizada à localidade espacial 2D, favorecendo o acesso aos dados de regiões vizinhas, algo semelhante ao mostrado na Figura 6.4. Entretanto, quando alocada como um vetor unidimensional, ela tende a funcionar como uma *cache* linear. Assim, analisando o problema e as estruturas de dados, os dados que podem desfrutar dessa localidade espacial são as probabilidades de emissão.

O uso da memória de textura requer algumas mudanças mais trabalhosas no código, que vão desde a alocação, transferência e até a leitura que é realizada com um comando especial denominado *tex1Dfetch()*. Como pode ser visto na Tabela 6.11, essa otimização proporcionou ganho de desempenho no *kernel* SSV, alcançando *speedups* de até 1,16, enquanto no *kernel* MSV a situação foi oposta com resultados piores para a maioria dos casos de teste.

Em suma, nos casos de teste do *kernel* MSV a memória de textura com sua *cache* limitada a 8 KB por SM e otimização para localidade espacial não se mostrou eficiente o suficiente para superar em desempenho a memória global, que por sua vez possui uma

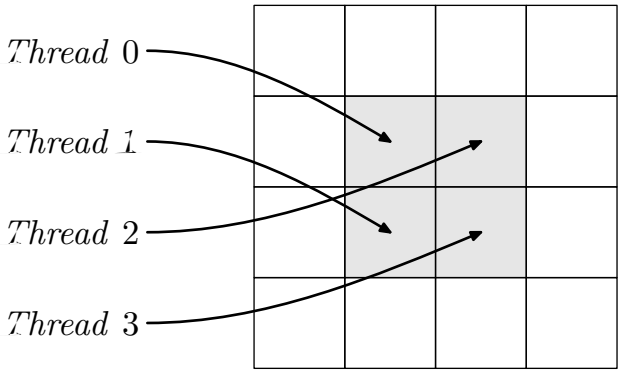


Figura 6.4: Exemplo de caso de uso que usufrui da *cache* da memória de textura que é otimizada para localidade espacial 2D

Tabela 6.11: Tempos de execução e *speedups* da solução com memória de textura em relação à solução inicial, para os *kernels* SSV e MSV

Família	*Kernel* SSV			*Kernel* MSV		
	Tempo de execução (ms)		*Speedup*	Tempo de execução (ms)		*Speedup*
	Solução inicial	Solução otimizada		Solução inicial	Solução otimizada	
Avian_gp85	611,82	609,58	1,00	2.233,67	2.261,05	0,99
CABIT	611,84	609,59	1,00	2.233,68	2.261,02	0,99
DUF530	1.320,16	1.222,91	1,08	5.023,62	5.003,32	1,00
PaRep2b	1.320,18	1.222,92	1,08	5.023,81	5.003,36	1,00
Flu_PB2	2.188,73	1.888,10	1,16	8.659,01	8.773,75	0,99
Totivirus_coat	2.188,72	1.888,10	1,16	8.659,01	8.773,76	0,99
ACR_tran	3.785,47	3.677,02	1,03	16.053,90	16.409,10	0,98
RdRP_5	-	-	-	-	-	-
Bac_GDH	-	-	-	-	-	-
AvrE	-	-	-	-	-	-

cache L1 que também explora localidade espacial e no dispositivo usado possui 48 KB. Todavia, representou uma contribuição positiva para a melhoria do *kernel* SSV.

6.9 Solução com *Padding* nas Probabilidades de Emissão

Visto que para obter acesso coalescido, o endereço base acessado deve ser múltiplo do valor na coluna Alinhamento da Tabela 2.1, manter os dados alinhados é fundamental para alcançar esse objetivo.

A estrutura de dados das probabilidades de emissão pode ser vista como uma matriz de 29 linhas e Q colunas, onde cada célula contém um valor de 32 bits referente a

uma probabilidade. Cada *thread* acessa endereços contíguos e crescentes desta estrutura durante a execução, portanto o tamanho Q do *profile* HMM vai ditar se há alinhamento ou não.

Em geral, o alinhamento é obtido adicionando algumas colunas vazias e sem uso ao final da matriz. Essas células, denominadas *halo cells* ou por vezes *ghost cells*, representam desperdício de espaço, porém proporcionam alinhamento. O modelo CUDA oferece as funções *cudaMallocPitch()* e *cudaMemcpy2D()* para facilitar o desenvolvimento utilizando a técnica de *padding*. A primeira aloca memória incluindo espaço extra para as *halo cells*, já a segunda copia uma matriz para memória do dispositivo, alinhando o início de cada linha da matriz através da inserção de *halo cells* ao final das linhas, se necessário.

Como pode ser visto na Tabela 6.12, essa otimização proporcionou ganhos de desempenho no *kernel* SSV, com *speedups* de até 1,05 quando comparado com a solução inicial, porém proporcionou resultados piores do que a solução inicial no *kernel* MSV.

Tabela 6.12: Tempos de execução e *speedups* da solução com *padding* nas probabilidades de emissão em relação à solução inicial, para os *kernels* SSV e MSV

Família	*Kernel* SSV			*Kernel* MSV		
	Tempo de execução (ms)		*Speedup*	Tempo de execução (ms)		*Speedup*
	Solução inicial	Solução otimizada		Solução inicial	Solução otimizada	
Avian_gp85	611,82	611,36	1,00	2.233,67	2.270,52	0,98
CABIT	611,84	611,35	1,00	2.233,68	2.270,52	0,98
DUF530	1.320,16	1.303,13	1,01	5.023,62	5.110,99	0,98
PaRep2b	1.320,18	1.303,13	1,01	5.023,81	5.111,00	0,98
Flu_PB2	2.188,73	2.087,30	1,05	8.659,01	9.104,78	0,95
Totivirus_coat	2.188,72	2.087,30	1,05	8.659,01	9.104,80	0,95
ACR_tran	3.785,47	3.670,38	1,03	16.053,90	16.401,60	0,98
RdRP_5	-	-	-	-	-	-
Bac_GDH	-	-	-	-	-	-
AvrE	-	-	-	-	-	-

6.10 Solução com *Padding* nas Sequências

A técnica de *padding* também pode ser adotada com as sequências, porém dada a maior disparidade entre os comprimentos das sequências, isso representaria um maior desperdício de espaço de memória. A quantidade de colunas da matriz teria que ser ao menos o tamanho da maior sequência, adicionando *halo cells* em todas as linhas que representam sequências menores.

Esse desperdício pode ser minimizado ordenando as sequências por comprimento.

Dessa forma, os *batches* são formados por sequências de tamanho semelhante o que reduz as disparidades nos comprimentos, além do que a ordenação das sequências também favorece o balanceamento de carga na solução uma vez que cada bloco agora possui uma quantidade de trabalho semelhante.

Como pode ser visto na Tabela 6.13, essa otimização proporcionou *speedups* de até 1,03 para o *kernel* SSV, porém proporcionou resultados piores do que a solução inicial na maioria dos casos de teste no *kernel* MSV.

