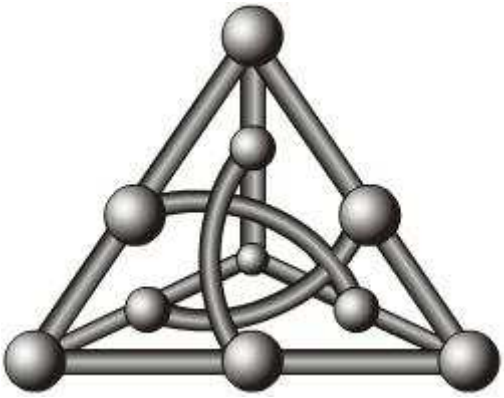
Estratégias de Otimização em GPU para Análise de Sequências Biológicas

Samuel Benjoino Ferraz Aquino

Dissertação de Mestrado apresentada à
Faculdade de Computação da
Universidade Federal de Mato Grosso do Sul



Orientadora: Profa. Dra. Nahri Balesdent Moreano

Campo Grande, Outubro de 2012

Resumo

Uma importante tarefa na área de Bioinformática é comparar uma sequência em relação a uma família de sequências e, dependendo do resultado obtido, incluir essa sequência na família em questão. HMMer [17, 18] é um conjunto de ferramentas bastante utilizado para realizar essa tarefa e aplica um algoritmo denominado algoritmo de Viterbi. Existem implementações do HMMer buscando ganhos de desempenho nas mais variadas plataformas. Entretanto, o tamanho das bases de sequências biológicas vem crescendo muito nos últimos anos, fazendo com que a comparação de sequências utilizando essas bases de dados se torne cada vez mais custosa em termos de tempo de processamento. Poucas implementações utilizam como plataforma de execução a GPU e avaliam esse dispositivo, que possui grande capacidade computacional e evoluiu muito nos últimos anos. Assim, este trabalho apresenta o desenvolvimento de soluções em GPU para o algoritmo de Viterbi aplicado à análise de sequências biológicas e avalia as maneiras mais eficientes de utilizar os recursos disponíveis nessa plataforma. O acelerador proposto alcança um ótimo desempenho, com speedup médio de 48,82 e máximo de 102,83, em relação ao HMMer2 executado em um computador convencional. O desempenho obtido também é superior ao alcançado por outros aceleradores em GPU descritos na literatura.

Palavras-chave: Alinhamento Sequência-*Profile*, Algoritmo de Viterbi, GPU, Acelerador.

Abstract

Comparing a biological sequence to a family of sequences and, depending on the results, including this sequence into the family is an important task in Bioinformatics. HMMer [17, 18] is a set of tools widely used to perform this task and applies an algorithm called Viterbi algorithm. There are several implementations of the Viterbi algorithm that try to achieve performance gains on several different platforms. However, the size of biological sequence databases has been growing exponentially recently, making the comparison process more computationally demanding. A GPU is a hardware device with a high capability of parallel processing that has evolved very much lately, nevertheless, just a few implementations of the Viterbi algorithm use and evaluate it for this problem. This work presents the development of solutions to the Viterbi algorithm applied to biological sequence analysis on GPUs and evaluate the most efficient ways to use their resources. The accelerator proposed achieves speedups up to 102,83 and on average 48,82, with respect to HMMer's execution on a general purpose computer. The performance achieved is higher than the ones achieved by other accelerators described in the literature.

Keywords: Sequence-Profile Alignment, Viterbi Algorithm, GPU, Accelerator.

Sumário

Lista de Figuras

Lista de Tabelas

Lista de Algoritmos

Capítulo 1

Introdução

Uma sequência biológica é modelada por uma cadeia de símbolos que carregam informação biológica [15]. Uma família de sequências biológicas é um conjunto de sequências que possuem funções similares, estrutura 2D/3D similar ou uma evolução histórica comum [30]. Uma família pode ser representada por um alinhamento múltiplo das sequências, que constitui uma das possíveis maneiras de organizar um conjunto de sequências de forma a encontrar regiões similares entre elas [9]. Existem milhares de famílias de sequências, cada família com seu respectivo papel.

Comparar uma nova sequência em relação a uma família de sequências é uma importante tarefa na área de Biologia Molecular, pois permite concluir algumas informações a respeito das funcionalidades da sequência e sua inclusão ou não na família em questão. O foco principal deste trabalho é propor soluções que reduzam o tempo necessário para realizar essa comparação utilizando GPUs, uma recente plataforma de computação de alto desempenho, e estudar técnicas de otimização, que visem o ganho de desempenho, de programas desenvolvidos para essa plataforma.

1.1 Motivação

O tamanho das bases de dados de sequências biológicas cresceu muito nos últimos anos, tornando a tarefa de comparação e classificação de sequências cada vez mais difícil e custosa. Existem diversas bases de famílias e sequências que comprovam esse fato.

UniProtKB/Swiss-Prot [71] é uma base de dados de proteínas categorizada e revisada manualmente. Sua versão atual contém 536.789 sequências, sendo que cada sequência possui anotações referentes à sua função, domínio, dentre outras informações. A Figura 1.1 mostra a evolução do número total de sequências dessa base, mostrando que seu tamanho duplica a cada 6 anos.

Como o processo de inclusão de sequências na UniProtKB/Swiss-Prot é dispendioso, criou-se uma base de sequências suplementar, chamada UniProtKB/TrEMBL [20]. Além de conter as sequências de UniProtKB/Swiss-Prot, UniProtKB/TrEMBL contém sequências que foram categorizadas utilizando apenas processos automatizados e que não foram revisadas manualmente. Essas novas sequências constituem uma espécie de candidatas a fazerem parte da base UniProtKB/Swiss-Prot. Atualmente, UniProtKB/TrEMBL possui 23.165.610 sequências e seu tamanho praticamente dobra a cada 2 anos. A Figura 1.2 mostra o crescimento exponencial dessa base.

Pfam [65] é uma base de dados de famílias de sequências de proteínas, onde as famílias são representadas por alinhamentos múltiplos e modelos ocultos de Markov [10], construída a partir das bases UniProtKB/Swiss-Prot e UniProtKB/TrEMBL. A Figura 1.3 apresenta o número de famílias dessa base, mostrando que, desde sua criação, ela não parou de crescer e está, atualmente, sete vezes maior que sua primeira versão.

A criação e evolução das bases de sequências e famílias descritas exigem, dentre outras técnicas,

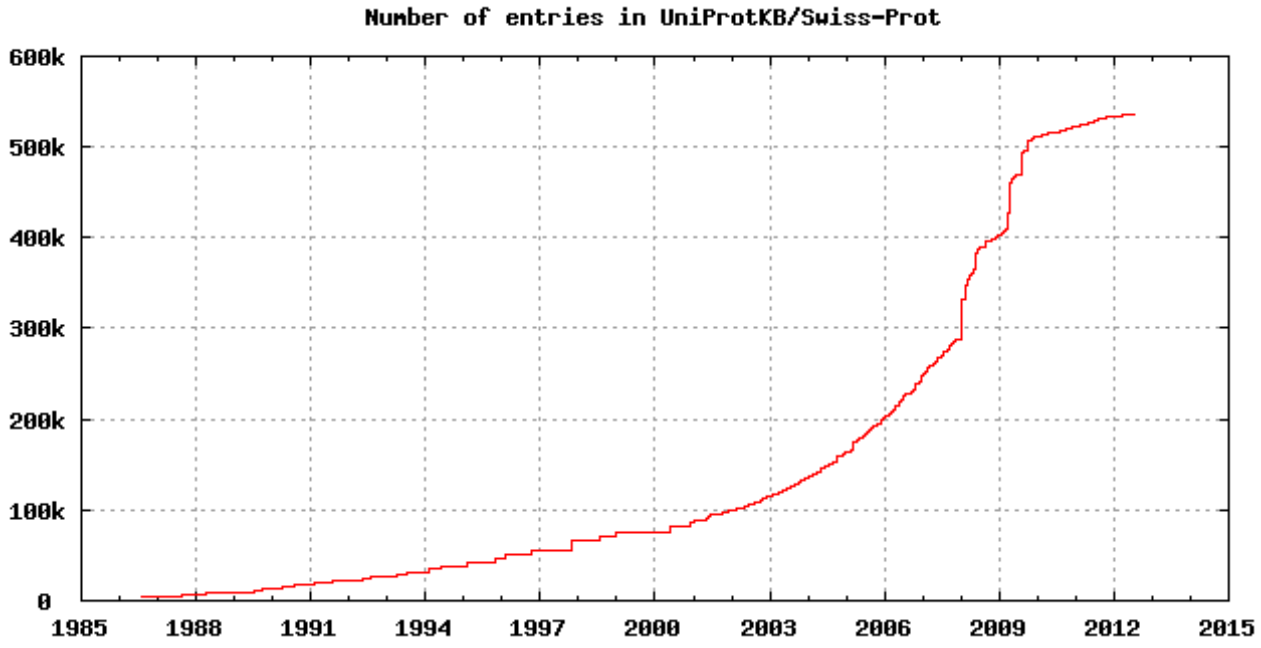


Figura 1.1: Número de sequências na base de dados UniProtKB/Swiss-Prot [71]

uma ferramenta que realize a comparação entre sequências biológicas e famílias. HMMer [17, 18] é uma das principais ferramentas utilizadas para esse fim, e é baseada em um importante algoritmo denominado algoritmo de Viterbi [23].

Embora muito utilizada, essa ferramenta demanda muito tempo de processamento quando grandes bases de dados são analisadas. Dependendo do tamanho da base de sequências e da base de famílias, o processo de comparação das sequências com as famílias pode demorar até 500 dias para finalizar em um processador convencional executando HMMer [40]. Por essa razão, a comunidade científica vem se esforçando nos últimos anos para reduzir o tempo de execução do HMMer, através da computação de alto desempenho.

Existem implementações e otimizações do algoritmo de Viterbi e do HMMer para as mais variadas plataformas. As principais propostas encontradas na literatura são:

- Otimizações no código do HMMer para processadores convencionais, processador Opteron [11] e processador Cell/B.E. [33];
- Implementações que utilizam extensões vetoriais do conjunto de instruções, como Intel SSE2 [68] e PowerPC AltiVec [72];
- Implementações para execução paralela em *clusters*, utilizando as bibliotecas MPI [42] e PVM [25];
- Implementações para FPGAs [41];
- Implementações para GPUs (*Graphics Processing Units*) [62] utilizando BrookGPU [6] e CUDA [47].

GPUs são dispositivos com alta capacidade de processamento paralelo, que avançaram muito nos últimos anos, tanto em aspectos de desempenho quanto em facilidade de programação. Há cerca de uma década, GPUs eram utilizadas apenas para executar aplicações gráficas, possuindo um rígido modelo de execução que dificultava a programação de propósito geral. Desde então, as GPUs evoluíram e se transformaram em processadores cada vez mais poderosos e flexíveis, permitindo que aplicações

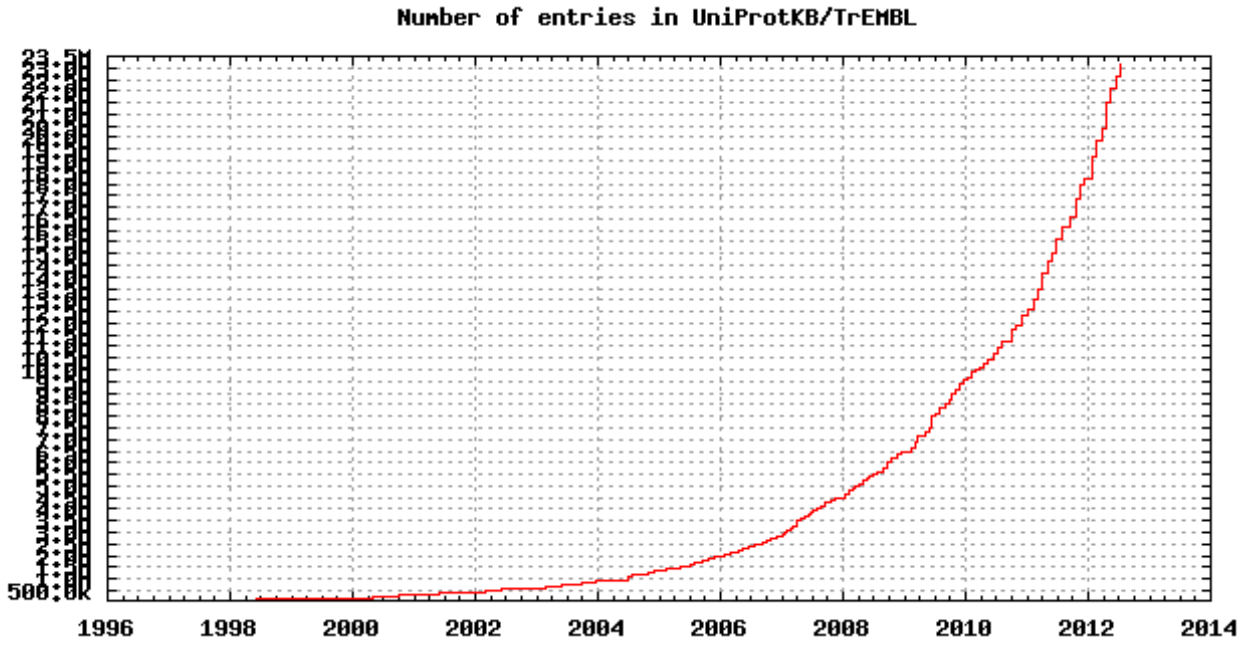


Figura 1.2: Número de sequências na base de dados UniProtKB/TrEMBL [20]

de propósito geral explorem a capacidade de processamento desses dispositivos através de modelos de programação mais amigáveis [62].

A grande capacidade de processamento paralelo das GPUs, aliada ao seu baixo custo, propicia a utilização desses dispositivos como uma plataforma de computação de alto desempenho para a implementação de soluções de problemas computacionalmente intensivos, como a comparação de sequências biológicas com famílias de sequências.

1.2 Objetivos do Trabalho

Este trabalho tem como objetivo analisar a utilização de GPUs como plataforma de computação de alto desempenho, estudando a arquitetura e organização desses dispositivos e a forma de programá-los. Deseja-se investigar e avaliar diversas técnicas de otimização de soluções em GPU, que tenham o propósito de oferecer ganhos de desempenho.

O trabalho também visa o desenvolvimento de um acelerador em GPU para a comparação de uma base de sequências biológicas com uma família de sequências, que retorne um valor de similaridade para cada sequência e permita concluir se ela faz parte da família ou não. Dado o volume dos dados biológicos envolvidos, o objetivo é construir um acelerador que tenha um bom desempenho, em termos de tempo de processamento.

Para alcançar esses objetivos, algumas etapas de desenvolvimento são necessárias. Inicialmente, é importante compreender por completo o problema da comparação de sequências e família de sequências, a aplicação do algoritmo de Viterbi para resolvê-lo e as dependências de dados envolvidas.

As principais implementações do algoritmo de Viterbi, aplicado à comparação sequência-*profile*, nos variados dispositivos de alto desempenho devem ser analisadas, investigando suas ideias principais e formas de exploração de paralelismo e avaliando se é possível adaptá-las para as GPUs.

É necessário conhecer a plataforma de computação de alto desempenho a ser utilizada, isto é, as principais características arquiteturais das GPUs e os modelos de programação disponíveis devem ser estudados, com o objetivo de realizar um mapeamento eficiente do problema na plataforma.

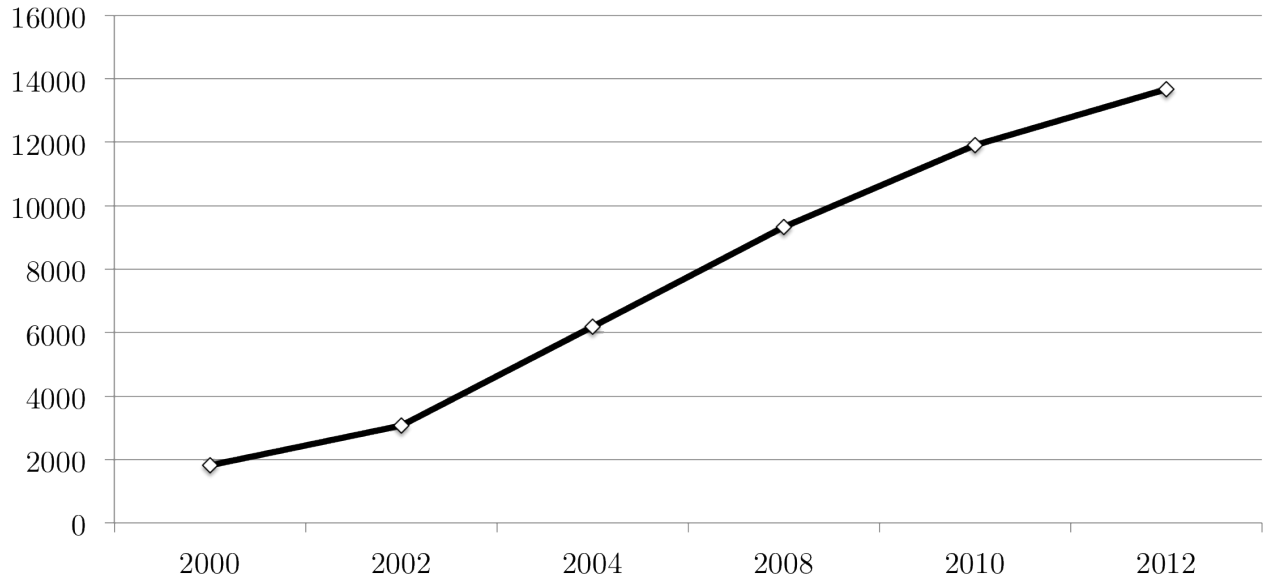


Figura 1.3: Número de famílias de sequências na base de dados Pfam

Por fim, pretende-se realizar uma extensa avaliação experimental de desempenho do acelerador proposto e das técnicas de otimização implementadas, com o objetivo de alcançar resultados e conclusões que permitam estudos futuros na área.

1.3 Organização do Texto

Este texto está organizado em 11 capítulos. O Capítulo 2 apresenta conceitos básicos de Bioinformática e de modelos ocultos de Markov, necessários para compreender a modelagem de famílias de sequências biológicas através de *profile*-HMMs e a operação de comparação de sequências com famílias.

O Capítulo 3 apresenta o algoritmo de Viterbi, utilizado neste trabalho para a comparação e o alinhamento de uma sequência com uma família. São discutidas as dificuldades envolvidas na paralelização do algoritmo de Viterbi, assim como a aplicação de filtros para a comparação sequência-*profile*.

No Capítulo 4 são descritas as implementações de alto desempenho do algoritmo de Viterbi, aplicado à comparação sequência-*profile*, encontradas na literatura da área, destacando-se as principais funcionalidades do HMMer, as implementações existentes para GPUs, *clusters* e FPGAs. Para cada tipo de implementação, a forma de paralelismo explorado, bem como o desempenho alcançado são discutidos.

O Capítulo 5 apresenta as GPUs, descrevendo sua arquitetura e hierarquia de memórias, para permitir uma maior compreensão sobre sua capacidade de processamento paralelo. Também descreve o modelo de programação OpenCL, utilizado neste trabalho para o desenvolvimento do acelerador para comparação sequência-*profile* em GPUs. As principais otimizações que podem ser aplicadas na implementação de soluções em GPUs visando ganhos de desempenho são descritas no Capítulo 6.

O acelerador básico para comparação sequência-*profile* em GPU, desenvolvido neste trabalho, ainda sem a aplicação de otimizações, é apresentado no Capítulo 7, juntamente com suas estruturas de dados. São descritas a plataforma de desenvolvimento e execução do acelerador e as bases de dados biológicos reais utilizadas nas avaliações experimentais. O acelerador é avaliado e os resultados preliminares são analisados.

Os Capítulos 8 e 9 apresentam a aplicação de diversas otimizações no acelerador básico em GPU, com o objetivo de melhorar o seu desempenho. Para cada otimização, experimentos são realizados e os resultados obtidos discutidos.

Com base nos resultados dos Capítulos 8 e 9, o Capítulo 10 apresenta a construção e avaliação do acelerador otimizado final em GPU, criado a partir da acelerador básico acrescido das otimizações que produzem maior ganho de desempenho. Experimentos são realizados com o objetivo de investigar a influência da ocupação da GPU e da granularidade do *kernel* no desempenho do acelerador.

Por fim, o Capítulo 11 sumariza os resultados alcançados neste trabalho e sugere experimentos, análises e linhas de pesquisa para trabalhos futuros na área.

Capítulo 2

Conceitos Fundamentais de Bioinformática

A evolução da Biologia Molecular fez com que, nas últimas décadas, uma grande quantidade de dados biológicos fosse gerada. A Bioinformática acompanhou essa evolução, auxiliando na identificação de informações relevantes a partir desses dados, ou através do uso de técnicas matemáticas e computacionais com o intuito de resolver problemas referentes à Biologia Molecular [67]. Este capítulo apresenta os principais conceitos de Bioinformática utilizados neste trabalho.

2.1 Sequências Biológicas, Famílias de Sequências e Alinhamentos

Todos os organismos existentes na terra são compostos por cadeias chamadas de sequências biológicas. Essas cadeias são modeladas por sequências de símbolos e podem ser de dois tipos [67]:

- Proteínas: indicam as funções desempenhadas por um ser. Proteínas são cadeias formadas por aminoácidos, representados pelos símbolos A, C, D, E, F, G, H, I, K, L, M, N, P, Q, R, S, T, V, W e Y;
- Ácidos nucléicos: responsáveis pela codificação das informações necessárias para produzir proteínas e pela transferência dessas informações para gerações seguintes. Ácidos nucléicos são cadeias formadas por bases nitrogenadas, representadas pelos símbolos A, C, G, T e U, e podem ser de dois tipos: DNA (formado pelas bases A, C, G e T) e RNA (formado pelas bases A, C, G e U).

A presença de determinadas subsequências em comum entre duas sequências pode indicar que ambas compartilham determinadas funcionalidades. Para que essas subsequências em comum sejam identificadas com mais facilidade, alinhamentos das sequências são construídos.

Dadas as sequências biológicas $S_1 = ACGTACGTACG$ e $S_2 = ATCGTACGTACG$, elas possuem uma estrutura em comum, evidenciada quando uma é colocada abaixo da outra, como mostra a Figura 2.1.

$$\begin{array}{lllllllllllll} S_1 & = & A & - & C & G & T & A & C & G & T & A & C & G \\ S_2 & = & A & T & C & G & T & A & C & G & T & A & C & G \end{array}$$

Figura 2.1: Alinhamento par-a-par de sequências

Para a similaridade entre as duas sequências ser destacada, um símbolo '–' representando um espaço (*gap*) é inserido em S_1. A representação mostrada na Figura 2.1 é um alinhamento par-a-par entre S_1 e S_2, conforme a definição seguinte.

Definição 1. *Um **alinhamento par-a-par** de duas sequências biológicas S_1 e S_2 de tamanhos quaisquer consiste na inserção de espaços (gaps) no decorrer das sequências, com o objetivo de deixá-las com o mesmo tamanho e criar uma correspondência entre os símbolos de S_1 e S_2 [67].*

Um **alinhamento par-a-par global** entre as sequências S_1 e S_2 alinha todos os símbolos dessas sequências. Se apenas uma subsequência de S_1 e uma subsequência de S_2 são alinhadas, trata-se de um **alinhamento par-a-par local**.

O conceito de alinhamento pode ser estendido para mais de duas sequências, com a definição de alinhamento múltiplo.

Definição 2. *Um **alinhamento múltiplo** de um conjunto de sequências $S = \{S_1, S_2, ..., S_n\}$ de tamanhos quaisquer consiste na inserção de espaços (gaps) no decorrer das as sequências, com o objetivo de deixá-las com o mesmo tamanho e criar uma correspondência entre os símbolos das sequências de S [67].*

A Figura 2.2 mostra um alinhamento múltiplo do conjunto de sequências $S = \{ACGTACGTACG, AACGTACGTACG, ACGACGTCGT, ACTAGTACG\}$. Novamente, para que a correspondência entre os elementos das sequências se torne evidente, é necessária a inserção de espaços.

$$\begin{array}{lcccccccccccc} S_1 & = & A & - & C & G & T & A & C & G & T & A & C & G \\ S_2 & = & A & A & C & G & T & A & C & G & T & A & C & G \\ S_3 & = & A & C & G & - & A & C & G & T & - & C & G & T \\ S_4 & = & A & C & - & T & A & - & G & T & A & C & G & - \end{array}$$

Figura 2.2: Alinhamento múltiplo de sequências

Podem existir diversos alinhamentos par-a-par diferentes entre duas sequências e diversos alinhamentos múltiplos diferentes para um mesmo conjunto de sequências. Uma função de custo é utilizada para ponderar símbolos alinhados, símbolos desalinhados e inserção de espaços, produzindo um valor de similaridade para o alinhamento. Algoritmos para alinhamento par-a-par e para alinhamento múltiplo podem ser encontrados em [67].

Uma família de sequências biológicas é um conjunto de sequências que possuem funções similares, estrutura 2D/3D similar ou uma evolução histórica comum [30]. As funcionalidades de uma sequência biológica estão geralmente atreladas à família a qual ela pertence.

Comparar uma sequência biológica qualquer com uma família de sequências é uma operação importante na área de Biologia Molecular, pois permite concluir algumas informações a respeito das funcionalidades da sequência e sua inclusão ou não na família.

Para que uma nova sequência S possa ser comparada a uma família de sequências, é necessário que esta última esteja representada por um modelo teórico que expresse as características da família como um todo. A comparação de S com a família modelada produz um valor de similaridade, ou *score*, para o alinhamento de S com a família. Esse *score* é utilizado para concluir se S faz parte da família ou não.

As famílias de sequências utilizadas neste trabalho são representadas por um modelo teórico descrito na próxima seção.

2.2 Modelos Ocultos de Markov

Um modelo probabilístico teórico muito utilizado para a representação de famílias de sequências biológicas é denominado modelo oculto de Markov. Modelos ocultos de Markov também são aplicados

na solução de outros problemas na área de bioinformática [36] e em diversas outras áreas, tais como reconhecimento de fala [66], inteligência artificial [5, 22] e mineração de dados [44].

Definição 3. *Um **modelo de Markov** é um modelo composto por um conjunto de Q estados $\{E_1, E_2, ..., E_Q\}$ capazes de, em um instante de tempo t, gerar uma variável aleatória s_t conforme uma distribuição de probabilidade. Além disso, modelos de Markov obedecem à **propriedade de Markov**: a probabilidade de uma variável aleatória ser s_t gerada em um instante t depende apenas da variável aleatória s_{t-1} gerada no instante $t-1$. Ou seja: $P(s_t|s_1, s_2, ..., s_{t-1}) = P(s_t|s_{t-1})$ [10].*

Os modelos de Markov utilizados neste trabalho são discretos no espaço de estados e no tempo, pois o problema estudado apresenta essas características. Entretanto, modelos de Markov também podem ser contínuos [66].

Em modelos de Markov convencionais, um estado E_j é capaz de gerar sempre a mesma variável aleatória s_t. Entretanto, essa restrição impede a representação de alguns problemas utilizando esses modelos. Por essa razão uma extensão do modelo foi formulada, denominada modelo oculto de Markov.

Definição 4. *Um **modelo oculto de Markov** (HMM – Hidden Markov Model) é composto por [15]:*

- *Um conjunto de Q estados distintos $\{E_1, E_2, ..., E_Q\}$;*
- *Um conjunto de probabilidades de transição Pt, de tal maneira que Pt_{E_i,E_j} indica a probabilidade de estar no estado E_i e transitar para o estado E_j. O valor Pt_{E_i,E_j} depende exclusivamente do estado anterior E_i, de acordo com a propriedade de Markov;*
- *Um conjunto de símbolos de saída $\Sigma = \{a_1, a_2, ..., a_{|\Sigma|}\}$;*
- *Um conjunto de probabilidades de emissão Pe associado a cada estado, de tal maneira que $Pe_{E_j}(a_i)$ indica a probabilidade de emitir o símbolo de saída a_i no estado E_j;*
- *Um conjunto de probabilidades de iniciação π, tal que π_{E_j} indica a probabilidade da emissão da sequência de observação iniciar no estado E_j.*

A principal diferença deste modelo em relação aos modelos de Markov convencionais é que, enquanto nos modelos de Markov a emissão de símbolos de saída é direta (uma vez que os símbolos são os próprios estados), nos modelos ocultos de Markov a emissão é indireta, pois são funções probabilísticas dos estados. Sendo assim, não existe uma correspondência direta entre os estados e os símbolos de saída, por essa razão os estados são chamados de ocultos e o modelo é denominado modelo oculto de Markov.

2.3 *Profile* HMMs para Representação de Famílias de Sequências

Profile HMMs constituem uma importante representação de famílias de sequências biológicas e são construídos a partir de um alinhamento múltiplo destas sequências. Eles são capazes de representar informações específicas de cada coluna desse alinhamento múltiplo, baseando-se na posição dos símbolos nas sequências, revelando quão conservada uma coluna do alinhamento múltiplo é, além de indicar quais símbolos são mais prováveis de acontecer [15, 16].

Dado um alinhamento múltiplo das sequências de uma família, em que os símbolos alinhados formam colunas, é possível construir um HMM que modele as principais características da família através dos seguintes passos:

1. Para cada coluna j do alinhamento múltiplo, um estado M_j é criado no HMM;
2. Para cada estado M_j, uma transição de M_j para o estado M_{j+1} é criada, com probabilidade de transição $Pt_{M_j,M_{j+1}} = 1$;

3. O conjunto Σ de símbolos de saída é definido como o conjunto de aminoácidos ou bases nitrogenadas que formam as sequências;

4. Para cada estado M_j e cada símbolo s_i de Σ, a probabilidade de emissão $Pe_{M_j}(s_i)$, é determinada por:

$$Pe_{M_j}(s_i) = \frac{número\ de\ ocorrências\ de\ s_i\ na\ coluna\ j\ do\ alinhamento\ múltiplo}{número\ de\ sequências\ do\ alinhamento\ múltiplo}.$$

5. As probabilidade de iniciação são definidas tal que $\pi_{M_1} = 1$ e $\pi_{M_j} = 0, \forall j = 2...Q$

A partir dos passos acima obtém-se um HMM que pode ser representado graficamente por um autômato. Por exemplo, para o alinhamento múltiplo das sequências de uma família apresentado na Figura 2.3, o HMM criado para modelar a família é representado pelo autômato mostrado na Figura 2.4.

$$\begin{array}{lcccccc} & & \overbrace{}^{\text{colunas do alinhamento}} & & & & \\ & & 1 & 2 & 3 & 4 & 5 \\ S_1 & = & A & C & G & T & A \\ S_2 & = & A & C & G & A & A \\ S_3 & = & A & C & T & T & A \\ S_4 & = & A & C & T & C & A \\ S_5 & = & A & C & C & C & G \end{array}$$

Figura 2.3: Alinhamento múltiplo de uma família de sequências

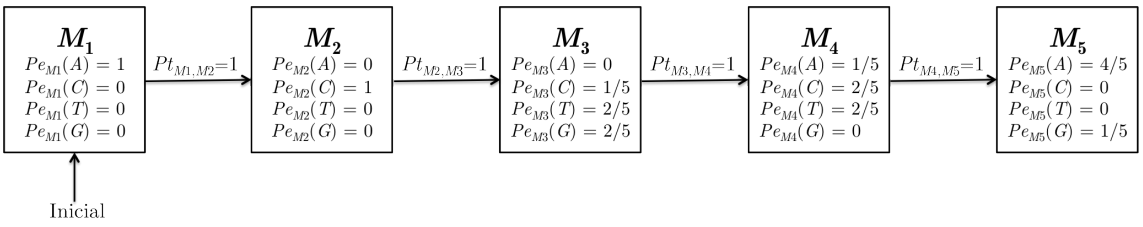


Figura 2.4: HMM que representa uma família de sequências, obtido a partir do alinhamento múltiplo da Figura 2.3

Os estados da Figura 2.4, chamados estados ***Match*** (M) e representados por retângulos, modelam cada uma das colunas do alinhamento múltiplo. Qualquer sequência de saída gerada por esse HMM deve começar no estado M_1, chamado de **estado inicial** do HMM. As possíveis transições de estado são representadas pelas setas que levam de um estado a outro. As probabilidades de transição P_t estão indicadas sobre estas setas. Dentro de cada estado estão indicadas as probabilidades de emissão P_e dos símbolos de saída daquele estado. Modelos que utilizam probabilidades associadas às colunas de um alinhamento múltiplo de sequências são chamados de perfis (*profiles*) [15]. Por essa razão, estes modelos são denominados *profile* HMMs.

Um *profile* HMM pode ser utilizado para calcular a probabilidade $P(S)$ de uma sequência S ser emitida pelo HMM, e assim fazer parte da família representada por ele. Para isso, os símbolos da sequência são alinhados aos estados do HMM e multiplica-se as probabilidades de emissão e transição obtidas no decorrer do processo, conforme a definição a seguir.

Definição 5. *A* ***probabilidade de observação*** $P(S)$ *de uma sequência* $S = s_1 s_2 ... s_Q$ *ser emitida por um profile HMM com estados* M_1, M_2, ..., M_Q *organizados linearmente é:*

$$P(S) = \pi_{M_1} \times Pe_{M_1}(s_1) \times Pt_{M_1,M_2} \times Pe_{M_2}(s_2) \times Pt_{M_2,M_3} \times ... \times Pt_{M_{Q-1},M_Q} \times Pe_{e_Q}(s_Q)$$

Por exemplo, a probabilidade $P(S_7)$ da sequência $S_7 = ACTTA$ fazer parte da família representada pelo *profile* HMM da Figura 2.4 é:

$$\begin{aligned} P(S_7) &= \pi_{M_1} \times Pe_{M_1}(A) \times Pt_{M_1,M_2} \times Pe_{M_2}(C) \times ... \times Pe_{M_4}(T) \times Pt_{M_4,M_5} \times Pe_{M_5}(A) \\ &= 1 \times 1 \times 1 \times 1 \times 1 \times \frac{1}{5} \times 1 \times \frac{2}{5} \times 1 \times \frac{4}{5} \\ &= \frac{8}{125} \end{aligned}$$

Assim, os estados *Match* alinham um símbolo da sequência sendo investigada com uma coluna do alinhamento múltiplo que representa a família de sequências. Entretanto, a Definição 5 apresenta a restrição de que para que uma sequência S seja comparada a um *profile* HMM de Q estados, ela deve conter exatamente Q símbolos.

Dado o alinhamento múltiplo das sequências de uma família mostrado na Figura 2.5, as colunas 1, 2, 5, e 6 são utilizadas para construir o *profile* HMM da família, representado na Figura 2.7. As colunas 3 e 4 não são consideradas por possuírem poucos símbolos nas sequências. Neste HMM há quatro estados *Match*, M_1, M_2, M_3 e M_4, correspondentes às colunas 1, 2, 5 e 6 do alinhamento múltiplo, respectivamente.

		1	2	3	4	5	6
S_1	=	A	C	G	A	T	A
S_2	=	C	C	$-$	$-$	T	G
S_3	=	C	C	A	$-$	G	G
S_4	=	A	C	$-$	$-$	G	A
S_5	=	A	$-$	$-$	$-$	T	G
S_6	=	A	C	C	$-$	T	G

Figura 2.5: Alinhamento múltiplo de sequências de uma família

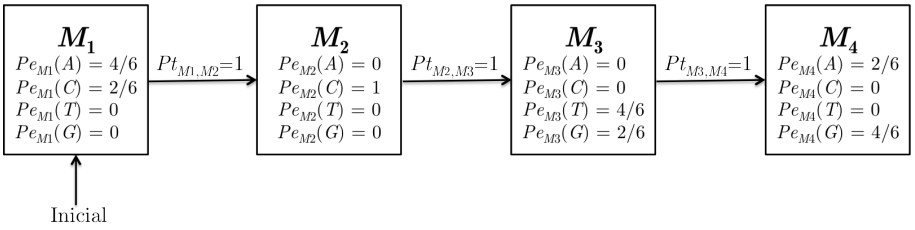


Figura 2.6: *Profile* HMM obtido a partir do alinhamento múltiplo da Figura 2.5

A probabilidade de duas novas sequências $S_8 = AGG$ e $S_9 = ACATG$ fazerem parte da família representada pelo *profile* HMM da Figura 2.6 é igual a 0, embora elas tenham similaridades com as sequências da família em questão. Isso se deve ao fato da construção do *profile* HMM desconsiderar as colunas 3 e 4 do alinhamento múltiplo, deixando de representar informações importantes da família.

A sequência S_8 possui probabilidade 0 pois nenhum símbolo seu é alinhado com o estado M_2, exatamente como a sequência S_5 do alinhamento múltiplo. Ou seja, este *profile* HMM não modela situações em que sequências pulam estados *Match* sem emitir símbolos de saída.

A sequência S_9 possui probabilidade 0 pois o símbolo A na terceira posição não é alinhado com o *profile* HMM, embora a sequência S_3 do alinhamento possua esse comportamento. Ou seja, este *profile* HMM não modela situações em que as sequências possuem símbolos que não são emitidos pelos estados *Match*.

Para que o modelo seja capaz de representar inserções e ausências de símbolos nas sequências comparadas com um *profile* HMM, dois novos tipos de estados são definidos: *Insert* e *Delete*.

Definição 6. *Estados* ***Insert*** *(I) permitem a existência de símbolos na sequência sendo comparada que não são alinhados aos estados Match do* profile *HMM.*

Definição 7. *Estados* ***Delete*** *(D) permitem que a sequência sendo comparada salte estados Match do* profile *HMM sem emitir símbolos de saída.*

O *profile* HMM da Figura 2.7 representa o mesmo alinhamento múltiplo da Figura 2.5, porém admite inserções e ausências de símbolos nas sequências comparadas ao HMM, e portanto modela a família de sequências de forma mais precisa e completa.

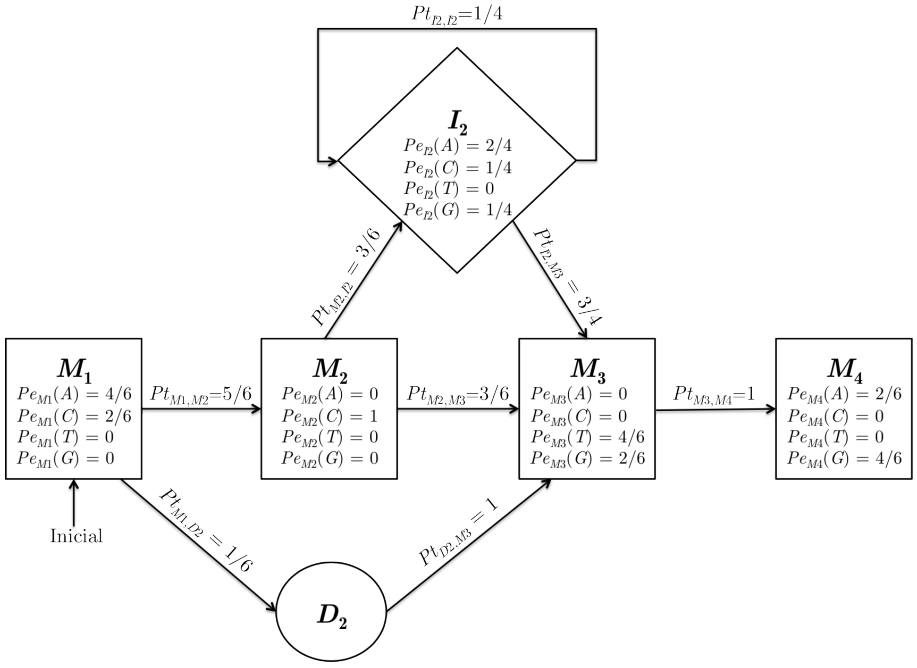


Figura 2.7: *Profile* HMM obtido a partir do alinhamento múltiplo da Figura 2.5 e que permite inserções e remoções

No alinhamento múltiplo da Figura 2.5, cinco das seis sequências têm um símbolo na coluna 2 do alinhamento, correspondente ao estado M_2, e apenas uma sequência não o possui. Isto é modelado no *profile HMM* pelo estado *Delete* D_2, representado por uma elipse na Figura 2.7, tal que $Pt_{M_1,D_2} = 1/6$ e $Pt_{M_1,M_2} = 5/6$. O estado D_2 permite que o alinhamento da sequência de observação ao HMM passe de M_1 para M_3, sem transitar por M_2 e sem emitir símbolos.

Após a coluna 2 do alinhamento múltiplo, três das seis sequências entram em uma região denominada **região de inserção** (colunas 3 e 4 na Figura 2.5), enquanto as outras três sequências partem para a próxima coluna de alinhamento (coluna 5), representada por M_3. Essa região de inserção é modelada no *profile HMM* pelo estado *Insert* I_2, representado por um losango na Figura 2.7, tal que $Pt_{M_2,I_2} = 3/6$ e $Pt_{M_2M_3} = 3/6$. O estado I_2 permite que símbolos da sequência de observação sejam emitidos entre os estados M_2 e M_3. As probabilidades de emissão em I_2 são obtidas contando todas as ocorrências dos símbolos na região de inserção.

Após as sequências S_1, S_3 e S_6 terem feito uma inserção cada, resta ainda uma inserção (feita por S_1). Assim, $Pt_{I_2,I_2} = 1/4$ e $Pt_{I_2,M_3} = 3/4$.

Utilizando o *profile* HMM da Figura 2.7, a sequência $S_8 = AGG$ pode ser emitida pela sequência de estados $M_1 \rightarrow D_2 \rightarrow M_3 \rightarrow M_4$, e S_9 pode ser emitida pela sequência de estados $M_1 \rightarrow M_2 \rightarrow I_2 \rightarrow M_3 \rightarrow M_4$. Desta forma, a probabilidade dessas sequências fazerem parte da família representada pelo HMM passa a ser:

$$P(S_8) = \frac{4}{6} \times \frac{1}{6} \times 1 \times \frac{2}{6} \times 1 \times \frac{4}{6} = \frac{2}{81} \cong 0.02$$

$$P(S_9) = \frac{4}{6} \times \frac{5}{6} \times 1 \times \frac{3}{6} \times \frac{2}{4} \times \frac{3}{4} \times \frac{4}{6} \times 1 \times \frac{4}{6} = \frac{5}{108} \cong 0.04$$

Os *profile* HMMs estudados até o momento permitem apenas o alinhamento global de uma nova sequência em relação a uma família, porém em sequências reais esses alinhamentos globais raramente acontecem. Por essa razão, os *profile* HMMs utilizados neste trabalho seguem uma arquitetura de HMMs que permite alinhamentos mais complexos, denominada **Plan7**.

2.3.1 Arquitetura Plan7

A arquitetura Plan7 foi proposta por Eddy *et al.* [17] e consiste de um modelo de *profile* HMMs que permite o alinhamento global, local e *multi-hit* de uma sequência sendo investigada com a família de sequências representada pelo modelo.

Definição 8. ***Plan7*** *é uma arquitetura de HMMs utilizada para representar uma família de sequências, modelada a partir de um alinhamento múltiplo de suas sequências, usando os seguintes estados:*

- M_j *(Match): emite símbolos de saída associados à coluna* j *do alinhamento múltiplo, isto é, alinha um símbolo da sequência investigada com a coluna* j *do alinhamento múltiplo;*
- I_j *(Insert): emite símbolos de saída após a coluna* j *do alinhamento múltiplo, indicando a inserção de símbolos não pertencentes ao alinhamento múltiplo na sequência investigada;*
- D_j *(Delete): permite que a sequência investigada passe pela coluna* j *do alinhamento múltiplo sem emitir um símbolo de saída, isto é, sem que nenhum símbolo da sequência seja alinhado a esta coluna;*
- S *(Start): não emite símbolos de saída. Qualquer sequência de observação deve iniciar em* S*, que é o único estado inicial do HMM;*
- N*: emite símbolos de saída somente quando transições para o próprio estado* N *são tomadas. Permite que símbolos iniciais da sequência investigada sejam consumidos antes de começar o alinhamento da sequência com as colunas do alinhamento múltiplo;*
- B *(Begin): não emite símbolos de saída. Ao chegar em* B*, inicia-se o alinhamento da sequência investigada (ou de uma subsequência dela) com as colunas do alinhamento múltiplo, tomando uma transição para algum estado* M_j*. Qualquer sequência de observação deve passar por* B*;*
- E *(End): não emite símbolos de saída. No estado* E *encerra-se o alinhamento da sequência investigada (ou de uma subsequência dela) com as colunas do alinhamento múltiplo. Qualquer sequência de observação deve passar por* E*;*
- C*: emite símbolos de saída somente quando transições para o próprio* C *são tomadas. Permite que símbolos finais da sequência investigada sejam consumidos após finalizar o alinhamento da sequência com as colunas do alinhamento múltiplo;*
- J*: emite símbolos de saída somente quando transições para o próprio* J *são tomadas. Permite o alinhamento de uma nova subsequência da sequência investigada com as mesmas colunas do alinhamento múltiplo;*
- T *(Termination): não emite símbolos de saída. Qualquer sequência de observação deve acabar em* T*, que é o estado final do HMM.*

A Figura 2.8 mostra graficamente um *profile* HMM com a arquitetura Plan7 e com os estados descritos na Definição 8.

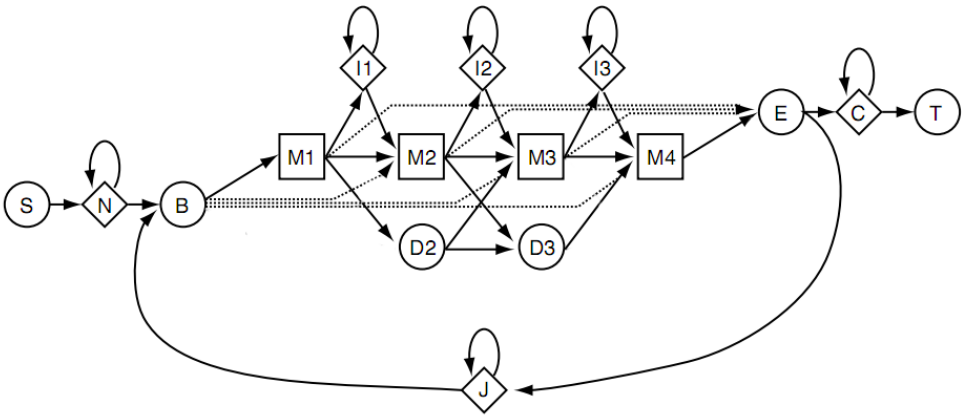


Figura 2.8: HMM com arquitetura Plan7 com quatro nós [17]

O conjunto $\{I_j, M_j, D_j\}$ de estados forma o **nó** j do HMM, e por haver exatamente sete transições saindo de cada nó, essa arquitetura é denominada Plan7. O número de nós de um HMM é utilizado como medida do seu comprimento.

Na arquitetura Plan7, os nós $\{I_j, M_j, D_j\}$ constituem a parte principal do HMM, que modela as colunas do alinhamento múltiplo da família de sequências. Os estados especiais N, B, E, C e J formam a parte do HMM que modela o algoritmo de alinhamento sequência-*profile* utilizado na comparação da sequência investigada com o HMM.

2.4 Operações entre *Profile* HMMs e Sequências

O ideal é que o *profile* HMM represente a família de sequências biológicas da melhor maneira possível. Entretanto, não existe uma maneira única de se criar um HMM [21], como exemplificado com a modelagem do alinhamento múltiplo da Figura 2.5. Dado um *profile* HMM criado a partir de um alinhamento múltiplo qualquer, existem métodos iterativos capazes de otimizar os parâmetros desse HMM a partir de novas sequências de observação. Essa operação, denominada **treinamento do HMM** [64], é realizada pelo algoritmo *Baum-Welch* [64, 21], dentre outros. Esse algoritmo analisa o comportamento do HMM durante a emissão de novas sequências de observação e, com base nesse comportamento, modifica as probabilidades de transição e emissão do HMM. Dessa forma, novas sequências podem ser incluídas na família modelada pelo HMM.

A Figura 2.9 representa um *profile* HMM de uma família de sequências. As probabilidades de emissão e transição, omitidas por questão de simplicidade, são todas maiores que zero. A figura mostra também dois caminhos que poderiam ser tomados nesse HMM para gerar a mesma sequência $S = s_1s_2s_3s_4s_5$. Um caminho, indicado pela linha pontilhada na figura, representa o seguinte alinhamento de S ao HMM:

$$(S,-) \to (N,-) \to (B,-) \to (M_1,s_1) \to (I_1,s_2) \to (M_2,s_3) \to (I_2,s_4) \to (M_3,s_5) \to (E,-) \to (C,-) \to (T,-),$$

onde (e, s) corresponde à emissão do símbolo s no estado e e as setas indicam as transições de estado. O outro caminho, indicado pela linha tracejada, representa o alinhamento:

$$(S,-) \to (N,-) \to (N,s_1) \to (B,-) \to (M_1,s_2) \to (M_2,s_3) \to (M_3,s_4) \to (E,-) \to (C,-) \to (C,s_5) \to (T,-).$$

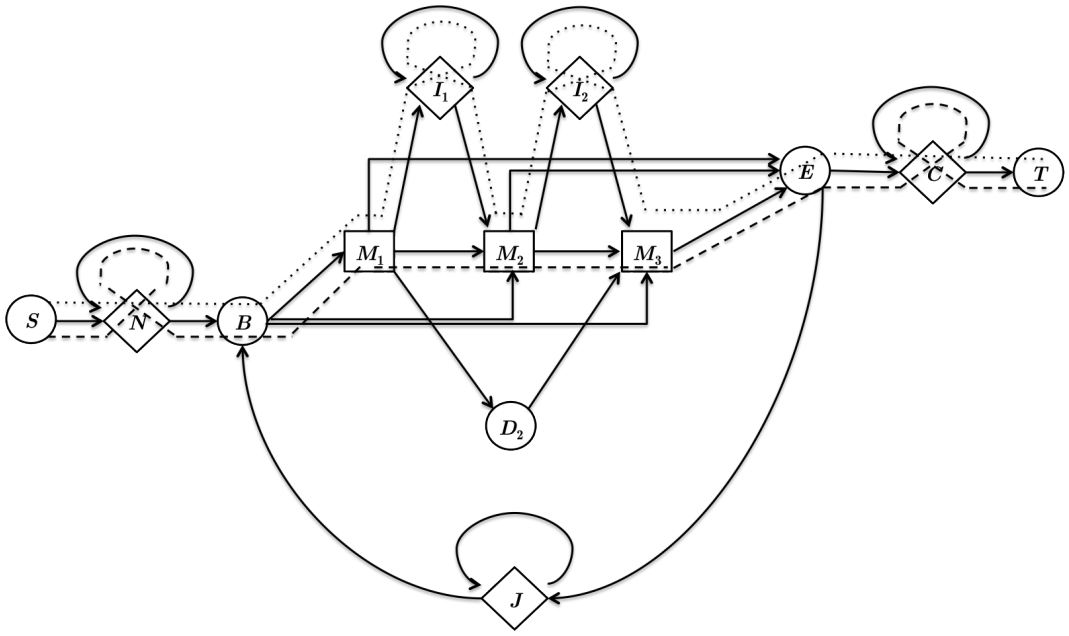


Figura 2.9: Dois possíveis alinhamentos da sequência $S = s_1 s_2 s_3 s_4 s_5$ ao *profile* HMM

Qualquer alinhamento da sequência S a esse *profile* HMM começa no estado inicial S (*Start*) e em seguida faz uma transição para o estado N. A partir de N, dois caminhos são possíveis: fazer uma transição para o próprio N, emitindo um símbolo; ou para o estado B, a partir do qual é possível realizar transições para M_1, M_2 ou M_3.

Para se chegar em um estado e, existe um conjunto de estados a partir dos quais uma transição pode ser tomada para e. Por exemplo, o estado M_3 pode ser alcançado a partir de quatro estados diferentes, M_2, I_2, D_2 e B.

Portanto, para calcular a probabilidade de uma sequência S ser gerada por um *profile* HMM, é necessário levar em consideração todas as transições que alcançam cada estado e do HMM. Essa operação, denominada **avaliação da sequência**, é realizada por dois algoritmos diferentes, porém equivalentes: o algoritmo *Forward* e o algoritmo *Backward* [21]. Nesses algoritmos, todos os caminhos no HMM que podem ser tomados para gerar S são contabilizados para calcular o *score* de S em relação ao HMM.

2.5 Alinhamento Sequência-*Profile*

Dados uma sequência $S = s_1 s_2 ... s_{|S|}$ e um *profile* HMM H representando uma família de sequências, o alinhamento sequência-*profile* de S com H pode ser construído utilizando diferentes abordagens e obtendo-se diferentes *scores*. Essas abordagens são: alinhamento global, alinhamento local em relação à sequência, alinhamento local em relação ao HMM e alinhamento *multi-hit*.

Um **alinhamento global** ocorre quando todos os elementos de S são alinhados aos nós do HMM. Esses alinhamentos acontecem quando a sequência inteira se enquadra nas características da família. A Figura 2.10 mostra um exemplo de alinhamento global de S com o HMM, representado por uma linha pontilhada. O alinhamento inicia no primeiro estado de *Match* (M_1), que gera o primeiro símbolo de S (s_1), e termina no último estado de *Match* (M_5), que gera o último símbolo da sequência ($s_{|S|}$).

Um **alinhamento local em relação à sequência** de S com um *profile* HMM ocorre quando apenas uma subsequência de S é alinhada ao HMM. Para isso, os estados N e C permitem que símbolos iniciais e finais de S sejam descartados. Esses alinhamentos acontecem quando apenas uma parte da sequência contém as características da família representada pelo HMM.

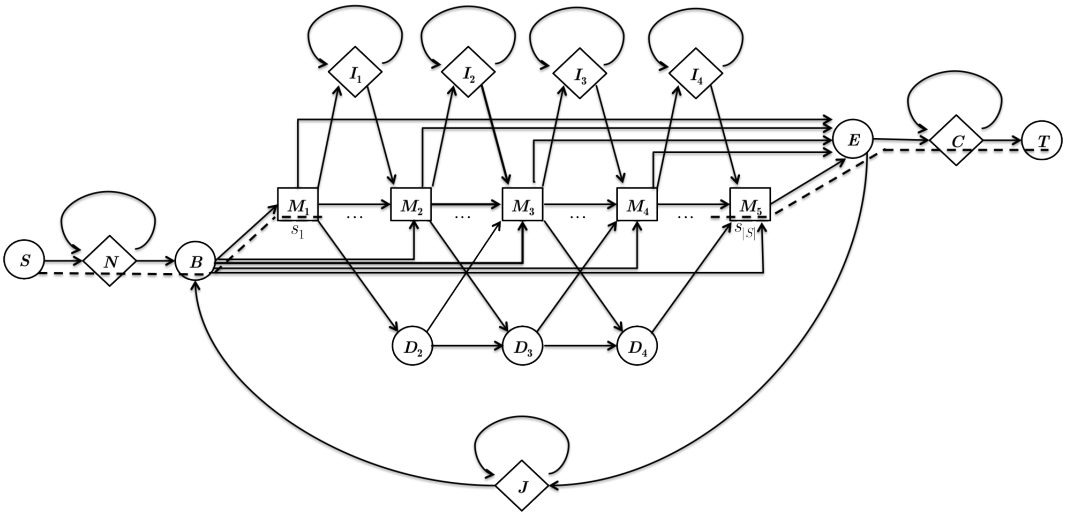


Figura 2.10: Alinhamento global

A Figura 2.11 mostra um exemplo de alinhamento local em relação à sequência de S com o HMM, representado por uma linha pontilhada. Os símbolos iniciais $s_1, ..., s_i$ foram descartados pelo estado N e os símbolos finais $s_{j+1}, ..., s_S$ foram descartados pelo estado C, permitindo apenas o alinhamento dos símbolos $s_{i+1}, ..., s_j$.

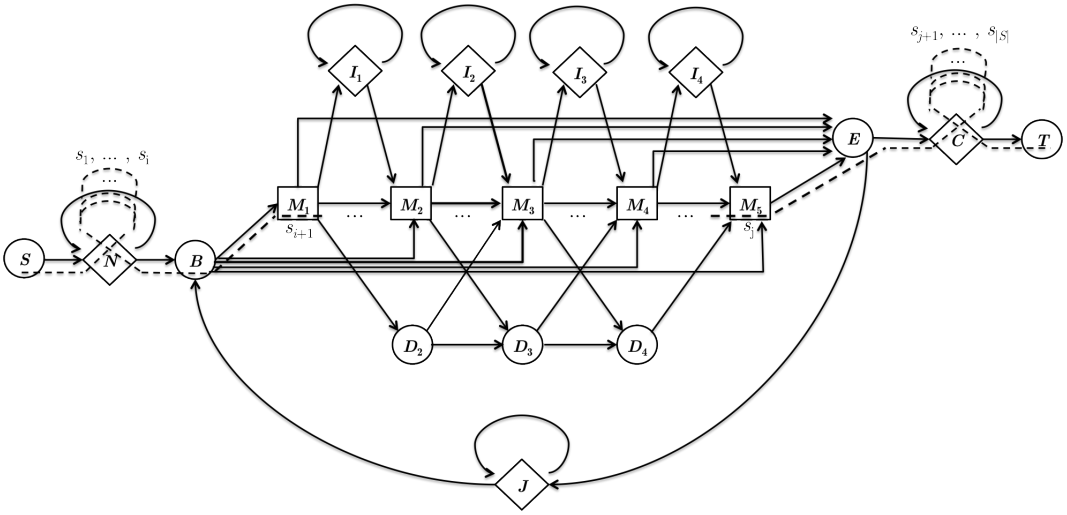


Figura 2.11: Alinhamento local em relação à sequência

Um **alinhamento local em relação ao *profile* HMM** ocorre quando apenas alguns nós do HMM são utilizados no alinhamento de S. As transições do estado B para os estados M_j e destes para E permitem que nós iniciais e finais do HMM não façam parte do alinhamento. Esses alinhamentos acontecem quando a sequência possui apenas algumas características da família representada pelo HMM.

A Figura 2.12 mostra um exemplo de alinhamento local de S em relação ao HMM, representado por uma linha pontilhada. O primeiro estado de *Match* (M_1) e o último estado de *Match* (M_5) não foram utilizados no alinhamento.

Um **alinhamento *multi-hit*** ocorre quando a sequência S é alinhada passando sucessivas vezes pelos nós do *profile* HMM. O estado J permite que várias subsequências de S sejam alinhadas ao HMM. Esses alinhamentos acontecem quando S é formada por vários segmentos que contêm as características da família.

A Figura 2.13 mostra um exemplo de alinhamento *multi-hit* de S com o HMM. Inicialmente,

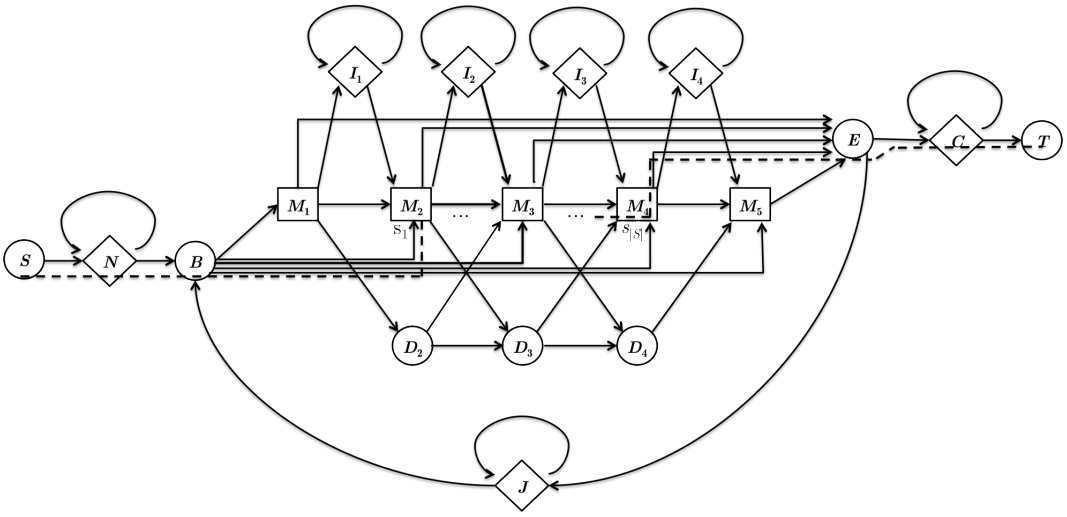


Figura 2.12: Alinhamento local em relação ao *profile* HMM

os símbolos $s_1, ..., s_j$ foram alinhados com o HMM. Logo após, o estado J descartou os símbolos $s_{j+1}, ..., s_k$ e os símbolos $s_{k+1}, ..., s_{|S|}$ foram novamente alinhados com o HMM. Desta maneira, foi possível descartar parte dos símbolos de S e alinhar duas subsequências de S com o mesmo HMM.

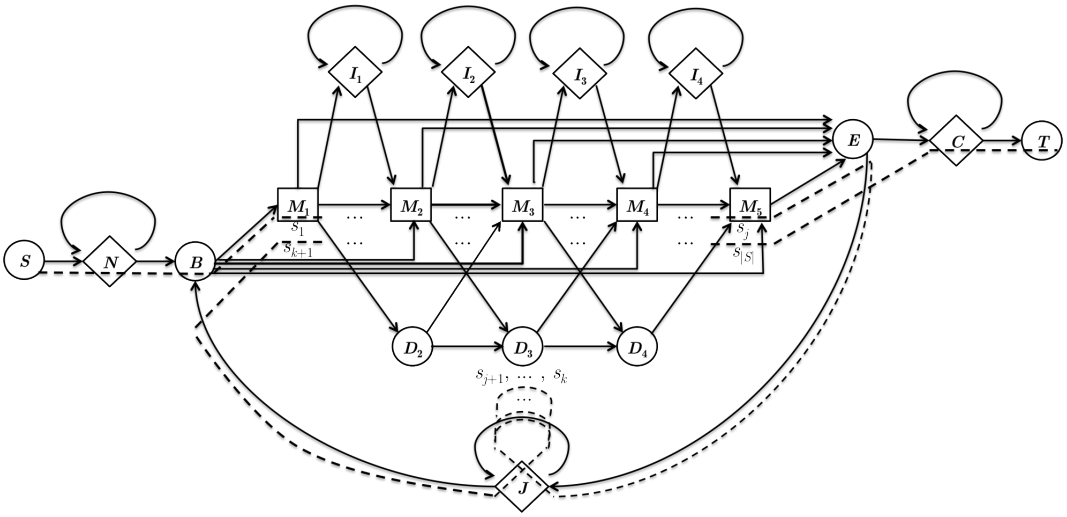


Figura 2.13: Alinhamento *multi-hit*

A arquitetura Plan7 permite definir os tipos de alinhamento que podem ser construídos através da definição do valor das probabilidades de transição nos estados especiais do HMM. Por exemplo, para não permitir a construção de um alinhamento *multi-hit*, a probabilidade de transição $E \rightarrow J$ deve ser $\emptyset$.

Em geral, todas as probabilidades de transição e emissão de um HMM são convertidas para valores logarítmicos [3]. Essa conversão permite que o cálculo da probabilidade de observação de uma nova sequência seja obtido através da soma de valores logarítmicos, ao invés da multiplicação das probabilidades originais, obtendo-se assim um *score* que representa um valor de similaridade entre a sequência e o HMM. Isto diminui a complexidade aritmética dos cálculos e reduz a possibilidade de ocorrer *underflow*, decorrente das multiplicações sucessivas de valores de probabilidade entre 0,0 e 1,0.

Além disso, uma transição com valor $-\infty$ indica uma transição com probabilidade zero, ou seja, uma transição inválida. Transições inválidas podem existir com o intuito de forçar novas sequências a passarem por determinados caminhos no HMM, por exemplo permitindo ou não inserções e remoções, impondo um alinhamento global, ou permitindo ou não alinhamentos *multi-hit*.

Dados um *profile* HMM H e uma sequência S a ser investigada, a operação de avaliação da sequência calcula a probabilidade de S ser gerada por H, considerando todos os caminhos em H que podem ser tomados para gerar S. De forma diferente, a operação de **encontrar o melhor alinhamento** de S em relação a H, determina a sequência de estados de H com maior probabilidade de gerar S. Para isso, dentre todos os caminhos em H que podem emitir S, apenas aquele de maior probabilidade é considerado.

Capítulo 3

Alinhamento Sequência-*Profile* e o Algoritmo de Viterbi

Este trabalho estuda o problema do alinhamento sequência-*profile*, no qual dados um *profile* HMM criado a partir de um alinhamento múltiplo de sequências de uma família e uma sequência a ser investigada, deseja-se encontrar o melhor alinhamento dessa sequência com o HMM e calcular o *score* desse alinhamento. Se o *score*, que representa um valor de similaridade entre a sequência e o HMM, for significativo, conclui-se que a sequência faz parte da família e o alinhamento pode ser utilizado para incluí-la no HMM.

3.1 Algoritmo de Viterbi

Os algoritmos *Forward* e *Backward* calculam a probabilidade de uma sequência S ser gerada por um *profile* HMM H, considerando todos os caminhos em H que podem ser tomados para gerar S. De forma diferente, o **algoritmo de Viterbi** [23] considera, a cada passo, dentre todos os caminhos em H que podem emitir S, apenas aquele de maior probabilidade. Para isso, o algoritmo considera somente a transição que alcança cada estado do HMM com maior probabilidade. Assim, o *score* calculado pelo algoritmo de Viterbi corresponde à probabilidade do melhor caminho que gera S em H. Este caminho representa o melhor alinhamento de S em relação à família representada pelo HMM.

O número de possíveis alinhamentos da sequência S ao *profile* HMM H é exponencial em relação ao comprimento de S [15], o que inviabiliza uma solução que enumere explicitamente todos esses possíveis alinhamentos para encontrar o melhor dentre eles. O algoritmo de Viterbi utiliza a técnica de programação dinâmica [12] e constrói um alinhamento ótimo de S a H utilizando alinhamentos ótimos de prefixos de S a H, construídos previamente.

O algoritmo de Viterbi foi projetado originalmente para operar HMMs de qualquer natureza. Como HMMs para a análise de sequências biológicas seguem a arquitetura Plan7 ou arquiteturas similares, o algoritmo de Viterbi adaptado para HMMs com a arquitetura Plan7 é descrito. Dada a sequência $S = s_1 s_2 \dots s_{|S|}$, de comprimento $|S|$, a ser alinhada ao *profile* HMM H de Q nós, o algoritmo de Viterbi calcula o *score* do alinhamento de cada prefixo $s_1 s_2 \dots s_i$ de S, chegando a cada estado de H, utilizando as seguintes estruturas de dados:

- Uma matriz de *scores* M para os estados *Match*, tal que $M[i, j]$ indica o *score* do melhor alinhamento que emite os $i - 1$ primeiros símbolos da sequência S e alcança o estado M_j, responsável por emitir o símbolo s_i;
- Uma matriz de *scores* I para os estados *Insert*, tal que $I[i, j]$ indica o *score* do melhor alinhamento que emite os $i - 1$ primeiros símbolos da sequência S e alcança estado I_j, responsável por emitir símbolo s_i;

Algoritmo 3.1 Algoritmo de Viterbi para a arquitetura Plan7

Entradas: HMM H com Q nós e probabilidades de emissão Pe e transição Pt
Sequência $S = s_1 s_2 ... s_{|S|}$

Saída: *Score* do melhor alinhamento de S com H

$B[0] = Pt_{N,B}$

$N[0] = 0$

$E[0] = -\infty$

$C[0] = -\infty$

$J[0] = -\infty$

for $i = 1$ to $|S|$ **do**

  $M[i,0] = -\infty$

  $I[i,0] = -\infty$

  $D[i,0] = -\infty$

  for $j = 1$ to Q **do**

    $M[0,j] = -\infty$

    $I[0,j] = -\infty$

    $D[0,j] = -\infty$

$$M[i,j] = Pe_{M_j}(s_i) + max \begin{cases} M[i-1,j-1] + Pt_{M_{j-1},M_j} \\ I[i-1,j-1] \ + Pt_{I_{j-1},M_j} \\ D[i-1,j-1] \ + Pt_{D_{j-1},M_j} \\ B[i-1] \qquad + Pt_{B,M_j} \end{cases}$$

$$I[i,j] = Pe_{I_j}(s_i) + max \begin{cases} M[i-1,j] + Pt_{M_j,I_j} \\ I[i-1,j] \ + Pt_{I_j,I_j} \end{cases}$$

$$D[i,j] = max \begin{cases} M[i,j-1] + Pt_{M_{j-1},D_j} \\ D[i,j-1] \ + Pt_{D_{j-1},D_j} \end{cases}$$

  end for

  $N[i] = N[i-1] + Pt_{N,N}$

$$E[i] = max \begin{cases} M[i,1] \ + Pt_{M_1,E} \\ M[i,2] \ + Pt_{M_2,E} \\ \quad \vdots \\ M[i,Q] + Pt_{M_Q,E} \end{cases}$$

$$J[i] = max \begin{cases} J[i-1] + Pt_{J,J} \\ E[i] \qquad + Pt_{E,J} \end{cases}$$

$$B[i] = max \begin{cases} N[i] + Pt_{N,B} \\ J[i] \ + Pt_{J,B} \end{cases}$$

$$C[i] = max \begin{cases} C[i-1] + Pt_{C,C} \\ E[i] \qquad + Pt_{E,C} \end{cases}$$

end for

$score = C[|S|] + Pt_{C,T}$

- Uma matriz de *scores* D para os estados *Delete*, tal que $D[i, j]$ indica o *score* do melhor alinhamento que emite os i primeiros símbolos da sequência S e alcança o estado D_j. Uma diferença da matriz D em relação às matrizes M e I é que, quando se calcula o *score* $D[i, j]$, o símbolo s_i já foi emitido antes de se chegar a D_j;

- Vetores de *scores* N, E, J, B e C, tal que a posição i de cada vetor indica o *score* do melhor alinhamento que emite os i primeiros símbolos da sequência S e alcança o estado associado ao vetor em questão.

O Algoritmo 3.1 apresenta o algoritmo de Viterbi, onde Pt_{e_1,e_2} e $Pe_e(s)$ correspondem às probabilidades de transição do estado e_1 para e_2 e de emissão do símbolo s no estado e, respectivamente. O resultado produzido é o *score* do melhor alinhamento que emite os $|S|$ símbolos da sequência S e alcança estado T (*Termination*) final do HMM H.

O algoritmo de Viterbi para um HMM qualquer possui complexidade de tempo $O(Q^2 \times |S|)$, pois permite transições de cada estado para todos os outros estados. No entanto, o algoritmo de Viterbi para a arquitetura Plan7 apresentado no Algoritmo 3.1 possui complexidade de tempo $O(Q \times |S|)$, pois há um número constante de transições possíveis a partir de um estado. O algoritmo basicamente calcula todas as posições das matrizes e vetores de *scores* M, I, D, N, E, J, B e C.

A Figura 3.1 mostra um HMM H com $Q = 5$ nós, extraído do conjunto de exemplos de HMMs disponibilizado juntamente com a ferramenta HMMer [17]. As probabilidades de transição de estado estão indicadas próximas às setas que representam as transições na figura. As probabilidades de emissão dos estados *Match* e *Insert* são mostradas na Tabela 3.1. Todas as probabilidades estão convertidas para valores logarítmicos. A primeira coluna da tabela mostra o estado que cada linha representa e as colunas restantes apresentam o valor da probabilidade de emissão de cada um dos símbolos do alfabeto Σ naquele estado. Por simplicidade, apenas as colunas de alguns símbolos são mostradas.

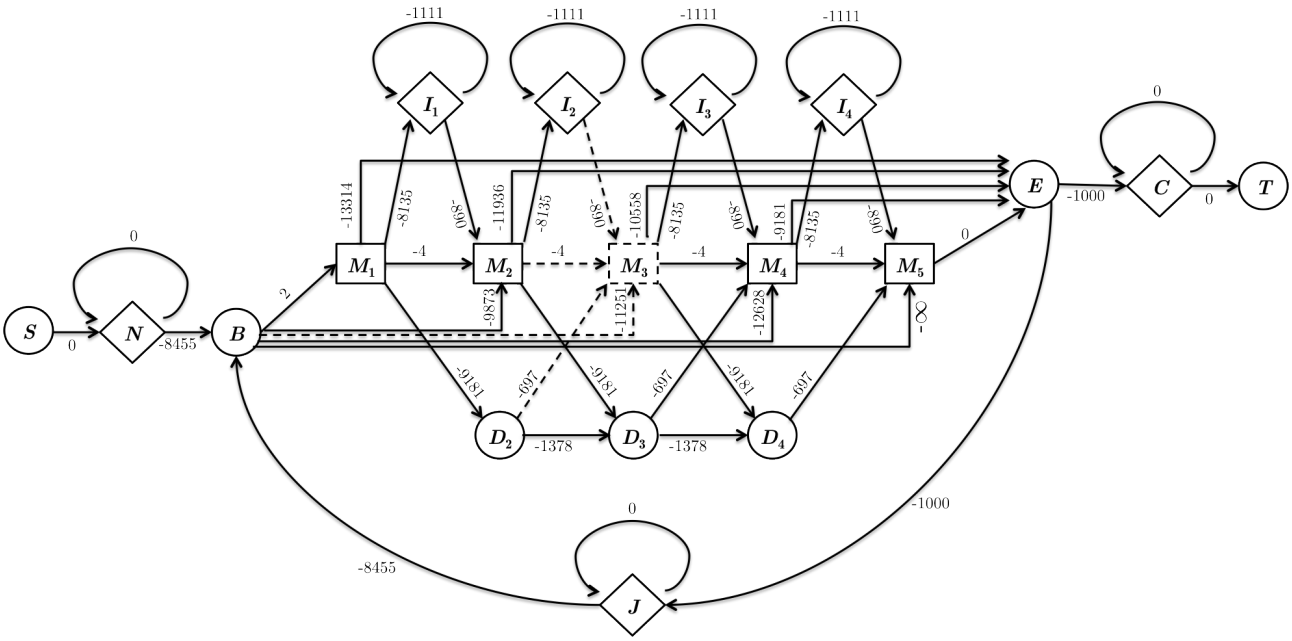


Figura 3.1: *Profile* HMM H com $Q = 5$ nós

O funcionamento do algoritmo de Viterbi para a arquitetura Plan7 é exemplificado para a sequência $S = AYPPQ$ e o HMM H da Figura 3.1. A Tabela 3.2 mostra as matrizes de *scores* M, I e D calculadas pelo algoritmo para a obtenção do melhor alinhamento de S a H. Cada coluna j corresponde ao nó j de H e cada grupo i de três linhas corresponde ao símbolo s_i de S. Cada célula $[i, j]$ da tabela apresenta os *scores* $I[i, j]$, $M[i, j]$ e $D[i, j]$, nesta ordem. A coluna 0 não corresponde a um nó de H e

Tabela 3.1: Probabilidades de emissão dos estados *Match* e *Insert* do HMM H da Figura 3.1

Estado	Símbolos do alfabeto Σ				
	A	P	Q	Y	...
M_1	2343	−3045	−3814	−4616	
M_2	1802	−3710	−3483	4331	
M_3	−4914	4339	−5623	−5713	
M_4	−4914	4339	−5623	−5713	
M_5	1018	2229	3491	−3307	

Estado	Símbolos do alfabeto Σ				
	A	P	Q	Y	...
I_1	−210	423	70	−176	
I_2	−210	423	70	−176	
I_3	−210	423	70	−176	
I_4	−210	423	70	−176	

a linha 0 não representa um símbolo de S, porém elas são necessárias para a iniciação das estruturas de dados do algoritmo.

Tabela 3.2: Matrizes de *scores* I, M e D para $S = AYPPQ$ e HMM H da Figura 3.1

i	Símbolo de S	Nós do HMM (j)						Estado
		0	1	2	3	4	5	
0	−	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$	I
		$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$	M
		$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$	D
1	A	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$	0	I
		$-\infty$	**-6110**	−16526	−24620	−25997	−21443	M
		$-\infty$	$-\infty$	−15291	−16669	−18047	−19425	D
2	Y	$-\infty$	−14421	−24837	−32931	−34308	0	I
		$-\infty$	−13069	**-1783**	−21701	−23079	−22051	M
		$-\infty$	$-\infty$	−22250	−10964	−12342	−13720	D
3	P	$-\infty$	−15109	−9495	−29413	−30791	0	I
		$-\infty$	−11498	−16783	**2552**	−7322	−10810	M
		$-\infty$	$-\infty$	−20679	−22057	−6629	−8007	D
4	P	$-\infty$	−15797	−10183	−5160	−15034	0	I
		$-\infty$	−11498	−15212	−6046	**6887**	−5097	M
		$-\infty$	$-\infty$	−20679	−22057	−15227	−2294	D
5	Q	$-\infty$	−16838	−11224	−6201	−1178	0	I
		$-\infty$	−12267	−14985	−16696	−11673	**10374**	M
		$-\infty$	$-\infty$	−21448	−22826	−24204	−20854	D

A Tabela 3.3 mostra os vetores de *scores* N, E, J, B e C calculados pelo algoritmo de Viterbi. Cada coluna corresponde a um dos vetores e cada linha i corresponde ao símbolo s_i de S. A linha i da tabela apresenta os *scores* $N[i]$, $E[i]$, $J[i]$, $B[i]$ e $C[i]$, nesta ordem.

Por exemplo, o alinhamento ótimo dos três primeiros símbolos de S (o prefixo AYP) que alcança o estado M_3 de H, com $s_3 = P$ sendo emitido por M_3, é formado pelo alinhamento ótimo do prefixo AY que alcança um estado e imediatamente anterior a M_3, seguido pela transição de e para M_3, com maior *score* possível. As transições de estado que chegam a M_3 são $M_2 \to M_3$, $I_2 \to M_3$, $D_2 \to M_3$ e $B \to M_3$, e estão tracejadas na Figura 3.1. Assim, para calcular o elemento $M[P, 3]$, isto é, o *score* do

Tabela 3.3: Vetores de *scores* N, E, J, B e C para $S = AYPPQ$ e HMM H da Figura 3.1

i	Símbolo de S	Estados especiais				
		N	B	E	C	J
0	–	0	-8455	$-\infty$	$-\infty$	$-\infty$
1	A	0	-8455	-19424	-20424	-20424
2	Y	0	-8455	-13719	-14719	-14719
3	P	0	-8455	-8006	-9006	-9006
4	P	0	-8455	-2294	-3294	-3294
5	Q	0	919	10374	9374	9374

alinhamento ótimo do prefixo AYP que alcança M_3 e emite $s_3 = P$ em M_3, o Algoritmo 3.1 realiza as operações a seguir. A transição escolhida é $M_2 \rightarrow M_3$, obtendo o valor $M[P, 3] = 2552$, mostrado na Tabela 3.2.

$$
\begin{aligned}
M[P,3] &= Pe_{M_3}(P) + max \begin{cases} M[Y,2] + Pt_{M_2,M_3} \\ I[Y,2] \ + Pt_{I_2,M_3} \\ D[Y,2] \ + Pt_{D_2,M_3} \\ B[Y] \quad + Pt_{B,M_3} \end{cases} \\
&= 4339 + max \begin{cases} -1783 \ + (-4) \\ -24837 + (-890) \\ -22250 + (-697) \\ -8455 \ + (-11251) \end{cases} \\
&= 4339 - 1787 = 2552
\end{aligned}
$$

3.1.1 Obtenção do alinhamento ótimo

O algoritmo de Viterbi apresentado no Algoritmo 3.1 calcula o *score* do melhor alinhamento de uma sequência S em relação a um *profile* HMM H, porém não constrói o alinhamento propriamente. Este alinhamento consiste do caminho em H que gerou S com maior *score*, isto é, de uma sequência de estados que inicia no estado inicial S (*Start*) e termina no estado final T (*Termination*), com os símbolos da sequência sendo emitidos pelos estados percorridos no caminho. Caso o alinhamento ótimo seja necessário, o algoritmo de Viterbi pode, ao final do cálculo do *score*, realizar um procedimento denominado *traceback* para obter este alinhamento.

O mecanismo de *traceback* é apresentado de maneira simplificada no Algoritmo 3.2 e utiliza as matrizes e vetores de *scores* calculados pelo algoritmo de Viterbi.

O Algoritmo 3.2 constrói o alinhamento ótimo do fim para o início, começando no estado final T. A cada iteração, o algoritmo consulta as matrizes de *scores* do algoritmo de Viterbi e identifica, dentre os estados predecessores do estado atual, aquele que alcançou o estado atual com maior *score*. Esse estado predecessor é então incluído no alinhamento e o procedimento é repetido até que o estado inicial S seja alcançado. A variável *alinhamento* armazena o alinhamento ótimo à medida que ele é construído, como uma lista de elementos (e, s), que representam a emissão do símbolo s no estado e. Para isso, uma operação de concatenação é realizada nas linhas 7 e 10.

Por exemplo, dada a execução do algoritmo de Viterbi mostrada nas Tabelas 3.2 e 3.3, para obter o alinhamento ótimo da sequência $S = AYPPQ$ ao HMM H da Figura 3.1, o procedimento de *traceback* é realizado. O mecanismo inicia no estado T e, como ele possui um único estado predecessor, conclui que o alinhamento ótimo vem do estado C.