Tabela 6.13: Tempos de execução e *speedups* da solução com *padding* nas sequências em relação à solução inicial, para os *kernels* SSV e MSV

Família	*Kernel* SSV			*Kernel* MSV		
	Tempo de execução (ms)		*Speedup*	Tempo de execução (ms)		*Speedup*
	Solução inicial	Solução otimizada		Solução inicial	Solução otimizada	
Avian_gp85	611,82	611,19	1,00	2.233,67	2.282,73	0,98
CABIT	611,84	611,21	1,00	2.233,68	2.282,74	0,98
DUF530	1.320,16	1.298,79	1,02	5.023,62	5.016,71	1,00
PaRep2b	1.320,18	1.298,80	1,02	5.023,81	5.016,81	1,00
Flu_PB2	2.188,73	2.167,05	1,01	8.659,01	8.720,23	0,99
Totivirus_coat	2.188,72	2.167,05	1,01	8.659,01	8.720,24	0,99
ACR_tran	3.785,47	3.687,79	1,03	16.053,90	16.017,70	1,00
RdRP_5	-	-	-	-	-	-
Bac_GDH	-	-	-	-	-	-
AvrE	-	-	-	-	-	-

6.11 Solução com *Loop Unrolling*

O modelo CUDA permite que um laço seja facilmente desenrolado simplesmente acrescentando uma linha com a diretiva *#pragma unroll* antes do mesmo. Além disso, é possível especificar o fator no qual esse laço será desenrolado acrescentando um número no fim dessa linha. Nos experimentos realizados os melhores resultados foram obtidos quando o fator foi definido em 8.

Como pode ser visto na Tabela 6.14, essa otimização proporcionou ganho de desempenho para o *kernel* SSV, com *speedups* de até 1,54 quando comparado com a solução inicial, porém proporcionou resultados piores do que a solução inicial na maioria dos casos de teste do *kernel* MSV.

Os resultados obtidos para o *kernel* SSV foram impressionantes visto que a única mudança feita no código foi a adição de uma simples diretiva. Todavia, como dito anteriormente, desenrolar um laço também tem implicações negativas, como o aumento

Tabela 6.14: Tempos de execução e *speedups* da solução com *loop unrolling* em relação à solução inicial, para os *kernels* SSV e MSV

Família	*Kernel* SSV			*Kernel* MSV		
	Tempo de execução (ms)		*Speedup*	Tempo de execução (ms)		*Speedup*
	Solução inicial	Solução otimizada		Solução inicial	Solução otimizada	
Avian_gp85	611,82	398,24	1,54	2.233,67	2.174,00	1,03
CABIT	611,84	398,25	1,54	2.233,68	2.174,03	1,03
DUF530	1.320,16	895,78	1,47	5.023,62	5.868,26	0,86
PaRep2b	1.320,18	895,78	1,47	5.023,81	5.868,29	0,86
Flu_PB2	2.188,73	1.528,52	1,43	8.659,01	13.137,80	0,66
Totivirus_coat	2.188,72	1.528,52	1,43	8.659,01	13.137,80	0,66
ACR_tran	3.785,47	2.545,66	1,49	16.053,90	15.328,00	1,05
RdRP_5	-	-	-	-	-	-
Bac_GDH	-	-	-	-	-	-
AvrE	-	-	-	-	-	-

no número de registradores alocados e isso, infelizmente, influenciou o desempenho do *kernel* MSV negativamente.

6.12 Solução com Múltiplos *Streams*

Nessa otimização, novos *batches* de dados são transferidos do *host* para a GPU enquanto ocorre a execução de *kernels* na GPU, mascarando o tempo de transferência que pode ser um gargalo na solução e garantindo um fluxo contínuo de dados para serem processados pela GPU.

Todavia, um ponto importante é saber o tamanho ideal de cada *batch* de forma atingir o máximo de velocidade nas transferências. No experimento realizado para determinação do número de sequências ideal do *batch*, foi utilizado um conjunto com 65.535 sequências com comprimento médio de 1.024. Oito casos de teste foram compostos objetivando determinar o tamanho de *batch* que apresenta melhor desempenho. Em cada caso, o conjunto de sequências foi dividido em *batches* de tamanhos iguais contendo de 512 a 65.535 sequências dependendo do caso. A Figura 6.5 apresenta os tempos de transferência obtidos no experimento.

Os resultados desse experimento empírico mostram que o melhor desempenho é alcançado com *batches* com ao menos 32.768 sequências e que o ganho de desempenho é praticamente irrelevante para *batches* maiores. Os resultados se assemelham aos obtidos em um experimento feito por Cook [4] que constatou que conjuntos de dados de ao menos 32 MB são suficientes para se alcançar os picos de velocidade durante as transferências

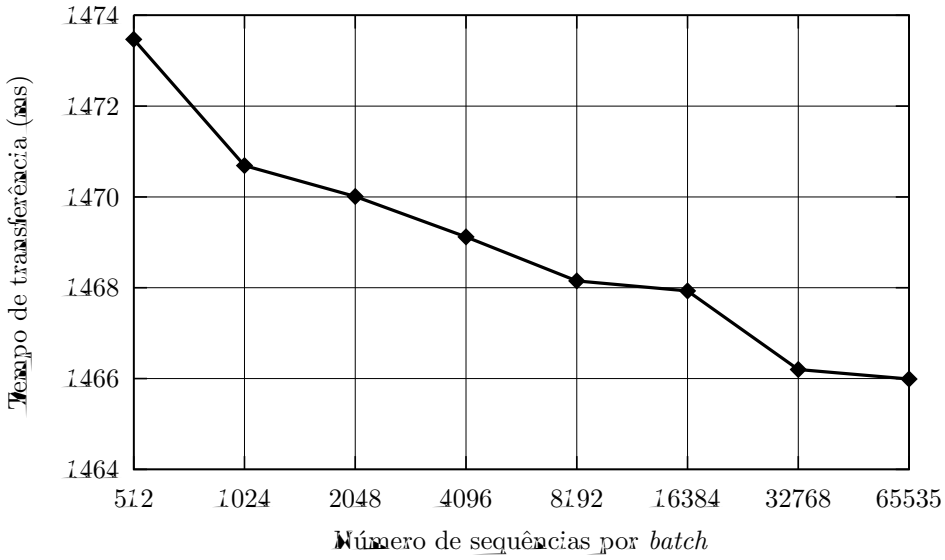


Figura 6.5: Determinação do tamanho do *batch* que proporciona velocidade máxima de transferência

de dados entre *host* e GPU.

Logo, observando a Tabela 6.2 onde o comprimento médio das sequências sendo usadas é de 355,24, fica claro que a solução deve usar *batches* de 65.535 para obter seu melhor desempenho. Ou seja, a solução resolve um *batch* de até 65.535 sequências, recupera os resultados da memória do dispositivo e então envia outro *batch* com no máximo 65.535 sequências, até que todas as sequências tenham seus *scores* calculados.

Assim como foi constatado na otimização com memória não-paginada, os tempos de transferências são apenas uma pequena fração do tempo total de execução, o que dificulta a identificação de melhoria de desempenho com a adoção de *streams*. Contudo, a Figura 6.6, proveniente da ferramenta Visual Profiler [39], oferecida pela NVIDIA para análise de desempenho de aplicações CUDA, mostra como as transferências de *batches* de sequências estão sendo sobrepostas com a execução de *kernels*. Cada linha representa um *stream*, os blocos verdes representam a execução de *kernels* e o bloco amarelo transferência de dados. Perto do fim da execução do *kernel* no primeiro *stream*, boa parte dos recursos do hardware já estão ociosos o que permite o início da execução do *kernel* no segundo *stream*, ou seja, o tempo que o hardware fica ocioso é minimizado. Além disso, após a execução do *kernel* no primeiro *stream* é também possível a realização de uma transferência concomitantemente com a execução de um *kernel* no segundo *stream*, ou seja, ao fim da execução desse *kernel* a GPU já possuirá dados em memória para a execução do próximo *kernel*, novamente minimizando o tempo de hardware ocioso.