Algoritmo 3.2 Algoritmo de *traceback* para arquitetura Plan7

Entradas: HMM H com Q nós e probabilidades de emissão Pe e transição Pt
Sequência $S = s_1 s_2 ... s_{|S|}$
Matrizes e vetores de *scores* M, I, D, N, E, J, B, C
Saída: Alinhamento ótimo de S com H

1: $i = |S|$
2: $alinhamento = (T, -)$
3: $estado_{atual} = T$
4: **while** $estado_{atual} \neq S$ **do**
5: $\quad estado_{predecessor}$ = estado predecessor que alcança $estado_{atual}$ com maior *score*
6: $\quad$ **if** $estado_{predecessor}$ emite símbolo **then**
7: $\quad\quad alinhamento = (estado_{predecessor}, s_i) \rightarrow alinhamento$
8: $\quad\quad i = i - 1$
9: $\quad$ **else**
10: $\quad\quad alinhamento = (estado_{predecessor}, -) \rightarrow alinhamento$
11: $\quad$ **end if**
12: $\quad estado_{atual} = estado_{predecessor}$
13: **end while**

No estado C, como há dois estados predecessores, E e o próprio C, o mecanismo analisa as seguintes alternativas:

- $C[P] + Pt_{C,C} = -3294 + 0 = -3294$, que corresponde à transição saindo do próprio C, onde o símbolo $s_5 = Q$ é emitido; ou
- $E[Q] + Pt_{E,C} = 10374 + (-1000) = 9374$, que corresponde à transição saindo de E, sem emitir nenhum símbolo de S.

Como $9374 > -3294$, o mecanismo conclui que o alinhamento ótimo vem de E e continua a análise a partir dele. Ao final, o alinhamento ótimo de S a H obtido é:

$$(S, -) \rightarrow (N, -) \rightarrow (B, -) \rightarrow (M_1, A) \rightarrow (M_2, Y) \rightarrow (M_3, P) \rightarrow (M_4, P) \rightarrow (M_5, Q) \rightarrow (E, -) \rightarrow (C, -) \rightarrow (T, -),$$

Em geral, o mecanismo de *traceback* é realizado apenas quando o *score* resultante é significativo, o que indica que a sequência em questão faz parte da família modelada pelo *profile* HMM utilizado. Nesse caso, o alinhamento ótimo obtido pelo mecanismo pode ser utilizado para guiar a inclusão da sequência no HMM.

3.1.2 Estruturas de Dados

Analisando mais detalhadamente as estruturas de dados utilizadas pelo algoritmo de Viterbi, é possível calcular a demanda de espaço em memória do Algoritmo 3.1. As entradas de dados do algoritmo são o *profile* HMM e a sequência investigada. Essa sequência de símbolos é representada por um vetor S de caracteres, de tamanho $|S|$.

O HMM é representado através do seu número de nós Q e das probabilidades de transição e emissão (em valores logarítmicos). As probabilidades de transição são organizadas como:

- Um vetor $Pt_{especiais}$ de valores inteiros, com oito posições, que contém as probabilidades de transição de estado especiais do HMM, como mostrado na Figura 3.2.

0	1	2	3	4	5	6	7
$Pt_{N,N}$	$Pt_{N,B}$	$Pt_{E,J}$	$Pt_{E,C}$	$Pt_{C,C}$	$Pt_{C,T}$	$Pt_{J,J}$	$Pt_{J,B}$

Figura 3.2: Probabilidades $Pt_{especiais}$ de transição de estado especiais

- Uma matriz $Pt_{regulares}$ de valores inteiros, de tamanho $Q \times 9$, contendo as probabilidades de transição de estado regulares do HMM. A Figura 3.3 mostra a organização dessa matriz, onde cada linha corresponde a um nó j do HMM e as colunas possuem as nove transições de estado relativas ao nó j.

nó	0	1	2	3	4	5	6	7	8
1									
$\vdots$									
j	$Pt_{M_j,M_{j+1}}$	$Pt_{M_j,D_{j+1}}$	$Pt_{M_j,I_{j+1}}$	$Pt_{M_j,E}$	Pt_{I_j,I_j}	$Pt_{I_j,M_{j+1}}$	$Pt_{D_j,D_{j+1}}$	$Pt_{D_j,M_{j+1}}$	Pt_{B,M_j}
$\vdots$									
Q									

Figura 3.3: Probabilidades $Pt_{regulares}$ de transição de estado regulares

As probabilidades de emissão dos estados *Match* e *Insert* são representadas pelas matrizes de inteiros Pe_M e Pe_I, respectivamente, de tamanho $Q \times |\Sigma|$. Cada matriz possui a estrutura mostrada na Figura 3.4, onde cada linha corresponde a um nó j do HMM e há uma coluna para cada símbolo do alfabeto Σ. Na figura o alfabeto de 20 aminoácidos das sequências de proteínas é utilizado. Por exemplo, a posição $[j, F]$ da matriz Pe_M possui a probabilidade $Pe_{M_j}(F)$ do símbolo F ser emitido pelo estado M_j.

símbolos do alfabeto Σ

nó	A	C	D	E	F	G	H	I	K	L	M	N	P	Q	R	S	T	V	W	Y
1																				
$\vdots$																				
j					$Pe_{M_j}(F)$															
$\vdots$																				
Q																				

Figura 3.4: Probabilidades Pe_M e Pe_I de emissão dos estados *Match* e *Insert*

As matrizes e vetores de *scores*, calculadas pelo algoritmo, são:

- Matrizes M, I e D de valores inteiros, cada uma de tamanho $|S| \times Q$;
- Vetores N, E, J, B e C de valores inteiros, cada um de tamanho $|S|$.

Por exemplo, uma entrada de dados típica do algoritmo de Viterbi consiste de um conjunto de 20 HMMs, extraídos da base de dados de famílias de sequências de proteínas Pfam [65], e que se enquadram em conjunto denominado *Top Twenty* das 20 famílias com maior número de sequências [82]. Os comprimentos médio e máximo desses HMMs são 173 e 488, respectivamente.

O conjunto de sequências a serem investigadas consiste da base de dados de sequências de proteínas UniProt Swiss-Prot [73, 71]. Essa base é composta por 530.264 sequências manualmente anotadas e revisadas, cujos tamanhos médio e máximo são 354 e 32.513, respectivamente. Por se tratarem de sequências de proteínas, o tamanho $|\Sigma|$ do alfabeto de símbolos é 20.

Supondo um *profile* HMM e uma sequência com os tamanhos médios e máximos apresentados, a Tabela 3.4 resume a quantidade de memória necessária para cada estrutura de dados apresentada, para a execução do algoritmo de Viterbi para uma sequência S e um HMM de comprimento Q. Considera-se nesse cálculo que um valor inteiro ocupa quatro bytes e um caracter ocupa um byte. De acordo com a tabela, uma instância do algoritmo de Viterbi precisa, em média, de pelo menos 758,08 KB para armazenar as estruturas de dados, como mostrado na penúltima coluna da Tabela 3.4. Para um HMM e uma sequência com os tamanhos máximos relatados, pelo menos 186.693,93 KB (182,32 MB) são necessários (última coluna da tabela).

Tabela 3.4: Estruturas de dados para comparação sequência-*profile*

Estrutura de dados		Tamanho (bytes)	Tamanho típico (KB)	
			Médio	Máximo
Sequência	S	$\|S\|$	0,35	31,75
Probabilidades de transição	$Pt_{regulares}$	$Q \times 9 \times 4$	6,08	17,16
	$Pt_{especiais}$	8×4	0,03	0,03
Probabilidades de emissão	Pe_M	$Q \times \|\Sigma\| \times 4$	13,52	38,13
	Pe_I	$Q \times \|\Sigma\| \times 4$	13,52	38,13
Scores	M, I, D	$3 \times \|S\| \times Q \times 4$	717,68	185.933,72
	N, E, J, B, C	$5 \times \|S\| \times 4$	6,91	635,02
Total			758,08	186.693,93

3.1.3 Dependências de dados

Apesar de possuir complexidade de tempo polinomial, o algoritmo de Viterbi é bastante custoso computacionalmente, tanto em termos de tempo de execução quanto de espaço em memória, devido ao tamanho das bases de dados de sequências biológicas atualmente.

Uma das abordagens utilizadas na otimização de algoritmos que calculam estruturas de dados como matrizes e vetores, como é o caso do algoritmo de Viterbi, é o cálculo em paralelo de vários elementos de uma estrutura. Entretanto, dependendo das dependências entre os dados calculados, otimizações do gênero não podem ser aplicadas pois produziriam resultados incorretos.

No algoritmo de Viterbi para comparação de uma sequência S com um *profile* HMM H, no cálculo de um elemento $[i, j]$ qualquer das matrizes de *scores* M, I e D, os seguintes valores são necessários:

- $M[i-1, j-1]$, $I[i-1, j-1]$, $D[i-1, j-1]$ e $B[i-1]$, usados no cálculo de $M[i, j]$;
- $M[i-1, j]$ e $I[i-1, j]$, usados no cálculo de $I[i, j]$;
- $M[i, j-1]$ e $D[i, j-1]$, usados no cálculo de $D[i, j]$.

Além disso:

- Todos os elementos $M[i-1, 1], M[i-1, 2], ..., M[i-1, Q]$ da linha $i-1$ de M são necessários para o cálculo de $E[i-1]$;
- $E[i-1]$ é usado no cálculo de $J[i-1]$;
- $J[i-1]$ é usado no cálculo de $B[i-1]$.

As dependências de dados descritas estão ilustradas na Figura 3.5, que mostra as estruturas de dados do algoritmo, com as matrizes M, I e D representadas agrupadas. As setas representam as dependências entre os dados, indicando quais elementos são necessários para o cálculo de outro elemento.

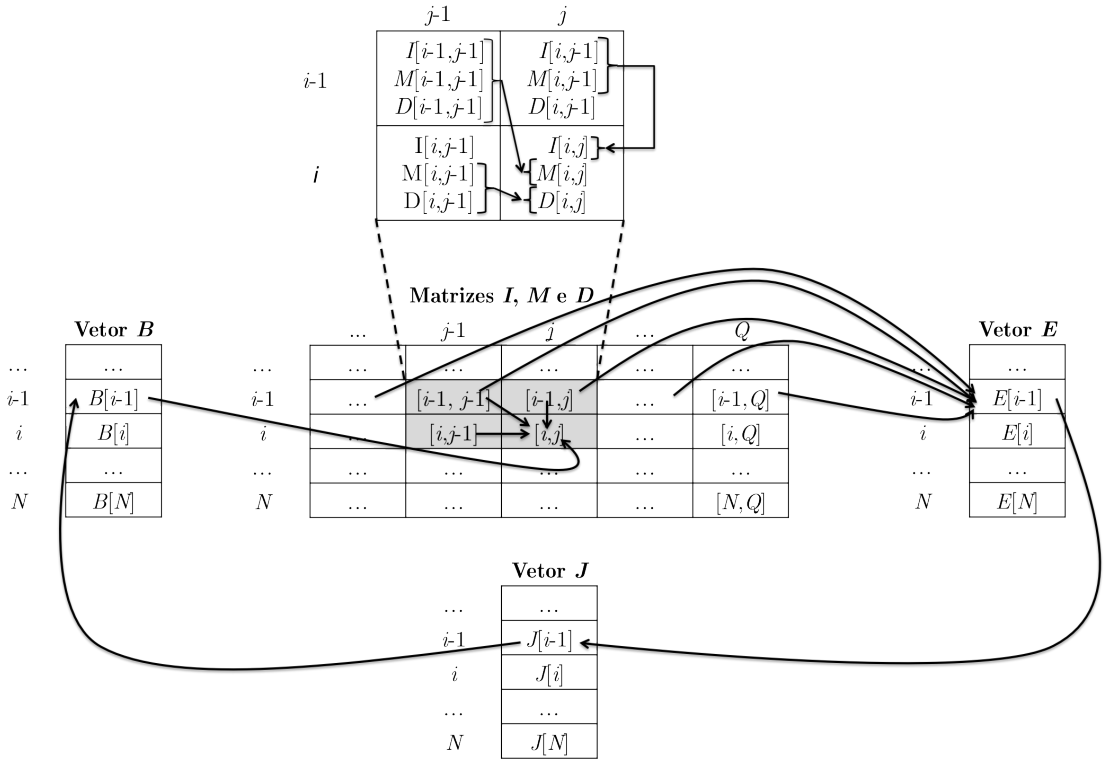


Figura 3.5: Dependências de dados do algoritmo de Viterbi para a arquitetura Plan7

A figura também mostra, de forma expandida, as dependências de dados entre os elementos das posições $[i-1, j-1]$, $[i-1, j]$, $[i, j-1]$ e $[i, j]$ das matrizes M, I e D. Estas dependências impõem que as três matrizes de *scores* sejam calculadas em conjunto. Isto é, não é possível calcular primeiro uma das matrizes inteiramente e depois as outras matrizes.

No cálculo de dois elementos distintos $[i, j]$ e $[k, l]$ das matrizes M, I e D, as seguintes situações podem ocorrer:

- Se $i = k$ então os dois elementos pertencem à mesma linha i das matrizes. Supondo, sem perda de generalidade, que $j < l$, então o elemento $[i, j]$ será necessário para o cálculo de $[i, l]$, pois existe uma cadeia de dependências de $[i, j]$ para $[i, j+1]$, deste para $[i, j+2]$, ..., até de $[i, l-1]$ para $[i, l]$;

- Se $j = l$ então os dois elementos pertencem à mesma coluna j das matrizes. Supondo, sem perda de generalidade, que $i < k$, então o elemento $[i, j]$ será necessário para o cálculo de $[k, l]$, pois existe uma cadeia de dependências de $[i, j]$ para $[i+1, j]$, deste para $[i+2, j]$, ..., até de $[k-1, j]$ para $[k, j]$;

- Se $i \neq k$ e $j \neq l$ então os dois elementos pertencem a linhas e colunas distintas das matrizes. Supondo, sem perda de generalidade, que $i < k$, então o elemento $[i, j]$ será necessário para o cálculo de $[k, l]$, devido a seguinte cadeia de dependências: $[i, j]$ será necessário no cálculo de $E[i]$, que será utilizado no cálculo de $J[i]$ e $B[i]$, sendo este necessário para o cálculo de todos os elementos da linha $i + 1$. Se esse ciclo de dependências for calculado para todas as linhas seguintes das matrizes M, I e D, em algum momento o elemento $B[k-1]$ será necessário para o cálculo de $[k, l]$, demonstrando que $[k, l]$ depende de $[i, j]$.

Portanto, se todas as dependências de dados são mantidas, é impossível calcular em paralelo dois elementos distintos das matrizes de *scores* M, I e D. Essa é a principal dificuldade enfrentada quando se deseja otimizar o algoritmo de Viterbi para a arquitetura Plan7 visando a redução do seu tempo de execução. As dependências entre os dados impedem a exploração de paralelismo no cálculo dos elementos das matrizes e vetores de *scores* no alinhamento de uma sequência a um *profile* HMM.

3.2 Filtro para Alinhamento Sequência-*Profile*

Em geral, para obter o melhor alinhamento de uma sequência com um *profile* HMM, os seguintes passos são realizados: a sequência e o HMM são fornecidos como entradas para o algoritmo de Viterbi, que gera o *score* do melhor ainhamento. Se esse *score* for significativo, o mecanismo de *traceback* é executado para a construção do alinhamento propriamente dito. Caso contrário, a sequência é descartada. A Figura 3.6 mostra esses passos.

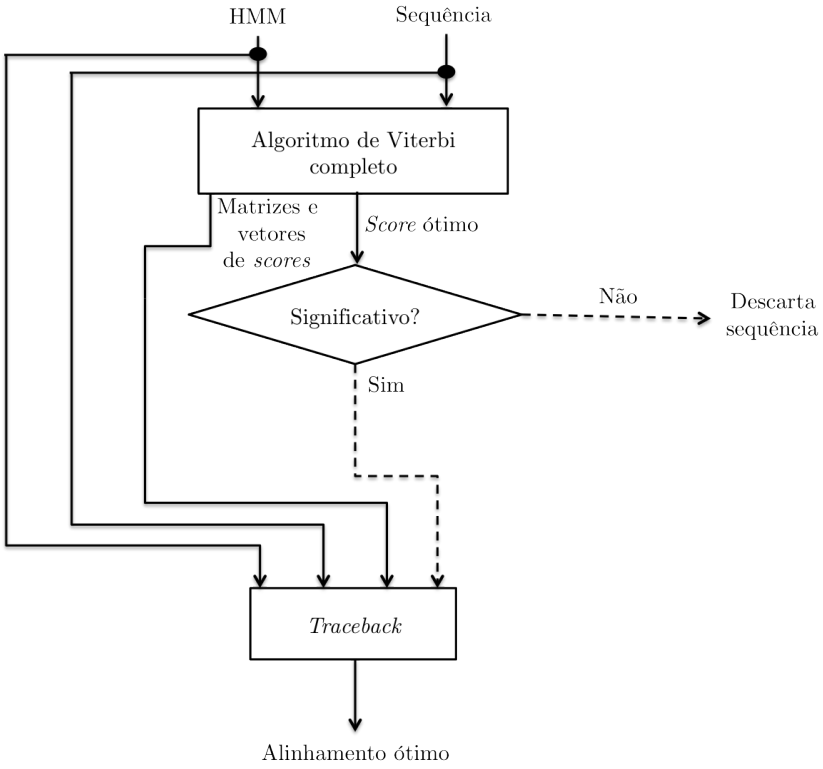


Figura 3.6: Fluxo de execução convencional para o alinhamento sequência-*profile*

Analisando as dependências de dados no cálculo das matrizes e vetores de *scores* do algoritmo de Viterbi descritas na Subseção 3.1.3, é possivel notar que, para o cálculo do elemento $[i, j]$ das matrizes M, I e D e do elemento $[i]$ dos vetores N, E, J, B e C, são necessários apenas dados das linhas i e $i-1$ das matrizes M, I e D e das posições i e $i-1$ dos vetores N, E, J, B e C. Assim, para o cálculo do *score* gerado pelo algoritmo de Viterbi, é necessário armazenar apenas as linhas corrente e anterior das matrizes M, I e D, bem como as posições corrente e anterior dos vetores N, E, J, B e C.

Entretanto, ao armazenar apenas duas linhas das matrizes e vetores de *scores*, não é possível utilizar os dados calculados durante o algoritmo de Viterbi para executar o mecanismo de *traceback* apresentado na Subseção 3.1.1, e assim obter o alinhamento ótimo e não apenas o seu *score*. Mesmo assim, essa abordagem é aplicada em várias soluções que visam otimizar o algoritmo de Viterbi.

Essas soluções implementam uma versão do algoritmo de Viterbi com alocação de apenas duas linhas das matrizes e vetores de *scores*. Se o *score* produzido for significativo, executa-se então o algortitmo de Viterbi completo, gerando as estruturas de dados de *scores* completas, para

posteriormente executar o *traceback*. Dessa forma, a versão do algoritmo de Viterbi de memória reduzida funciona como um filtro que evita a execução do algortitmo de Viterbi completo para todas as sequências investigadas. A Figura 3.7 mostra os passos realizados no alinhamento sequência-*profile*, quando um filtro é utilizado.

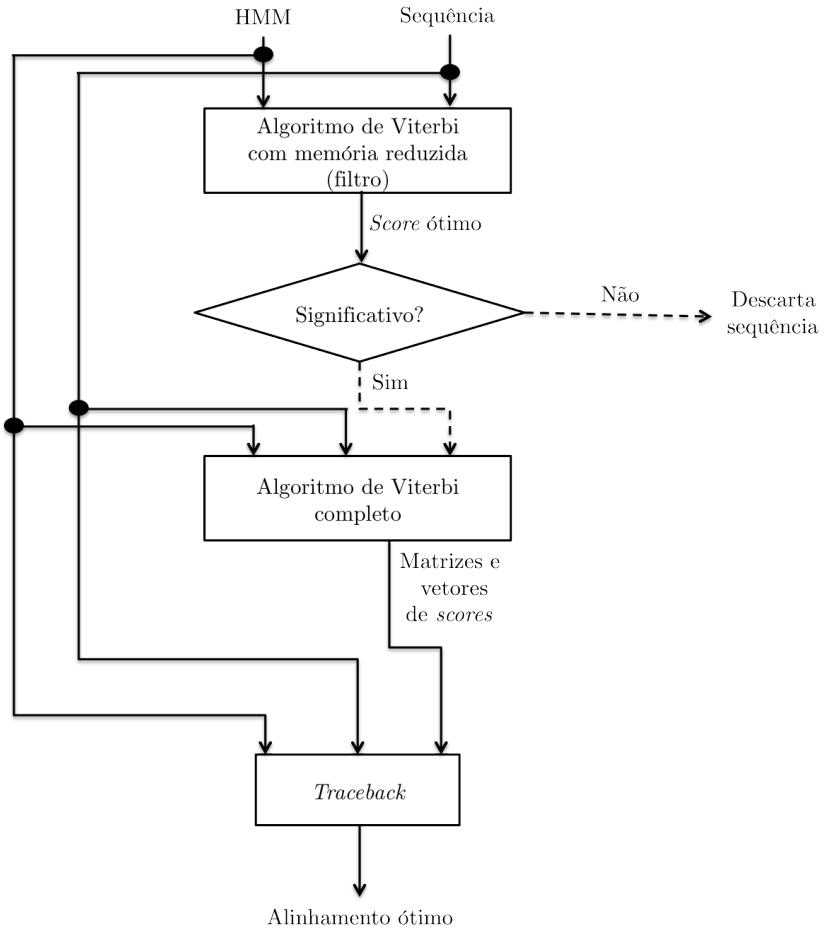


Figura 3.7: Fluxo de execução com filtro para o alinhamento sequência-*profile*

A Tabela 3.5 mostra a quantidade de memória necessária para a execução do algoritmo de Viterbi para uma sequência S e um HMM de comprimento Q, com a alocação de apenas duas linhas das estruturas de dados de *scores*. As duas últimas colunas da tabela mostram esta demanda de memória para a entrada de dados típica descrita na Subseção 3.1.2, considerando um HMM e uma sequência com os tamanhos médios e máximos, respectivamente.

Tabela 3.5: Estruturas de dados para filtro para comparação sequência-*profile*

Estrutura de dados		Tamanho (bytes)	Tamanho típico (KB)	
			Médio	Máximo
Sequência	S	$\|S\|$	0,35	0,35
Probabilidades de transição	$Pt_{regulares}$	$Q \times 9 \times 4$	6,08	17,16
	$Pt_{especiais}$	8×4	0,03	0,03
Probabilidades de emissão	Pe_M	$Q \times \|\Sigma\| \times 4$	13,52	38,13
	Pe_I	$Q \times \|\Sigma\| \times 4$	13,52	38,13
Scores	M, I, D	$3 \times 2 \times Q \times 4$	4,05	11,44
	N, E, J, B, C	$5 \times 2 \times 4$	0,04	0,04
Total			37,58	105,26

3.3 Comparação de uma Base de Sequências com um *Profile* HMM

Em geral, na investigação de novas sequências biológicas, todas as sequências de um conjunto são comparadas a um mesmo *profile* HMM. Essa operação recebe como entradas um conjunto de sequências $S = \{S_1, S_2, ..., S_N\}$ e um HMM H de comprimento Q e utiliza o algoritmo de Viterbi para calcular o *score* do melhor alinhamento de cada sequência $S_k \in S$ com H. O Algoritmo 3.3 mostra simplificadamente a operação. O laço mais externo itera sobre o conjunto de sequências S, enquanto os dois laços internos executam o algoritmo de Viterbi para cada sequência S_k do conjunto.

Algoritmo 3.3 Comparação de base de sequências com *profile* HMM

Entradas: HMM H com Q nós e probabilidades de emissão Pe e transição Pt
Conjunto de sequências $S = \{S_1, S_2, ..., S_N\}$
Saída: *Score* do melhor alinhamento de cada sequência de S com H

```
1:  for k = 1 to N do
2:      // Executa o algoritmo de Viterbi apresentado no Algoritmo 3.1 para a sequência S_k
3:      ...
4:      for i = 1 to |S_k| do
5:          ...
6:          for j = 1 to Q do
7:              ...
8:          end for
9:          ...
10:     end for
11:     ...
12: end for
```

Para execução em processadores convencionais, as diferentes iterações do laço mais externo do Algoritmo 3.3 são executadas sequencialmente. Não há necessidade de replicação das estruturas de dados para cada sequência, pois as instâncias do algoritmo de Viterbi são criadas sequencialmente e as estruturas de dados podem ser reaproveitadas para cada instância. É necessário apenas ter espaço suficiente para armazenar as estruturas de dados para a maior sequência. A Tabela 3.6 mostra a demanda de memória para a comparação sequencial de uma base de sequências S com um HMM de comprimento Q.

Tabela 3.6: Estruturas de dados para comparação base de sequências-*profile*, sequencialmente

Estrutura de dados		Tamanho (bytes)	Tamanho típico (KB)	
			Médio	Máximo
Sequências	$S = \{S_1, ..., S_N\}$	$\max_{1 \le k \le N} \lvert S_k \rvert$	31,75	31,75
Probabilidades de transição	$Pt_{regulares}$	$Q \times 9 \times 4$	6,08	17,16
	$Pt_{especiais}$	8×4	0,03	0,03
Probabilidades de emissão	Pe_M	$Q \times \lvert\Sigma\rvert \times 4$	13,52	38,13
	Pe_I	$Q \times \lvert\Sigma\rvert \times 4$	13,52	38,13
Scores	M, I, D	$3 \times \max_{1 \le k \le N} \lvert S_k \rvert \times Q \times 4$	65.915,03	185.933,72
	N, E, J, B, C	$5 \times \max_{1 \le k \le N} \lvert S_k \rvert \times 4$	635,02	635,02
Total			66.614,94	186.693,93

Diversas soluções propostas exploram alguma forma de paralelismo na comparação de uma base de sequências com um HMM, com o objetivo de reduzir o tempo total de processamento dessa operação. Na comparação de um conjunto de sequências $S = \{S_1, S_2, ..., S_N\}$ com um mesmo HMM através do algoritmo de Viterbi, o cálculo das matrizes e vetores de *scores* de uma sequência $S_k \in S$ não

depende do cálculo das matrizes e vetores de *scores* de uma sequência $S_l \in S$, para $k \neq l$, dado que esses cálculos operam sobre sequências diferentes. Logo, as estruturas de dados de *scores* de S_k podem ser calculadas ao mesmo tempo que as de S_l. Na comparação da base S de N sequências de entrada, existem portanto N instâncias independentes do algoritmo de Viterbi que executam as diferentes iterações do laço mais externo do Algoritmo 3.3. Essas instâncias podem ser executadas em paralelo, cada uma comparando uma sequência diferente de S com o mesmo HMM e explorando uma forma de paralelismo de granularidade grossa denominada **paralelismo entre sequências**.

Para isso, é necessária a replicação das estruturas de dados que armazenam informações específicas do processamento de cada sequência. As estruturas de dados que armazenam a sequência e as matrizes e vetores de *scores* terão conteúdos distintos para cada sequência $S_k \in S$, e portanto devem ser replicadas para permitir que as diferentes instâncias do algoritmo de Viterbi sejam executadas em paralelo. Dependendo da arquitetura utilizada, não é necessária a replicação das estruturas de dados que representam o HMM (probabilidades de transição e emissão), dado que todas as sequências são comparadas ao mesmo HMM, e essas estruturas podem ser compartilhadas entre as várias instâncias do algoritmo de Viterbi.

A Tabela 3.7 mostra a quantidade de memória necessária para implementar a comparação de uma base de sequências S com um HMM de comprimento Q, explorando paralelismo entre sequências. Supõe-se nesse cálculo que não é necessário replicar as informações sobre o HMM (estruturas de dados $Pt_{regulares}$, $Pt_{especiais}$, Pe_M e Pe_I).

Tabela 3.7: Estruturas de dados para comparação base de sequências-*profile*, explorando paralelismo entre sequências

Estrutura de dados		Tamanho (bytes)	Tamanho típico (KB)	
			Médio	Máximo
Sequências	$S = \{S_1, ..., S_N\}$	$\sum_{k=1}^{N} \lvert S_k \rvert$	185.440,97	185.440,97
Probabilidades de transição	$Pt_{regulares}$	$Q \times 9 \times 4$	6,08	17,16
	$Pt_{especiais}$	8×4	0,03	0,03
Probabilidades de emissão	Pe_M	$Q \times \lvert\Sigma\rvert \times 4$	13,52	38,13
	Pe_I	$Q \times \lvert\Sigma\rvert \times 4$	13,52	38,13
Scores	M, I, D	$3 \times \sum_{k=1}^{N} \lvert S_k \rvert \times Q \times 4$	384.975.475,45	1.085.942.381,63
	N, E, J, B, C	$5 \times \sum_{k=1}^{N} \lvert S_k \rvert \times 4$	3.708.819,61	3.708.819,61
Total			388.869.769,19	1.089.836.735,65

Para a entrada de dados típica da Subseção 3.1.2, com um HMM com os tamanhos médio e máximo relatados e considerando a base de sequências UniProt Swiss-Prot inteira, a demanda de memória para a comparação base de sequências-*profile*, explorando paralelismo entre sequências, é 388.869.769,19 KB (370,86 GB) e 1.089.836.735,65 KB (1039,35 GB), respectivamente, como mostrado nas duas últimas colunas da Tabela 3.7.

O grande volume de memória necessário pode inviabilizar a implementação dessa operação em diversas arquiteturas. Nesses casos, aplica-se a abordagem descrita na Seção 3.2, de implementar um filtro com o algoritmo de Viterbi com a alocação de apenas duas linhas das estruturas de dados de *scores*. Se ainda assim, a demanda de memória não pode ser atendida pela arquitetura alvo, é possível dividir a base de sequências em subconjuntos, denominados *batches*. O paralelismo entre sequências é explorado entre as sequências de um mesmo *batch*, enquanto diferentes *batches* são executados sequencialmente.

Capítulo 4

Implementações do Algoritmo de Viterbi

O algoritmo de Viterbi é uma importante ferramenta no processo de comparação de sequências biológicas com famílias de sequências. Porém, a existência de muitas dependências de dados no algoritmo de Viterbi dificulta a aplicação de otimizações. Este capítulo apresenta soluções já desenvolvidas para o algoritmo de Viterbi em variadas plataformas, propondo otimizações para melhorar seu desempenho, isto é, reduzir seu tempo de execução. Os ganhos de desempenho são apresentados na forma de *speedups* obtidos com base no tempo de execução da implementação ou na quantidade de células das matrizes de *scores* que são calculadas por segundo (*Cell Updates Per Second* – CUPS).

4.1 Ferramenta HMMer

A ferramenta HMMer [17, 18] é composta por um conjunto de programas para análise de sequências biológicas. Ela utiliza *profile* HMMs para representar informações estatísticas de famílias de sequências e, através do algoritmo de Viterbi, permite a comparação de novas sequências com as famílias. As duas principais operações desta ferramenta são:

- *hmmsearch*: Dados um HMM e um conjunto de sequências biológicas, compara cada sequência com o HMM fornecido utilizando o algoritmo de Viterbi e retorna informações estatísticas a respeito das similaridades encontradas;
- *hmmpfam* ou *hmmscan*: Dados um conjunto de HMMs e um conjunto de sequências biológicas, compara cada sequência com cada HMM fornecido utilizando o algoritmo de Viterbi e retorna informações estatísticas a respeito das similaridades encontradas.

Desenvolvida em C/C++, a versão 2 do HMMer [17] é dividida em duas diferentes implementações:

- *core_ algorithms*: Implementa o algoritmo de Viterbi e seus relacionados de maneira simples e legível, porém pouco eficiente. Essa implementação não é utilizada como padrão de execução do HMMer e geralmente é usada apenas para estudos introdutórios da ferramenta;
- *fast_ algorithms*: Implementa o algoritmo de Viterbi e seus relacionados realizando diversas otimizações de código que tornam esta implementação mais rápida que a anterior. É utilizada como padrão na execução do HMMer.

O fluxo de execução do HMMer2 para o alinhamento sequência-*profile* segue o esquema convencional mostrado na Figura 3.6. Uma sequência e um HMM são as entradas do algoritmo de Viterbi completo, que gera um *score*. Se esse *score* for considerado significativo, o *traceback* é realizado, caso contrário, a sequência é descartada.

O HMMer2 disponibiliza alguns recursos para exploração de paralelismo entre sequências, como o suporte para *threads* POSIX para multiprocessadores e suporte para a biblioteca de troca de mensagens PVM (*Parallel Virtual Machine*) [25] para *clusters* de computadores. Oferece também otimizações de código para processadores PowerPC usando o conjunto de instruções AltiVec [72], que permite executar algumas iterações do laço mais interno do algoritmo de Viterbi em paralelo, utilizando uma forma limitada do modelo SIMD (*Single Instruction, Multiple Data*) de computação.

O desempenho do HMMer2 foi avaliado por vários artigos, relatando valores de CUPS que variam de 21 MCUPS [4] até 55 MCUPS [31], executando em um processador Intel Pentium 4 de 2,8 GHz com 1GB de memória RAM.

A versão 3 do HMMer [18] tem como principal modificação os critérios utilizados para determinar o *score* de uma sequência. Na versão 2, o algoritmo de Viterbi é utilizado para calcular o *score* da sequência comparada ao HMM e se esse *score* for considerado significativo, conclui-se que a sequência pertence à família representada pelo HMM e o mecanismo de *traceback* é realizado. Já na versão 3, existe um conjunto de filtros responsáveis por selecionar as sequências que devem ou não ser pós-processadas.

O primeiro filtro é um algoritmo denominado *MSV* (*Multiple Segment Viterbi*) [19]. Esse algoritmo é uma versão simplificada do algoritmo de Viterbi e gera um *score* inicial para a sequência em relação ao HMM utilizado. Se esse *score* for considerado significativo, essa sequência é então submetida ao segundo filtro, que é o algoritmo de Viterbi. Se o *score* calculado pelo algoritmo de Viterbi for significativo, essa sequência é então submetida ao terceiro filtro, que é o algoritmo *Forward-Backward*. Se o *score* produzido por este algoritmo também for considerado significativo, então conclui-se que a sequência pertence à família representada pelo HMM e o *traceback* é executado. A Figura 4.1 resume os principais passos realizados na comparação de uma sequência com um HMM no HMMer3.

A ferramenta HMMer é amplamente utilizada na área de Biologia Molecular e, devido ao crescimento do tamanho das bases de dados de sequências biológicas, diversas otimizações que melhoram seu desempenho computacional vêm sendo propostas.

4.2 Soluções em GPU

Esta seção apresenta as principais implementações do algoritmo de Viterbi para análise de sequências biológicas em GPU, plataforma de computação de alto desempenho utilizada para solucionar problemas de propósito geral com grande demanda computacional. Detalhes específicos referentes a GPUs são apresentados no Capítulo 5.

ClawHMMer [29] é uma implementação da comparação base de sequências-*profile* em GPU e explora o paralelismo entre sequências. Os HMMs utilizados possuem apenas os estados M, I e D e, por não possuírem o estado J, não permitem a construção de alinhamentos *multi-hit*.

Na época da publicação do ClawHMMer, qualquer problema implementado em GPU precisava se adaptar a uma modelagem gráfica para que fosse executado. ClawHMMer segue esse conceito e foi implementado com a linguagem de programação Brook para GPUs [6]. Por não se tratar de um problema de computação gráfica, a implementação de programas dessa forma é bastante complexa.

Dentre as estratégias aplicadas no ClawHMMer, as principais são:

- A base de sequências biológicas é dividida em partes menores (chamadas de *batches*) que caibam na memória da GPU. Diferentes *batches* são executados um por vez em uma GPU ou paralelamente em um *cluster* de GPUs;
- A base de sequências é ordenada antes do processamento, buscando que as sequências de um mesmo *batch* tenham tamanhos próximos e assim, exijam quantidades similares de processamento;

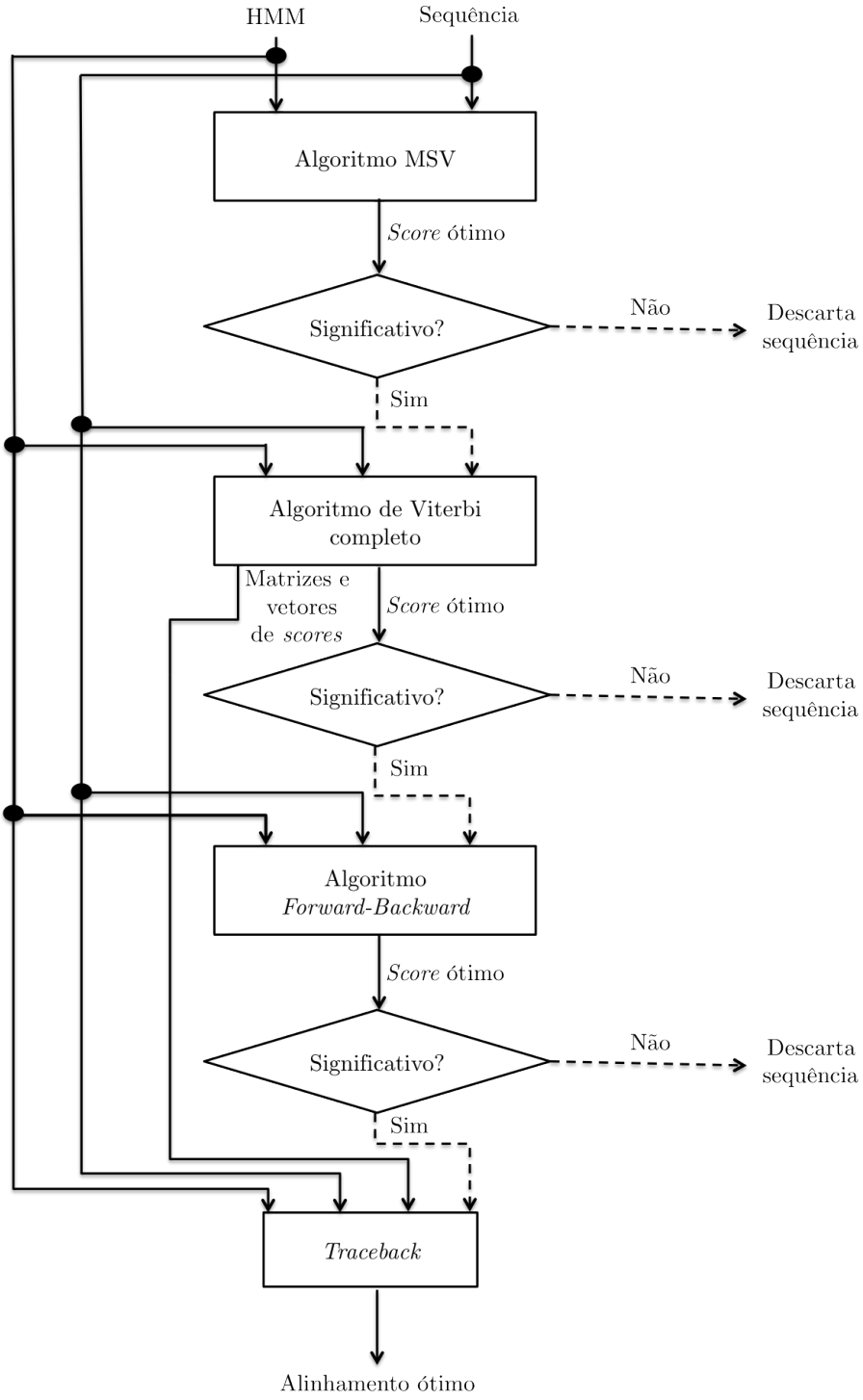


Figura 4.1: Fluxo de execução do HMMer3 para o alinhamento sequência-*profile*

- As probabilidades de transição são carregadas em registradores, pois estes possuem acesso mais rápido que a memória principal;
- Para as matrizes de *scores* M, I e D, apenas a linha corrente e a linha anterior são armazenadas, diminuindo assim a quantidade de memória necessária para processar uma sequência. Por essa razão, o *traceback* não é realizado na GPU.

A avaliação de desempenho comparou o ClawHMMer, executando em diferentes GPUs, com o HMMer2, utilizando uma base de proteínas do NCBI [46]. O ClawHMMer obteve ganhos de desempenho em relação ao HMMer para todos os HMMs, inclusive, em alguns casos, em relação à

versão otimizada para instruções AltiVec. O *speedup* máximo obtido em relação à execução do HMMer2 em um Intel Pentium 4 2,8 GHz foi 36, com a GPU ATI R520 [2] para um HMM de comprimento 1543.