Portanto, a introdução do uso de *streams* na solução inicial proporcionou melhor aproveitamento dos recursos da GPU na realização simultânea de algumas tarefas que

antes eram realizadas sequencialmente e, consequentemente, um pequeno ganho de desempenho.

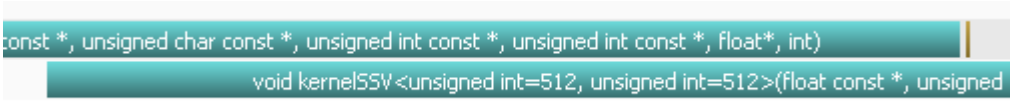


Figura 6.6: Análise visual da sobreposição da execução de *kernels* e transferências de memória quando usado *streams* no filtro SSV em GPU

6.13 Solução com Acesso Vetorizado às Probabilidades de Emissão

Com as GPUs modernas atingindo teraflops de poder de processamento e as memórias não acompanhando esse crescimento, *kernels* que são *bandwidth bound* ficam mais comuns. Assim, eliminar gargalos que prejudicam a largura de banda nos acessos à memória é essencial. Para isso, é preciso realizar operações que leiam/escrevam mais dados na memória ao mesmo tempo e também minimizar as instruções sendo executadas.

O modelo CUDA fornece tipos especiais de dados *int2*, *int4*, *float2*, *float4*, etc. Esses tipos de dados permitem acesso vetorizado à memória e podem ser facilmente manipulados pelo *host* como tipos convencionais (*int* e *float*) e pela GPU por *casting* nesses tipos convencionais através do comando *reinterpret_cast<T>()* do C++.

6.13.1 *Scores* Representados com Números Reais e Acesso Vetorizado com Fator 2

Nessa otimização, a ideia é que cada *thread* calcule duas células de cada linha da matriz M, permitindo ao código ganho de generalidade, uma vez que o mesmo torna-se capaz de tratar *profile* HMMs de tamanho até 2.048, e também ganho de desempenho ao utilizar o acesso vetorizado às probabilidades.

Como pode ser visto na Tabela 6.15, a solução com números reais e acesso vetorizado com fator 2 proporcionou ganho de desempenho em ambos os *kernels*, alcançando *speedups* de até 2,6 quando comparada com a solução inicial.

Visto que essa solução otimizada é a primeira capaz de lidar com *profile* HMMs com mais que 1.024 nós e que até então não havia tempos para comparação desses casos de teste, os tempos obtidos nessa solução para as famílias maiores que 1.024 serão considerados como os tempos de referência daqui por diante.

Tabela 6.15: Tempos de execução e *speedups* da solução com números reais e acesso vetorizado às probabilidades de emissão com fator 2 em relação à solução inicial, para os *kernels* SSV e MSV

Família	*Kernel* SSV			*Kernel* MSV		
	Tempo de execução (ms)		*Speedup*	Tempo de execução (ms)		*Speedup*
	Solução inicial	Solução otimizada		Solução inicial	Solução otimizada	
Avian_gp85	611,82	413,75	1,48	2.233,67	1.363,46	1,64
CABIT	611,84	413,74	1,48	2.233,68	1.363,46	1,64
DUF530	1.320,16	899,59	1,47	5.023,62	2.534,28	1,98
PaRep2b	1.320,18	899,60	1,47	5.023,81	2.534,30	1,98
Flu_PB2	2.188,73	1.607,75	1,36	8.659,01	4.496,30	1,93
Totivirus_coat	2.188,72	1.607,74	1,36	8.659,01	4.496,31	1,93
ACR_tran	3.785,47	2.188,60	1,73	16.053,90	6.185,99	2,60
RdRP_5	-	2.890,42	-	-	9.083,23	-
Bac_GDH	-	3.422,32	-	-	10.559,90	-
AvrE	-	4.524,00	-	-	17.377,10	-

6.13.2 *Scores* Representados com Números Reais e Acesso Vetorizado com Fator 4

Seguindo a proposta da otimização anterior, essa otimização objetiva saber se aumentar o fator do acesso vetorizado de 2 para 4 resulta em ganhos ainda maiores de desempenho. Como pode ser visto na Tabela 6.16, a solução com números reais e acesso vetorizado com fator 4 não proporcionou resultados melhores que a solução com fator 2, para ambos os *kernels*.

6.13.3 *Scores* Representados com Números Naturais e Acesso Vetorizado com Fator 2

Uma vez que tanto a otimização com números naturais quanto a otimização com acesso vetorizado proporcionaram ganho de desempenho, ao menos ao *kernel* SSV, essa otimização resume-se a unir o acesso vetorizado com números naturais. Como pode ser visto na Tabela 6.17, a solução com números naturais e acesso vetorizado com fator 2 proporcionou ganho de desempenho ainda maior, alcançando *speedups* de até 2,62 quando comparada com a solução inicial.

Tabela 6.16: Tempos de execução e *speedups* da solução com números reais e acesso vetorizado às probabilidades de emissão com fator 4 em relação à solução inicial, para os *kernels* SSV e MSV

Família	*Kernel* SSV			*Kernel* MSV		
	Tempo de execução (ms)		*Speedup*	Tempo de execução (ms)		*Speedup*
	Solução inicial	Solução otimizada		Solução inicial	Solução otimizada	
Avian_gp85	611,82	509,03	1,20	2.233,67	1.642,40	1,36
CABIT	611,84	509,05	1,20	2.233,68	1.642,41	1,36
DUF530	1.320,16	973,58	1,36	5.023,62	3.394,34	1,48
PaRep2b	1.320,18	973,58	1,36	5.023,81	3.394,49	1,48
Flu_PB2	2.188,73	1.820,80	1,20	8.659,01	5.850,68	1,48
Totivirus_coat	2.188,72	1.820,81	1,20	8.659,01	5.850,67	1,48
ACR_tran	3.785,47	2.481,52	1,53	16.053,90	9.333,66	1,72
RdRP_5	2.890,42	3.135,45	0,92	9.083,23	9.981,57	0,91
Bac_GDH	3.422,32	3.746,62	0,91	10.559,90	11.478,15	0,92
AvrE	4.524,00	4.510,27	1,00	17.377,10	18.101,15	0,96

Tabela 6.17: Tempos de execução e *speedups* da solução com números naturais e acesso vetorizado às probabilidades de emissão com fator 2 em relação à solução inicial, para os *kernels* SSV e MSV