GPU-HMMER [75, 77] implementa a operação *hmmsearch* do HMMer2 em GPU usando HMMs com a arquitetura Plan7. Essa solução utiliza o modelo de programação CUDA, que mapeia a GPU como um dispositivo de programação de propósito geral, sem a necessidade de programação baseada em uma modelagem gráfica.

GPU-HMMER explora o paralelismo entre sequências, ou seja, cada *thread* disparada é responsável por executar o algoritmo de Viterbi para uma sequência diferente. As seguintes estratégias são aplicadas:

- Ordenação da base de sequências, para que as *threads* de um mesmo bloco operem provavelmente quantidades de dados similares e realizem aproximadamente um mesmo volume de processamento;
- *Loop unrolling* do laço mais interno do algoritmo de Viterbi, responsável por iterar os nós do HMM;
- Apenas a linha corrente e a linha anterior das matrizes de *scores* M, I e D são armazenadas, diminuindo a quantidade de memória utilizada, porém impedindo a execução do *traceback* na GPU;
- As probabilidades de transição e as sequências são armazenadas na memória de constantes e na memória de texturas, respectivamente, pois essas memórias possuem cache;
- Acesso coalescido às matrizes de *scores* M, I e D na memória global, acelerando o acesso a essas estruturas;
- As probabilidades de emissão dos estados M e I dependem do símbolo atual da sequência e são acessadas ao final do cálculo de cada posição das matrizes de *scores* correspondentes. Por essa razão, esse símbolo é armazenado na memória compartilhada no início de cada iteração do laço mais externo do algoritmo de Viterbi (que itera nos símbolos da sequência), evitando que ele seja lido da memória de texturas em toda iteração do laço mais interno.

O número de sequências que podem ser processadas paralelamente pelo GPU-HMMER depende de dois fatores: a quantidade de memória disponível para armazenar os dados das sequências e o número de registradores usados por cada *thread*.

GPU-HMMER foi avaliado utilizando a GPU NVIDIA GeForce GTX 8800 Ultra [57] com 768 MB de memória e 128 núcleos executando a uma frequência de 612 MHz. O desempenho obtido foi comparado com o HMMer2 executando em um processador AMD Athlon 275 de 2,2 GHz para uma base de sequências do NCBI. Os *speedups* variaram entre 12 e 38,6, dependendo do HMM utilizado.

Stivala [69] realizou uma avaliação experimental de desempenho do GPU-HMMER utilizando a base de sequências ASTRAL SCOP [8] e executando em uma GPU NVIDIA GeForce GTX 280 [57]. O *speedup* médio obtido foi 12, comparado à execução do HMMer2 em um computador com processador Intel Q8200 (com 4 núcleos) de 2,33 GHz e 4 GB de memória RAM.

CuHMMer [84] também é uma implementação da operação *hmmsearch* do HMMer para GPUs que adota o paralelismo entre sequências e utiliza a arquitetura Plan7 de HMMs. Geralmente, nas implementações que utilizam GPU, o processador ao qual a GPU está acoplada (*host*) fica ocioso enquanto a GPU realiza o processamento. A ideia dessa implementação é aproveitar esse tempo ocioso do *host* para também processar sequências, em paralelo à GPU. As principais estratégias adotadas são:

- Balanceamento do processamento realizado pelas *threads* na GPU através de uma estrutura que agrupa sequências por intervalos de tamanho, sem a necessidade de ordenação. Assim, provavelmente, sequências com tamanhos próximos são executadas ao mesmo tempo na GPU;

- Criação de *threads* no processador *host*, responsáveis por dividir a computação entre o *host* e a GPU de modo a manter a GPU sempre ocupada com dados que utilizem ao máximo a sua capacidade de processamento. Enquanto isso, o *host* executa o HMMer convencional para outras sequências;
- A implementação em GPU funciona como um filtro, portanto não executa o *traceback*;
- As probabilidades de transição e emissão são armazenadas na memória compartilhada da GPU. Caso essa memória não seja suficiente para todas as informações, a memória de texturas é utilizada.

CuHMMer foi avaliado [84] utilizando a GPU NVIDIA GeForce GTX 8800 com 768 MB de memória e 128 núcleos executando a uma frequência de 575 MHz, em conjunto com o processador Intel Core2 E7200 executando o HMMer2. As principais bases de sequências utilizadas foram a Uniprot SwissProt e Trembl [73], juntamente com HMMs com tamanhos entre 37 e 461. Os resultados foram comparados com a execução do HMMer2 no processador AMD Athlon64 X2 Dual Core Processor 3800+. O *speedup* variou de 13 a 45, dependendo do HMM e da base de sequências utilizados.

Du *et al.* [14] implementam o algoritmo de Viterbi em GPU, utilizando HMMs com apenas os estados M, I e D, o que impede a construção de alinhamentos *multi-hit* por não haver o estado J. Esta simplificação é utilizada para permitir a exploração de uma forma diferente de paralelismo.

A Subseção 3.1.3 apresentou as dependências de dados do algoritmo de Viterbi para a arquitetura Plan7, demonstrando que, se todas as dependências de dados forem mantidas, é impossível calcular em paralelo duas posições distintas das matrizes de *scores* M, I e D de uma mesma sequência. Por exemplo, todos os elementos $M[i-1,1],...,M[i-1,Q]$ da linha $i-1$ de M são usados no cálculo de $E[i-1]$, e $B[i-1]$ é usado no cálculo de todos os elementos $M[i,1],...,M[i,Q]$ da linha i de M.

Analisando as dependências de dados mostradas na Figura 3.5, ao remover o estado J do HMM, as dependências de $E[i-1]$ para $J[i-1]$ e deste para $B[i-1]$ são eliminadas. Portanto, a dependência de todos os elementos da linha $i-1$ de M para, necessariamente, todos os elementos da linha i de M deixa de existir. Em consequência, os cálculos das células de uma mesma anti-diagonal d das matrizes de *scores* M, I e D tornam-se independentes entre si e podem ser realizados em paralelo. Essa abordagem permite a exploração de uma forma de paralelismo de granularidade fina, denominada **paralelismo de anti-diagonal**.

A Figura 4.2 ilustra a exploração do paralelismo de anti-diagonal no algoritmo de Viterbi, mostrando as matrizes de *scores* M, I e D e as suas quatro primeiras anti-diagonais. Todas as células de uma mesma anti-diagonal (representada por uma linha pontilhada) podem ser calculadas ao mesmo tempo. As diferentes anti-diagonais são calculadas sequencialmente, d_1, em seguida d_2, e assim sucessivamente, para que as demais dependências de dados do algoritmo de Viterbi sejam respeitadas.

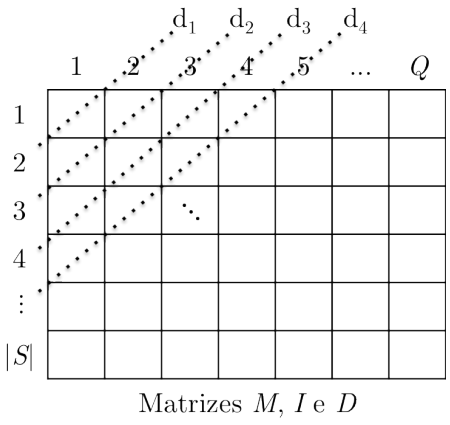


Figura 4.2: Paralelismo de anti-diagonal no cálculo das matrizes de *scores* M, I e D

Du *et al.* [14] exploram o paralelismo de anti-diagonal e apresenta três soluções diferentes:

1. As matrizes M, I e D são armazenadas na memória global da GPU e são calculadas em sucessivos passos. A cada passo, os elementos de uma anti-diagonal são calculados em paralelo. A Figura 4.3(a) representa as matrizes M, I e D e as linhas pontilhadas representam as células que são calculadas em paralelo. As matrizes são armazenadas na memória de forma que as células de uma mesma anti-diagonal ocupem posições consecutivas de memória. A Figura 4.3(b) mostra o padrão de acesso às matrizes na memória, permitindo assim que a GPU realize esses acessos eficientemente. O problema dessa solução é que armazenar as matrizes M, I e D inteiras na GPU limita o tamanho das sequências que podem ser operadas;

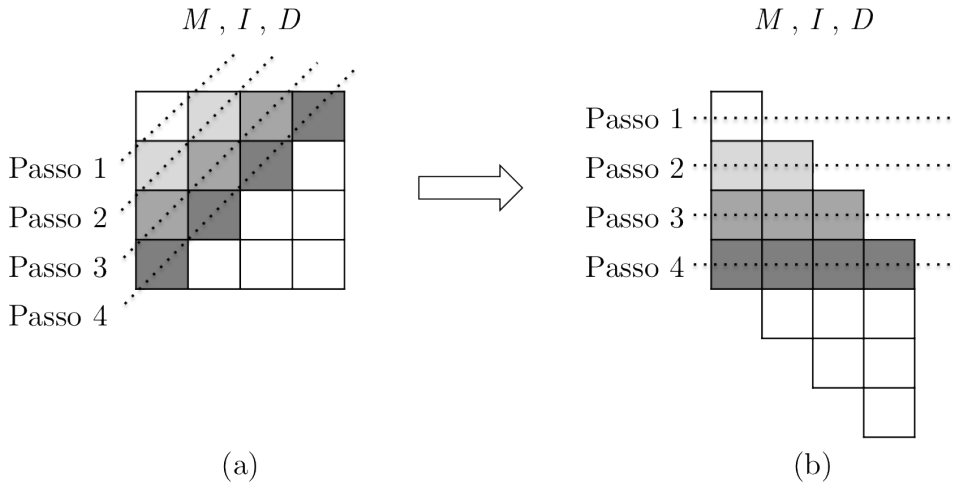


Figura 4.3: Anti-diagonais e padrão de acesso às matrizes M, I e D em [14]

2. Nessa solução, quando uma anti-diagonal d é calculada em paralelo na GPU, apenas as anti-diagonais $d-1$ e $d-2$ estão disponíveis na memória da GPU. Quando o cálculo de d termina, os valores são transferidos para a memória do *host* enquanto a anti-diagonal $d+1$ é calculada, e assim sucessivamente, até que todas as anti-diagonais sejam calculadas, sobrepondo a transferência de dados entre *host* e GPU com a execução na GPU. Com isso, o *traceback* só pode ser realizado no *host*, uma vez que a GPU não possui as matrizes de *scores* completas. Além disso, a transferência de dados frequente entre *host* e GPU influencia negativamente no desempenho;

3. Um pré-processamento é realizado no *host* com o objetivo de encontrar nas sequências investigadas segmentos homólogos (alinhamentos locais com *scores* relevantes). Uma vez encontrados os segmentos, o algoritmo de Viterbi é utilizado para calcular o *score* dos alinhamentos das partes restantes da sequência. As diferentes partes da sequência são processadas em paralelo ou sequencialmente na GPU, dependendo da quantidade de memória disponível.

Essa implementação foi avaliada utilizando a GPU NVIDIA Geforce 9800 GTX com 512 MB de memória e 128 núcleos executando a uma frequência de 675 MHz, e comparada com uma implementação sequencial do algoritmo de Viterbi executada em um processador Intel Dual Core de 2,83 GHz. As três soluções desenvolvidas foram testadas com sequências e famílias geradas artificialmente e os *speedups* obtidos variaram entre 1,97 e 72,21. O maior *speedup* foi alcançado com a terceira solução e o menor com a primeira.

Ganesan *et al.* [24] propõem uma solução em GPU para a comparação base de sequências-*profile*, para HMMs com a arquitetura Plan7, mantendo todas as dependências de dados do algoritmo de Viterbi, porém calculando em paralelo algumas células das matrizes de *scores*. As sucessivas linhas das

matrizes são calculadas sequencialmente, porém os elementos de uma mesma linha são calculados em paralelo.

As dependências de dados que impedem o cálculo em paralelo dos elementos da linha i das matrizes M, I e D são relativas à matriz D, uma vez que o elemento $D[i, j]$ depende de $D[i, j-1]$ e $M[i, j-1]$. A ideia utilizada para resolver esse problema é apresentada na Figura 4.4, que representa a linha i da matriz D. Os elementos de índice 0, k, $2k$ e $3k$ dessa linha são chamados de **âncoras**, e as posições entre duas âncoras formam uma partição.

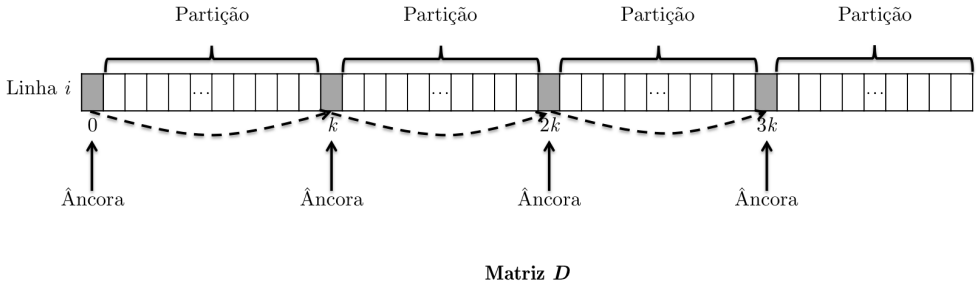


Figura 4.4: Âncoras e partições da linha i da matriz de *scores* D [24]

A relação de recorrência da matriz D é desenvolvida até que os elementos de uma partição dependam apenas do valor da âncora que inicia a partição. Assim, $D[i, 1]$, $D[i, 2]$, ... , $D[i, k]$ dependem apenas de $D[0]$, os elementos $D[i, k+1]$, $D[i, k+2]$, ... , $D[i, 2k]$ dependem apenas de $D[k]$, e assim sucessivamente. O mesmo processo é aplicado para os elementos de M, uma vez que o cálculo de D também depende do cálculo de M.

O processamento de uma linha das matrizes de *scores* 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, 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.

Essa abordagem é implementada em GPU de forma a explorar o paralelismo entre sequências e o paralelismo entre elementos de uma mesma linha. A implementação foi testada com uma base de proteínas da NCBI utilizando HMMs de comprimento 128, 256 e 507. Foram utilizadas 4 GPUs Tesla C1060 com 4 GB de memória e 240 núcleos executando a uma frequência de 1,3 GHz. Os resultados foram comparados com o GPU-HMMER executando nas mesmas GPUs e com uma versão sequencial do algoritmo de Viterbi executando no processador AMD Opteron de 2,33 GHz. A implementação alcançou um *speedup* de 5 a 8 em relação ao GPU-HMMER e um *speedup* de 100 em relação ao algoritmo sequencial.

A Tabela 4.1 apresenta um resumo das principais implementações do algoritmo de Viterbi em GPU. Para cada trabalho desenvolvido, indicado na primeira coluna da tabela, a segunda coluna indica a arquitetura de *profile* HMMs implementada e a terceira mostra a GPU utilizada na avaliação de desempenho. O processador base comparado com a GPU é indicado na quarta coluna, e a última coluna mostra o *speedup* alcançado pela GPU em relação a este processador.

As implementações vistas nesta seção demonstram o interesse da comunidade científica em utilizar GPUs na comparação de sequências biológicas. Essa é uma linha de pesquisa recente na computação, pois a programação de aplicações de propósito geral para GPUs tornou-se viável recentemente [62].

4.3 Soluções em *Cluster*

Nas soluções que implementam o algoritmo de Viterbi em *clusters*, a plataforma de execução consiste em um conjunto de processadores (nós) ligados por uma rede de interconexão. Essas soluções

Tabela 4.1: Principais implementações do algoritmo de Viterbi em GPU

Referência	Arquitetura do HMM	GPU	Processador base	*Speedup*
[29]	M, I e D	ATI R520	Intel Pentium 4	36 (máximo)
[75]	Plan7	NVIDIA GeForce GTX 8800 Ultra	AMD Athlon 275	12 a 38,6
[69]	Plan7	NVIDIA GeForce GTX 280	Intel Q8200	12 (médio)
[84]	Plan7	NVIDIA GeForce GTX 8800 e *host* Intel Core2 E7200	AMD Athlon64 X2 Dual Core 3800+	13 a 45
[14]	M, I e D	NVIDIA Geforce 9800 GTX	Intel Dual Core	1,97 a 72,21
[24]	Plan7	4 NVIDIA Tesla C1060	AMD Opteron	100

exploram o paralelismo entre sequências para reduzir o tempo de execução da comparação base de sequências-*profile*.

MPI-HMMER [78] utiliza a biblioteca MPI [42] em conjunto com um esquema mestre-escravo para implementar as operações *hmmsearch* e *hmmpfam* do HMMer2. Os nós do *cluster* são organizados em diversas sub-redes, cada uma composta por um nó mestre e diversos nós escravos subordinados a esse mestre, como mostrado na Figura 4.5.

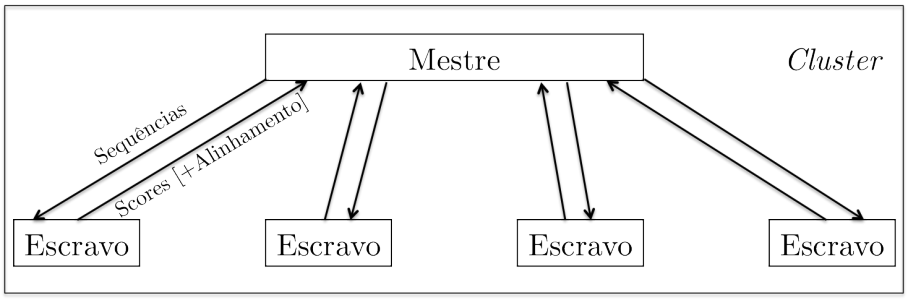


Figura 4.5: Esquema mestre-escravo utilizado no MPI-HMMER [32]

Para a operação *hmmsearch*, distribui-se um subconjunto da base de sequências a serem investigadas para cada sub-rede do *cluster* e cada nó mestre distribui o trabalho entre os seus nós escravos, que realizam independentemente o processamento das sequências recebidas e retornam o resultado para o mestre. Os nós escravos são capazes de executar a versão original do HMMer2 e uma versão otimizada com instruções Intel SSE2 [68], baseada em uma forma limitada do modelo SIMD de computação. A operação *hmmpfam* é implementada de modo similar, exceto por algumas otimizações de E/S realizadas na distribuição dos HMMs.

Devido à sobrecarga de alguns nós mestres, muitos escravos ficavam ociosos até que o mestre enviasse novas sequências para processamento. Esse problema foi atenuado com a otimização da comunicação entre mestres e escravos, através da técnica de *double-buffering* e do empacotamento de várias sequências em uma única mensagem.

O *speedup* obtido ao processar uma amostra de 100 MB de uma base de sequências do NCBI em um cluster com 16 nós executando o HMMer2 com instruções SSE2 foi 7,71, em relação à execução do HMMer2 com instruções SSE2 em apenas um processador. Walters *et al.* [79] afirmam que este *speedup* aumenta conforme o crescimento do número de nós da rede.

Walters *et al.* [76] propõem otimizações no MPI-HMMER, tornando-o mais escalável devido a melhorias na paralelização de E/S. Tanto na operação *hmmsearch* quanto na operação *hmmpfam*, a principal mudança é a manipulação da base de sequências e da base de HMMs utilizando índices de acesso, ao invés do envio direto dos dados para os nós escravos. Além disso, os nós mestres calculam o alinhamento ótimo propriamente dito apenas das sequências que obtiverem *score* significativo, diferentemente do MPI-HMMER, onde os escravos realizam o *traceback* de todas as sequências processadas.

Essa implementação se comporta melhor em *clusters* de grande proporção, chegando a alcançar um *speedup* de 190 em relação ao MPI-HMMER básico, para a operação *hmmsearch* em um *cluster* com 256 processadores. Para *clusters* menores, o MPI-HMMER básico obtém melhor desempenho, mantendo um *speedup* linear para *clusters* com até 64 nós.

Jiang *et al.* [32] propõem outra implementação paralela das operações *hmmsearch* e *hmmpfam* utilizando MPI. As duas principais diferenças em relação às implementações anteriores são: mudança no esquema mestre-escravo, introduzindo outros níveis de hierarquia que diminuem a sobrecarga dos nós mestres; e para a operação *hmmpfam*, a criação de uma *cache* que armazena os HMMs nos nós escravos, evitando o reenvio dos mesmos toda vez que um conjunto de sequências precisar ser calculado. Em conjunto com outras pequenas otimizações, essa implementação obteve uma maior escalabilidade, permitindo a execução em *clusters* com até 2048 nós, e alcançando um *speedup* de até 4 em relação à implementação em MPI sem *cache*.

4.4 Soluções em Hardware

Nas soluções que implementam o algoritmo de Viterbi em hardware, a plataforma de execução consiste de uma FPGA (*Field-Programmable Gate Array*) [41], dispositivo de lógica programável formado por blocos lógicos que podem ser configurados para realizar uma tarefa modelada pelo projetista. Devido a limitações de memória e capacidade das FPGAs, as soluções em hardware em geral não exploram o paralelismo entre sequências. A maioria dessas soluções implementa a estrutura de um *array* sistólico [38] e explora o paralelismo de anti-diagonal.

Um *array* sistólico é um conjunto de unidades interligadas linearmente, denominadas elementos de processamento (PEs – *Processing Elements*). Em um *array* sistólico, as informações fluem entre os PEs como em um *pipeline*, ao longo de sucessivos passos de processamento. A cada passo, cada PE_j recebe dados do PE_{j-1}, anterior a ele no *array*, realiza uma operação simples com estes dados e encaminha os seus resultados gerados para o PE_{j+1} seguinte, que os utilizará com entrada no próximo passo de processamento.

Na implementação do algoritmo de Viterbi com esta estrutura, há no *array* sistólico um PE para cada nó do HMM (de comprimento Q). Cada PE_j é responsável por calcular os elementos da coluna j das matrizes de *scores* M, I e D, uma posição a cada passo de processamento. Os valores de $M[i,j]$, $I[i,j]$ e $D[i,j]$ são calculados em paralelo pelos circuitos em hardware de PE_j, dado que não há dependência de dados entre eles. As células de uma mesma anti-diagonal das matrizers são calculadas em paralelo pelos diferentes PEs, de forma que as sucessivas anti-diagonais são calculadas sequencialmente, a cada passo de processamento. A principal desvantagem desta abordagem é que, como não há estado J, não é possível realizar alinhamentos *multi-hit*.

Os símbolos da sequência $S = s_1 s_2 ... s_{|S|}$ a ser investigada são fornecidos ao PE_1 inicial e fluem da esquerda para a direita pelos PEs do *array* sistólico, um símbolo a cada passo. No passo 1, o símbolo s_1 é fornecido ao PE_1, que calcula a posição $[1,1]$ das matrizes de *scores*. Os demais PEs ficam ociosos. No passo 2, o símbolo s_1 é repassado para o PE_2, que calcula a posição $[1,2]$ das matrizes, ao mesmo tempo que o símbolo s_2 é fornecido ao PE_1, que calcula a posição $[2,1]$. Os demais PEs ficam ociosos. No passo 3, o símbolo s_1 é repassado para o PE_3, que calcula a posição $[1,3]$, ao mesmo tempo que o símbolo s_2 é repassado para o PE_2, que calcula a posição $[2,2]$, e que o símbolo s_3 é fornecido ao PE_1, que calcula a posição $[3,1]$. Os demais PEs ficam ociosos. A operação prosegue, até que, no passo final, o símbolo $s_{|N|}$ é repassado para o PE_Q final, que calcula a posição $[S,Q]$ das matrizes de *scores*.

A Figura 4.6 mostra um *array* sistólico com 4 PEs, usado para calcular as matrizes de *scores* M, I e D, para um HMM de 4 nós e uma sequência de tamanho 5. Cada linha pontilhada indica uma anti-diagonal, cujas células são calculadas em paralelo, em um passo de processamento. A cada passo $t = 1, ..., 8$, as células da anti-diagonal d_t são calculadas.

Benkrid *et al.* [4] utilizam a estrutura do *array* sistólico para implementar em FPGA a comparação

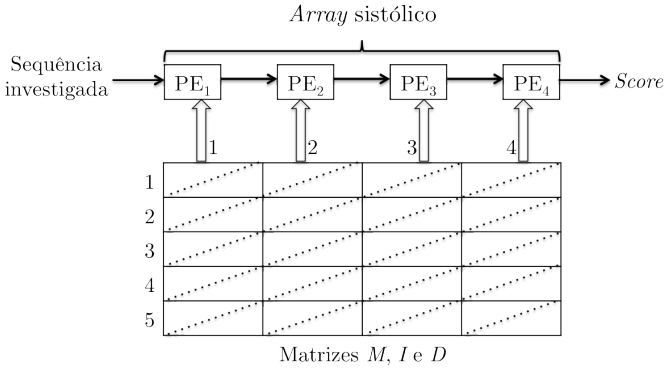


Figura 4.6: *Array* sistólico de PEs para cálculo das matrizes *scores* M, I e D explorando paralelismo de anti-diagonal

base de sequências-*profile* utilizando o algoritmo de Viterbi. O HMM utilizado segue a arquitetura Plan7, mas não possui o estado J, permitindo que o paralelismo de anti-diagonal seja explorado. Executando em uma FPGA Xilinx Virtex-II Pro 2VP100 com 90 PEs a uma frequência de 100 MHz e utilizando um conjunto de sequências de tamanho médio 367 e um *profile* HMM com comprimento máximo 90, essa implementação obteve o desempenho de 5,2 GCUPS (GigaCUPS, onde 1 GCUPS = 10^9 CUPS), enquanto a versão sequencial do HMMer2 executando em um computador Intel Pentium 4 de 2,8 GHz com 1 GB de memória RAM alcançou 21 MCUPS (MegaCUPS, onde 1 MCUPS = 10^6 CUPS), o que representa um *speedup* de 247.

Jacob *et al.* [31] também implementam o algoritmo de Viterbi em FPGA com a estrutura de *array* sistólico para HMMs com os estados M, I e D apenas. Essa implementação explora, além do paralelismo de anti-diagonal, uma forma limitada de paralelismo entre duas sequências. As sequências de entrada são intercaladas, símbolo a símbolo. A operação realizada por cada PE é dividida em dois estágios de um *pipeline*, onde cada estágio opera um símbolo de uma sequência diferente. Executando em uma FPGA Xilinx Virtex-II 6000 com 68 PEs a uma frequência de 180 MHz para amostras de sequências extraídas das bases UniProt Swiss-Prot [73] e HMMs extraídos da PFAM [65], essa implementação atingiu 10,6 GCUPS, enquanto uma versão otimizada do HMMer2 executando em um Intel Pentium 4 de 2,8 GHz com 1 GB de memória RAM alcançou 55 MCUPS, representando um *speedup* de 190.

Maddimsetty *et al.* [40] utilizam o mesmo paralelismo de anti-diagonal para HMMs com os estados M, I e D apenas, porém propõem um mecanismo para reduzir a perda de precisão por não permitirem alinhamentos *multi-hit*. O HMM é duplicado e, ligando as duas cópias do HMM, existe um estado L (*Link*) que desempenha o papel do estado J, permitindo que alinhamentos com até dois *hits* sejam identificados. Os autores estimam que essa solução poderia alcançar um *speedup* de 100 ou mais em relação à implementação sequencial do HMMer2 em um Intel Pentium 4 de 2,8 GHz com 1 GB de memória RAM.

Sun *et al.* [70] propõem a mesma abordagem de anti-diagonal utilizada nas outras implementações, porém recalculam parte das matrizes de *scores* caso detectem que o *score* obtido com a utilização do estado J seria maior. Executando em uma FPGA Xilinx Virtex-5 110T com 25 PEs a uma frequência de 130 MHz, essa implementação alcançou 3,2 GCUPS, enquanto a versão sequencial do HMMer2 executando em um Intel Core2 Duo de 2,33 GHz com 4 GB de memória RAM atingiu 35,5 MCUPS, o que representa um *speedup* de 90.

Derrien e Quinton [13] apresentam um modelo teórico para o estudo das diferentes maneiras de explorar paralelismo no cálculo das posições das matrizes de *scores* de várias sequências, sem que o estado J seja eliminado. Algumas abordagens são apresentadas e uma delas, que explora o paralelismo entre sequências, é implementada na FPGA Xilinx Spartan-3 4000, alcançando 33 MCUPS. O *speedup* obtido foi 1,4 em relação ao HMMer2 sequencial executando em um processador Intel Pentium 4, que

atingiu 24 MCUPS.

Walters *et al.* [80] apresenta uma implementação que explora o paralelismo entre sequências utilizando um *cluster* de computadores, organizados no esquema mestre-escravo, sendo que cada nó escravo pode ter uma FPGA conectada a ele. O desempenho obtido é superior ao alcançado utilizando apenas o *cluster* ou apenas a FPGA.

As Tabelas 4.2 e 4.3 apresentam um resumo das principais implementações do algoritmo de Viterbi em FPGAs. Para cada trabalho indicado na primeira coluna, as três colunas seguintes indicam a arquitetura dos *profile* HMMs implementados, a FPGA utilizada na avaliação de desempenho e o processador base comparado com a FPGA. As duas colunas finais mostram o desempenho obtido pela implementação em FPGA em GCUPS e o *speedup* alcançado em relação ao processador base. A segunda tabela detalha a capacidade máxima das implementações, em termos do número máximo de PEs do *array* sistólico, a frequência máxima do *clock* obtida com a FPGA e o comprimento máximo do HMM e das sequências tratadas. Alguns dados não são informados nas referências.

Tabela 4.2: Principais implementações do algoritmo de Viterbi em FPGA

Referência	Arquitetura do HMM	FPGA	Processador base	GCUPS	*Speedup* máximo
[4]	Plan7 sem J	Xilinx Virtex-II Pro 2VP100	Intel Pentium 4	5,2	247
[31]	M, I e D	Xilinx Virtex-II 6000	Intel Pentium 4	10,6	190
[40]	M, I e D, HMM duplicado	não sintetizado	Intel Pentium 4	–	100 (estimado)
[70]	Plan7 sem J, com correção	Xilinx Virtex-5 110T	Intel Core2 Duo	3,2	90 56,8 (médio)
[13]	Plan7	Xilinx Spartan-3 4000	Intel Pentium 4	0,033	1,4

Tabela 4.3: Capacidade máxima das implementações do algoritmo de Viterbi em FPGA

Referência	Número máximo de PEs	Frequência máxima do *clock* (MHz)	Comprimento máximo do HMM	Tamanho máximo das sequências
[4]	90	100	90	367 (médio)
[31]	68	180	544	1024
[40]	50	200 (estimado)	215 (médio)	–
[70]	25	130	–	–
[13]	1	33	400	–

4.5 Considerações Finais

Implementações em clusters costumam ter bom ganho de desempenho em redes com muitos nós, o que as faz ter um alto custo. Já implementações em FPGA têm custo mais acessível, mas em geral, para obterem bom desempenho, eliminam o estado J, diminuindo a precisão no cálculo da similaridade entre uma sequência e uma família, fazendo com que sequências que teriam *score* significativo por conterem várias subsequências com alinhamentos locais com o HMM, formando assim um alinhamento *multi-hit*, sejam descartadas.

GPUs surgem como plataformas com custo-benefício razoável, pois GPUs com quantidade de núcleos intermediária (como é o caso da GPU utilizada nos experimentos deste trabalho) costumam ser mais acessíveis que FPGAs, além de possuírem interação mais amigável com a plataforma hospedeira. Soluções nessa plataforma têm bom desempenho e, dependendo da solução projetada, não têm perda de precisão dos resultados. Além disso, como a utilização de GPUs como plataforma de computação de alto desempenho de propósito geral é recente, há diversas otimizações que podem ser investigadas e exploradas no desenvolvimento de aplicações nesta plataforma. Por isso, as GPUs são escolhidas

como plataforma de alto desempenho para o desenvolvimento do acelerador do algoritmo de Viterbi proposto neste trabalho.

Capítulo 5

Dispositivo de Computação Paralela GPU

Uma **GPU** (*Graphics Processing Unit*) é um dispositivo de hardware desenvolvido inicialmente com o objetivo de realizar operações gráficas tais como geração de imagens e vídeos, mas que, devido ao seu crescente poder computacional e capacidade de execução paralela, passou a ser utilizado como plataforma de computação de alto desempenho para solucionar problemas de propósito geral com grande demanda computacional [62].

GPGPU (*General-Purpose Computation on Graphics Hardware*) representa a utilização de GPUs para processamento de aplicações não gráficas [62]. Programar aplicações de propósito geral para GPUs era, inicialmente, uma tarefa difícil, pois os problemas precisavam ser mapeados em um *pipeline* gráfico, que na maioria dos casos não se adequava à estrutura do problema. Com o passar do tempo, as GPUs tornaram-se mais genéricas e, atualmente, é possível programá-las utilizando modelos de programação que, de maneira simples, mapeiam e executam o código solicitado na GPU em questão. Bioinformática [74], Banco de Dados [27] e Inteligência Artifical [61] são exemplos de áreas que usam GPUs para otimizar o tempo de execução de suas tarefas.

5.1 Modelo de Programação OpenCL

Com o crescente interesse de outras áreas na capacidade de processamento paralelo das GPUs, tornou-se necessária a criação de modelos de programação que permitissem a implementação de um problema de propósito geral em GPU de modo mais amigável. O modelo de programação OpenCL, baseado na arquitetura CUDA [53], é um desses modelos.

OpenCL (*Open Computing Language*) [34, 83] é um modelo de programação criado por um consórcio de empresas de computação [35] e aos poucos está sendo incorporado pelos principais fabricantes de GPUs. A ideia principal é fornecer aos fabricantes de dispositivos uma linguagem, uma API (*Application Programming Interface*) e bibliotecas para que eles implementem *drivers* que permitam a execução de códigos OpenCL em seus dispositivos. OpenCL foi projetado para executar em plataformas heterogêneas tais como processadores convencionais, GPUs e outros processadores como o IBM Cell [33]. Sendo assim, código produzido apresenta portabilidade.

A NVIDIA disponibiliza apenas uma versão beta do *driver* da versão 1.1 [60] do OpenCL, portanto, o *driver* oficial mais atual da NVIDIA implementa a versão 1.0. Todos os conceitos referentes ao OpenCL apresentados nesta seção baseiam-se nesta versão e as diferenças entre as versões podem ser encontradas em [34].

A NVIDIA também disponibiliza um *toolkit* englobando ferramentas que permitem a compilação de programas escritos em CUDA C/C++ [52] ou utilizando bibliotecas como o OpenCL [34]. O *toolkit* também fornece a ferramenta de análise *OpenCL Visual Profiler*, que permite extrair informações de utilização da GPU durante a execução de programas [55]. Além do *toolkit*, um SDK (*Software*

Development Kit) [58] disponível fornece um conjunto de exemplos de programas escritos em CUDA C/C++ e OpenCL que auxiliam no desenvolvimento.

5.1.1 Modelo da plataforma

O modelo de plataforma sobre o qual aplicações OpenCL executam é mostrado na Figura 5.1. O modelo consiste de:

- Um computador *host*, ao qual são conectados um ou mais dispositivos de computação, e responsável por fornecer comandos a serem executados nesses dispositivos;
- **Dispositivos de computação** (*Compute Devices*), capazes de receber e executar programas OpenCL enviados pelo *host*. Um dispositivo de computação (por exemplo GPUs) é formado por várias unidades de computação;
- **Unidades de computação** (*Compute Units*), que são agrupamentos de elementos de processamento;
- **Elementos de processamento** (*Processing Elements* – PEs), que são processadores responsáveis por realizar as computações.

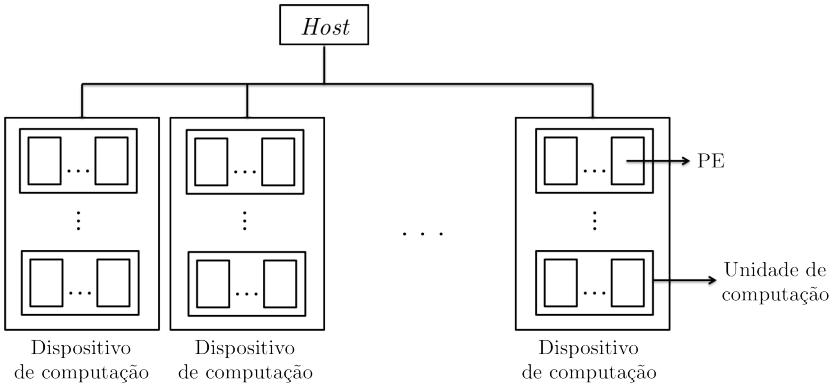


Figura 5.1: Modelo da plataforma OpenCL [34]

PEs de uma mesma unidade de computação executam o mesmo conjunto de instruções, podendo se comportar como unidades SIMD (*Single Instruction Multiple Data*) que executam em passos síncronos ou como unidades SPMD (*Single Program Multiple Data*), cada PE com o seu PC (*program counter*). Neste trabalho, a abordagem SIMD é adotada.

Embora possua um modelo de plataforma, OpenCL executa de acordo com a plataforma hospedeira, mapeando suas estruturas nas estruturas existentes no modelo da plataforma hospedeira.

5.1.2 Modelo de execução

Os comandos submetidos por um *host* a um dispositivo de computação devem seguir um modelo de execução. Os termos envolvidos no modelo de execução do OpenCL são:

- *Kernel*: programa escrito em uma linguagem formada por um subconjunto do padrão *C99* da linguagem de programação C, executado em paralelo nos PEs;
- *Work-item*: instância de um *kernel*;
- *Work-group*: agrupamento de *work-items*.

O modelo de execução do OpenCL pode ser de dois tipos:

- Baseado no **paralelismo de tarefas**, onde apenas uma instância de um *kernel* é executada, ou seja, um *work-group* com um *work-item*. O paralelismo é explorado, por exemplo, utilizando tipos de dados vetoriais implementados pelo dispositivo de computação. Ideal quando a plataforma hospedeira é um processador vetorial;
- Baseado no **paralelismo de dados**, onde a computação é definida em termos de sequências de instruções aplicadas a múltiplos elementos de memória. *Work-items* x e y referentes ao mesmo *kernel* calculam posições de memória diferentes, definidas pelos seus identificadores de *work-item* e *work-group*. Esse modelo é utilizado neste trabalho, pois se adequa ao modelo de arquitetura das GPUs.

Todos os *work-items* associados a um mesmo *kernel* executam o mesmo código e *work-items* de um mesmo *work-group* são executados concorrentemente. Embora o nível de paralelismo seja alto, OpenCL permite a sincronização de *work-items* de um mesmo *work-group* através de barreiras. Já os *work-groups* não possuem sincronismo entre si, condição necessária para que o OpenCL escalone os *work-items* de maneira a manter os PEs sempre ocupados.

Quando um *kernel* é submetido pelo *host*, um espaço de execução é definido pelo usuário e um *work-item* é executado para cada ponto neste espaço. Cada *work-item* possui uma identificação única global, criada a partir de suas coordenadas no espaço de execução. *Work-groups* também possuem uma identificação única global. Essas identificações são usadas para determinar o subproblema de atuação de cada *work-item*.

O espaço de execução utilizado pelo OpenCL pode possuir uma, duas ou três dimensões. A Figura 5.2 mostra um exemplo de um espaço de execução com duas dimensões. Os valores presentes nessa figura têm os seguintes significados:

- G_x: quantidade total de *work-items* na dimensão x;
- G_y: quantidade total de *work-items* na dimensão y;
- S_x: quantidade de *work-items* na dimensão x dentro de um *work-group*;
- S_y: quantidade de *work-items* na dimensão y dentro de um *work-group*.

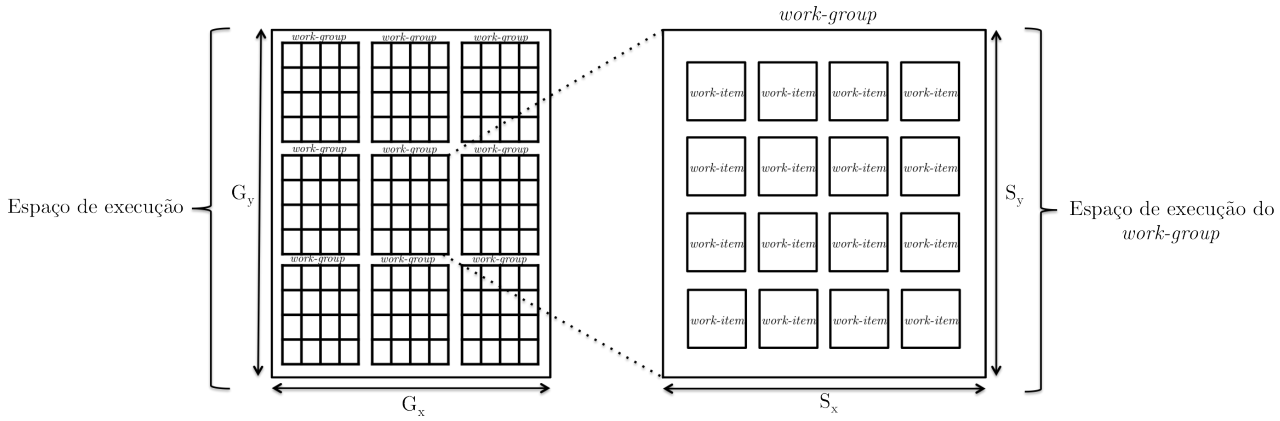


Figura 5.2: Espaço de execução com duas dimensões no modelo de programação OpenCL [34]

Todos esses valores devem ser informados pelo *host* quando um *kernel* é invocado, e então a plataforma hospedeira se encarrega de escalonar e executar os *work-items* nos PEs.