Família	*Kernel* SSV			*Kernel* MSV		
	Tempo de execução (ms)		*Speedup*	Tempo de execução (ms)		*Speedup*
	Solução inicial	Solução otimizada		Solução inicial	Solução otimizada	
Avian_gp85	611,82	390,18	1,57	2.233,67	1.311,33	1,70
CABIT	611,84	390,19	1,57	2.233,68	1.311,32	1,70
DUF530	1.320,16	850,44	1,55	5.023,62	2.433,94	2,06
PaRep2b	1.320,18	850,44	1,55	5.023,81	2.433,97	2,06
Flu_PB2	2.188,73	1.527,64	1,43	8.659,01	4.407,12	1,96
Totivirus_coat	2.188,72	1.527,63	1,43	8.659,01	4.407,13	1,96
ACR_tran	3.785,47	2.094,67	1,81	16.053,90	6.137,33	2,62
RdRP_5	2.890,42	2.755,10	1,05	9.083,23	9.099,87	1,00
Bac_GDH	3.422,32	3.269,39	1,05	10.559,90	10.646,90	0,99
AvrE	4.524,00	4.432,35	1,02	17.377,10	16.949,10	1,03

6.13.4 *Scores* Representados com Números Naturais e Acesso Vetorizado com Fator 4

Novamente seguindo a tendência da otimização com números reais, o aumento do fator de 2 para 4 não acarretou em resultados melhores. Como pode ser visto na Tabela 6.18, a solução com números naturais e acesso vetorizado com fator 4 não proporcionou resultados melhores que a solução com fator 2, para ambos os *kernels*.

Tabela 6.18: Tempos de execução e *speedups* da solução com números naturais e acesso vetorizado às probabilidades de emissão com fator 4 em relação à solução inicial, para os *kernels* SSV e MSV

Família	*Kernel* SSV			*Kernel* MSV		
	Tempo de execução (ms)		*Speedup*	Tempo de execução (ms)		*Speedup*
	Solução inicial	Solução otimizada		Solução inicial	Solução otimizada	
Avian_gp85	611,82	484,03	1,26	2.233,67	1.692,17	1,32
CABIT	611,84	484,05	1,26	2.233,68	1.692,17	1,32
DUF530	1.320,16	939,20	1,41	5.023,62	3.159,51	1,59
PaRep2b	1.320,18	939,21	1,41	5.023,81	3.159,63	1,59
Flu_PB2	2.188,73	1.754,19	1,25	8.659,01	5.659,48	1,53
Totivirus_coat	2.188,72	1.754,21	1,25	8.659,01	5.659,49	1,53
ACR_tran	3.785,47	2.367,23	1,60	16.053,90	8.361,41	1,92
RdRP_5	2.890,42	2.999,61	0,96	9.083,23	9.766,91	0,93
Bac_GDH	3.422,32	3.547,93	0,96	10.559,90	11.604,29	0,91
AvrE	4.524,00	4.268,04	1,06	17.377,10	17.552,63	0,99

6.13.5 *Scores* Representados com Números Naturais e Acesso Vetorizado com Fator 16

A ideia nessa otimização é aumentar ainda mais o número de células calculadas por cada *thread* sem, no entanto, aumentar o número de transações a memória e, dessa forma, aumentar o número de operações aritméticas realizadas por cada *thread* contribuindo para a proporção computação/memória de cada *thread* que muitas vezes é tido como uma métrica para determinar a eficiência do código.

Para tal, cada valor natural, que originalmente possui 32 bits, sofre transformações através de operações de *shift* e *and* lógico para se extrair quatro valores de 8 bits cada. Em outras palavras, nessa otimização cada *thread* é responsável pelo cálculo de 16 células. Como pode ser visto na Tabela 6.19, a solução com números naturais e acesso vetorizado com fator 16 proporcionou ganhos de desempenho ainda maiores, alcançando *speedup* de até 6,48.

Os experimentos mostraram que o acesso vetorizado às probabilidades de emissão com algum fator proporciona melhoria de desempenho. Contudo, o acesso vetorizado aumenta o número de registradores usados e, portanto, para *kernels* que já excederam o número de registradores ideal, essa abordagem agrava ainda mais o problema.

Tabela 6.19: Tempos de execução e *speedups* da solução com números naturais e acesso vetorizado às probabilidades de emissão com fator 16 em relação à solução inicial, para os *kernels* SSV e MSV

Família	*Kernel* SSV			*Kernel* MSV		
	Tempo de execução (ms)		*Speedup*	Tempo de execução (ms)		*Speedup*
	Solução inicial	Solução otimizada		Solução inicial	Solução otimizada	
Avian_gp85	611,82	627,54	0,97	2.233,67	1.065,30	2,10
CABIT	611,84	627,57	0,97	2.233,68	1.065,31	2,10
DUF530	1.320,16	711,26	1,86	5.023,62	1.409,71	3,56
PaRep2b	1.320,18	711,22	1,86	5.023,81	1.409,73	3,56
Flu_PB2	2.188,73	1.037,81	2,11	8.659,01	1.839,38	4,71
Totivirus_coat	2.188,72	1.037,81	2,11	8.659,01	1.839,37	4,71
ACR_tran	3.785,47	1.331,40	2,84	16.053,90	2.477,04	6,48
RdRP_5	2.890,42	1.754,72	1,65	9.083,23	3.159,26	2,88
Bac_GDH	3.422,32	2.165,42	1,58	10.559,90	4.109,87	2,57
AvrE	4.524,00	2.387,09	1,90	17.377,10	4.321,36	4,02

6.14 Solução com Acesso Vetorizado às Sequências

O acesso vetorizado com fator 2 proporciona melhor desempenho e os bits provenientes da cada transação podem sofrer transformações através de operações de *shift* e *and* lógico para se extrair valores de oito bits cada. Nessa otimização, a ideia é carregar vários símbolos (caracteres) da sequência de uma vez, reduzindo os acessos a memória, e não deixar a eficiência dessas requisições apenas a cargo da memória *cache*.

Para tal, uma requisição de 64 bits dados é realizada, e quanto atendida, oito caracteres são armazenados nos registradores/memória local, garantido que as próximas sete iterações do laço não realizarão mais requisições de caracteres. Como pode ser visto na Tabela 6.20, essa otimização proporcionou *speedups* de até 1,51 no *kernel* SSV, quando comparado com a solução inicial, porém proporcionou resultados piores para o *kernel* MSV.

6.15 Solução com *Tiling*

Nessa solução, a ideia é tornar cada *thread* responsável por uma porcentagem maior da computação do problema. Para tal, uma porção do problema é resolvida por um conjunto de *threads* que após a conclusão resolve uma nova porção e assim sucessivamente. Dessa forma, a ocupação do dispositivo é favorecida e também a solução ganha em generalidade, visto que a mesma consegue também lidar com *profile* HMMs maiores que

Tabela 6.20: Tempos de execução e *speedups* da solução com acesso vetorizado às sequências em relação à solução inicial, para os *kernels* SSV e MSV

Família	*Kernel* SSV			*Kernel* MSV		
	Tempo de execução (ms)		*Speedup*	Tempo de execução (ms)		*Speedup*
	Solução inicial	Solução otimizada		Solução inicial	Solução otimizada	
Avian_gp85	611,82	405,44	1,51	2.233,67	2.856,54	0,78
CABIT	611,84	405,45	1,51	2.233,68	2.856,53	0,78
DUF530	1.320,16	918,24	1,44	5.023,62	6.158,26	0,82
PaRep2b	1.320,18	918,23	1,44	5.023,81	6.158,29	0,82
Flu_PB2	2.188,73	1.522,35	1,44	8.659,01	10.458,90	0,83
Totivirus_coat	2.188,72	1.522,37	1,44	8.659,01	10.458,91	0,83
ACR_tran	3.785,47	2.583,65	1,47	16.053,90	17.649,90	0,91
RdRP_5	-	-	-	-	-	-
Bac_GDH	-	-	-	-	-	-
AvrE	-	-	-	-	-	-

1.024.