5.1.3 Hierarquia de memórias

Dentro da arquitetura do OpenCL, os *work-items* que executam um *kernel* podem acessar posições de memória que contenham informações necessárias para o cálculo dos subproblemas. O conjunto de memórias do OpenCL é organizado na forma de uma hierarquia, composta por quatro tipos distintos de regiões de memória:

- **Memória global**: Região de memória que pode ser acessada por qualquer *work-item* do espaço de execução. Ou seja, uma mesma posição da memória global pode ser acessada por um *work-item* a pertencente a um *work-group* w_1 e por um *work-item* b pertencente a um *work-group* w_2;
- **Memória constante**: Região da memória global que permanece constante durante toda a execução do *kernel*. Possui as mesmas condições de acesso da memória global;
- **Memória local**: Região de memória compartilhada entre todos os *work-items* do mesmo *work-group*. Ou seja, a posição m da memória local acessada por um *work-item* a pertencente a um *work-group* w_1 é diferente da posição m da memória local acessada por um *work-item* b pertencente a um *work-group* w_2;
- **Memória exclusiva**: Região de memória exclusiva a um *work-item*. Ou seja, a posição m da memória exclusiva de um *work-item* a é diferente da posição m da memória exclusiva de um *work-item* b, independente do *work-group* ao qual eles pertencem.

A Figura 5.3 mostra as regiões de memória e como elas interagem com cada um dos componentes da plataforma OpenCL. O *host* e a GPU têm diferentes permissões de acesso a essas memórias, como mostra a Tabela 5.1.

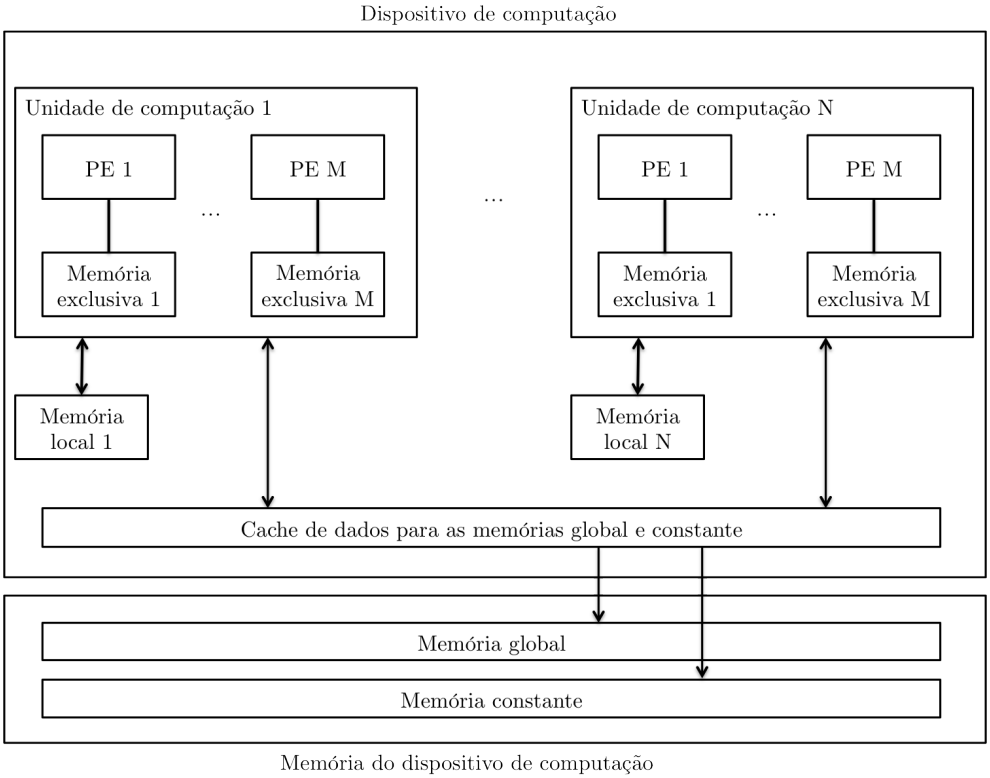


Figura 5.3: Hierarquia de memórias da plataforma OpenCL [34]

Existem duas maneiras de um *kernel* utilizar variáveis alocadas nas memórias da GPU: receber por parâmetro as variáveis alocadas pelo *host* na memória da GPU ou criá-las dentro do próprio *kernel*.

Tabela 5.1: Permissões de acesso às memórias no modelo OpenCL

Memória	Permissão de acesso		Alocação	
	Host	GPU	*Host*	GPU
Global	Leitura e escrita	Leitura e escrita	Dinâmica	Estática
Constante	Leitura e escrita	Leitura	Dinâmica	Estática
Local	Sem acesso	Leitura e escrita	Dinâmica	Estática
Exclusiva	Sem acesso	Leitura e escrita	Sem acesso	Estática

Para identificar em qual região de memória uma variável está armazenada, um qualificador deve ser colocado antes de sua declaração: `__global`, `__constant`, `__local` ou `__private`.

Por exemplo, no Algoritmo 5.1 é apresentado um *kernel* que recebe um parâmetro `a` alocado na memória global e atribui um de seus elementos para uma variável `b` alocada na memória local. A variável `id` não possui um qualificador de espaço de memória. Quando isso acontece, a variável é colocada no espaço de memória genérico, que depende do local da sua declaração. Como a variável `id` foi declara dentro do *kernel*, seu espaço de memória genérico é a memória exclusiva.

Algoritmo 5.1 Exemplo de *kernel* e uso de qualificadores de espaço de memória

```
1: __kernel void Assign(__global int* a)
2: {
3:       int id = get_global_id(0);
4:       __local int b = a[id];
5: }
```

A nomenclatura das memórias apresentada é adotada pelo modelo de programação OpenCL. Outros modelos de programação para GPU adotam uma nomenclatura diferente, porém com conceitos de organização análogos aos aqui descritos.

5.1.4 Programa exemplo

O problema da soma de dois vetores, definido a seguir, é usado para exemplificar a implementação usando o modelo de programação OpenCL.

Definição 9. *A soma de dois vetores a e b de tamanho N resulta em um vetor c de tamanho N, cujas posições são calculadas da seguinte maneira:* $c[i] = a[i] + b[i]$, $0 \leq i \leq N - 1$.

Supondo, sem perda de generalidade, que $N = 64$, o problema pode ser dividido da seguinte maneira: cada *work-item* i ($0 \leq i \leq 63$) é responsável por somar $a[i]$ com $b[i]$ e armazenar o resultado em $c[i]$, como mostra a Figura 5.4.

Figura 5.4: Tarefa de cada *work-item* na soma de dois vetores

A quantidade de dimensões do espaço de execução de um *kernel* não influencia no desempenho de um *kernel*, mas costuma ser definida com base no problema sendo resolvido e na quantidade de dimensões dos dados sendo manipulados, pois desta maneira os identificadores únicos dos *work-groups* e dos *work-items* auxiliam no acesso aos dados de cada *work-item*. Neste exemplo, como os dados possuem apenas uma dimensão, define-se um espaço de execução com apenas uma dimensão também.

A Seção 6.3 apresenta heurísticas que orientam a escolha do tamanho de um *work-group*. Nesse exemplo, o tamanho 32 é usado para os *work-groups*, como mostra a Figura 5.5. Assim, o cálculo das 64 posições do vetor c é dividido entre dois *work-groups*, cada um com 32 *work-items* responsáveis por calcular uma das posições de c.

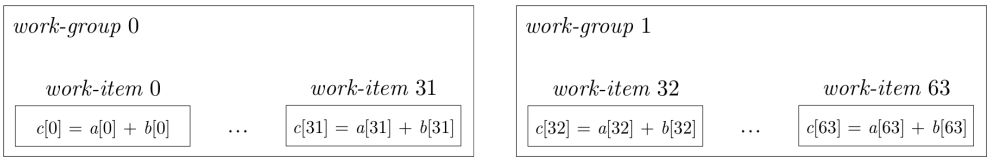


Figura 5.5: Agrupamento de *work-items* em *work-groups*

Uma implementação em OpenCL possui pelo menos dois arquivos: um com extensão .cpp ou .c, que executa no *host* com o objetivo de configurar o ambiente de execução, alocar espaço de memória na GPU e no *host* e invocar os *kernels* na GPU, e outro com extensão **.cl**, que contém o código dos *kernels* que serão executados na GPU.

Para a implementação da soma de dois vetores em OpenCL, o Algoritmo 5.2 mostra o programa executado no *host*. Por simplicidade, alguns detalhes de implementações foram omitidos. Nas linhas 2 e 3, as variáveis `global_work_size` e `local_work_size` armazenam a quantidade total de *work-items* e o tamanho de cada *work-group*, respectivamente. Nas linhas 4 e 5, os vetores `a` e `b`, alocados na memória global da GPU, recebem seus valores dos vetores `a_host` e `b_host`, alocados na memória do *host*. A linha 6 cria o ambiente de execução com dois *work-groups* de 32 *work-items* utilizando as variáveis `global_work_size` e `local_work_size`. A linha 7 copia o vetor resultante `c`, alocado na memória da GPU, para a variável `c_host`, alocado na memória do *host*. A variável `command_queue`, passada como parâmetro nas funções `clEnqueueWriteBuffer`, `clEnqueueNDRangeKernel` e `clEnqueueReadBuffer`, representa a fila de comandos a serem executados da GPU.

Algoritmo 5.2 Programa executado no *host* para soma de dois vetores em GPU

```
1: ...
2: global_work_size = 64;
3: local_work_size = 32;
4: error_code = clEnqueueWriteBuffer(command_queue, a, CL_TRUE, 0, sizeof(cl_float) *
   global_work_size, a_host, 0, NULL, NULL);
5: error_code = clEnqueueWriteBuffer(command_queue, b, CL_TRUE, 0, sizeof(cl_float) *
   global_work_size, b_host, 0, NULL, NULL);
6: error_code = clEnqueueNDRangeKernel(command_queue, kernel, 1, NULL,
   &global_work_size, &local_work_size, 0, NULL, NULL);
7: error_code = clEnqueueReadBuffer(command_queue, c, CL_TRUE, 0, sizeof(cl_float) *
   global_work_size, c_host, 0, NULL, NULL);
8: ...
```

O *kernel* responsável pela soma de dois vetores a e b em GPU é apresentado no Algoritmo 5.3, onde:

- A palavra reservada `__kernel` indica que a função declarada é um *kernel*;
- Os vetores a e b são os dois vetores que serão somados, o vetor c armazena o resultado e o valor inteiro N é o tamanho dos vetores a e b;
- A palavra reservada `__global` antes de cada parâmetro indica que a variável em questão está armazenada na memória global da GPU;
- A função `get_global_id(0)` retorna a identificação global de cada *work-item*;

- A atribuição `c[id] = a[id] + b[id]` armazena na posição *id* de *c* o resultado da soma dos valores na posição *id* de *a* e *b*. Como *id* representa a identificação global do *work-item* e essa identificação varia entre 0 e 63, todas as posições são calculadas quando as 64 instâncias do *kernel* são invocadas.

Algoritmo 5.3 *Kernel* para soma de dois vetores em GPU

```
1: __kernel void VectorAdd(__global const float* a, __global const float* b, __global
   float* c, int N)
2: {
3:       int id = get_global_id(0);
4:       if(id ≥ N)
5:            return;
6:       c[id] = a[id] + b[id];
7: }
```

Programas desenvolvidos em OpenCL executam de acordo com a plataforma hospedeira, que neste trabalho, é uma GPU com arquitetura CUDA.

5.2 CUDA

CUDA (*Compute Unified Device Architecture*) é uma arquitetura de software e hardware criada pela NVIDIA (*circa* 2006), capaz de executar computações em GPUs NVIDIA sem a necessidade de utilizar APIs gráficas [53, 37].

As arquiteturas de hardware Fermi [48] e Kepler [59] foram criadas recentemente pela NVIDIA. A GPU utilizada nos experimentos deste trabalho pertence à arquitetura Fermi, portanto, ela é utilizada como referência neste texto. Esta arquitetura, mostrada na Figura 5.6, é representativa de várias GPUs recentes, entretanto outros fabricantes como a ATI [2] possuem sua própria arquitetura, que é similar em alguns aspectos. Comparativos entre essas arquiteturas podem ser encontrados em [43].

Os principais componentes dessa arquitetura são:

- *Streaming Processor* (SP): núcleo capaz de executar uma operação de inteiro ou ponto flutuante a cada ciclo de *clock*. Uma GPU com arquitetura Fermi pode conter até 512 SPs. Um SP equivale a um PE do OpenCL;
- *Streaming Multiprocessor* (SM): agrupamento de 32 SPs, com algumas características adicionadas apresentadas na Figura 5.7:
  - 16 unidades de *load* e *store*, responsáveis por ler e escrever na cache ou na memória global;
  - Unidades de funções especiais (*Special Function Units* – SFU), responsável por executar funções como seno e cosseno;
  - Dois escalonadores de *warps*[1], responsáveis por escolher dois *warps* para serem executados concorrentemente;
  - Conjunto de registradores;
  - 64 KB de memória *on-chip* de acesso mais rápido que a memória global, denominada *memória compartilhada*. Essa memória corresponde à memória local do OpenCL, e parte dela pode ser utilizada como *cache* L1.
- **DRAM**: memória global com até 6GB, representada pelo mesmo nome no OpenCL;

[1] Um *warp* é um conjunto de *threads* executadas paralelamente e será melhor definido mais adiante nesta seção

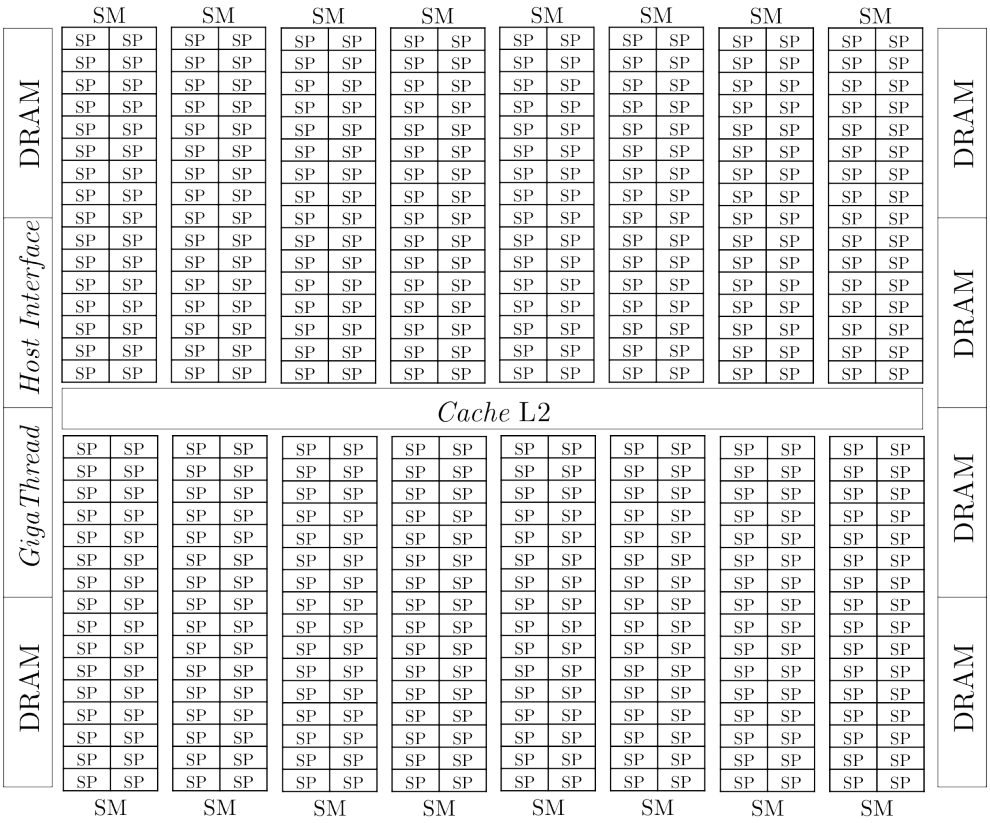


Figura 5.6: Modelo de arquitetura NVIDIA Fermi [48]

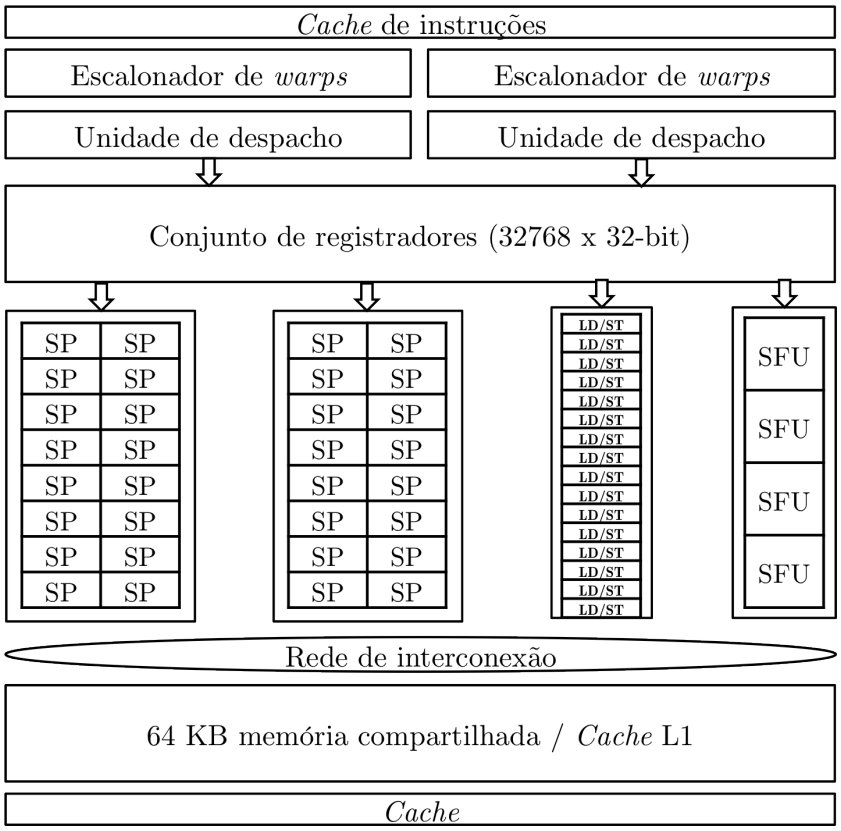


Figura 5.7: *Streaming Multiprocessor* [48]

- **Cache L2**: Cache de 768 KB que atende operações de *load* e *store* feitas à DRAM por qualquer SM;
- **GigaThread**: escalonador responsável por encaminhar blocos de *threads* para os SMs;
- **Host Interface**: Interface de conexão da GPU com o *host*, geralmente através de um barramento PCI Express [63].

CUDA possui um modelo de programação próprio e os principais conceitos referentes a este modelo estão representados na Figura 5.8. Esses conceitos podem ser mapeados nos conceitos do OpenCL da seguinte maneira:

- Uma *thread* CUDA é equivalente a um *work-item* OpenCL;
- Um bloco de *threads* CUDA é equivalente a um *work-group* OpenCL;
- Um conjunto de blocos de *threads* CUDA é denominado de *grid*. Em OpenCL, não há uma nomenclatura específica para este conceito.

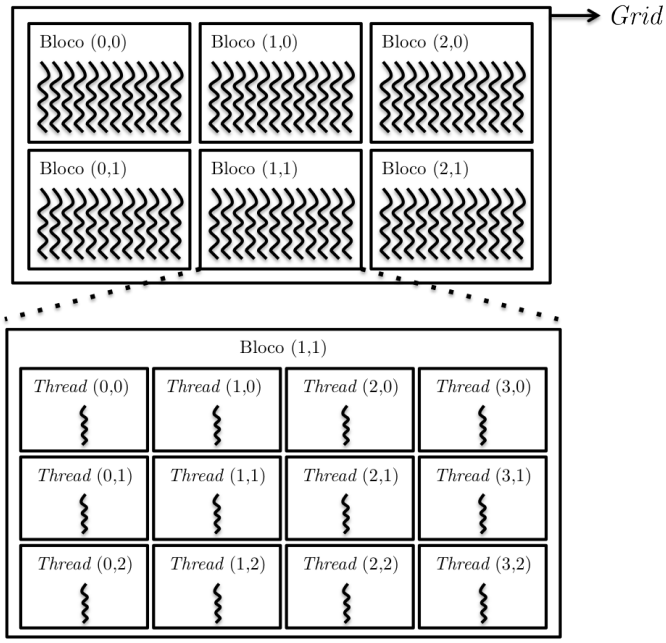


Figura 5.8: Espaço de execução no modelo de programação CUDA [53]

Quando um programa escrito em OpenCL é executado na plataforma CUDA, é criada uma *thread* para cada *work-item*, um bloco de *threads* para cada *work-group* e um *grid* contendo esses blocos [51].

Threads de um mesmo bloco são executadas no mesmo SM em conjuntos de 32 *threads*, chamados *warps*. Quando um SM recebe um bloco de *threads* para executar, ele é particionado em *warps* que são escalonados para execução pelo escalonador de *warps*. Esse particionamento acontece de tal maneira que cada *warp* contém *threads* consecutivas, começando da *thread* 0 até a última *thread* do bloco. Por exemplo, um bloco de 64 *threads* é particionado em dois *warps*, de tal maneira que o primeiro *warp* contém as *threads* $0, ..., 31$ e o segundo *warp* contém as *threads* $32, ..., 63$. *Warps* são executados utilizando instruções SIMD (*Single Instruction, Multiple Data*), que executam paralelamente a mesma operação sobre diferentes dados. Dessa forma, as *threads* de um mesmo *warp* executam de forma síncrona.

A Figura 5.9 mostra simplificadamente a execução de uma instrução de um *warp* gerado pela invocação do *kernel* apresentado no Algoritmo 5.3. Os SPs de um SM executam a mesma instrução `c[id] = a[id] + b[id]`, cada SP trabalhando com uma posição de `a`, `b` e `c`.

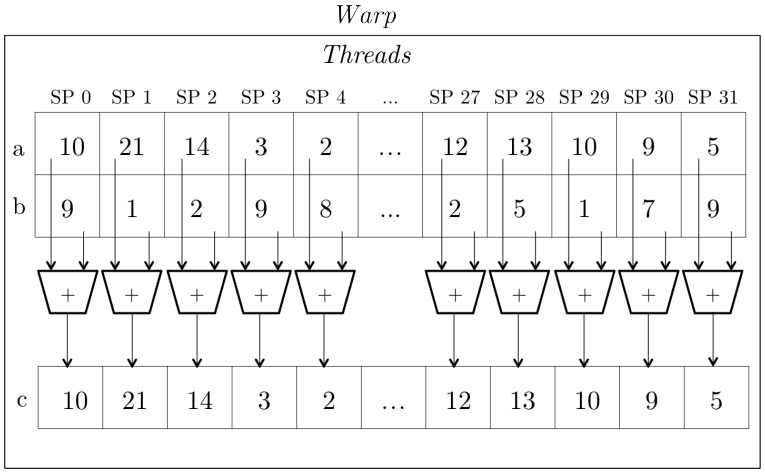


Figura 5.9: Exemplo de execução de um *warp* do *kernel* de soma de dois vetores

5.2.1 Capacidade de computação

A capacidade de computação de uma GPU NVIDIA é representada por um número com o formato $X.Y$, onde X indica a versão da arquitetura e Y a versão das melhorias realizadas nesta arquitetura [51]. A capacidade de computação 1.0 foi a precursora, enquanto a capacidade de computação 3.5 é a mais recente, disponível para a arquitetura Kepler.

As diferenças entre as capacidades de computação são evidentes principalmente nos seguintes aspectos:

- Quantidade de SPs;
- Quantidade e capacidade das unidades de operações aritméticas;
- Quantidade de escalonadores de *warp*;
- Tamanho da memória local;
- Quantidade de bancos na memória local;
- Requisitos necessários para a coalescência de memória global;
- Existência de *cache* de DRAM.

Como a capacidade de computação é definida pela arquitetura do dispositivo, ela também determina os requisitos necessários para que algumas otimizações possam ser aplicadas. A GPU utilizada nos experimentos deste trabalho possui capacidade de computação 2.1, que servirá de referência para as otimizações.

5.2.2 Comparação entre OpenCL e CUDA

O OpenCL e o modelo de programação utilizado por CUDA são bastante similares, porém existem vantagens e desvantagens de cada modelo.

CUDA foi desenvolvida pela NVIDIA e a evolução dessa arquitetura, bem como do seu modelo de programação, depende apenas dos esforços da própria empresa. Já o OpenCL é gerenciado por um consórcio de empresas, e para que futuras modificações sejam aceitas, as empresas participantes precisam colaborar e entrar em acordo. Por essa razão, atualizações no OpenCL demoram mais tempo para serem lançadas.

OpenCL executa utilizando a arquitetura CUDA em GPUs da NVIDIA, e *kernels* escritos em OpenCL C, após compilados, são transformados em código PTX, que é um conjunto de instruções nativas da arquitetura CUDA. Experimentos realizados demonstraram que, executando um *kernel* com a mesma funcionalidade, CUDA executou 6.4× mais rápido que OpenCL para um volume de dados pequeno e 1.4× mais rápido para um volume maior de dados [81].

Uma das condições para que um determinado problema seja implementado em GPU de modo eficiente é que exista paralelismo de dados suficiente para ocupar os processadores, ou seja, implementações em GPUs manipulam grandes quantidades de dados. Dado que a aplicação estudada neste trabalho, análise de sequências biológicas, manipula grandes volumes de dados, a perda de desempenho do OpenCL em relação ao CUDA será provavelmente pequena, e essa perda de desempenho é compensada pela principal vantagem do OpenCL: portabilidade.

Experimentos realizados mostram que o mesmo *kernel* escrito em OpenCL pode ser executado em diferentes dispositivos com pouca ou nenhuma alteração de código [81]. Portanto, quando se deseja testar a implementação de um programa em várias plataformas diferentes e realizar comparações entre elas, ou utilizar uma plataforma híbrida composta de diferentes dispositivos, OpenCL se apresenta como uma ótima alternativa.

É importante ressaltar que a similaridade entre os modelos de programação OpenCL e CUDA faz com que a transcrição de implementações entre esses modelos não seja uma tarefa árdua [50].

5.3 Aplicabilidade das GPUs

Com a evolução das GPUs nos últimos anos, muitos problemas implementados em outras arquiteturas de alto desempenho passaram a ser implementados também em GPUs. Entretanto, para que um problema qualquer alcance algum ganho de desempenho nessa arquitetura, é necessário que ele atenda alguns requisitos básicos [49]:

- O problema deve oferecer bastante paralelismo de dados, isto é, não deve existir uma forte dependência entre os dados manipulados por diferentes *threads*;
- A quantidade de dados utilizados deve ser suficiente para manter todos ou quase todos os processadores da GPU ocupados;
- O acesso aos dados deve seguir um padrão que facilite o agrupamento de acessos na memória global, denominado coalescência (Subseção 6.1.1);
- As transferências de memória entre o *host* e a GPU devem ser mínimas. Problemas que necessitam enviar e receber dados da GPU a todo instante dificilmente terão ganho de desempenho;
- Os dados transferidos para a GPU devem ser submetidos a uma quantidade considerável de operações aritméticas, para que o custo da transferência seja amortizado pelas operações realizadas em paralelo.

Capítulo 6

Otimizações em GPU

No decorrer deste trabalho será possível compreender que problemas implementados em GPUs podem apresentar pouco ou nenhum ganho de desempenho em relação a implementações em outras arquiteturas caso otimizações específicas não sejam aplicadas, isto é, se o problema for implementado de forma análoga àquela utilizada para uma arquitetura convencional.

Neste capítulo são apresentadas as principais otimizações utilizadas na implementação de soluções em GPUs. O principal objetivo dessas otimizações é reduzir o tempo de execução.

6.1 Utilização das Memórias

O uso correto das memórias da GPU, respeitando a latência, tamanho e largura de banda de cada memória, é uma recomendação de alta prioridade nesta plataforma, pois provê bons ganhos de desempenho [49].

6.1.1 Acessos coalescidos à memória global

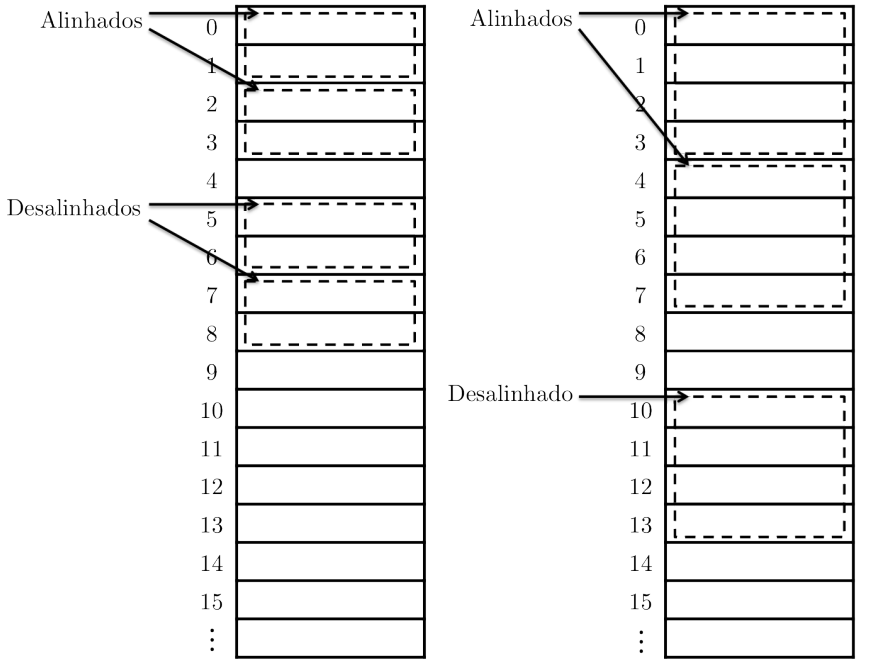
Em diversas arquiteturas, acessos à memória possuem tamanho maior que 1 byte. Nesses casos, é importante que esses acessos estejam alinhados, conforme a Definição 10.

Definição 10. *Um acesso a um dado de b bytes em uma memória está alinhado se o endereço inicial do dado for múltiplo de b [28].*

A Figura 6.1 mostra acessos de diversos tamanhos à memória e as situações nas quais esses acessos estão alinhados ou desalinhados. Em 6.1(a), os dados são de 2 bytes, portanto, acessos aos endereços 0 e 2 estão alinhados, pois 0 e 2 são múltiplos de 2. Já acessos aos endereços 5 e 7 não estão alinhados, pois ambos não são múltiplos de 2. A mesma regra vale para 6.1(b), para acessos a dados de tamanho 4 bytes.

A Figura 6.2 mostra exemplos de segmentos de memória de 32, 64 e 128 bytes. Acessos à memória global da GPU são realizados em transações, que trazem um ou mais segmentos de memória. Um segmento é um bloco de memória cujo acesso é alinhado. Se algumas condições forem obedecidas, é possível agrupar acessos a diferentes palavras de memória realizados pelas *threads* de um *warp* em apenas uma transação, gerando acessos coalescidos à memória.

A coalescência de memória acontece quando *threads* de um *warp* acessam palavras pertencentes ao mesmo segmento de 128 bytes da memória global, podendo haver *threads* ociosas. Dessa maneira, apenas uma transação é gerada para atender todas as solicitações. Caso contrário, duas ou mais transações são necessárias para atender requisições, dependendo do padrão de acesso às posições [51].



(a) Para dados de 2 bytes (b) Para dados de 4 bytes

Figura 6.1: Acessos alinhados e desalinhados à memória

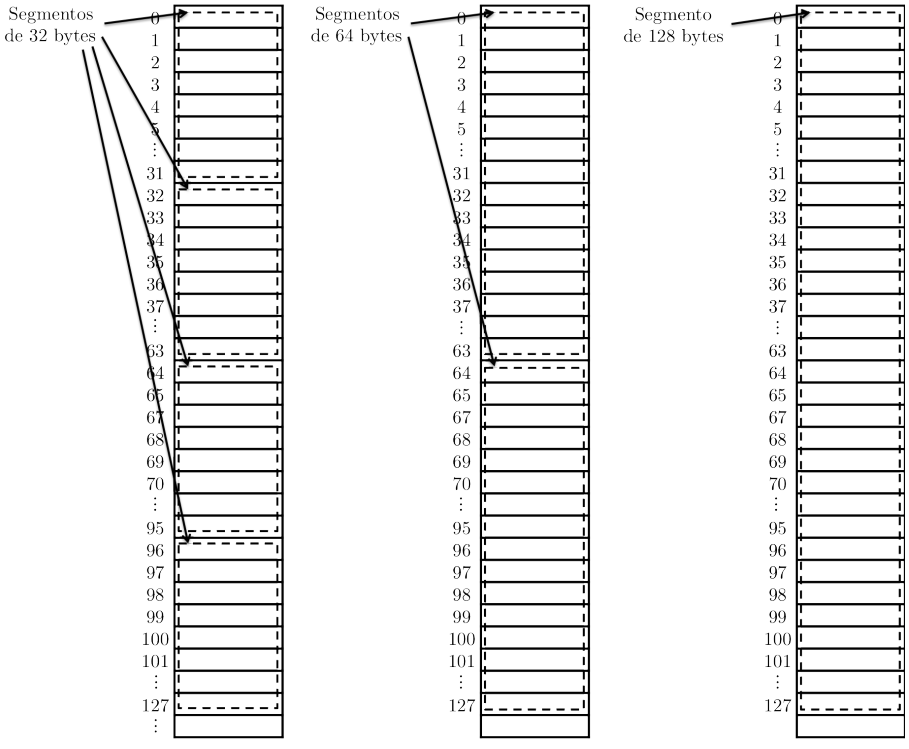


Figura 6.2: Segmentos da memória global de uma GPU

Supondo que cada acesso à memória traga 4 bytes e uma GPU com capacidade de computação 2.1, as Figuras 6.3 e 6.4 mostram um conjunto de acessos não coalescidos e um conjunto de acessos coalescidos, respectivamente. Na Figura 6.3, todos os acessos caem em segmentos diferentes, fazendo com que, para cada acesso, uma transação de 128 bytes seja gerada, bem como um desperdício de banda de 124 bytes. Já na Figura 6.4, todos os acessos caem no mesmo segmento de 128 bytes e apenas uma transação é necessária para atender aos 32 acessos, além de não haver desperdício de banda.

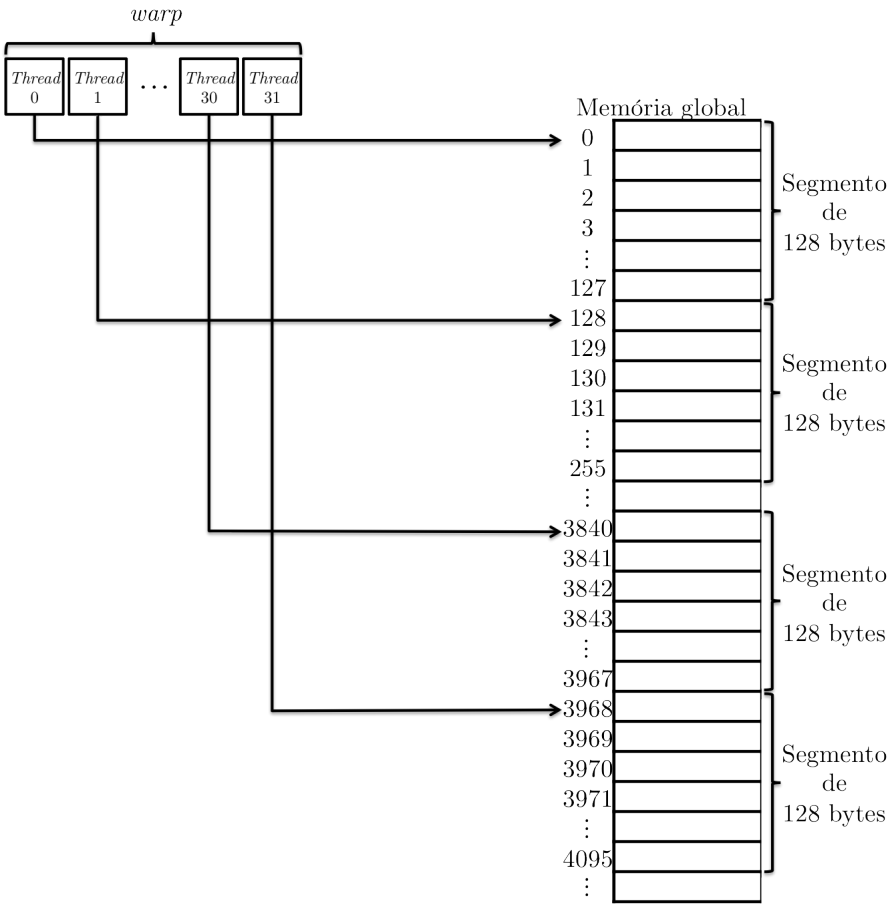


Figura 6.3: Conjunto de acessos não coalescidos à memória global

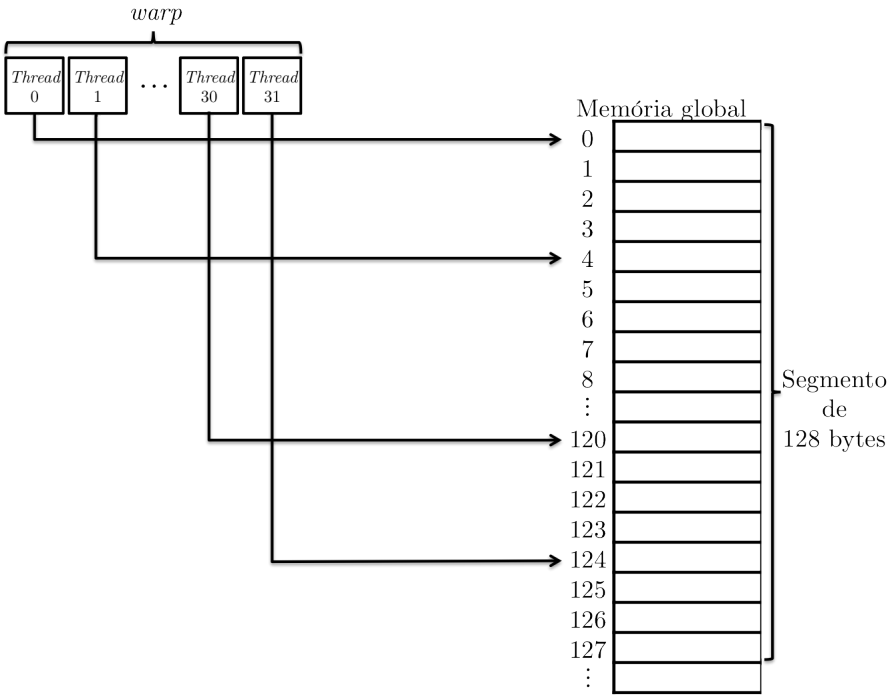


Figura 6.4: Conjunto de acessos coalescidos à memória global

Em GPUs com capacidade de computação 1.3, o tamanho da transação é variável, podendo ser

de 32 bytes, 64 bytes ou 128 bytes e não há *cache* de memória global. A quantidade de transações necessárias para atender uma requisição, bem como o tamanho de cada transação depende do padrão de acesso de um meio-*warp*, que corresponde a 16 *threads* de um warp. Mais informações a respeito da coalescência nesses dispositivos podem ser encontradas em [51].

Para exemplificar a coalescência de memória, o *kernel* apresentado no Algoritmo 6.1 copia as posições do vetor `a` para o vetor `b`, ambos recebidos como argumentos e alocados na memória global. Suponha que os vetores `a` e `b` possuem 32 posições inteiras de 4 bytes. A invocação desse *kernel* é apresentada no Algoritmo 6.2, onde 32 instâncias do *kernel* são criadas e a leitura do vetor `b` produz acessos a um mesmo segmento de memória, de acordo com o padrão apresentado na Figura 6.5, gerando apenas uma transação de 128 bytes.