6.15.1 *Tiling* com Fator 2

Como pode ser visto na Tabela 6.21, a solução com *tiling* com fator 2 proporcionou ganho de desempenho em ambos os *kernels*, alcançando *speedups* de até 2,74 quando comparados com a solução inicial.

Tabela 6.21: Tempos de execução e *speedups* da solução com *tiling* com fator 2 em relação à solução inicial, para os *kernels* SSV e MSV

Família	*Kernel* SSV			*Kernel* MSV		
	Tempo de execução (ms)		*Speedup*	Tempo de execução (ms)		*Speedup*
	Solução inicial	Solução otimizada		Solução inicial	Solução otimizada	
Avian_gp85	611,82	423,21	1,45	2.233,67	1.375,10	1,62
CABIT	611,84	423,21	1,45	2.233,68	1.375,12	1,62
DUF530	1.320,16	864,06	1,53	5.023,62	2.591,20	1,94
PaRep2b	1.320,18	864,05	1,53	5.023,81	2.591,21	1,94
Flu_PB2	2.188,73	1.448,72	1,51	8.659,01	4.289,68	2,02
Totivirus_coat	2.188,72	1.448,74	1,51	8.659,01	4.289,70	2,02
ACR_tran	3.785,47	2.022,51	1,87	16.053,90	5.855,42	2,74
RdRP_5	2.890,42	2.653,99	1,09	9.083,23	8.812,66	1,03
Bac_GDH	3.422,32	3.255,37	1,05	10.559,90	9.970,96	1,06
AvrE	4.524,00	4.678,14	0,97	17.377,10	16.255,60	1,07

6.15.2 *Tiling* com Fator 4

Seguindo a proposta da otimização anterior, essa otimização objetiva saber se reduzir a quantidade de *threads* e aumentar ainda mais a quantidade de computação que cada *thread* realiza resulta em ganhos ainda maiores de desempenho. Como pode ser visto na Tabela 6.22, a solução com *tiling* com fator 4 proporcionou ganhos de desempenho ainda maiores, alcançando *speedups* de até 4,87.

Tabela 6.22: Tempos de execução e *speedups* da solução com *tiling* com fator 4 em relação à solução inicial, para os *kernels* SSV e MSV

Família	*Kernel* SSV			*Kernel* MSV		
	Tempo de execução (ms)		*Speedup*	Tempo de execução (ms)		*Speedup*
	Solução inicial	Solução otimizada		Solução inicial	Solução otimizada	
Avian_gp85	611,82	379,55	1,61	2.233,67	1.089,93	2,05
CABIT	611,84	379,58	1,61	2.233,68	1.089,95	2,05
DUF530	1.320,16	643,47	2,05	5.023,62	1.596,33	3,15
PaRep2b	1.320,18	643,49	2,05	5.023,81	1.596,34	3,15
Flu_PB2	2.188,73	1.124,94	1,95	8.659,01	2.445,99	3,54
Totivirus_coat	2.188,72	1.124,95	1,95	8.659,01	2.445,72	3,54
ACR_tran	3.785,47	1.574,42	2,40	16.053,90	3.294,03	4,87
RdRP_5	2.890,42	2.017,16	1,43	9.083,23	4.459,37	2,04
Bac_GDH	3.422,32	2.531,65	1,35	10.559,90	5.740,46	1,84
AvrE	4.524,00	2.984,22	1,52	17.377,10	6.270,65	2,77

6.15.3 *Tiling* com Fator 8

Infelizmente continuar aumentando o fator de *tiling* não proporcionou ganhos ainda maiores de desempenho. Como pode ser visto na Tabela 6.23, a solução com *tiling* com fator 8 não proporcionou resultados melhores que a solução com fator 4, para ambos os *kernels*.

6.16 Solução Otimizada Final

Analisando todas as otimizações realizadas e comparadas com a solução inicial, a Tabela 6.24 lista as otimizações julgadas interessantes para a associação e construção da solução final de cada filtro.

Uma vez que basicamente todas as otimizações resultaram em ganho de desempenho no *kernel* SSV, apenas a otimização com memória constante e a otimização

Tabela 6.23: Tempos de execução e *speedups* da solução com *tiling* com fator 8 em relação à solução inicial, para os *kernels* SSV e MSV

Família	*Kernel* SSV			*Kernel* MSV		
	Tempo de execução (ms)		*Speedup*	Tempo de execução (ms)		*Speedup*
	Solução inicial	Solução otimizada		Solução inicial	Solução otimizada	
Avian_gp85	611,82	1.241,49	0,49	2.233,67	1.734,04	1,29
CABIT	611,84	1.241,48	0,49	2.233,68	1.734,02	1,29
DUF530	1.320,16	1.757,95	0,75	5.023,62	2.060,66	2,44
PaRep2b	1.320,18	1.757,94	0,75	5.023,81	2.060,69	2,44
Flu_PB2	2.188,73	2.908,00	0,75	8.659,01	2.939,57	2,95
Totivirus_coat	2.188,72	2.908,10	0,75	8.659,01	2.939,56	2,95
ACR_tran	3.785,47	4.301,60	0,88	16.053,90	3.994,27	4,02
RdRP_5	2.890,42	5.347,62	0,54	9.083,23	5.226,70	1,74
Bac_GDH	3.422,32	7.030,36	0,49	10.559,90	6.756,85	1,56
AvrE	4.524,00	7.953,21	0,57	17.377,10	7.528,73	2,31

Tabela 6.24: Otimizações adotadas na solução final dos filtros SSV e MSV

Otimização	*kernel* SSV	*kernel* MSV
Memória não-paginada	•	•
Árvore de máximos e redução otimizada	•	•
Scores representados com números inteiros		
Scores representados com números naturais	•	•
Probabilidades de emissão na memória constante		
Probabilidades de emissão na memória de textura	•	
Padding nas probabilidades de emissão	•	
Padding nas sequências	•	
Ordenação das sequências	•	•
Loop unrolling	•	•
Múltiplos *streams*	•	•
Acesso vetorizado às probabilidades de emissão	•	•
Acesso vetorizado às sequências	•	
Tiling	•	•

com números inteiros, que é mutuamente exclusiva com a otimização com números naturais, não foram adotadas. Já para o *kernel* MSV, apenas algumas poucas otimizações resultaram em ganho de desempenho. Entretanto, apesar de algumas otimizações terem resultados negativos isoladamente, como é o caso do *loop unrolling* e do acesso vetorizado às probabilidades de emissão, em conjunto final, elas resultaram em ganho de desempenho. Com a introdução do *tiling*, por exemplo, um novo laço foi criado, com uma nova possibilidade de *loop unrolling*, e nesse caso a utilização da técnica gerou ganho de desempenho.