Algoritmo 6.1 *Kernel* de cópia de um vetor para outro

```
1: __kernel void VectorCopy(__global int* a, __global int* b)
2: {
3:       int id = get_global_id(0);
4:       a[id] = b[id];
5: }
```

Algoritmo 6.2 Invocação de 32 instâncias do *kernel* de cópia de vetor do Algoritmo 6.1

```
1: ...
2: ckKernel = clCreateKernel (program, "VectorCopy", &errcode)
3: szGlobalWorkSize = 32;
4: szLocalWorkSize = 32;
5: ciErr = clEnqueueNDRangeKernel(cqCommandQueue, ckKernel, 1, NULL,
   &szGlobalWorkSize, &szLocalWorkSize, 0, NULL);
6: ...
```

De acordo com os Algoritmos 6.1 e 6.2, 32 *threads* serão invocadas e a leitura do vetor b produzirá acessos a um mesmo segmento de memória, de acordo com o padrão apresentado na Figura 6.5, gerando apenas uma transação de 128 bytes.

6.1.2 Acessos coalescidos à memória exclusiva

A memória exclusiva é uma seção de memória *off-chip* utilizada pelo compilador para guardar variáveis criadas dentro de um *kernel* que não couberam em registradores ou que consomem muitos registradores, como por exemplo, vetores. Quando uma variável é armazenada na memória exclusiva, ocorre o que é denominado *register spilling*.

Por estar mapeada na memória global do dispositivo, a memória exclusiva possui as mesmas restrições da memória global: pequena largura de banda, alta latência e um conjunto de requisitos necessários para que os acessos sejam coalescidos. A única diferença da memória exclusiva em relação à memória global está na sua organização. A memória exclusiva é organizada de tal maneira que *threads* consecutivas acessam endereços de memória consecutivos, como mostra a Figura 6.6. Ou seja, acessos a uma estrutura de dados armazenada na memória exclusiva serão coalescidos se as *threads* de um mesmo *warp* estiverem acessando o mesmo endereço relativo [51].

Por exemplo, no *kernel* apresentado no Algoritmo 6.3, cada instância do *kernel* possui um vetor `v` alocado na memória exclusiva. Supondo que 32 *work-items* são invocados, todos eles acessam as mesmas posições relativas do vetor `v`, fazendo com que as *threads* acessem posições consecutivas da memória global. Assim, os acessos caem em um mesmo segmento da memória global, permitindo a coalescência dos acessos.

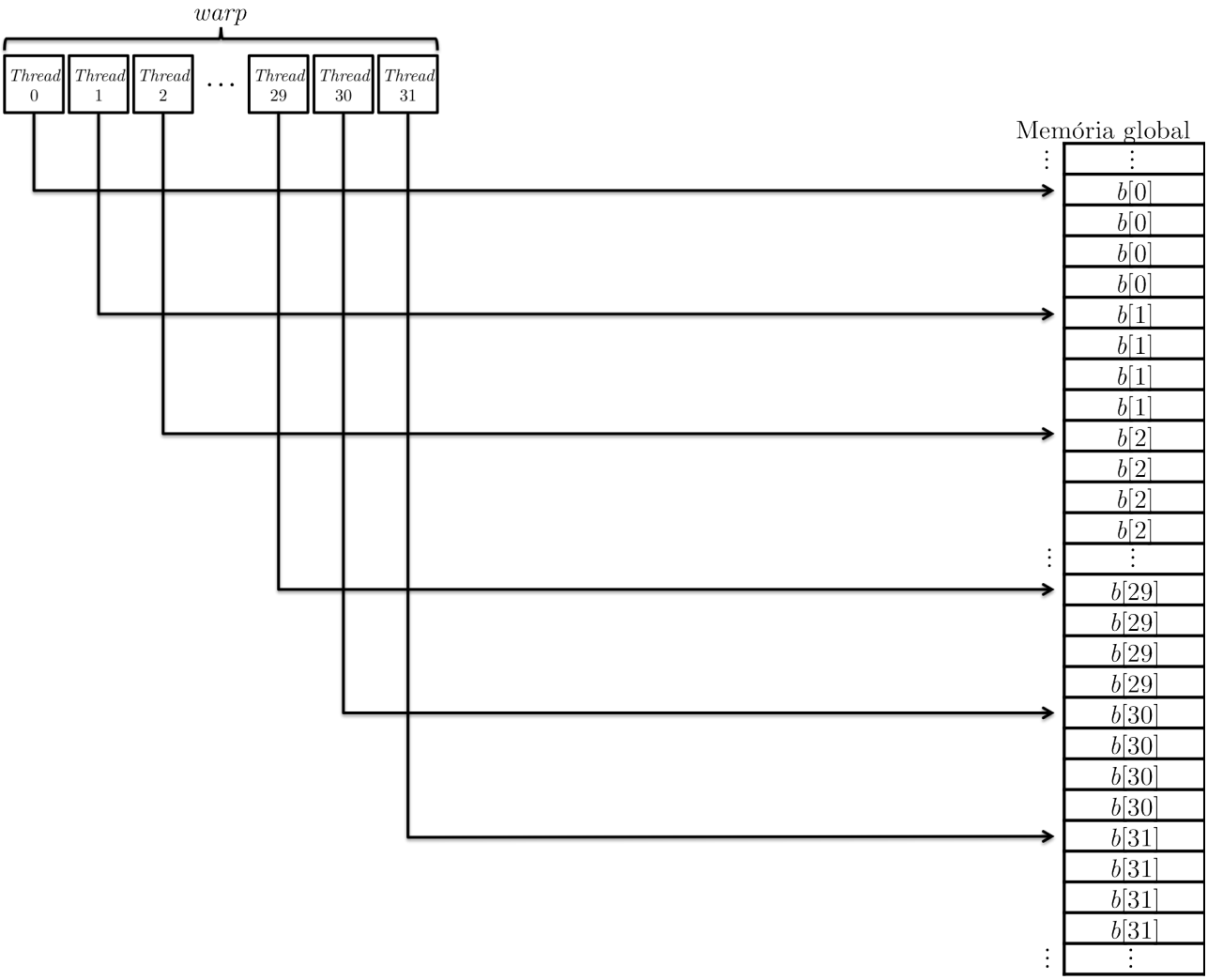


Figura 6.5: Acessos coalescidos à memória global realizados pelo Algoritmo 6.1

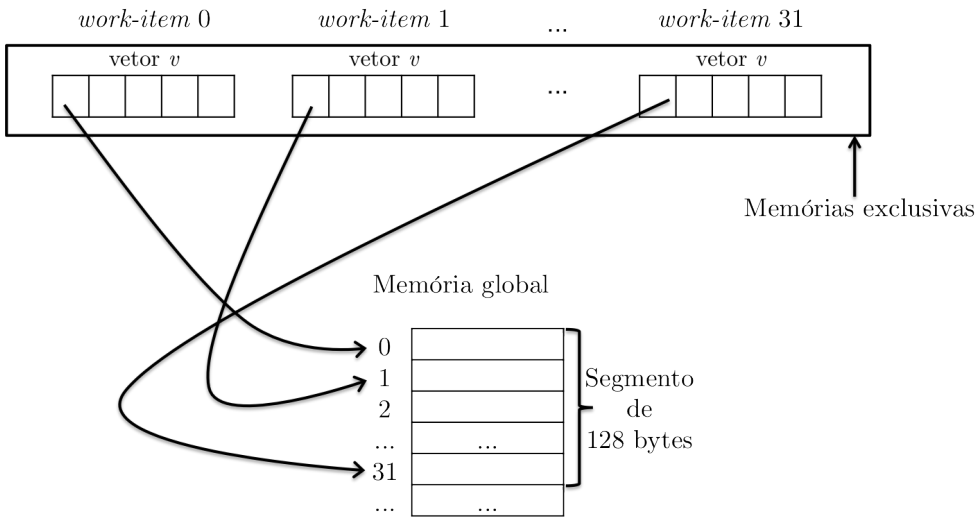


Figura 6.6: Mapeamento das memórias exclusivas na memória global e acessos coalescidos

Algoritmo 6.3 *Kernel* com acessos coalescidos às memórias exclusivas

```
1: __kernel void CoalescencePrivate()
2: {
3:       int id = get_global_id(0), i, v[32];
4:       for (i = 0 ; i < 32 ; i++)
5:            v[i] = 1;
6: }
```

6.1.3 Memória local

A memória local é uma pequena memória *on-chip* presente em cada um dos SMs. É uma memória extremamente rápida e, quando utilizada apropriadamente, pode possuir uma latência de acesso até 100 vezes menor que a memória global [49].

Essa memória é compartilhada entre os *work-items* de um mesmo *work-group*. Ou seja, um *work-item* x pertencente a um *work-group* w_1 tem acesso a uma memória local diferente de um *work-item* y pertencente ao *work-group* w_2.

A memória local é comumente usada para evitar acessos desnecessários à memória global [49]. Por exemplo, considere o problema da multiplicação $c = a \times b$ de duas matrizes de dimensão $N \times N$, representado na Figura 6.7.

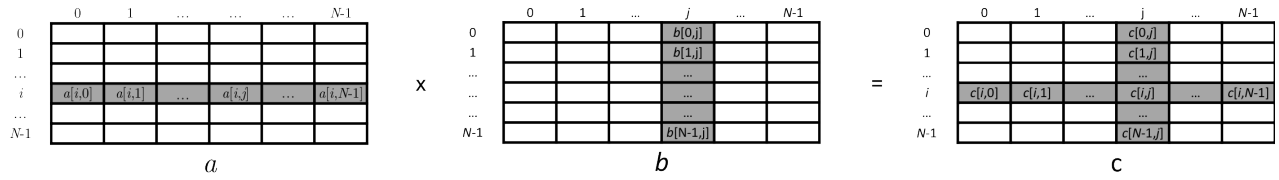


Figura 6.7: Multiplicação de duas matrizes a e b, resultando em c

Em GPUs, é muito comum dividir o cálculo da matriz resultante c em várias submatrizes e fazer com que cada *work-group* calcule uma submatriz diferente de c. Supondo que cada *work-group* seja responsável por calcular uma linha da matriz c, o elemento $c[i, j]$ correponde ao produto escalar da linha i de a com a coluna j de b, como mostra a Figura 6.7. Ou seja, todos os elementos pertencentes à mesma linha de c usam todos os elementos da mesma linha de a nos seus cálculos. Se a matriz a estiver armazenada na memória global, cada *work-item* de um *work-group* lerá todos os N elementos da linha de a. Ao todo, os *work-items* de um *work-group* lerão a linha toda de a N vezes.

Para evitar esses acessos desnecessários à memória global, a linha de a que é utilizada por um *work-group* pode ser trazida para a memória local no início da execução do *kernel*, e todos os acessos à a passam a ser feitos na memória local, cuja latência de acesso é menor em relação à memória global.

Quando não é possível coalescer os acessos à memória global, a memória local também pode ser utilizada. Os dados podem ser lidos da memória global de maneira coalescida e trazidos para a memória local. A partir daí, os *work-items* acessam os dados a partir da memória local, que não sofre a mesma degradação de desempenho sofrida pela memória global quando os acessos não são coalescidos [49].

A memória local está dividida em bancos de memória, de tal maneira que acessos de leitura ou escrita em endereços consecutivos estão mapeados em bancos de memória consecutivos [49]. A Figura 6.8 mostra um vetor a de 32 posições mapeado em uma memória local com 16 bancos.

Acessos em paralelo à diferentes bancos da memória local são atendidos ao mesmo tempo. Acessos a um mesmo banco são atendidos de forma serializada por banco. Ou seja, acessos a essa memória devem ser feitos de modo a evitar conflitos de banco.

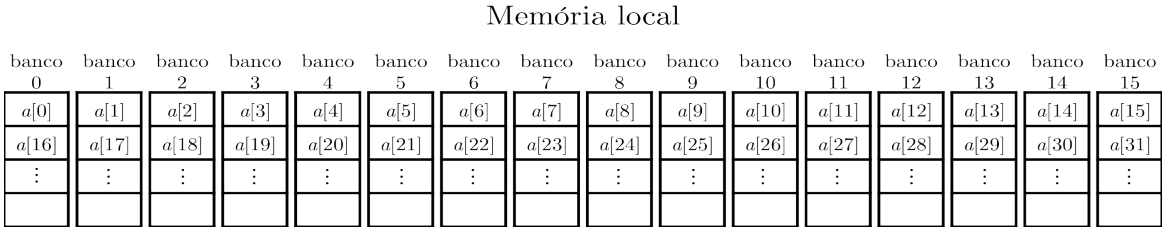


Figura 6.8: Vetor `a` de 32 posições mapeado em 16 bancos da memória local

6.1.4 Memória de constantes

A memória de constantes é uma região de 64 KB da memória global que só pode ser modificada pelo *host*. Uma leitura na memória de constantes é tão rápida quanto um acesso a um registrador, se todas as *threads* de um meio-*warp* estiverem acessando o mesmo endereço relativo [49].

O vetor `b` presente no Algoritmo 6.4 é um exemplo de uma estrutura que pode ser alocada na memória de constantes. Como as *threads* de um *warp* executam o *kernel* de forma síncrona, o comando `a[id] = a[id] + b[i]` é executado com a mesma posição `i` de `b` para todas essas *threads*. Sendo assim, o mesmo endereço relativo é acessado por todas as *threads* de um *warp*, tornando vantajosa a alocação de `b` na memória de constantes, dado que ele não é atualizado dentro do *kernel*.

Algoritmo 6.4 *Kernel* com uso da memória constante

```
1: __kernel void ConstantMemory(__global int* a, __constant int* b)
2: {
3:       int id = get_global_id(0), i;
4:       a[id] = 0;
5:       for (i = 0 ; i < 16 ; i++)
6:            a[id] = a[id] + b[i];
7: }
```

6.1.5 Largura de banda da memória global

A máxima utilização da largura de banda de memória global é um dos principais fatores que devem ser considerados para a otimização de um programa em GPU. Como este objetivo, existem duas importantes medidas: largura de banda teórica e largura de banda efetiva.

A largura de banda teórica LB_t representa a largura de banda total oferecida pela memória. LB_t é calculada pela Equação 11 (em GBytes/segundo), onde $freq_{mem}$ é a frequência do *clock* da memória global (em MHz) e L_{mem} é a largura da interface de memória (em bytes) [49]:

Equação 11.

$$LB_t = \frac{freq_{mem} \times 10^6 \times L_{mem} \times 2}{10^9}$$

A largura de banda efetiva LB_e indica a largura de banda realmente utilizada em uma execução, e é calculada pela Equação 12 (em GBytes/segundo), onde B_l é a quantidade de bytes lidos, B_e é a quantidade de bytes escritos e t_exec é o tempo total de execução em segundos [49]:

Equação 12.

$$LB_e = \frac{B_l + B_e}{10^9 \times t_{exec}}$$

É importante que LB_e esteja próxima de LB_t, indicando que o programa está desperdiçando o mínimo possível da largura de banda fornecida pela GPU.

Uma outra medida de desempenho, denominada *throughput*, é similar à LB_e mas com uma diferença: LB_e é calculada com base apenas nos bytes lidos e escritos pelo programa, já o *throughput* considera também os bytes que foram lidos ou escritos devido a um acesso não coalescido de memória [49].

6.2 Transferência de Dados entre *Host* e GPU

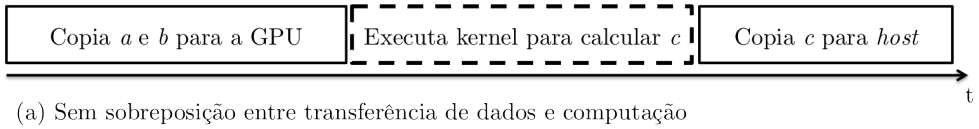
Um dos principais gargalos na implementação de problemas em GPU é a largura de banda para a realização de transferências de dados entre a memória do *host* e a memória da GPU. Por essa razão, alguns cuidados devem ser tomados para que este gargalo não comprometa o desempenho.

Para diminuir o impacto das transferências entre o *host* e a GPU, é aconselhável realizar o mínimo de transferências possível, transferir a maior quantidade possível de dados de uma só vez e realizar operações extras na GPU que evitem novas transferências de dados [49].

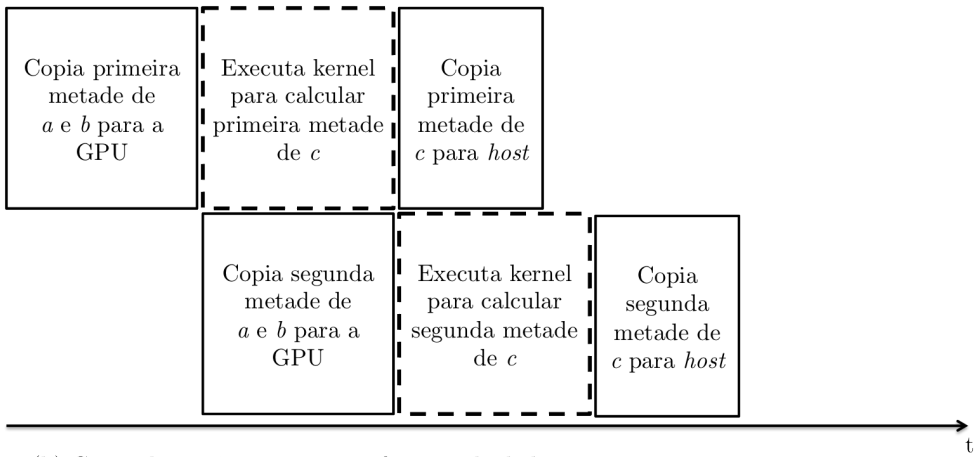
Em alguns casos, o tempo gasto com transferências de dados entre *host* e GPU é próximo ao tempo gasto com execuções de *kernel*. Em GPUs com capacidade de computação maior ou igual a 2.0, é possível amenizar os efeitos negativos dessas transferências sobrepondo-as com execuções de *kernel* (*overlapping*) [49].

Supondo um *kernel* responsável por somar dois vetores `a` e `b` e armazenar o resultado em `c`, inicialmente os valores de `a` e `b` são copiados do *host* para a GPU e, ao final da execução, o valor de `c` é copiado da GPU para o *host*.

Em um dispositivo sem capacidade de sobreposição, a transferência de `a` e `b` do *host* para a memória da GPU precisa ser finalizada para que a computação na GPU aconteça, e só ao término desta operação o vetor *c* é copiado da GPU para o *host*, como mostrado na Figura 6.9(a).



(a) Sem sobreposição entre transferência de dados e computação



(b) Com sobreposição entre transferência de dados e computação

Figura 6.9: Transferência de dados entre *host* e GPU sem e com sobreposição da computação na GPU [49]

Já em um dispositivo com capacidade de sobreposição, é possível transferir a primeira metade de *a* e *b* do *host* para a GPU, realizar a computação da primeira metade de *c* ao mesmo tempo que copia a segunda metade de *a* e *b*, e seguir sobrepondo transferências com execuções dos *kernels* até o final, como mostrado na Figura 6.9 (b).

Para que o *overlapping* otimize o tempo de execução de um *kernel*, alguns fatores devem ser considerados. Um dos principais fatores é o balanceamento entre as fases de cópia e execução, de tal maneira que, de fato, haja um paralelismo entre a cópia de dados e a execução de um *kernel*. Encontrar a proporção ideal de cópia de memória e execução é muitas vezes realizado através de um pré-processamento antes da invocação do *kernel*, passo este chamado de aquecimento da GPU [49].

6.3 Ocupação da GPU

Os recursos necessários para executar um *work-group* são alocados em um *Streaming Multiprocessor* (SM) no início de sua execução, e esta só acontecerá se todos os recursos por ele solicitados estiverem disponíveis no SM. Portanto, os *work-items* de um mesmo *work-group* executam no mesmo SM e, consequentemente, o mesmo acontece com os *warps* de um mesmo *work-group*.

Uma GPU é capaz de alocar uma determinada quantidade máxima de *warps* em cada SM, e essa quantidade depende da capacidade de computação do dispositivo. Porém, dependendo da demanda de recursos exigidos pelos *work-groups* (registradores e memória local), a quantidade efetiva de *warps* alocados pode ser inferior à capacidade máxima da GPU.

Definição 13. *A* ***ocupação*** *de uma GPU é a medida referente a quantidade de warps ativos em um SM dividida pela quantidade máxima possível de warps ativos em um SM [51].*

Ou seja, a ocupação indica se o programa em execução está utilizando ao máximo os recursos disponibilizados pela GPU.

A NVIDIA oferece uma planilha, mostrada na Figura 6.10, que simula a ocupação de uma GPU [54] com base nos seguintes parâmetros:

- Capacidade de computação do dispositivo, que permite descobrir a quantidade de registradores e memória local por SM;
- Quantidade de registradores alocada para cada *work-item*, que determinará os seguintes itens:
  - Quantidade de registradores necessários por *work-group*, que afeta o número de *work-groups* que podem estar ativos em um SM. Caso a quantidade de registradores necessária para executar um *work-group* seja maior que a quantidade de registradores disponibilizada por um SM, a execução do *kernel* falha;
  - Quantidade de registradores necessários por *warp*, que influencia na quantidade de *warps* ativos em um SM.
- Quantidade de memória local alocada por *work-group*. Existe uma memória local por *SM* e a quantidade de memória local utilizada por cada *work-group* influencia na quantidade de *work-groups* ativos em um *SM*.
- Quantidade de *work-items* (*threads*) por *work-group*, que influencia na quantidade total de registradores necessários para a execução de um *work-group*, o que pode afetar significativamente a ocupação da GPU.

Com esses parâmetros, é possível calcular um valor de ocupação teórico da GPU. A partir dessas informações, a planilha gera valores estatísticos e gráficos referente à ocupação da GPU.

A princípio, quanto maior a ocupação, melhor o desempenho do programa na GPU. Entretanto, a partir de um determinado valor, o aumento da ocupação não necessariamente implica melhoria de desempenho [51].

É importante que o tamanho dos *work-groups* e a quantidade de *work-groups* disponíveis para execução sejam suficientes para manter a GPU sempre ocupada. Embora esses valores devam ser

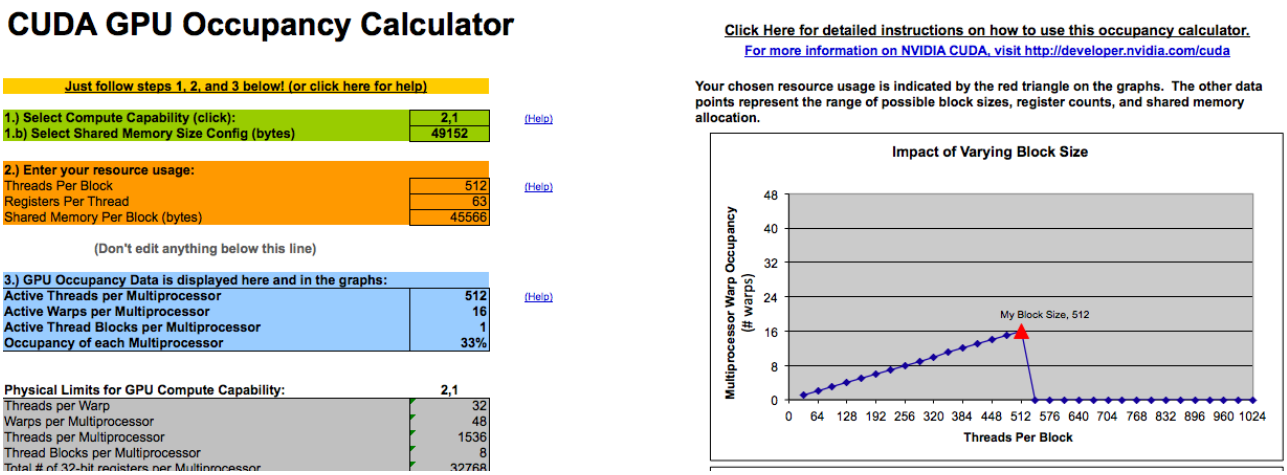


Figura 6.10: Planilha de simulação da ocupação da GPU fornecida pela NVIDIA [54]

determinados levando em conta o problema implementado e a GPU utilizada, existem algumas heurísticas que auxiliam nessa tarefa [49]:

- A quantidade de *work-groups* deve ser maior ou igual a quantidade de SMs disponíveis, pois dessa maneira cada SM terá pelo menos um *work-group* para executar;
- O tamanho dos *work-groups* deve ser múltiplo do tamanho do *warp*, evitando a existência de *warps* incompletos e facilitando a coalescência de memória;
- A escolha do tamanho dos *work-groups* entre 128 e 256 deve ser avaliada empiricamente.

6.4 Dependências de Dados entre Instruções

Para o *kernel* mostrado no Algoritmo 6.5, suponha que foi criado apenas um *work-group*, com 32 *work-items*.

Algoritmo 6.5 *Kernel* com dependências de dados entre instruções e divergência

```
1: __kernel void DependencyExample(__global int* a, __global int* b)
2: {
3:       int id = get_global_id(0);
4:       if(id < 16)
5:            a[id] = 10;
6:       a[id] = a[id] * 5;
7:       b[id] = a[id];
8: }
```

A linha 6 escreve em uma posição do vetor `a` que será utilizada na linha 7. Essa dependência de dados é denominada *read-after-write* e a instrução que lê o valor de `a[id]` na linha 7 precisa aguardar 24 ciclos do *clock* até que o resultado multiplicação feita na linha 6 esteja pronto, pois este é o tempo médio necessário para a realização dessa operação aritmética em GPUs NVIDIA [49].

Para evitar que essas dependências de dados causem perda de desempenho, é aconselhável que sempre existam *warps* disponíveis para serem executados, para que enquanto um *warp* aguarda por algum resultado proveniente de uma operação aritmética, outro *warp* disponível possa ser executado. Com uma ocupação razoável da GPU, a quantidade de *warps* disponíveis é suficiente para esconder os atrasos causados por dependências de dados entre instruções [49].

6.5 Fluxo de Controle da Execução e Divergências

Um *warp* é a menor unidade de execução na GPU, e corresponde a um conjunto de *threads* que executam de forma síncrona a mesma instrução utilizando o modelo SIMD de paralelismo. Entretanto, quando *threads* de um mesmo *warp* tomam diferentes caminhos de execução dentro de um *kernel*, criam-se as chamadas divergências, e esses caminhos são executados sequencialmente até que as *threads* juntem-se novamente em um mesmo caminho de execução.

Por exemplo, no Algoritmo 6.5, considerando os *work-items* 15 e 16 do único *warp* em execução. O *work-item* 15 executará a linha 5, enquanto o *work-item* 16 não a executará. Sendo assim, duas *threads* de um mesmo *warp* tomam caminhos de execução diferentes, gerando uma divergência. O *work-item* 16 aguarda o fim da execução do `if` para que os caminhos de execução juntem-se e a execução do *warp* possa continuar. Divergências reduzem o paralelismo na execução na GPU, por isso é recomendável que, sempre que possível, elas sejam evitadas, pois podem degradar o desempenho do *kernel* [49].

6.6 Opções de Compilação e Otimização de Instruções

Existem algumas opções de compilação que fazem com que o compilador para GPU gere código de forma que determinadas operações aritméticas consumam menos ciclos na execução: a opção *–cl-mad-enable* sinaliza para o compilador que operações de adição e multiplicação podem ser agrupadas em uma única instrução especial chamada FMAD e a opção *–cl-fast-relaxed-math* também permite que o compilador realize otimizações nas operações aritméticas. O uso dessas diretivas pode trazer ganho de desempenho, mas geralmente é acompanhado por uma perda de precisão nos cálculos [49].

As principais recomendações quanto à otimização de instruções são:

- Funções matemáticas nativas devem ser usadas sempre que possível;
- Operações de divisão ou módulo devem ser evitadas, pois são particularmente custosas. Ao invés delas, operações de deslocamento (*shift*) e operações lógicas (and/or) podem ser utilizadas;
- Conversões (*casts*) automáticas entre tipos devem ser evitadas.

6.7 Considerações Finais

A partir das otimizações descritas, é possível enumerar algumas recomendações que devem ser aplicadas na implementação de um programa a ser executado em GPU, com o objetivo de melhorar o seu desempenho:

- Coalescer acessos à memória global e à memória exclusiva. É importante coalescer os acessos sempre que possível, reduzindo o desperdício de largura de banda e a quantidade de transações necessárias para atender um conjunto de acessos;
- Usar a memória local sempre que possível;
- Usar a memória de constantes para valores somente-leitura cuja posição relativa acessada é igual para as *threads* de um *warp*;
- Otimizar o uso da largura de banda da memória global;
- Minimizar a transferência de dados entre *host* e GPU;
- Gerar um bom valor de ocupação da GPU para um *kernel* em execução, minimizando também o impacto das dependências de dados entre instruções;

- Reduzir ao máximo a quantidade de divergências na execução de um *kernel*;
- Quando possível, utilizar opções de compilação para otimização de instruções aritméticas.

Capítulo 7

Acelerador Básico do Algoritmo de Viterbi em GPU

Embora o algoritmo de Viterbi tenha complexidade de tempo polinomial, devido ao volume de dados de entrada (muitas sequências e sequências longas), o seu tempo de execução pode ser muito longo. Experimentos mostram que o algoritmo de Viterbi consome certa de 97% do tempo de execução das operações *hmmsearch* e *hmmpfam* do HMMer [39, 75], portanto, para reduzir o tempo de execução dessas operações, é necessário otimizar a implementação do algoritmo de Viterbi.

Neste trabalho desenvolvemos um acelerador em GPU para o algoritmo de Viterbi aplicado à análise de sequências biológicas. Este capítulo apresenta a versão inicial, isto é, sem otimizações, do acelerador, bem como alguns resultados preliminares.

7.1 Plataformas, Ferramentas, Bases de Dados e Metodologias Utilizadas

A GPU utilizada para executar as implementações desenvolvidas é uma NVIDIA GeForce GTX 460 [57], com 1 GB de memória DDR5 e 336 núcleos de processamento. Ela é conectada um computador *host* com processador AMD Athlon II X3 [1] de 3.2 GHz e 4 GB de memória RAM, com o sistema operacional Ubuntu 10.10 [7] através da interface padrão PCI-Express [63].

O *Toolkit* e SDK versão 3.2.16 [56] e o *driver* versão 260.19.26 fornecidos pela NVIDIA são utilizados, permitindo a execução de programas escritos em OpenCL. A versão 1.0 do OpenCL é utilizada pois é a versão oficial mais recente disponibilizada pela NVIDIA. Ainda assim, as versões 1.1 e 1.2 não oferecem vantagens significativas para as implementações desenvolvidas neste trabalho. O *Toolkit* também fornece a ferramenta *OpenCL Visual Profiler*, que retorna informações estatísticas sobre a execução e acesso às memórias referentes à execução de um *kernel* na GPU. Os dados fornecidos por esta ferramenta são aplicados na avaliação experimental de desempenho das implementações desenvolvidas.

As sequências biológicas utilizadas nos experimentos pertencem à base de dados de sequências de proteínas UniProtKB Swiss-Prot [73, 71], composta por sequências manualmente anotadas e revisadas. Um subconjunto de 70.000 sequências com tamanho mínimo 550 e tamanho máximo 1.500, extraídas aleatoriamente dessa base, é utilizado nos experimentos preliminares. Os experimentos finais são realizados com a base completa. A Tabela 7.1 mostra algumas informações sobre a base completa e sobre o subconjunto de sequências.

Os HMMs usados foram extraídos da base de dados de famílias de proteínas Pfam [65]. Essa base é composta por milhares de famílias e, para a realização dos experimentos, foram selecionadas as famílias que pertencem ao conjunto denominado *Top twenty*, pois correspondem às 20 famílias com

Tabela 7.1: Base de sequências UniProt Swiss-Prot

Medida		Base completa	Subconjunto extraído
Número total de sequências		530.264	70.000
Tamanho das sequências	mínimo	2	550
	médio	354	786
	máximo	32.513	1.500

maior número de sequências [82]. A Tabela 7.2 mostra as famílias utilizadas, indicando para cada família, o seu nome, sua identificação na base Pfam e a quantidade de nós do HMM que a representa.

Tabela 7.2: Famílias *Top twenty* de sequências da base Pfam

Família	Identificação	Comprimento do HMM (número de nós)
ABC_tran	PF00005	118
Acetyltransf_1	PF00583	83
adh_short	PF00106	167
BPD_transp_1	PF00528	185
COX1	PF00115	447
Cytochrom_B_C	PF00032	102
Cytochrom_B_N	PF00033	188
GP120	PF00516	488
HATPase_c	PF02518	111
Helicase_C	PF00271	78
HTH_1	PF00126	60
MFS_1	PF07690	354
Oxidored_q1	PF00361	270
Pkinase	PF00069	260
Response_reg	PF00072	112
RVP	PF00077	100
RVT_1	PF00078	214
RVT_thumb	PF06817	70
WD40	PF00400	39
zf-C2H2	PF00096	23
Comprimento	mínimo	23
	médio	173,45
	máximo	488

Os resultados obtidos neste trabalho são comparados à versão 2 do software HMMer, executada no mesmo computador *host*. O compilador utilizado no *host* é o GCC (*GNU Compiler Collection*) [26] versão 4.4.5, com as opções de compilação `-g` e `-O2`, que correspondem às opções padrão desta versão do HMMer. A versão 3 do HMMer tem ganho de desempenho em relação a versão 2, porém sofre uma perda de precisão nos resultados gerados [69]. Embora as implementações desenvolvidas neste trabalho visem a otimização da operação *hmmsearch* do HMMer2, elas também podem ser aplicadas no fluxo da operação *hmmsearch* do HMMer3, que utiliza o algoritmo de Viterbi como um dos seus passos.

7.2 Acelerador com Abordagem de Granularidade Grossa

Neste trabalho o algoritmo de Viterbi é implementado em GPU para a comparação base de sequências-*profile*, e recebe como entrada um conjunto de sequências $S = \{S_1, S_2, ..., S_N\}$ e um HMM H de comprimento Q e calcula o *score* do melhor alinhamento de cada sequência $S_k \in S$ com H. A implementação em GPU explora o paralelismo entre sequências, ou seja, existem N instâncias do algoritmo de Viterbi executando em paralelo, cada instância trabalhando com uma sequência $S_k \in S$ diferente.

Na programação para GPU utilizando modelos de programação, a GPU é vista como um dispositivo de computação que recebe um *kernel* e executa em paralelo diversas instâncias desse *kernel*, operando sobre diferentes dados. Em OpenCL, a quantidade de instâncias do *kernel* corresponde ao tamanho do espaço de execução definido.

A abordagem mais intuitiva para a implementação da comparação base de sequências-*profile* em GPU utilizando OpenCL é fazer com que cada *kernel* instanciado execute o algoritmo de Viterbi inteiro para uma sequência diferente e um HMM. A Figura 7.1 ilustra essa abordagem de granularidade grossa, mostrando o espaço de execução da implementação para um conjunto de $N = 64$ sequências. São criados 64 *work-items*, agrupados em dois *work-groups* com dimensão 32. Todos os *work-items* executam o algoritmo de Viterbi para o mesmo HMM H, mas cada *work-item* opera sobre uma sequência $S_k \in S$ diferente.

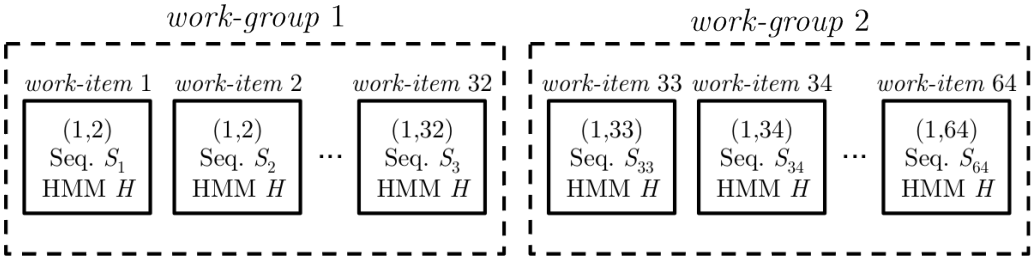


Figura 7.1: Espaço de execução do acelerador em GPU para comparação de 64 sequências $S_1, S_2, ..., S_{64}$ com o *profile* HMM H

Para que esse processamento seja possível, a GPU deve possuir memória suficiente para armazenar as estruturas de dados necessárias para a execução em paralelo das N instâncias do algoritmo de Viterbi. A Tabela 3.7 na página 30 mostra a quantidade de memória necessária para implementar a comparação base de sequências-*profile* explorando paralelismo entre sequências, para uma entrada de dados típica que consiste da base de sequências Swiss-Prot inteira e os HMMs das famílias *Top twenty*. Considera-se inicialmente que apenas as estruturas de dados que armazenam informações específicas do processamento de cada sequência (sequências e matrizes e vetores de *scores*) precisam ser replicadas para cada *work-item*. Ou seja, as informações sobre o HMM ($Pt_{regulares}$, $Pt_{especiais}$, Pe_M e Pe_I) não são, a princípio, replicadas, pois podem ser colocadas na memória global da GPU, que pode ser acessada por vários *work-items* executando em paralelo.

Esta demanda de até ≈ 1039 GB de memória inviabiliza a implementação do acelerador em diversas GPUs, como por exemplo a GPU utilizada para realização dos experimentos neste trabalho, que possui apenas 1 GB de memória.

Implementando o algoritmo de Viterbi como um filtro, a quantidade de memória necessária para sua execução diminui consideravelmente, pois apenas duas linhas das estruturas de dados de *scores* são armazenadas. A Tabela 7.3 mostra a quantidade de memória necessária para execução do filtro para comparação base de sequências-*profile*, explorando paralelismo entre sequências.

Considerando novamente a entrada de dados típica que consiste da base de sequências Swiss-Prot inteira e dos HMMs das famílias *Top twenty* da Pfam, a demanda média de memória para executar a operação descrita é 2,25 GB, como mostrado na Tabela 7.3, demanda esta que pode ser atendida pelas

Tabela 7.3: Estruturas de dados para filtro para comparação base de sequências-*profile*, explorando paralelismo entre sequências

Estrutura de dados		Tamanho (bytes)	Tamanho típico (KB)	
			Médio	Máximo
Sequências	$S = \{S_1, ..., S_N\}$	$\sum_{k=1}^{N} \lvert S_k \rvert$	185.400,98	185.400,98
Probabilidades de transição	$Pt_{regulares}$	$Q \times 9 \times 4$	6,08	17,16
	$Pt_{especiais}$	8×4	0,03	0,03
Probabilidades de emissão	Pe_M	$Q \times \lvert\Sigma\rvert \times 4$	13,52	38,13
	Pe_I	$Q \times \lvert\Sigma\rvert \times 4$	13,52	38,13
Scores	M, I, D	$N \times 3 \times 2 \times Q \times 4$	2.150.054,81	6.064.894,50
	N, E, J, B, C	$N \times 5 \times 2 \times 4$	20.713,44	20.713,44
Total			2.356.242,38	6.271.142,36

memórias de algumas GPUs atuais.

Entretanto, ao armazenar apenas parte das matrizes de *scores*, não é possível aproveitar os *scores* calculados pelo algoritmo de Viterbi para executar o mecanismo de *traceback*. Portanto, se o *score* obtido para uma sequência é considerado significativo, ela precisa ser processada novamente, desta vez utilizando o algoritmo de Viterbi completo, para que o *traceback* possa ser realizado.

A Tabela 7.4 mostra os resultados obtidos com a execução da operação *hmmsearch* do HMMer2 para a base de sequências Swiss-Prot completa e os HMMs das famílias *Top twenty*. A segunda e a terceira colunas apresentam, respectivamente, o número de *hits*, isto é, a quantidade de sequências classificadas como integrantes da família em questão (por obterem um *score* significativo quando comparadas ao HMM que representa a família), e a porcentagem de *hits* em relação ao total de sequências da base.

A maior porcentagem de *hits* obtida foi 0,91% e a média de *hits* é 0,29%. Portanto apenas para essa pequena porcentagem de sequências é necessária a execução do *traceback* posteriormente.

A grande demanda de memória para armazenamento das estruturas de *scores* combinada com a pequena ocorrência de *hits* motivam o desenvolvimento do acelerador em GPU utilizando a abordagem de um filtro. O fluxo de execução da solução desenvolvida neste trabalho para a comparação base de sequências-*profile* é mostrado na Figura 7.2.

O HMM e a base de sequências são submetidos ao acelerador em GPU, que executa as várias instâncias do algoritmo de Viterbi, armazenando apenas as linhas corrente e anterior das estruturas de *scores*, e retorna os *scores* ótimos, resultantes das comparações. Se o *score* de uma sequência S_k for relevante, isto é, se ocorrer um *hit* para S_k, o HMM e a sequência são submetidos a uma implementação sequencial do algoritmo de Viterbi (por exemplo, o software HMMer2), que é executada no *host* armazenando as estruturas de *scores* completas, para que o mecanismo de *traceback* seja posteriormente executado e retorne o alinhamento ótimo de S_k ao HMM.