A Tabela 6.25 apresenta os tempos de execução da solução otimizada final em GPU e do HMMER3, para cada configuração sua. Na Tabela 6.26 os resultados apresentados referem-se ao HMMER3.1b.

Tabela 6.25: Tempos de execução da solução final em GPU e do HMMER3

Família	Tempo de execução (ms)				
	HMMER3 1 *core*	HMMER3 4 *cores*	HMMER3 1 *core*/SSE2	HMMER3 4 *cores*/SSE2	Solução em GPU
Avian_gp85	41.240	15.750	1.340	490	829,33
CABIT	41.320	15.710	1.370	460	829,51
DUF530	81.980	44.400	2.590	770	892,96
PaRep2b	82.060	44.320	2.470	740	892,97
Flu_PB2	121.940	69.600	3.950	1.190	1.372,46
Totivirus_coat	122.010	69.780	3.860	1.120	1.372,48
ACR_tran	164.450	93.490	5.020	1.420	1.371,62
RdRP_5	205.900	116.320	6.240	1.870	1.651,75
Bac_GDH	247.000	140.100	7.270	2.080	1.653,46
AvrE	285.510	162.440	8.360	2.370	2.301,94

Tabela 6.26: Tempos de execução da solução final em GPU e do HMMER3.1b

Família	Tempo de execução (ms)				
	HMMER3.1b 1 *core*	HMMER3.1b 4 *cores*	HMMER3.1b 1 *core*/SSE2	HMMER3.1b 4 *cores*/SSE2	Solução em GPU
Avian_gp85	41.190	15.770	510	260	307,35
CABIT	41.210	15.560	490	270	307,33
DUF530	82.230	45.840	960	340	310,01
PaRep2b	82.130	46.590	960	340	310,00
Flu_PB2	121.580	69.540	1.420	450	543,75
Totivirus_coat	123.510	69.440	1.440	440	543,75
ACR_tran	164.240	93.440	1.930	590	535,39
RdRP_5	205.580	116.440	2.400	710	827,67
Bac_GDH	247.230	139.880	2.990	910	817,87
AvrE	287.850	162.670	3.500	1.020	1.094,94

A Tabela 6.27 apresenta os *speedups* da solução otimizada final em GPU em relação a cada configuração do HMMER3, enquanto na Tabela 6.28 a comparação é feita com o HMMER3.1b.

Analisando a Tabela 6.27 referente à solução com apenas o filtro MSV, foi possível superar o HMMER3 usando SSE2 e um *core* em todos os casos de teste, com *speedups* de até 4,40. Entretanto, quando os quatro *cores* do processador são usados, a solução em GPU só é capaz de ganhar em casos onde os *profile* HMMs possuem mais de 1.000 nós.

Tabela 6.27: *Speedup* da solução final em GPU em comparação com o HMMER3

Família	*Speedup*			
	HMMER3 1 *core*	HMMER3 4 *cores*	HMMER3 1 *core*/SSE2	HMMER3 4 *cores*/SSE2
Avian_gp85	49,73	18,99	1,62	0,59
CABIT	49,81	18,94	1,65	0,55
DUF530	91,81	49,72	2,90	0,86
PaRep2b	91,90	49,63	2,77	0,83
Flu_PB2	88,85	50,71	2,88	0,87
Totivirus_coat	88,90	50,84	2,81	0,82
ACR_tran	119,89	68,16	3,66	1,04
RdRP_5	124,66	70,42	3,78	1,13
Bac_GDH	**149,38**	**84,73**	**4,40**	**1,26**
AvrE	124,03	70,57	3,63	1,03

Tabela 6.28: *Speedup* da solução final em GPU em comparação com o HMMER3.1b

Família	*Speedup*			
	HMMER3.1b 1 *core*	HMMER3.1b 4 *cores*	HMMER3.1b 1 *core*/SSE2	HMMER3.1b 4 *cores*/SSE2
Avian_gp85	134,02	51,31	1,66	0,85
CABIT	134,09	50,63	1,59	0,88
DUF530	265,25	147,87	3,10	1,10
PaRep2b	264,94	150,29	3,10	1,10
Flu_PB2	223,60	127,89	2,61	0,83
Totivirus_coat	227,14	127,71	2,65	0,81
ACR_tran	**306,77**	**174,53**	3,60	1,10
RdRP_5	248,38	140,68	2,90	0,86
Bac_GDH	302,29	171,03	**3,66**	**1,11**
AvrE	262,89	148,57	3,20	0,93

Analisando a Tabela 6.28 referente à solução com filtros SSV e MSV, fica claro que a solução em GPU alcançou *speedups* significativos de até 306,77 se comparados com a execução sequencial dos algoritmos, e que com algumas otimizações foi possível alcançar resultados mais que duas vezes melhores que a solução inicial. Com esta solução, foi possível superar o HMMER3.1b usando SSE2 e um *core* em todos os casos de teste, com *speedups* de até 3,66. Porém, quando os quatro *cores* do processador são usados, a solução em GPU só é capaz de ganhar em alguns casos, mas, no geral, mantém os resultados muito próximos.

Outra observação recorrente nas Tabelas 6.27 e 6.28 é o melhor desempenho de ambas as soluções para as famílias ACR_tran e Bac_GDH. Isso ocorre graças ao número de nós dos *profile* HMMs dessas famílias que são 1.021 e 1.528, respectivamente. Dada a

modelagem do problema, a quantidade de nós do *profile* HMM está diretamente associada a quantidade de *threads* usadas na solução e os valores anteriormente citados proporcionam melhores taxas de ocupação do dispositivo. A ocupação do dispositivo pode ser mensurada através da ferramenta CUDA *Occupancy Calculator* [36], de forma que quanto maior a taxa de ocupação, menor é a porcentagem de recursos do hardware ocioso durante a execução.

7 Conclusão

A solução desenvolvida em GPU para o problema de determinar se uma nova sequência biológica é homóloga a uma família de sequências conhecida, demonstrou desempenho superior à solução sequencial em software, mesmo quando sem otimizações. Isso é fruto apenas de uma modelagem bem-feita do problema na plataforma em questão que é massivamente paralela.

Os resultados deixam claro que a exploração do paralelismo de tarefas na ferramenta HMMER traz ganhos de desempenho proporcionais à quantidade de *cores* do processador, tanto na sua versão 3 quanto na 3.1b. Contudo, a exploração do paralelismo de dados traz ganhos muito mais significativos, e quando associado à exploração do paralelismo de tarefas, o desempenho é expressivo.

Os resultados obtidos mostram que a solução otimizada final desenvolvida em GPU alcançou *speedup* de até 302,29 quando comparado com o HMMER sem SSE2. Porém, é apenas um pouco melhor que o HMMER com SSE2 usando um *core*, com *speedup* de até 4,40, e quando os quatro *cores* do processador são usados, a solução em GPU fica com resultados próximos ao HMMER com *speedups* entre 0,55 e 1,26.

A GPU utilizada nos experimentos data de 2010, enquanto o *host* (e seu processador) data de 2012, tempo considerável quando se trata da evolução do hardware e que provavelmente influenciou de forma negativa na comparação dos resultados obtidos. Porém, é possível constatar que mesmo uma GPU que não seja de última geração é capaz de gerar resultados expressivos para problemas bem modelados e que a compreensão da plataforma e das limitações do hardware do dispositivo usado possibilita otimizações nos *kernels* que garantem ganho de desempenho consideráveis.