Se o acelerador em GPU, em termos de tempo de execução, tiver um desempenho melhor que a implementação no *host*, o fluxo de execução proposto faz com que o uso de memória no acelerador seja reduzido e o desempenho total (incluindo a execução na GPU e no *host*) seja bom, dado que apenas poucas sequências são reprocessadas no *host*.

O Algoritmo 7.1 apresenta o algoritmo de Viterbi executado por cada *work-item* do acelerador básico desenvolvido para GPU neste trabalho. As diferenças desse algoritmo em relação ao Algoritmo 3.1 são:

- Os operadores de máximo são substituídos por sucessivas instruções condicionais;
- Apenas duas linhas, a atual e a anterior, das matrizes M, I e D e dos vetores B, C e J de *scores* são mantidas.

Tabela 7.4: *Hits* na operação *hmmsearch* do HMMer2 para base Swiss-Prot completa e famílias *Top twenty* da Pfam

Família	Número de *hits*	% de *hits*
ABC_tran	4830	0,91%
Acetyltransf_1	657	0,12%
adh_short	981	0,19%
BPD_transp_1	612	0,12%
COX1	239	0,05%
Cytochrom_B_C	1874	0,35%
Cytochrom_B_N	2099	0,40%
GP120	85	0,02%
HATPase_c	2475	0,47%
Helicase_C	3135	0,59%
HTH_1	435	0,08%
MFS_1	2103	0,40%
Oxidored_q1	1242	0,23%
Pkinase	3796	0,72%
Response_reg	900	0,17%
RVP	199	0,40%
RVT_1	612	0,12%
RVT_thumb	111	0,02%
WD40	2716	0,51%
zf-C2H2	2107	0,40%
Média	1560,40	0,29%

- O *score* $E[i]$ é calculado parcialmente a cada iteração do laço interno (que itera sobre os nós do HMM), nas linhas 31 a 33, sem necessidade do laço adicional utilizado na linha 19 do Algoritmo 3.1;

- Durante uma iteração do laço externo (que itera sobre os símbolos da sequência), apenas a posição atual do vetor E é utilizada. Por essa razão, ao invés de duas posições, apenas uma posição de memória por *work-item* é destinada para o armazenamento de E;

- No Algoritmo 3.1, o vetor de *scores* N é calculado na linha 18 através da relação de recorrência $N[i] = N[i-1] + Pt_{N,N}$, com a condição inicial $N[1] = Pt_{N,N}$. Essa recorrência pode ser resolvida através do método iterativo [12], da seguinte forma:

$$\begin{aligned} N[i] &= N[i-1] + Pt_{N,N} \\ &= N[i-2] + Pt_{N,N} + Pt_{N,N} = N[i-2] + 2 \times Pt_{N,N} \\ &= N[i-3] + Pt_{N,N} + 2 \times Pt_{N,N} = N[i-3] + 3 \times Pt_{N,N} \\ &\vdots \\ &= N[i-(i-1)] + (i-1) \times Pt_{N,N} = N[1] + (i-1) \times Pt_{N,N} = i \times Pt_{N,N} \end{aligned}$$

  Calculando $N[i]$ através da equação $N[i] = i \times Pt_{N,N}$ (linha 40 do Algoritmo 7.1), apenas o valor da posição atual do vetor é utilizado e nenhuma posição de memória por *work-item* é destinada para o seu armazenamento.

Com essas modificações, a quantidade de memória necessária para execução do acelerador em GPU proposto é a mesma mostrada na Tabela 7.3, com exceção dos vetores de *scores* N, E, J, B e C, que passam a consumir $N \times (3 \times 2 + 1) \times 4$ bytes, isto é, $N \times 3 \times 4$ bytes a menos.

Algoritmo 7.1 Algoritmo de Viterbi executado por cada *work-item* do acelerador básico em GPU

Entradas: HMM H com Q nós e probabilidades de emissão Pe e transição Pt
Sequência $S_k = s_1 s_2 ... s_{|S_k|}$
Saída: *Score* do melhor alinhamento de S com H

1: $B[0] = Pt_{N,B}$
2: $C[0] = J[0] = -\infty$
3: $i_{atual} = 1$
4: $i_{anterior} = 0$
5: **for** $i = 1$ to $|S|$ **do**
6: $M[i_{atual}, 0] = I[i_{atual}, 0] = D[i_{atual}, 0] = -\infty$
7: $E = -\infty$
8: **for** $j = 1$ to Q **do**
9: $M[0, j] = I[0, j] = D[0, j] = -\infty$
10: $aux = M[i_{anterior}, j-1] + Pt_{M_{j-1},M_j}$
11: **if** $I[i_{anterior}, j-1] + Pt_{I_{j-1},M_j} > aux$ **then**
12: $aux = I[i_{anterior}, j-1] + Pt_{I_{j-1},M_j}$
13: **end if**
14: **if** $D[i_{anterior}, j-1] + Pt_{D_{j-1},M_j} > aux$ **then**
15: $aux = D[i_{anterior}, j-1] + Pt_{D_{j-1},M_j}$
16: **end if**
17: **if** $B[i_{anterior}] + Pt_{B,M_j} > aux$ **then**
18: $aux = B[i_{anterior}] + Pt_{B,M_j}$
19: **end if**
20: $M[i_{atual}, j] = aux + Pe_{M_j}(s_i)$
21: $aux = M[i_{anterior}, j] + Pt_{M_j,I_j}$
22: **if** $I[i_{anterior}, j] + Pt_{I_j,I_j} > aux$ **then**
23: $aux = I[i_{anterior}, j] + Pt_{I_j,I_j}$
24: **end if**
25: $I[i_{atual}, j] = aux + Pe_{I_j}(s_i)$
26: $aux = M[i_{atual}, j-1] + Pt_{M_{j-1},D_j}$
27: **if** $D[i_{atual}, j-1] + Pt_{D_{j-1},D_j} > aux$ **then**
28: $aux = D[i_{atual}, j-1] + Pt_{D_{j-1},D_j}$
29: **end if**
30: $D[i_{atual}, j] = aux$
31: **if** $M[i_{atual}, j] + Pt_{M_j,E} > E$ **then**
32: $E = M[i_{atual}, j] + Pt_{M_j,E}$
33: **end if**
34: **end for**
35: $aux = J[i_{anterior}] + Pt_{J,J}$
36: **if** $E + Pt_{E,J} > aux$ **then**
37: $aux = E + Pt_{E,J}$
38: **end if**
39: $J[i_{atual}] = aux$
40: $aux = (i \times Pt_{N,N}) + Pt_{N,B}$
41: **if** $J[i_{atual}] + Pt_{J,B} > aux$ **then**
42: $aux = J[i_{atual}] + Pt_{J,B}$
43: **end if**
44: $B[i_{atual}] = aux$
45: $aux = C[i_{anterior}] + Pt_{C,C}$
46: **if** $E + Pt_{E,C} > aux$ **then**
47: $aux = E + Pt_{E,C}$
48: **end if**
49: $C[i_{atual}] = aux$
50: $i_{anterior} = i_{atual}$
51: $i_{atual} = 1 - i_{atual}$
52: **end for**
53: $score = C[i_{anterior}] + Pt_{C,T}$

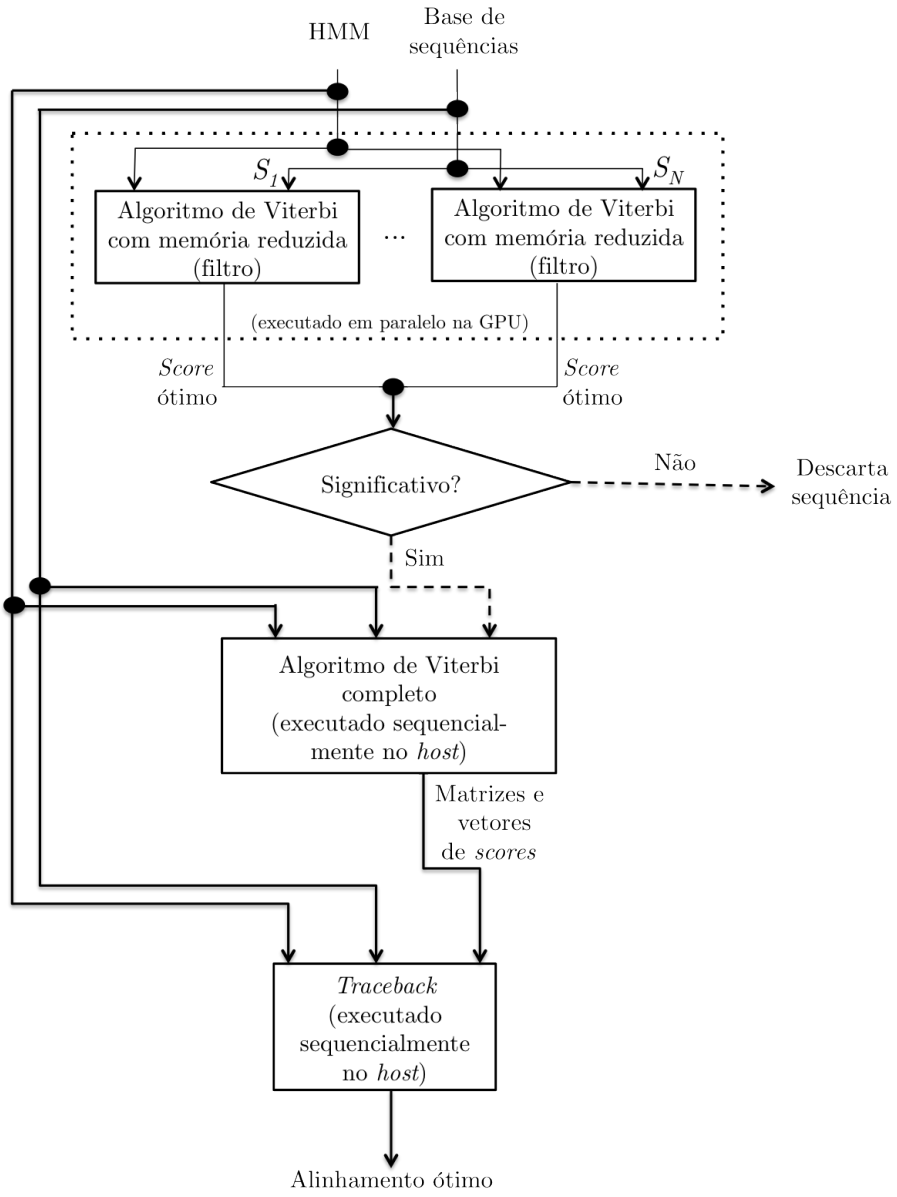


Figura 7.2: Fluxo de execução do acelerador em GPU como filtro

7.2.1 Organização das estruturas de dados

Em OpenCL, estruturas de dados com mais de uma dimensão devem ser manipuladas como vetores unidimensionais. Portanto, as matrizes de *scores* M, I e D, a matriz de transição $Pt_{regulares}$ e as matrizes de emissão Pe_M e Pe_I são convertidas para vetores unidimensionais.

Na implementação do acelerador básico, embora cada instância do algoritmo de Viterbi tenha seus próprios vetores de *scores*, há apenas um vetor para cada estrutura de dados do problema, armazenado na memória global e acessível a todos os *work-items*. Cada *work-item* possui informações que permitem identificar e acessar apenas as posições pertinentes à sua execução.

Para um conjunto de sequências $S = \{S_1, S_2, ..., S_N\}$ e um HMM H de comprimento Q, a Figura 7.3 representa como cada estrutura de dados é organizada na memória global da GPU que executa o acelerador básico. A Figura 7.3(a) representa um dos três vetores de *scores* globais M, I e D, nos quais cada $2 \times Q$ posições são associadas a um *work-item*, dado que o acelerador é utilizado como um filtro. A Figura 7.3(b) representa um dos vetores de *scores* globais B, C e J, nos quais cada duas posições são associadas a um *work-item*. A Figura 7.3(c) representa um dos vetores com as

probabilidades de emissão Pe_M e Pe_I, e a Figura 7.3(d) o vetor com as probabilidades de transições regulares $Pt_{regulares}$. O vetor com as probabilidades de transições especiais $Pt_{especiais}$ é alocado de acordo com a Figura 3.2. Cada posição de todos esses vetores ocupa quatro bytes.

As sequências também são armazenadas na memória global e a alocação dessa estrutura é mostrada na Figura 7.3(e). Existe apenas um vetor S que é acessível a todos os *work-items* e nele estão concatenadas as cadeias de símbolos das sequências de entrada, sendo que cada símbolo ocupa um byte.

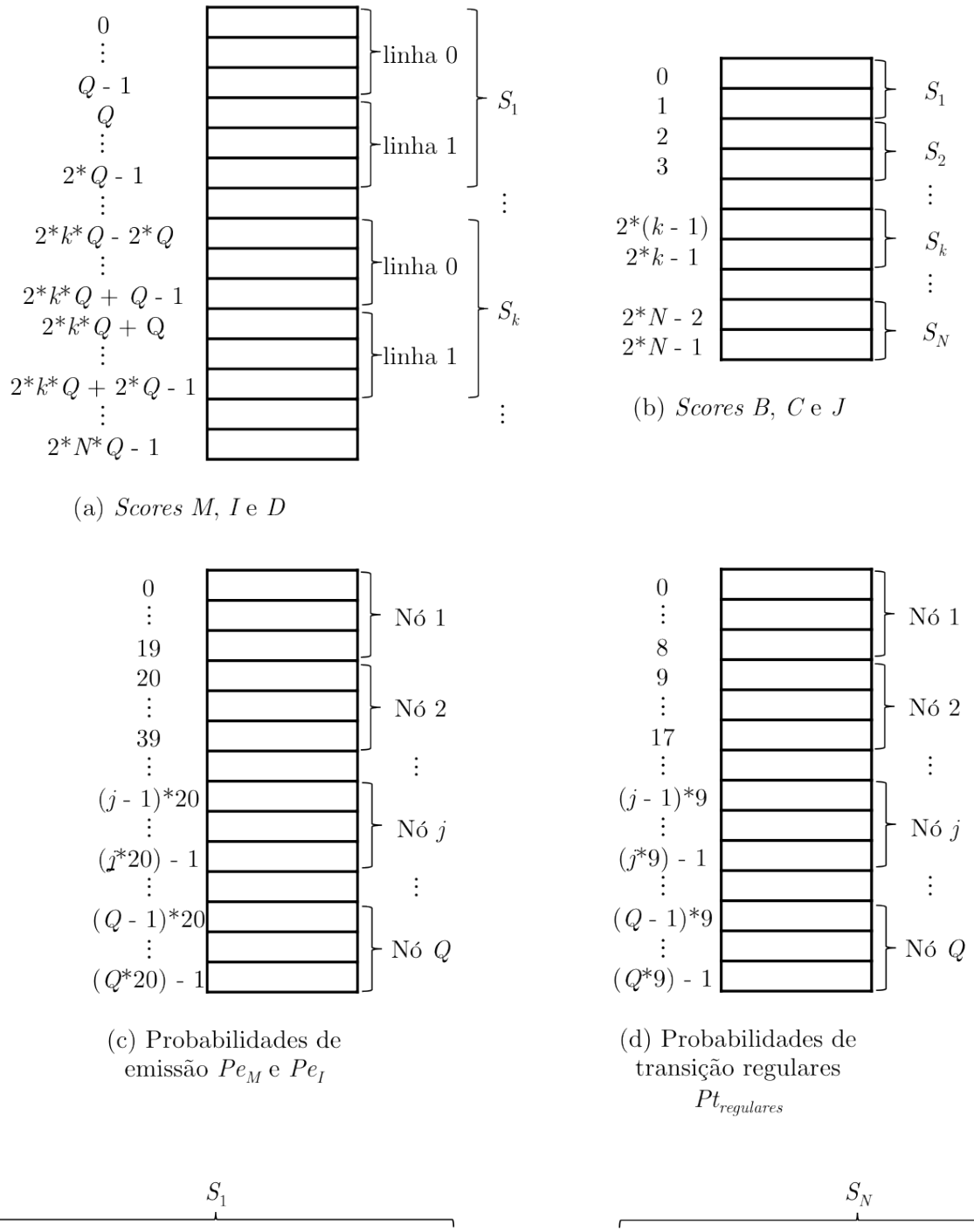
7.3 Resultados Preliminares

No acelerador básico desenvolvido para GPU, todas as estruturas de dados necessárias para a execução do algoritmo de Viterbi são armazenadas na memória global do dispositivo e nenhuma otimização é realizada.

Usando as famílias *Top twenty* da base Pfam e o subconjunto de 70.000 sequências da base UniProt Swiss-Prot, o acelerador básico é executado na GPU e o seu desempenho é comparado com o obtido pela ferramenta HMMer2, executada no *host*. A Tabela 7.5 apresenta o tempo de execução da comparação base de sequências-*profile* nos respectivos ambientes, para cada família. O tempo de execução em GPU inclui o tempo de transferência de dados entre o *host* e a GPU e o tempo de execução do *kernel* e o tempo de execução HMMer inclui apenas a execução do algoritmo de Viterbi. Experimentos preliminares mostram que há uma variação insignificante no tempo de execução do acelerador e do HMMer2 em sucessivas execuções para a mesma entrada de dados, por essa razão, os dados apresentados correspondem à última execução em cada plataforma. Note que, em várias execuções, a implementação em GPU não teve ganho de desempenho em relação à execução do HMMer2.

Tabela 7.5: Tempo de execução do acelerador básico em GPU (sem otimizações) e do HMMer2

Família	Tempo de execução (s)	
	Acelerador básico em GPU	HMMer2
ABC_tran	97,34	94,50
Acetyltransf_1	57,34	56,37
adh_short	91,27	91,86
BPD_transp_1	48,99	48,52
COX1	155,96	149,48
Cytochrom_B_C	154,34	154,49
Cytochrom_B_N	138,78	140,64
GP120	217,05	206,26
HATPase_c	373,26	356,28
Helicase_C	295,91	280,28
HTH_1	414,40	389,51
MFS_1	63,80	62,74
Oxidored_q1	221,92	225,25
Pkinase	31,59	74,10
Response_reg	17,80	19,50
RVP	82,33	80,22
RVT_1	84,19	82,09
RVT_thumb	68,15	67,21
WD40	91,88	94,53
zf-C2H2	176,21	170,46
Média	144,12	142,21



(a) *Scores M*, *I* e *D*

(b) *Scores B*, *C* e *J*

(c) Probabilidades de emissão Pe_M e Pe_I

(d) Probabilidades de transição regulares $Pt_{regulares}$

(e) Sequências

Figura 7.3: Organização das estruturas de dados na memória da GPU, para acelerador básico

A partir desse experimento conclui-se que implementar algoritmos em GPU de forma similar à implementação em processadores convencionais e sem aplicar otimizações específicas para GPUs pode não gerar ganho de desempenho em relação às implementações convencionais. Os próximos capítulos apresentam otimizações para GPUs que podem ser aplicadas ao algoritmo de Viterbi para a comparação base de sequências-*profile*, juntamente com os resultados obtidos.

Capítulo 8

Otimizações Aplicadas ao Acelerador em GPU

Este capítulo apresenta as otimizações aplicadas com o objetivo de melhorar o desempenho do acelerador em GPU desenvolvido para a comparação base de sequências-*profile*, bem como os resultados alcançados.

8.1 Escalonamento de Instruções

Existe uma dependência de dados verdadeira entre duas instruções I_1 e I_2 de um programa quando I_1 precede I_2 na ordem do programa e I_1 produz um resultado que é utilizado por I_2 [28]. Dependendo da arquitetura computacional utilizada e de quão próximas no programa I_1 e I_2 estão, essa dependência pode originar um conflito de dados RAW (*Read-After-Write*) entre as instruções, que por sua vez pode causar o atraso da execução de I_2, enquanto o resultado de I_1 não está disponível.

O procedimento de determinar a ordem de execução das instruções de um programa com o intuito de reduzir os atrasos causados por conflitos entre instruções, diminuindo o tempo total de execução do programa, é denominado *escalonamento de instruções*. O escalonamento estático de instruções é realizado pelos compiladores nas fases de otimização e geração de código [45].

Na GPU utilizada, a latência das instruções aritméticas é 24 ciclos, ou seja, se uma instrução I_1 de um *warp* W_1 realiza uma operação aritmética cujo resultado é utilizado pela próxima instrução I_2 de W_1, I_2 precisa aguardar 24 ciclos para iniciar sua execução. Para que a GPU não fique ociosa durante esses 24 ciclos, se houver um *warp* W_2 disponível para execução, ele será escalonado e executará enquanto a instrução aritmética de W_1 é realizada. Neste caso, ocorrem trocas de contexto entre os *warps* W_1 e W_2.

De acordo com o manual de programação OpenCL da NVIDIA [49], os conflitos entre instruções não causam perdas de desempenho contanto que existam *warps* disponíveis para executar na GPU enquanto outros *warps* executam operações aritméticas. Para dispositivos com capacidade de computação 1.2 ou superior, isso acontece quando a ocupação da GPU é maior ou igual a 25%.

Utilizando a planilha disponibilizada pela NVIDIA para cálculo da ocupação da GPU e fornecendo as informações de que o acelerador básico desenvolvido possui 256 *work-items* por *work-group*, 37 registradores por *work-item* e não utiliza a memória compartilhada/local, obtém-se o resultado de que o acelerador gera uma ocupação da GPU de 50%, como mostra a Figura 8.1. Com esse valor de ocupação, a princípio é possivel estimar que os conflitos entre instruções não causam perda de desempenho ao acelerador.

Mesmo com uma boa ocupação da GPU, um escalonamento de instruções, no nível da linguagem de programação alto nível, é realizado a partir do algoritmo de Viterbi apresentado no Algoritmo 7.1,

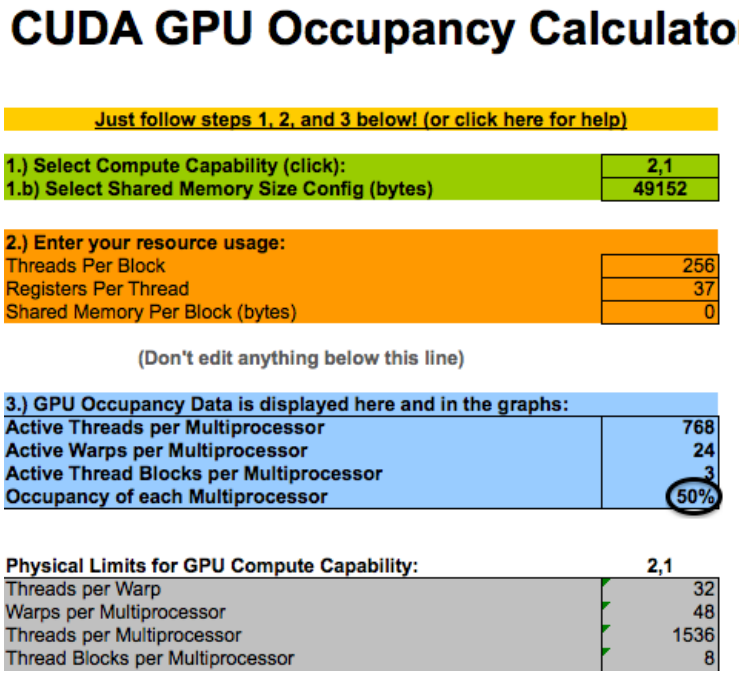


Figura 8.1: Ocupação da GPU pelo acelerador básico

com o intuito de aumentar a distância entre instruções com dependências de dados verdadeiras. O Algoritmo 8.1 mostra o algoritmo de Viterbi, executado por cada *work-item* do acelerador em GPU, resultante do escalonamento de instruções. As setas indicam exemplos de dependências de dados verdadeiras entre as instruções do programa.

No Algoritmo 8.1 são utilizadas várias variáveis auxiliares, aux_M, aux_I, aux_D, etc, ao invés da única variável *aux* do Algoritmo 7.1. Esse procedimento corresponde, no nível da linguagem de programação alto nível, à técnica de *renomeação de registradores*, uma vez que variáveis são em geral mapeadas em registradores pelo compilador [45]. O aumento na quantidade de variáveis aumenta o número de registradores necessários para executar essa versão do acelerador, que passa de 37 para 39 registradores por *work-item*.

A Tabela 8.1 mostra o tempo de execução do acelerador em GPU com escalonamento de instruções para o conjunto de 70.000 sequências da base Swiss-Prot utilizando os HMMs *Top Twenty* e compara com o tempo da execução do acelerador básico, sem escalonamento. Em todas as execuções houve ganho de desempenho do acelerador com escalonamento em relação ao acelerador básico. Concluímos assim que, mesmo quando há uma boa ocupação da GPU, afastar instruções com conflitos RAW pode trazer ganho de desempenho. Um compilador pode produzir um resultado equivalente, através do escalonamento estático de instruções, da análise de variáveis vivas e da alocação de registradores [45], no entanto o experimento mostra que o código gerado pelo compilador para GPU não é totalmente otimizado neste aspecto.

8.2 *Loop Unrolling*

A técnica de *loop unrolling* consiste em substituir o corpo de um laço por várias cópias do mesmo corpo, ajustando suas variáveis e as condições de parada do laço [45]. Para aplicar o *loop unrolling*, inicialmente é necessário decidir quantas cópias do corpo do laço haverá em cada iteração do novo laço. A quantidade de cópias criadas é denominada fator (*unroll factor*) [45].

Uma das vantagens do *loop unrolling* é que, havendo menos iterações no novo laço, reduz-se o *overhead* do laço, isto é, o número de instruções executadas para realizar o fluxo de controle do laço

Algoritmo 8.1 Algoritmo de Viterbi com escalonamento para reduzir conflitos entre instruções

```
1: B[0] = Pt_{N,B}
2: C[0] = J[0] = −∞
3: i_atual = 1
4: i_anterior = 0
5: for i = 1 to |S| do
6:     M[i_atual, 0] = I[i_atual, 0] = D[i_atual, 0] = −∞
7:     E = −∞
8:     for j = 1 to Q do
9:         M[0, j] = I[0, j] = D[0, j] = −∞
10:        aux_M = M[i_anterior, j − 1] + Pt_{M_{j−1},M_j}
11:        aux_I = M[i_anterior, j] + Pt_{M_j,I_j}
12:        aux_D = M[i_atual, j − 1] + Pt_{M_{j−1},D_j}
13:        if I[i_anterior, j − 1] + Pt_{I_{j−1},M_j} > aux_M then
14:            aux_M = I[i_anterior, j − 1] + Pt_{I_{j−1},M_j}
15:        end if
16:        if I[i_anterior, j] + Pt_{I_j,I_j} > aux_I then
17:            aux_I = I[i_anterior, j] + Pt_{I_j,I_j}
18:        end if
19:        if D[i_atual, j − 1] + Pt_{D_{j−1},D_j} > aux_D then
20:            aux_D = D[i_atual, j − 1] + Pt_{D_{j−1},D_j}
21:        end if
22:        if D[i_anterior, j − 1] + Pt_{D_{j−1},M_j} > aux_M then
23:            aux_M = D[i_anterior, j − 1] + Pt_{D_{j−1},M_j}
24:        end if
25:        I[i_atual, j] = aux_I + Pe_{I_j}(s_i)
26:        D[i_atual, j] = aux_D
27:        if B[i_anterior] + Pt_{B,M_j} > aux_M then
28:            aux_M = B[i_anterior] + Pt_{B,M_j}
29:        end if
30:        M[i_atual, j] = aux_M + Pe_{M_j}(s_i)
31:        if M[i_atual, j] + Pt_{M_j,E} > E then
32:            E = M[i_atual, j] + Pt_{M_j,E}
33:        end if
34:    end for
35:    aux_J = J[i_anterior] + Pt_{J,J}
36:    aux_B = (i × Pt_{N,N}) + Pt_{N,B}
37:    aux_C = C[i_anterior] + Pt_{C,C}
38:    if E + Pt_{E,J} > aux_J then
39:        aux_J = E + Pt_{E,J}
40:    end if
41:    if aux_J + Pt_{J,B} > aux_B then
42:        aux_B = aux_J + Pt_{J,B}
43:    end if
44:    if E + Pt_{E,C} > aux_C then
45:        aux_C = E + Pt_{E,C}
46:    end if
47:    J[i_atual] = aux_J
48:    B[i_atual] = aux_B
49:    C[i_atual] = aux_C
50:    i_anterior = i_atual
51:    i_atual = 1 − i_atual
52: end for
53: score = C[i_anterior] + Pt_{C,T}
```

Tabela 8.1: Tempo de execução do acelerador básico e do acelerador com escalonamento de instruções

Família	Tempo de execução (s)	
	Acelerador básico	Acelerador com escalonamento
ABC_tran	97,34	88,91
Acetyltransf_1	68,15	62,74
adh_short	138,78	126,42
BPD_transp_1	154,34	141,90
COX1	373,26	341,82
Cytochrom_B_C	84,19	77,90
Cytochrom_B_N	155,96	142,78
GP120	414,40	377,01
HATPase_c	91,27	83,47
Helicase_C	63,80	58,50
HTH_1	48,99	44,94
MFS_1	295,91	271,37
Oxidored_q1	221,92	202,96
Pkinase	217,05	200,03
Response_reg	91,88	84,36
RVP	82,33	75,4
RVT_1	176,21	161,39
RVT_thumb	57,34	52,89
WD40	31,59	28,90
zf-C2H2	17,80	15,95
Média	144,13	131,98

fica proporcionalmente menor em relação as demais instruções do laço. Por exemplo, com fator 2, o número de vezes que a condição de parada do laço é testada é reduzido pela metade.

Outra vantagem é que, replicando o corpo do laço, é possível realizar um melhor escalonamento de instruções, pois instruções de diferentes cópias do corpo podem ser intercaladas, afastando instruções com dependências de dados entre si. Dessa forma, o *loop unrolling* expõe o paralelismo entre as instruções [28], isto é, permite que mais instruções independentes entre si sejam encontradas.

Para avaliar o possível ganho de desempenho proporcionado pelo *loop unrolling* em GPU, a técnica é aplicada no laço mais interno do algoritmo de Viterbi (que itera sobre os nós do HMM), a partir do Algoritmo 7.1 do acelerador básico. São criadas três novas versões do acelerador em GPU, com fator 2, 4 e 8, respectivamente. Os Algoritmos 8.2 e 8.3 mostram o algoritmo de Viterbi, executado por cada *work-item* do acelerador em GPU, resultante da aplicação do *loop unrolling* com fator 2.

As linhas 11 a 35 formam a primeira cópia do corpo do laço enquanto as linhas 37 a 61 formam a segunda cópia. O laço mais interno tem o passo com incremento 2 e uma nova condição de parada controlada pela variável *limite*, que indica quantas vezes esse laço pode ser executado por completo, considerando o fator do *loop unrolling*. Se o número de nós do HMM não for divisível pelo fator, as linhas 65 a 89 realizam o restante das iterações originais em um laço com passo 1 e sem replicações. As versões do algoritmo com fator 4 e a 8 são análogas ao Algoritmos 8.2 e 8.3, com a diferença de que o laço da linha 9 contém mais replicações do corpo do laço original.

Embora o *loop unrolling* reduza o *overhead* do laço, o paralelismo entre as instruções exposto pela técnica deve ser explorado para reduzir os conflitos RAW entre as instruções. Isso é obtido combinando o *loop unrolling* com o escalonamento de instruções. Os Algoritmos 8.4 e 8.5 mostram o algoritmo de Viterbi executado por cada *work-item* do acelerador em GPU, resultante da aplicação do *loop unrolling* com fator 2 e do escalonamento de instruções. Esse algoritmo é criado a partir

Algoritmo 8.2 Algoritmo de Viterbi com *loop unrolling* com fator 2

1: $B[0] = Pt_{N,B}$
2: $C[0] = J[0] = -\infty$
3: $i_{atual} = 1$
4: $i_{anterior} = 0$
5: $limite = (Q\ div\ fator) \times fator$
6: **for** $i = 1$ to $|S|$ **do**
7: $M[i_{atual}, 0] = I[i_{atual}, 0] = D[i_{atual}, 0] = -\infty$
8: $E = -\infty$
9: **for** $j = 1$ to $limite$ **step** 2 **do**
10: *//Primeira cópia do corpo do laço*
11: $M[0, j] = I[0, j] = D[0, j] = -\infty$
12: $aux = M[i_{anterior}, j-1] + Pt_{M_{j-1},M_j}$
13: **if** $I[i_{anterior}, j-1] + Pt_{I_{j-1},M_j} > aux$ **then**
14: $aux = I[i_{anterior}, j-1] + Pt_{I_{j-1},M_j}$
15: **end if**
16: **if** $D[i_{anterior}, j-1] + Pt_{D_{j-1},M_j} > aux$ **then**
17: $aux = D[i_{anterior}, j-1] + Pt_{D_{j-1},M_j}$
18: **end if**
19: **if** $B[i_{anterior}] + Pt_{B,M_j} > aux$ **then**
20: $aux = B[i_{anterior}] + Pt_{B,M_j}$
21: **end if**
22: $M[i_{atual}, j] = aux + Pe_{M_j}(s_i)$
23: $aux = M[i_{anterior}, j] + Pt_{M_j,I_j}$
24: **if** $I[i_{anterior}, j] + Pt_{I_j,I_j} > aux$ **then**
25: $aux = I[i_{anterior}, j] + Pt_{I_j,I_j}$
26: **end if**
27: $I[i_{atual}, j] = aux + Pe_{I_j}(s_i)$
28: $aux = M[i_{atual}, j-1] + Pt_{M_{j-1},D_j}$
29: **if** $D[i_{atual}, j-1] + Pt_{D_{j-1},D_j} > aux$ **then**
30: $aux = D[i_{atual}, j-1] + Pt_{D_{j-1},D_j}$
31: **end if**
32: $D[i_{atual}, j] = aux$
33: **if** $M[i_{atual}, j] + Pt_{M_j,E} > E$ **then**
34: $E = M[i_{atual}, j] + Pt_{M_j,E}$
35: **end if**
36: *//Segunda cópia do corpo do laço*
37: $M[0, j+1] = I[0, j+1] = D[0, j+1] = -\infty$
38: $aux = M[i_{anterior}, j] + Pt_{M_j,M_{j+1}}$
39: **if** $I[i_{anterior}, j] + Pt_{I_j,M_{j+1}} > aux$ **then**
40: $aux = I[i_{anterior}, j] + Pt_{I_j,M_{j+1}}$
41: **end if**
42: **if** $D[i_{anterior}, j] + Pt_{D_j,M_{j+1}} > aux$ **then**
43: $aux = D[i_{anterior}, j] + Pt_{D_j,M_{j+1}}$
44: **end if**
45: **if** $B[i_{anterior}] + Pt_{B,M_{j+1}} > aux$ **then**
46: $aux = B[i_{anterior}] + Pt_{B,M_{j+1}}$
47: **end if**
48: $M[i_{atual}, j+1] = aux + Pe_{M_{j+1}}(s_i)$
49: $aux = M[i_{anterior}, j+1] + Pt_{M_{j+1},I_{j+1}}$
50: **if** $I[i_{anterior}, j+1] + Pt_{I_{j+1},I_{j+1}} > aux$ **then**
51: $aux = I[i_{anterior}, j+1] + Pt_{I_{j+1},I_{j+1}}$
52: **end if**
53: $I[i_{atual}, j+1] = aux + Pe_{I_{j+1}}(s_i)$

Algoritmo 8.3 Algoritmo de Viterbi com *loop unrolling* com fator 2 (continuação)

54: $aux = M[i_{atual}, j] + Pt_{M_j, D_{j+1}}$
55: **if** $D[i_{atual}, j] + Pt_{D_j, D_{j+1}} > aux$ **then**
56: $aux = D[i_{atual}, j] + Pt_{D_j, D_{j+1}}$
57: **end if**
58: $D[i_{atual}, j+1] = aux$
59: **if** $M[i_{atual}, j+1] + Pt_{M_{j+1}, E} > E$ **then**
60: $E = M[i_{atual}, j+1] + Pt_{M_{j+1}, E}$
61: **end if**
62: **end for**
63: *//Laço complementar para quando Q não for divisível pelo fator*
64: **for** $j = limite + 1$ to Q **do**
65: $M[0, j] = I[0, j] = D[0, j] = -\infty$
66: $aux = M[i_{anterior}, j-1] + Pt_{M_{j-1}, M_j}$
67: **if** $I[i_{anterior}, j-1] + Pt_{I_{j-1}, M_j} > aux$ **then**
68: $aux = I[i_{anterior}, j-1] + Pt_{I_{j-1}, M_j}$
69: **end if**
70: **if** $D[i_{anterior}, j-1] + Pt_{D_{j-1}, M_j} > aux$ **then**
71: $aux = D[i_{anterior}, j-1] + Pt_{D_{j-1}, M_j}$
72: **end if**
73: **if** $B[i_{anterior}] + Pt_{B, M_j} > aux$ **then**
74: $aux = B[i_{anterior}] + Pt_{B, M_j}$
75: **end if**
76: $M[i_{atual}, j] = aux + Pe_{M_j}(s_i)$
77: $aux = M[i_{anterior}, j] + Pt_{M_j, I_j}$
78: **if** $I[i_{anterior}, j] + Pt_{I_j, I_j} > aux$ **then**
79: $aux = I[i_{anterior}, j] + Pt_{I_j, I_j}$
80: **end if**
81: $I[i_{atual}, j] = aux + Pe_{I_j}(s_i)$
82: $aux = M[i_{atual}, j-1] + Pt_{M_{j-1}, D_j}$
83: **if** $D[i_{atual}, j-1] + Pt_{D_{j-1}, D_j} > aux$ **then**
84: $aux = D[i_{atual}, j-1] + Pt_{D_{j-1}, D_j}$
85: **end if**
86: $D[i_{atual}, j] = aux$
87: **if** $M[i_{atual}, j] + Pt_{M_j, E} > E$ **then**
88: $E = M[i_{atual}, j] + Pt_{M_j, E}$
89: **end if**
90: **end for**
91: $aux = J[i_{anterior}] + Pt_{J,J}$
92: **if** $E + Pt_{E,J} > aux$ **then**
93: $aux = E + Pt_{E,J}$
94: **end if**
95: $J[i_{atual}] = aux$
96: $aux = (i \times Pt_{N,N}) + Pt_{N,B}$
97: **if** $J[i_{atual}] + Pt_{J,B} > aux$ **then**
98: $aux = J[i_{atual}] + Pt_{J,B}$
99: **end if**
100: $B[i_{atual}] = aux$
101: $aux = C[i_{anterior}] + Pt_{C,C}$
102: **if** $E + Pt_{E,C} > aux$ **then**
103: $aux = E + Pt_{E,C}$
104: **end if**
105: $C[i_{atual}] = aux$
106: $i_{anterior} = i_{atual}$
107: $i_{atual} = 1 - i_{atual}$
108: **end for**
109: $score = C[i_{anterior}] + Pt_{C,T}$

Tabela 8.2: Tempo de execução dos aceleradores básico e com *loop unrolling*, sem e com escalonamento de instruções

Família	Tempo de execução (s)							
	Sem escalonamento				Com escalonamento			
	Sem *loop unrolling*	Com *loop unrolling*			Sem *loop unrolling*	Com *loop unrolling*		
		Fator 2	Fator 4	Fator 8		Fator 2	Fator 4	Fator 8
ABC_tran	97,34	96,53	96,48	94,32	88,91	53,67	39,48	32,87
Acetyltransf_1	68,15	68,14	68,33	66,47	62,74	38,37	28,32	22,88
adh_short	138,78	137,86	138,08	134,82	126,42	78,18	56,43	46,40
BPD_transp_1	154,34	153,70	153,98	150,59	141,90	86,65	62,60	49,42
COX1	373,26	370,63	371,41	361,60	341,82	208,41	150,33	119,73
Cytochrom_B_C	84,19	84,05	84,07	82,12	77,90	46,66	34,28	28,99
Cytochrom_B_N	155,96	155,77	156,01	152,02	142,78	86,92	63,15	51,61
GP120	414,40	410,90	411,48	399,56	377,01	232,36	168,13	133,21
HATPase_c	91,27	91,04	90,94	88,66	83,47	50,88	36,86	31,08
Helicase_C	63,80	63,87	63,59	62,15	58,50	35,07	25,79	22,12
HTH_1	48,99	48,75	49,05	47,56	44,94	26,85	19,18	16,59
MFS_1	295,91	293,53	294,97	286,37	271,37	164,52	118,72	94,41
Oxidored_q1	221,92	220,34	221,26	215,83	202,96	122,86	88,62	71,71
Pkinase	217,05	216,79	218,85	212,66	200,03	122,70	89,10	71,95
Response_reg	91,88	91,82	91,78	89,42	84,36	50,81	36,34	28,80
RVP	82,33	82,46	81,93	79,97	75,47	45,81	32,93	27,47
RVT_1	176,21	175,05	175,57	170,86	161,39	97,50	70,77	58,09
RVT_thumb	57,34	57,18	57,50	55,81	52,89	31,63	23,44	20,09
WD40	31,59	31,52	31,49	30,62	28,90	17,43	13,11	12,16
zf-C2H2	17,80	17,72	17,76	16,93	15,95	9,81	7,63	7,70
Média	144,13	143,38	143,73	139,92	131,99	80,35	58,26	47,36

dos Algoritmos 8.2 e 8.3, realizando-se o escalonamento com o intuito de afastar as instruções com dependências de dados verdadeiras entre si. Diversas variáveis auxiliares são utilizadas, de forma que a quantidade de variáveis do algoritmo escalonado é maior que do algoritmo sem escalonamento, o que acarreta o uso de um maior número de registradores.