7.1 Trabalhos Futuros

A utilização de GPUs como plataformas de computação paralela mostrou-se promissora. O desempenho dessas plataformas melhora a cada nova geração, com preços acessíveis aos desenvolvedores. Contudo, apesar dos esforços no desenvolvimento dos modelos de programação CUDA e OpenCL visando facilitar a programação de propósito geral nesses dispositivos, há muitos detalhes inerentes do hardware que devem ser analisados com cuidado no desenvolvimento até de códigos simples. Um abordagem ingênua do uso de memória pode levar um código a executar de forma quase sequencial nesses dispositivos massivamente paralelos.

Os processadores mais modernos também apresentam novidades, como é o caso

do AVX (*Advanced Vector Extensions*), uma extensão do conjunto de instruções dos processadores Intel e AMD. Nessa extensão, o comprimento dos registradores para instruções SIMD foi aumentando de 128 bits, como era o caso do SSE, para 256 bits e posteriormente esses registradores foram novamente aumentados, dessa vez para 512 bits, em uma versão batizada AVX-512. Registradores maiores indicam capacidade de executar mais operações simultaneamente e/ou de executar a mesma quantidade de operações simultaneamente com valores representados em mais bits (maior precisão).

Portanto, esse projeto deixa espaço para melhorias e algumas sugestões para trabalhos futuros são:

- Teste e adequação do código para GPUs com *Compute Capability* 3.5. Novos dispositivos apresentam mais poder de processamento, além de novas funcionalidades como é o caso do paralelismo dinâmico presente na versão citada. Essas novas funcionalidades trazem novas possibilidades que podem ser benéficas para a solução implementada;

- Teste e adequação do código para execução em múltiplas GPUs. A utilização de várias GPUs acrescenta poder de processamento na solução de um problema e portanto, se bem modelada, a solução pode ganhar desempenho;

- O desenvolvimento de uma solução heterogênea. Os processadores atuais possuem vários *cores* e a manutenção destes ociosos durante o processamento das sequências na GPU é um desperdício. Uma solução heterogênea distribuiria parte do trabalho para a GPU e outra parte para os *cores* ociosos do processador, com essas unidades de processamento adicionais a solução ficaria mais rápida. Durante o desenvolvimento deste projeto, uma implementação do algoritmo MSV foi desenvolvida para testes usando SSE2 e OpenMP. Essa solução processou o mesmo conjunto de sequências da base de dados UniProtKB/Swiss-Prot completa em tempo semelhante ao do HMMER3. Portanto, uma solução heterogênea consistiria em unir a solução em GPU e esse código com SSE2 e OpenMP;

- Desenvolvimento de uma solução para o problema da comparação sequência-família utilizando a extensão SIMD do conjunto de instruções AVX ou AVX-512. Com as melhorias oferecidas e os registradores maiores, o desempenho de uma solução para o problema desse projeto usando esses recursos pode ser superior a aquele obtido com a extensão SSE2;

- Estudo e compreensão da solução proposta por Rognes [47] para ganho de desempenho no algoritmo de Smith-Waterman usando SIMD. Uma vez que sequências distintas são independentes entre si, nessa abordagem as sequências são intercaladas de modo que símbolos de 16 sequências distintas são processados

paralelamente, o que minimiza as dependências de dados. Em outras palavras, as instruções SIMD não são usadas para processar vários símbolos de uma única sequência simultaneamente, ao invés disso são usadas para processar várias sequências simultaneamente. Dessa forma, as ideias para ganho de desempenho apresentadas nesse artigo talvez possam ser adaptadas e aplicadas no problema desse projeto.

Referências Bibliográficas

[1] N. Abbas, S. Derrien, S. Rajopadhye, and P. Quinton. Accelerating HMMER on FPGA using Parallel Prefixes and Reductions. In *Proceedings of the International Conference on Field-Programmable Technology*, pages 37–44, 2010.

[2] F. Ahmed, S. Quirem, G. Min, and B. K. Lee. Hotspot Analysis Based Partial CUDA Acceleration of HMMER 3.0 on GPGPUs. *International Journal of Soft Computing and Engineering*, 2(4):91–95, 2012.

[3] A. C. de Araújo Neto. Otimização de um Acelerador em Hardware para Análise de Sequências Biológicas. Trabalho de Conclusão de Curso, Faculdade de Computação da Universidade Federal de Mato Grosso do Sul, 2011.

[4] S. Cook. *CUDA Programming: A Developer's Guide to Parallel Computing with GPUs*. Morgan Kaufmann, 2012.

[5] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. The MIT Press, 3rd edition, 2009.

[6] Z. Du, Z. Yin, and D. A. Bader. A Tile-based Parallel Viterbi Algorithm for Biological Sequence Alignment on GPU with CUDA. In *Proceedings of the IEEE International Symposium on Parallel & Distributed Processing*, pages 1–8, 2010.

[7] R. Durbin, S. Eddy, A. Krogh, and G. Mitchison. *Biological Sequence Analysis: Probabilistic Models of Proteins and Nucleic Acids*. Cambridge University Press, 1998.

[8] S. R. Eddy. *HMMER User's Guide*. HMMER Development Team, 3.0 edition, 2010.

[9] S. R. Eddy. Accelerated Profile HMM Searches. *PLoS Computational Biology*, 7(10):1–16, 2011.

[10] S. R. Eddy and T. J. Wheeler. *HMMER User's Guide*. HMMER Development Team, 3.1b1 edition, 2013.

[11] R. Farber. *CUDA Application Design and Development*. Morgan Kaufmann, 2011.

[12] S. Ferraz and N. Moreano. Evaluating Optimization Strategies for HMMer Acceleration on GPU. In *Proceedings of the International Conference on Parallel and Distributed Systems*, pages 59–68, 2013.

[13] R. D. Finn, A. Bateman, J. Clements, P. Coggill, R. Y. Eberhardt, S. R. Eddy, A. Heger, K. Hetherington, L. Holm, J. Mistry, E. L. L. Sonnhammer, J. Tate, and M. Punta. Pfam: the protein families database. *Nucleic Acids Research*, 42(D1):222–230, 2014.

[14] G. D. Forney Jr. The Viterbi Algorithm. *Proceedings of the IEEE*, 61(3):268–278, 1973.

[15] N. Ganesan, R. D. Chamberlain, J. Buhler, and M. Taufer. Accelerating HMMER on GPUs by Implementing Hybrid Data and Task Parallelism. In *Proceedings of the ACM International Conference on Bioinformatics and Computational Biology*, pages 418–421, 2010.

[16] M. Harris. Optimizing Parallel Reduction in CUDA. NVIDIA Developer Technology, 2007.

[17] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 5th edition, 2011.

[18] D. R. Horn, M. Houston, and P. Hanrahan. ClawHMMER: A Streaming HMMer-Search Implementation. In *Proceedings of the ACM/IEEE Supercomputing Conference*, page 11, 2005.

[19] Howard Hughes Medical Institute. HMMER. http://hmmer.janelia.org/. Acessado em 15/04/2014.

[20] Intel Corporation. Intel Core i7-3770S Processor Specifications. http://ark.intel.com/products/65524. Acessado em 15/05/2014.

[21] Intel Corporation. *Intel Architecture Optimization: Reference Manual*, 1999.

[22] K. Jiang, O. Thorsen, A. Peters, B. Smith, and C. P. Sosa. An Efficient Parallel Implementation of the Hidden Markov Methods for Genomic Sequence-Search on a Massively Parallel System. *IEEE Transactions on Parallel and Distributed Systems*, 19(1):15–23, 2008.

[23] Khronos Group. OpenCL. http://www.khronos.org/opencl/. Acessado em 15/04/2014.

[24] V. V. Kindratenko, J. J. Enos, G. Shi, M. T. Showerman, G. W. Arnold, J. E. Stone, J. C. Phillips, and W. Hwu. Gpu clusters for high-performance computing. In *Proceedings of the IEEE International Conference on Cluster Computing and Workshops*, pages 1–8, 2009.

[25] D. B. Kirk and W. W. Hwu. *Programming Massively Parallel Processors: A Hands-on Approach*. Morgan Kaufmann, 2nd edition, 2012.

[26] A. Krogh. *Computational Methods in Molecular Biology*, volume 32 of *New Comprehensive Biochemistry*, chapter An Introduction to Hidden Markov Models for Biological Sequences, pages 45–63. Elsevier, 1999.

[27] A. Krogh, M. Brown, S. Mian, K. Sjölander, and D. Haussler. Hidden Markov Models in Computational Biology. *Journal of Molecular Biology*, 235(5):1501–1531, 1994.

[28] X. Li, W. Han, G. Liu, H. An, M. Xu, W. Zhou, and Q. Li. A Speculative HMMER Search Implementation on GPU. In *Proceedings of the IEEE International Parallel and Distributed Processing Symposium*, pages 735–741, 2012.

[29] R. P. Maddimsetty. Acceleration of Profile-HMM Search for Protein Sequences in Reconfigurable Hardware. Master's thesis, Department of Computer Science & Engineering of the Washington University in St. Louis, 2006.

[30] R. P. Maddimsetty, J. Buhler, R. D. Chamberlain, M. A. Franklin, and B. Harris. Accelerator Design for Protein Sequence HMM Search. In *Proceedings of the Annual International Conference on Supercomputing*, pages 288–296, 2006.

[31] A. C. M. A. de Melo and N. Moreano. *Reconfigurable Embedded Control Systems: Applications for Flexibility and Agility*, chapter FPGA-Based Accelerators for Bioinformatics Applications, pages 311–341. IGI Global, 2010.

[32] S. S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997.

[33] National Human Genome Research Institute. Human Genome Project. http://www.genome.gov/11006943. Acessado em 15/05/2014.

[34] J. Nickolls, I. Buck, M. Garland, and K. Skadron. Scalable Parallel Programming with CUDA. *ACM Queue*, 6(2):40–53, 2008.

[35] NVIDIA Corporation. CUDA Device Query. http://docs.nvidia.com/cuda/cuda-samples/index.html#device-query. Acessado em 15/05/2014.

[36] NVIDIA Corporation. CUDA Occupancy Calculator. http://developer.download.nvidia.com/compute/cuda/CUDA_Occupancy_calculator.xls. Acessado em 19/05/2014.

[37] NVIDIA Corporation. CUDA Zone. https://developer.nvidia.com/cuda-zone. Acessado em 15/04/2014.

[38] NVIDIA Corporation. GeForce GTX 570 Specifications. http://www.geforce.com/hardware/desktop-gpus/geforce-gtx-570/specifications. Acessado em 02/04/2014.

[39] NVIDIA Corporation. NVIDIA Visual Profiler. https://developer.nvidia.com/nvidia-visual-profiler. Acessado em 01/05/2014.

[40] NVIDIA Corporation. NVIDIA's Next Generation CUDA Compute Architecture: Kepler GK110. www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf. Acessado em 01/08/2014.

[41] NVIDIA Corporation. CUDA C Best Practices Guide. http://docs.nvidia.com/cuda/cuda-c-best-practices-guide/, 2014. Acessado em 01/08/2014.

[42] T. Oliver, L. Y. Yeow, and B. Schmidt. High Performance Database Searching with HMMER on FPGAs. In *Proceedings of the IEEE International Parallel and Distributed Processing Symposium*, pages 1–7, 2007.

[43] T. F. Oliver, B. Schmidt, Y. Jakop, and D. L. Maskell. Accelerating the Viterbi Algorithm for Profile Hidden Markov Models Using Reconfigurable Hardware. In *Proceedings of the International Conference on Computational Science*, pages 522–529, 2006.

[44] PCI-SIG. Creating a PCI Express Interconnect. http://www.pcisig.com/specifications/pciexpress/resources/PCI_Express_White_Paper.pdf. Acessado em 19/05/2014.

[45] C. N. S. Pedersen. *Algorithms in Computational Biology*. PhD thesis, Department of Computer Science of the University of Aarhus, 1999.

[46] S. Quirem, F. Ahmed, and B. K. Lee. CUDA Acceleration of P7Viterbi Algorithm in HMMER 3.0. In *Proceedings of the IEEE International Performance Computing and Communications Conference*, pages 1–2, 2011.

[47] T. Rognes. Faster Smith-Waterman database searches with inter-sequence SIMD parallelisation. *BMC Bioinformatics*, 12(1):221, 2011.

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

[49] Y. Sun, P. Li, G. Gu, Y. Wen, Y. Liu, and D. Liu. Accelerating HMMer on FPGAs Using Systolic Array Based Architecture. In *Proceedings of the IEEE International Symposium on Parallel & Distributed Processing*, pages 1–8, 2009.

[50] T. Takagi and T. Maruyama. Accelerating HMMER Search Using FPGA. In *Proceedings of the International Conference on Field Programmable Logic and Applications*, pages 332–337, 2009.

[51] The UniProt Consortium. Activities at the Universal Protein Resource (UniProt). *Nucleic Acids Research*, 42(D1):191–198, 2014.

[52] J. P. Walters, V. Balu, S. Kompalli, and V. Chaudhary. Evaluating the use of GPUs in Liver Image Segmentation and HMMER Database Searches. In *Proceedings of the IEEE International Symposium on Parallel & Distributed Processing*, pages 1–12, 2009.

[53] J. P. Walters, R. Darole, and V. Chaudhary. Improving MPI-HMMER's Scalability with Parallel I/O. In *Proceedings of the IEEE International Symposium on Parallel & Distributed Processing*, pages 1–11, 2009.

[54] J. P. Walters, J. Landman, and V. Chaudhary. MPI-HMMER. http://www.mpihmmer.org/. Acessado em 02/09/2013.

[55] J. P. Walters, B. Qudah, and V. Chaudhary. Accelerating the HMMER Sequence Analysis Suite Using Conventional Processors. In *Proceedings of the International Conference on Advanced Information Networking and Applications*, pages 289–294, 2006.

[56] P. Yao, H. An, M. Xu, G. Liu, X. Li, Y. Wang, and W. Han. CuHMMer: A Load-balanced CPU-GPU Cooperative Bioinformatics Application. In *Proceedings of the International Conference on High Performance Computing and Simulation*, pages 24–30, 2010.