A Tabela 8.2 mostra os resultados da execução dos aceleradores básico e com *loop unrolling*, sem e com escalonamento de instruções, para o conjunto de 70.000 sequências da base Swiss-Prot e os HMMs *Top Twenty*. Dentre os aceleradores com *loop unrolling* e sem escalonamento de instruções, o que obtém melhor desempenho é o acelerador com fator 8. No entanto, o ganho de desempenho em relação ao acelerador básico é muito pequeno. Em algumas execuções dos aceleradores com fator 2 e 4 houve inclusive uma pequena perda de desempenho em relação ao acelerador básico. Os aceleradores com *loop unrolling* e escalonamento de instruções apresentam melhoras significativas de desempenho, a medida que o fator aumenta, demonstrando que o *loop unrolling* tem um bom efeito no desempenho quando acompanhado do escalonamento das instruções.

O ganho de desempenho da versão escalonada com fator 8 em relação à versão não escalonada com o mesmo fator é maior que o ganho de desempenho da versão escalonada com fator 4 em relação à versão não escalonada com o mesmo fator, que por sua vez também é maior que o ganho de desempenho da versão escalonada com fator 2 em relação à versão não escalonada com o memso fator. Ou seja, quanto maior o fator do *loop unrolling*, mais importante é realizar o escalonamento de instruções.

A versão escalonada com fator 8 é a que apresenta o melhor desempenho. No entanto, dentre as versões escalonadas, a melhora no desempenho da versão com fator 4 em relação à versão com fator 2 é maior que a melhora da versão com fator 8 em relação à versão com fator 4. Isto é, o ganho de desempenho obtido reduz a medida que o fator do *loop unrolling* aumenta [28].

O gráfico da Figura 8.2 mostra os resultados médios da Tabela 8.2, onde o acelerador básico pode ser visto como tendo fator 1, isto é, possui uma única cópia do corpo do laço interno do algoritmo de

Algoritmo 8.4 Algoritmo de Viterbi com *loop unrolling* com fator 2 e escalonamento

1: $B[0] = Pt_{N,B}$
2: $C[0] = J[0] = -\infty$
3: $i_{atual} = 1$
4: $i_{anterior} = 0$
5: $limite = (Q\ div\ fator) \times fator$
6: **for** $i = 1$ to $|S|$ **do**
7: $\quad M[i_{atual}, 0] = I[i_{atual}, 0] = D[i_{atual}, 0] = -\infty$
8: $\quad E = -\infty$
9: $\quad$ **for** $j = 1$ to Q **step** 2 **do**
10: $\quad\quad M[0, j] = I[0, j] = D[0, j] = -\infty$
11: $\quad\quad M[0, j+1] = I[0, j+1] = D[0, j+1] = -\infty$
12: $\quad\quad aux\_M\_first = M[i_{anterior}, j-1] + Pt_{M_{j-1},M_j}$
13: $\quad\quad aux\_M\_second = M[i_{anterior}, j] + Pt_{M_j,M_{j+1}}$
14: $\quad\quad aux\_I\_first = M[i_{anterior}, j] + Pt_{M_j,I_j}$
15: $\quad\quad aux\_I\_second = M[i_{anterior}, j+1] + Pt_{M_{j+1},I_{j+1}}$
16: $\quad\quad aux\_D\_first = M[i_{atual}, j-1] + Pt_{M_{j-1},D_j}$
17: $\quad\quad aux\_D\_second = M[i_{atual}, j] + Pt_{M_j,D_{j+1}}$
18: $\quad\quad$ **if** $I[i_{anterior}, j-1] + Pt_{I_{j-1},M_j} > aux\_M\_first$ **then**
19: $\quad\quad\quad aux\_M\_first = I[i_{anterior}, j-1] + Pt_{I_{j-1},M_j}$
20: $\quad\quad$ **end if**
21: $\quad\quad$ **if** $I[i_{anterior}, j] + Pt_{I_j,M_{j+1}} > aux\_M\_second$ **then**
22: $\quad\quad\quad aux\_M\_second = I[i_{anterior}, j] + Pt_{I_j,M_{j+1}}$
23: $\quad\quad$ **end if**
24: $\quad\quad$ **if** $I[i_{anterior}, j] + Pt_{I_j,I_j} > aux\_I\_first$ **then**
25: $\quad\quad\quad aux\_I\_first = I[i_{anterior}, j] + Pt_{I_j,I_j}$
26: $\quad\quad$ **end if**
27: $\quad\quad$ **if** $I[i_{anterior}, j+1] + Pt_{I_{j+1},I_{j+1}} > aux\_I\_second$ **then**
28: $\quad\quad\quad aux\_I\_second = I[i_{anterior}, j+1] + Pt_{I_{j+1},I_{j+1}}$
29: $\quad\quad$ **end if**
30: $\quad\quad$ **if** $D[i_{atual}, j-1] + Pt_{D_{j-1},D_j} > aux\_D\_first$ **then**
31: $\quad\quad\quad aux\_D\_first = D[i_{atual}, j-1] + Pt_{D_{j-1},D_j}$
32: $\quad\quad$ **end if**
33: $\quad\quad$ **if** $D[i_{atual}, j] + Pt_{D_j,D_{j+1}} > aux\_D\_second$ **then**
34: $\quad\quad\quad aux\_D\_second = D[i_{atual}, j] + Pt_{D_j,D_{j+1}}$
35: $\quad\quad$ **end if**
36: $\quad\quad$ **if** $D[i_{anterior}, j-1] + Pt_{D_{j-1},M_j} > aux\_M\_first$ **then**
37: $\quad\quad\quad aux\_M\_first = D[i_{anterior}, j-1] + Pt_{D_{j-1},M_j}$
38: $\quad\quad$ **end if**
39: $\quad\quad$ **if** $D[i_{anterior}, j] + Pt_{D_j,M_{j+1}} > aux\_M\_second$ **then**
40: $\quad\quad\quad aux\_M\_second = D[i_{anterior}, j] + Pt_{D_j,M_{j+1}}$
41: $\quad\quad$ **end if**
42: $\quad\quad I[i_{atual}, j] = aux\_I\_first + Pe_{I_j}(s_i)$
43: $\quad\quad I[i_{atual}, j+1] = aux\_I\_second + Pe_{I_{j+1}}(s_i)$
44: $\quad\quad D[i_{atual}, j] = aux\_D\_first$
45: $\quad\quad D[i_{atual}, j+1] = aux\_D\_second$
46: $\quad\quad$ **if** $B[i_{anterior}] + Pt_{B,M_j} > aux\_M\_first$ **then**
47: $\quad\quad\quad aux\_M\_first = B[i_{anterior}] + Pt_{B,M_j}$
48: $\quad\quad$ **end if**
49: $\quad\quad$ **if** $B[i_{anterior}] + Pt_{B,M_{j+1}} > aux\_M\_second$ **then**
50: $\quad\quad\quad aux\_M\_second = B[i_{anterior}] + Pt_{B,M_{j+1}}$
51: $\quad\quad$ **end if**
52: $\quad\quad M[i_{atual}, j] = aux\_M\_first + Pe_{M_j}(s_i)$
53: $\quad\quad M[i_{atual}, j+1] = aux\_M\_second + Pe_{M_{j+1}}(s_i)$

Algoritmo 8.5 Algoritmo de Viterbi com *loop unrolling* com fator 2 e escalonamento (continuação)

54: **if** $M[i_{atual}, j] + Pt_{M_j,E} > E$ **then**
55: $E = M[i_{atual}, j] + Pt_{M_j,E}$
56: **end if**
57: **if** $M[i_{atual}, j+1] + Pt_{M_{j+1},E} > E$ **then**
58: $E = M[i_{atual}, j+1] + Pt_{M_{j+1},E}$
59: **end if**
60: **end for**
61: *//Laço complementar para quando Q não for divisível pelo fator*
62: **for** $j = limite + 1$ to Q **do**
63: $M[0, j] = I[0, j] = D[0, j] = -\infty$
64: $aux_M = M[i_{anterior}, j-1] + Pt_{M_{j-1},M_j}$
65: $aux_I = M[i_{anterior}, j] + Pt_{M_j,I_j}$
66: $aux_D = M[i_{atual}, j-1] + Pt_{M_{j-1},D_j}$
67: **if** $I[i_{anterior}, j-1] + Pt_{I_{j-1},M_j} > aux_M$ **then**
68: $aux_M = I[i_{anterior}, j-1] + Pt_{I_{j-1},M_j}$
69: **end if**
70: **if** $I[i_{anterior}, j] + Pt_{I_j,I_j} > aux_I$ **then**
71: $aux_I = I[i_{anterior}, j] + Pt_{I_j,I_j}$
72: **end if**
73: **if** $D[i_{atual}, j-1] + Pt_{D_{j-1},D_j} > aux_D$ **then**
74: $aux_D = D[i_{atual}, j-1] + Pt_{D_{j-1},D_j}$
75: **end if**
76: **if** $D[i_{anterior}, j-1] + Pt_{D_{j-1},M_j} > aux_M$ **then**
77: $aux_M = D[i_{anterior}, j-1] + Pt_{D_{j-1},M_j}$
78: **end if**
79: $I[i_{atual}, j] = aux_I + Pe_{I_j}(s_i)$
80: $D[i_{atual}, j] = aux_D$
81: **if** $B[i_{anterior}] + Pt_{B,M_j} > aux_M$ **then**
82: $aux_M = B[i_{anterior}] + Pt_{B,M_j}$
83: **end if**
84: $M[i_{atual}, j] = aux_M + Pe_{M_j}(s_i)$
85: **if** $M[i_{atual}, j] + Pt_{M_j,E} > E$ **then**
86: $E = M[i_{atual}, j] + Pt_{M_j,E}$
87: **end if**
88: **end for**
89: $aux_J = J[i_{anterior}] + Pt_{J,J}$
90: $aux_B = (i \times Pt_{N,N}) + Pt_{N,B}$
91: $aux_C = C[i_{anterior}] + Pt_{C,C}$
92: **if** $E + Pt_{E,J} > aux_J$ **then**
93: $aux_J = E + Pt_{E,J}$
94: **end if**
95: **if** $aux_J + Pt_{J,B} > aux_B$ **then**
96: $aux_B = aux_J + Pt_{J,B}$
97: **end if**
98: **if** $E + Pt_{E,C} > aux_C$ **then**
99: $aux_C = E + Pt_{E,C}$
100: **end if**
101: $J[i_{atual}] = aux_J$
102: $B[i_{atual}] = aux_B$
103: $C[i_{atual}] = aux_C$
104: $i_{anterior} = i_{atual}$
105: $i_{atual} = 1 - i_{atual}$
106: **end for**
107: $score = C[i_{anterior}] + Pt_{C,T}$

Viterbi.

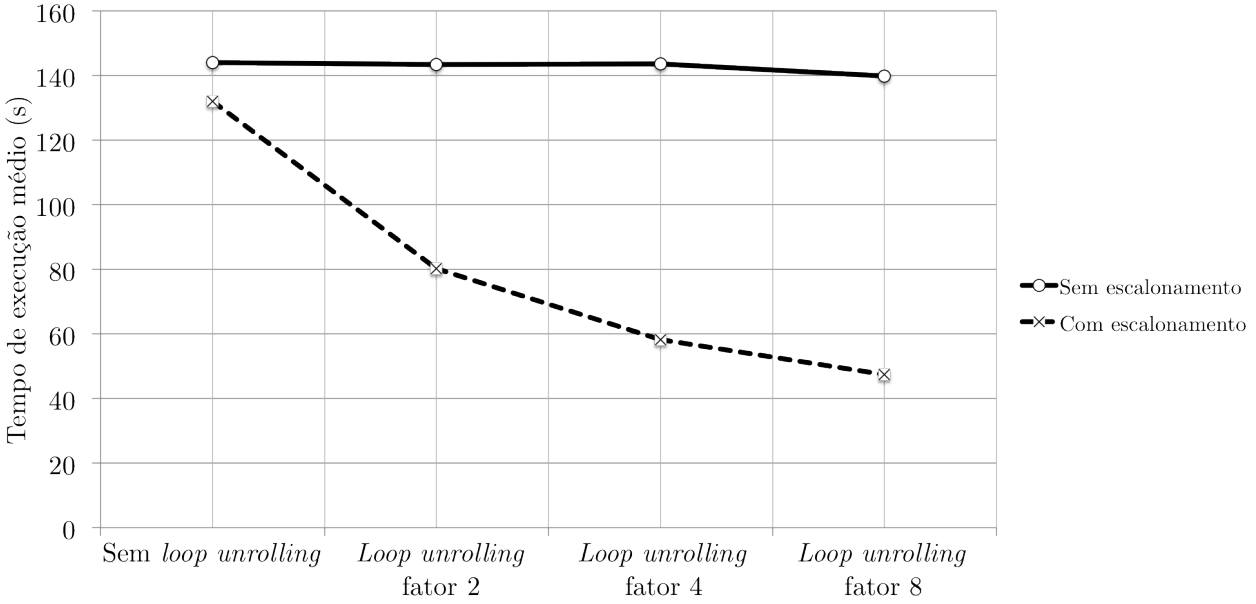


Figura 8.2: Tempo de execução médio dos aceleradores básico e com *loop unrolling*, sem e com escalonamento de instruções

Em geral, quanto maior o fator do *loop unrolling*, maior o ganho de desempenho. Entretanto, a partir de um determinado fator, o desempenho pode piorar. À medida que o fator aumenta, o número de registradores utilizados também cresce [28]. Quando o compilador não consegue manter todas as variáveis em registradores, ele as armazena na memória, que passa a ser acessada a cada vez que uma das variáveis é lida ou escrita. Esse mecanismo, denominado *register spilling*, causa perda de desempenho pois o acesso à memória é mais lento que o acesso aos registradores [45]. Além disso, o tamanho do código aumenta consideravelmente com o o aumento do fator do *loop unrolling*, o que pode prejudicar o desempenho da hierarquia de memórias, em relação à busca de instruções [28].

A Tabela 8.3 mostra a quantidade de registradores necessários, por *work-item*, para executar cada uma das versões dos aceleradores básico e com *loop unrolling*, e a ocupação da GPU de cada acelerador. O acelerador que obtém maior ganho de desempenho é também aquele que mais utiliza registradores na sua execução e um dos que produz o menor valor de ocupação (33%), ratificando que a diminuição no valor de ocupação não implica necessariamente em perda de desempenho.

Tabela 8.3: Número de registradores utilizados e ocupação da GPU dos aceleradores básico e com *loop unrolling*, sem e com escalonamento de instruções

Medida	Sem escalonamento				Com escalonamento			
	Sem *loop unrolling*	Com *loop unrolling*			Sem *loop unrolling*	Com *loop unrolling*		
		Fator 2	Fator 4	Fator 8		Fator 2	Fator 4	Fator 8
Número de registradores	37	39	41	45	39	44	63	63
Ocupação da GPU	50%	50%	50%	33%	50%	33%	33%	33%

Em geral, a partir de um determinado fator, o *loop unrolling* não proporciona ganho de desempenho significativo [28] e a redução do ganho de desempenho obtido com o fator 8 quando comparado ao ganho trazido pelo fator 4 mostra que o acelerador está alcançando esse limiar. Por esses motivos, a versão do acelerador com *loop unrolling* com fator 16 não foi desenvolvida. Além disso, a Tabela 8.3 mostra que os aceleradores com *loop unrolling* com fator 4 e 8 já utilizam o número máximo de registradores

(63) disponíveis por *work-item* da GPU utilizada, informado na planilha de ocupação da GPU [54], e o acelerador com fator 16 possivelmente demandaria mais registradores.

8.3 Transferência de Dados entre *Host* e GPU e *Overlapping*

Com o objetivo de investigar as situações nas quais a transferência de dados entre *host* e GPU pode ser reduzida e a técnica de *overlapping* pode ser aplicada, a forma de acesso das estruturas de dados do acelerador por cada *work-item* são analisadas.

A Figura 8.3 mostra as estruturas de dados necessárias para a execução do acelerador em GPU com N *work-items*. Uma seta partindo de um *work-item* w_k para uma estrutura ou para parte dela indica que w_k acessa essa estrutura.

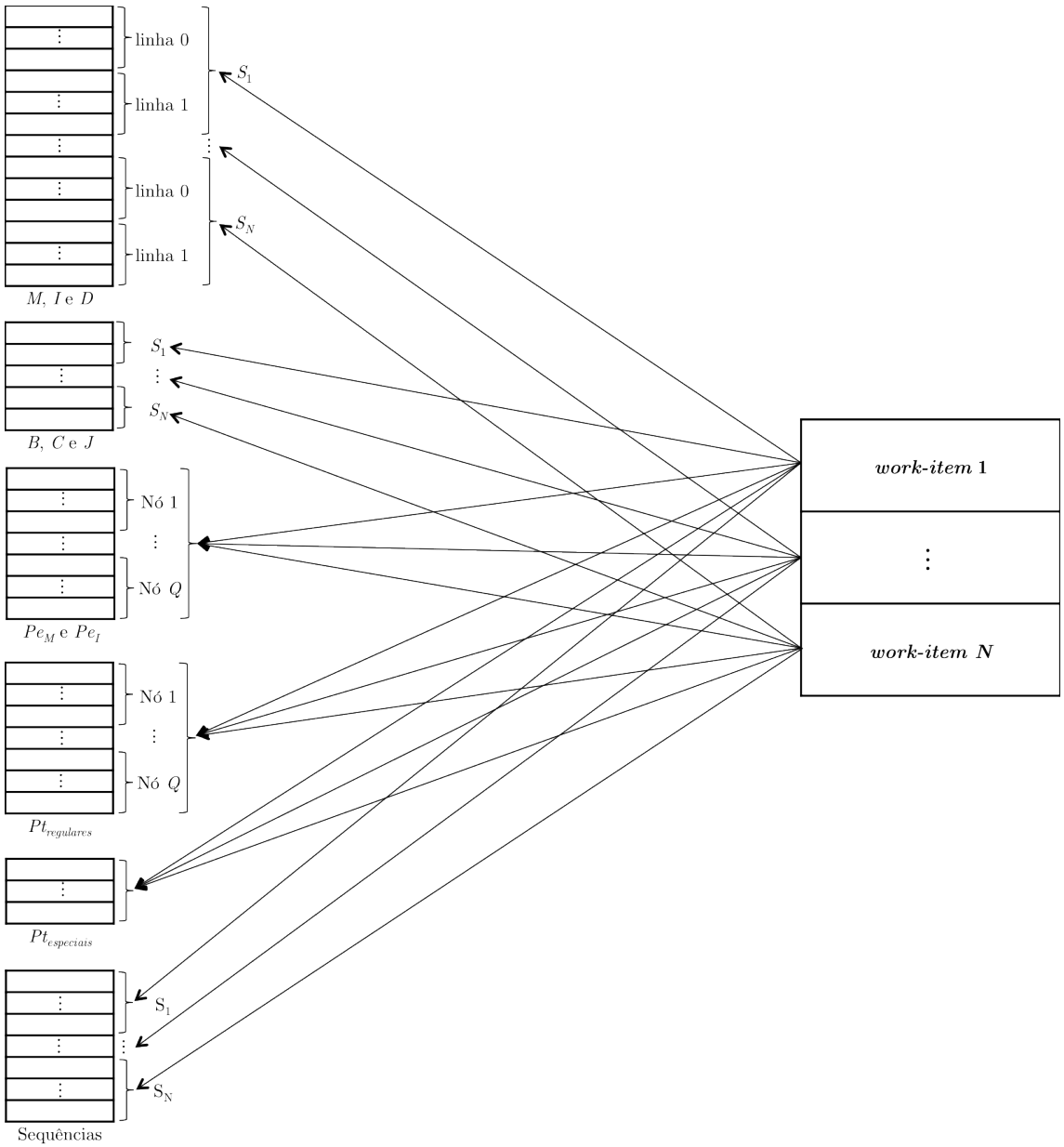


Figura 8.3: Estruturas de dados acessadas por cada *work-item* do acelerador em GPU

Cada *work-item* utiliza uma parte distinta de cada uma das estruturas de *scores* M, I, D, B, C e J. Os valores dessas estruturas são inicializados na memória global da GPU e acessados apenas por ela, não existindo transferências de dados entre o *host* e a GPU que envolvam estas estruturas.

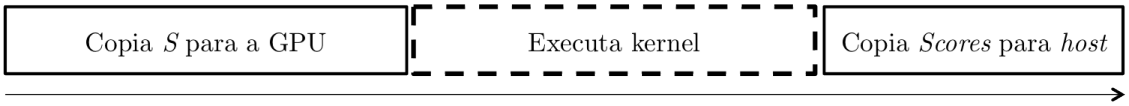
Em uma iteração do laço mais externo do algoritmo de Viterbi executado por cada *work-item*, todas as posições das estruturas $Pt_{regulares}$ e $Pt_{especiais}$ são lidas. Portanto, essas estruturas são copiadas do *host* para a memória global da GPU antes da invocação do *kernel* e são mantidas lá durante toda a execução, evitando novas transferências de dados.

As posições das estruturas Pe_M e Pe_I utilizadas por um *work-item* durante sua execução dependem da sequência atribuída a ele. É possível que todas as posições sejam acessadas ou que apenas parte delas sejam necessárias. Por exemplo, no processamento da sequência $S_1 = GALVIPFSTCYNQDERKHWM$ todas as posições dessas estruturas são lidas pelo *work-item* w_1, ao qual S_1 foi atribuída. Para a sequência $S_2 = GGGGGGGGGGGGGGG$ apenas as posições dessas estruturas correspondentes ao símbolo G são acessadas. Essas estruturas são copiadas do *host* para a memória global da GPU antes da invocação do *kernel* e são mantidas lá durante toda a execução.

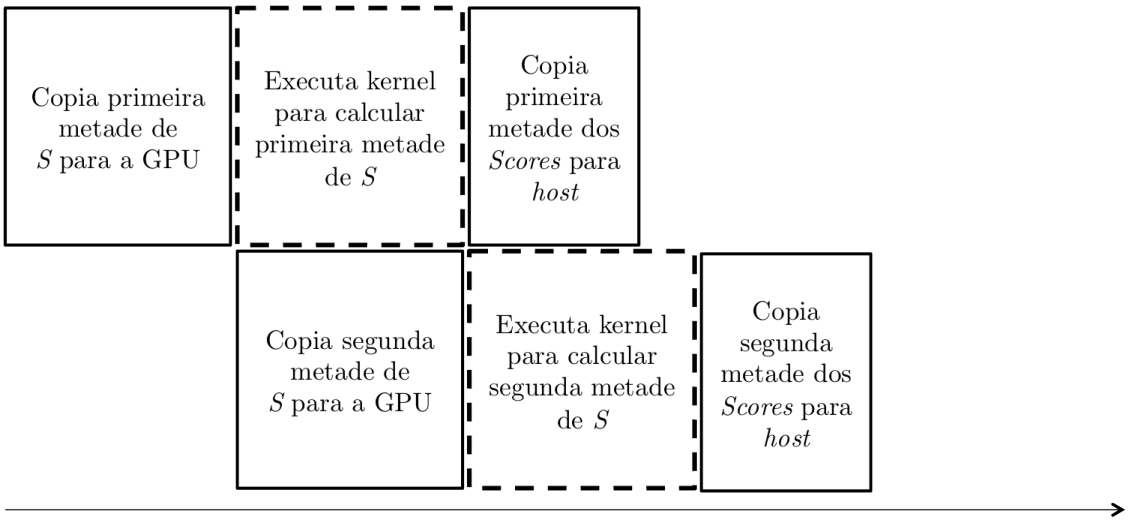
Uma cópia parcial e/ou sob demanda dessas estruturas exigiria uma análise prévia das necessidades de cada *work-item* ou uma sincronização dos *work-items*, o que aumentaria o tempo de execução. Dado o tamanho reduzido das estruturas (34 KB ao todo, para o maior HMM do conjunto *Top Twenty*), é mais vantajoso copiá-las por completo para a GPU, evitando novas transferências de dados.

A estrutura S armazena os símbolos das sequências que são processadas em GPU e cada *work-item* acessa todas as posições apenas da parte da estrutura referente à sequência atribuída a ele.

A Figura 8.4(a) mostra a forma convencional, sem *overlapping*, de execução do acelerador, em apenas um passo. Inicialmente toda a estrutura S, com N sequências, é copiada do *host* para a memória global da GPU, antes da invocação das N instâncias do *kernel*, e é mantida lá durante toda a execução. Em seguida, os N *work-items* calculam, na GPU, os *scores* das sequências, acessando a estrutura S. Por fim, os *scores* finais calculados são copiados da GPU para o *host*.



(a) Sem *overlapping*



(b) Com *overlapping*

Figura 8.4: Execução do acelerador sem e com sobreposição da transferência de dados *host*-GPU com computação na GPU

A Figura 8.4(b) mostra uma forma de execução do acelerador com *overlapping* em dois passos.

Inicialmente, apenas a primeira metade de S ($N/2$ sequências) é copiada do *host* para a memória global da GPU, antes da invocação das $N/2$ instâncias do *kernel*. Em seguida, enquanto os $N/2$ *work-items* calculam na GPU os *scores* da primeira metade das sequências, a segunda metade de S é copiada do *host* para a memória global da GPU, sobrepondo a computação na GPU com a transferência de dados *host*-GPU.

Idealmente, quando o acelerador termina o cálculo dos *scores* da primeira metade de S, a segunda metade de S já está disponível na memória global da GPU. Em seguida, $N/2$ novas instâncias do *kernel* são invocadas e, enquanto os novos *work-items* calculam na GPU os *scores* da segunda metade das sequências, os *scores* finais calculados para a primeira metade são copiados da memória global da GPU para o *host*. Novamente ocorre a sobreposição da computação na GPU com a transferência de dados *host*-GPU. Finalmente, quando o acelerador termina o cálculo dos *scores* da segunda metade de S, eles são copiados da memória global da GPU para o *host*.

A Tabela 8.4 mostra o tempo de execução dos aceleradores básico e com *overlapping*. Em nenhuma das execuções o tempo de execução do acelerador com *overlapping* foi melhor em relação à execução do acelerador básico. A sobreposição da transferência de dados *host*-GPU com computação na GPU tem um *overhead* de controle para o *host*, que precisa ser compensado pelo volume de sobreposição, para que o *overlapping* traga ganho de desempenho.

Tabela 8.4: Tempo de execução dos aceleradores básico e com *overlapping*

Família	Tempo de execução (s)	
	Acelerador básico	Acelerador com *overlapping*
ABC_tran	97,34	98,08
Acetyltransf_1	68,15	69,80
adh_short	138,78	139,40
BPD_transp_1	154,34	156,31
COX1	373,26	376,55
Cytochrom_B_C	84,19	85,84
Cytochrom_B_N	155,96	158,45
GP120	414,40	416,68
HATPase_c	91,27	91,95
Helicase_C	63,80	64,39
HTH_1	48,99	49,51
MFS_1	295,91	298,25
Oxidored_q1	221,92	223,56
Pkinase	217,05	220,47
Response_reg	91,88	92,90
RVP	82,33	83,44
RVT_1	176,21	177,25
RVT_thumb	57,34	58,57
WD40	31,59	31,97
zf-C2H2	17,80	18,01
Média	144,13	145,57

A Tabela 8.5 mostra a porcentagem do tempo de execução que o acelerador básico gasta com computação na GPU e com transferências de dados *host*-GPU da estrutura S de sequências e das demais estruturas. Em todas as execuções, o tempo gasto com computação na GPU é superior a 99%. Ou seja, por apenas um período muito pequeno de tempo existe de fato sobreposição entre transferência de dados e computação, não havendo benefício em explorar *overlapping*.

A última coluna da Tabela 8.5 mostra que, mesmo que fosse possível realizar o *overlapping* com outras estruturas de dados, além da estrutura S, também não haveria ganho de desempenho, pois o

Tabela 8.5: Porcentagem do tempo de execução do acelerador básico gasto com computação na GPU e transferências *host*-GPU

Família	% do tempo de execução		
	Computação na GPU	Transferências *host*-GPU	
		Estrutura S	Outras estruturas
ABC_tran	99,98%	0,02%	0,001%
Acetyltransf_1	99,97%	0,03%	0,001%
adh_short	99,98%	0,02%	0,001%
BPD_transp_1	99,98%	0,01%	0,001%
COX1	99,99%	0,01%	0,002%
Cytochrom_B_C	99,97%	0,03%	0,001%
Cytochrom_B_N	99,98%	0,01%	0,001%
GP120	99,99%	0,01%	0,001%
HATPase_c	99,97%	0,02%	0,001%
Helicase_C	99,96%	0,04%	0,001%
HTH_1	99,95%	0,05%	0,002%
MFS_1	99,99%	0,01%	0,001%
Oxidored_q1	99,99%	0,01%	0,000%
Pkinase	99,99%	0,01%	0,001%
Response_reg	99,97%	0,02%	0,001%
RVP	99,97%	0,03%	0,001%
RVT_1	99,99%	0,01%	0,001%
RVT_thumb	99,96%	0,04%	0,002%
WD40	99,92%	0,07%	0,003%
zf-C2H2	99,87%	0,13%	0,005%
Média	99,97%	0,03%	0,001%

tempo gasto com essas transferências de memória é muito pequeno em relação ao tempo de execução do *kernel*.

8.4 Fluxo de Controle e Divergências

Kernels executados em GPU deveriam, em um cenário ideal, não conter divergências durante sua execução, pois elas degradam o desempenho. As divergências existentes no acelerador em GPU para o algoritmo de Viterbi são analisadas e otimizações para amenizá-las são investigadas.

As divergências do algoritmo de Viterbi podem ser classificadas em dois tipos:

- Divergências internas: causadas pelas estruturas condicionais (*if-then*) existentes nas linhas 11 a 33 e linhas 36 a 48 do Algoritmo 7.1. *Work-items* de um *warp* executam usando o mesmo HMM mas operam sequências diferentes, portanto, suas estruturas de *scores* possivelmente são diferentes. Isso implica que cada uma das estruturas condicionais do algoritmo de Viterbi são candidatas a gerarem divergência;
- Divergências externas: causadas pelo laço mais externo (que itera sobre os símbolos da sequência) na linha 5 do Algoritmo 7.1. Como cada *work-item* w_k trabalha com uma sequência S_k diferente, o tamanho $|S_k|$ da sequência pode ser diferente para cada *work-item*. Assim, *work-items* de um mesmo *warp* podem obter resultados diferentes para a condição de parada $i > |S_k|$ do laço, ocasionando divergências.

O laço mais interno (que itera sobre os nós do HMM) na linha 8 do Algoritmo 7.1 não causa

divergências pois, como todos os *work-items* instanciados executam usando o mesmo HMM de comprimento Q, a condição de parada $j > Q$ gera o mesmo resultado para todos os *work-items* de um *warp*.

Para analisar o impacto das divergências internas no desempenho do acelerador em GPU, dois conjuntos de sequências sintéticas são criados:

- Conjunto 1: 70.000 sequências iguais com 32 símbolos;
- Conjunto 2: 70.000 sequências diferentes com 32 símbolos.

A execução do acelerador para um HMM qualquer não apresenta divergências externas para ambos os conjuntos, uma vez que todas as sequências possuem o mesmo tamanho. Em relação às divergências internas, a execução do acelerador utilizando o conjunto 1 não apresenta esse tipo de divergência, pois todas as sequências são iguais e, portanto, todos os *work-items* de um *warp* obtêm os mesmos resultados em todas as estruturas condicionais. Já a execução do acelerador utilizando o conjunto 2 pode apresentar divergências internas, uma vez que as sequências são diferentes e, consequentemente, os *work-items* de um mesmo *warp* podem obter resultados diferentes nas condições testadas nas estruturas condicionais.

A Tabela 8.6 mostra o resultado da execução do acelerador básico em GPU para os HMMs *Top Twenty* e os conjuntos 1 e 2 de sequências. O tempo de execução do conjunto 2 (com divergências) é praticamente igual ao tempo de execução do conjunto 1 (sem divergências). A princípio, esse comportamento não é esperado, uma vez que as divergências poderiam causar atraso na execução dos *warps*.

Tabela 8.6: Tempo de execução do acelerador básico para os conjuntos de sequências sintéticas sem e com divergência interna

Família	Tempo de execução (s)	
	Sem divergência interna	Com divergência interna
ABC_tran	3,72	3,72
Acetyltransf_1	2,63	2,63
adh_short	5,3	5,31
BPD_transp_1	5,94	5,95
COX1	14,31	14,34
Cytochrom_B_C	3,21	3,22
Cytochrom_B_N	5,98	5,99
GP120	15,79	15,83
HATPase_c	3,5	3,51
Helicase_C	2,45	2,45
HTH_1	1,88	1,87
MFS_1	11,33	11,34
Oxidored_q1	8,47	8,49
Pkinase	8,36	8,38
Response_reg	3,55	3,55
RVP	3,16	3,17
RVT_1	6,75	6,76
RVT_thumb	2,21	2,21
WD40	1,21	1,21
zf-C2H2	0,69	0,69
Média	5,52	5,53

A ferramenta de *profile* fornecida pela NVIDIA é utilizada para a execução do acelerador básico com o HMM de 78 nós correspondente à família Helicase_C da Pfam e os conjuntos 1 e 2 de sequências, permitindo a medição das divergências. A Figura 8.5 mostra o resultado, gerado pela ferramenta para ambos os conjuntos, de que não há divergências (valor 0 na figura) na execução do acelerador. Este resultado justifica os dados da Tabela 8.6, em que o tempo de execução é praticamente igual para os dois conjuntos de sequências.

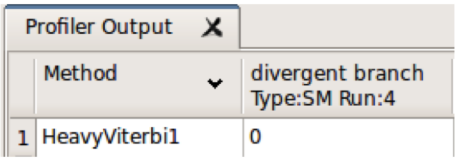


Figura 8.5: Divergências na execução do acelerador para os conjuntos 1 e 2 de sequências e o HMM Helicase_C

Através do mecanismo de predicação de desvios, instruções de desvio condicional correspondentes a estruturas condicionais (por exemplo, *if*) com poucas instruções, são substituídas por instruções predicadas. A instrução predicada realiza uma operação que é concretizada apenas se uma determinada condição (predicado) é satisfeita [28]. O compilador substitui um desvio condicional por instruções predicadas, somente se o número de instruções controladas pela condição do desvio é pequeno. Dessa forma, as divergências entre os *work-items* de um mesmo *warp* causadas pelos desvios condicionais são eliminadas [49]. Como as estruturas condicionais do Algoritmo 7.1 que poderiam causar divergências internas controlam a execução de poucas instruções, concluímos que o mecanismo de predicação é utilizado, substituindo os desvios condicionais, e em consequência, eliminando as divergências internas na execução do acelerador para o conjunto 2 de sequências.

Para analisar o impacto das divergências externas no desempenho do acelerador em GPU, outros dois conjuntos de sequências sintéticas são criados com as seguintes características:

- Conjunto 3: 10.016 sequências de tamanho mínimo 10 e máximo 499, tal que as sequências estão ordenadas de forma não decrescente do tamanho e cada grupo de 32 sequências consecutivas possui o mesmo tamanho, que é menor que o tamanho das sequências do grupo seguinte. Mais precisamente, $|S_1| = |S_2| = \ldots = |S_{32}| < |S_{33}| = \ldots = |S_{64}| < |S_{65}| = \ldots = |S_{96}| < \ldots < |S_{9.984}| = \ldots = |S_{10.016}|$.
- Conjunto 4: formado a partir de uma permutação aleatória das sequências do conjunto 3.

Uma execução do acelerador para qualquer HMM não produz divergências externas para o conjunto 3, uma vez que todas as sequências de um *warp* possuem o mesmo tamanho. Embora as sequências do conjunto 4 sejam as mesmas do conjunto 3, a ordem na qual elas aparecem no conjunto é diferente, havendo sequências de tamanhos diferentes designadas a *work-items* de um mesmo *warp*. Sendo assim, a execução do acelerador para esse conjunto produz divergências externas.

A Figura 8.6 mostra o resultado, gerado pela ferramenta de *profile*, da execução do acelerador básico com o HMM de 185 nós correspondente à família BPD_transp_1 da Pfam e os conjuntos 3 e 4 de sequências. Os resultados confirmam que, na execução do acelerador para o conjunto 3, não ocorrem divergências, mas para o conjunto 4 ocorrem.

A Tabela 8.6 mostra o resultado da execução do acelerador básico em GPU para os HMMs *Top Twenty* e os conjuntos 3 e 4 de sequências. Embora os conjuntos 3 e 4 possuam as mesmas sequências e a execução do acelerador com o conjunto 3 não apresente divergências, o tempo de execução do conjunto 4 é um pouco menor que o do conjunto 3, para todos os HMMs.

(a) Conjunto 3 (b) Conjunto 4

Figura 8.6: Divergências externas na execução do acelerador para os conjuntos 3 e 4 de sequências e o HMM BPD_transp_1

Tabela 8.7: Tempo de execução do acelerador básico para os conjuntos sintéticos de sequências sem e com divergência externa

Família	Tempo de execução (s)	
	Sem divergência externa	Com divergência externa
ABC_tran	4,63	4,39
Acetyltransf_1	3,30	3,09
adh_short	6,40	6,27
BPD_transp_1	7,37	6,99
COX1	17,76	16,94
Cytochrom_B_C	4,02	3,82
Cytochrom_B_N	7,42	7,05
GP120	19,72	18,65
HATPase_c	4,35	4,13
Helicase_C	3,05	2,88
HTH_1	2,34	2,21
MFS_1	14,06	13,40
Oxidored_q1	10,65	10,07
Pkinase	10,43	9,88
Response_reg	4,25	4,17
RVP	3,92	3,72
RVT_1	8,41	8,03
RVT_thumb	2,74	2,59
WD40	1,51	1,42
zf-C2H2	0,83	0,79
Média	6,86	6,52

A princípio, a perda de desempenho mesmo com uma diminuição das divergências não é esperada. Analisando os resultados gerados pela ferramenta de *profile*, conclui-se que o fato do conjunto 3 estar ordenado gera um padrão de acesso às memórias que é menos eficiente em relação ao padrão gerado pelo conjunto não ordenado. Embora a ordenação das sequências contribua negativamente neste caso, no próximo capítulo ela permite a otimização dos acessos às memórias e da largura de banda do acelerador.

Capítulo 9

Otimizações de Memória Aplicadas

Armazenar e acessar as estruturas de dados em GPU de maneira otimizada é um dos pontos mais importantes para obter ganho de desempenho nessa plataforma. Neste capítulo são apresentadas as otimizações aplicadas no armazenamento e acesso às estruturas de dados do acelerador em GPU para o algoritmo de Viterbi, bem como os resultados alcançados.

9.1 Sequências

A Figura 9.1 mostra o padrão de acessos dos *work-items* de um *warp* ao primeiro símbolo de um conjunto de sequências $S = \{S_1, S_2, ..., S_{32}\}$, tal que $|S_k| = 128$ para $1 \leq k \leq 32$, armazenado na memória global da GPU de acordo com a organização da Figura 7.3(e). De acordo com o Algoritmo 7.1, há duas leituras do mesmo símbolo para cada *work-item* a cada iteração do laço interno (que itera sobre os nós do HMM) do algoritmo de Viterbi. Para evitar que dois acessos sejam gerados por um *work-item*, no primeiro acesso o símbolo lido é armazenado em uma variável local (que deve ser mapeada pelo compilador em um registrador) e o segundo acesso é realizado diretamente nessa variável.

Como todas as sequências desse conjunto têm comprimento 128, o primeiro símbolo da sequência S_1 está distante na memória 128 bytes do primeiro símbolo de S_2, que também está distante 128 bytes do primeiro símbolo de S_3, e assim sucessivamente até S_{32}.

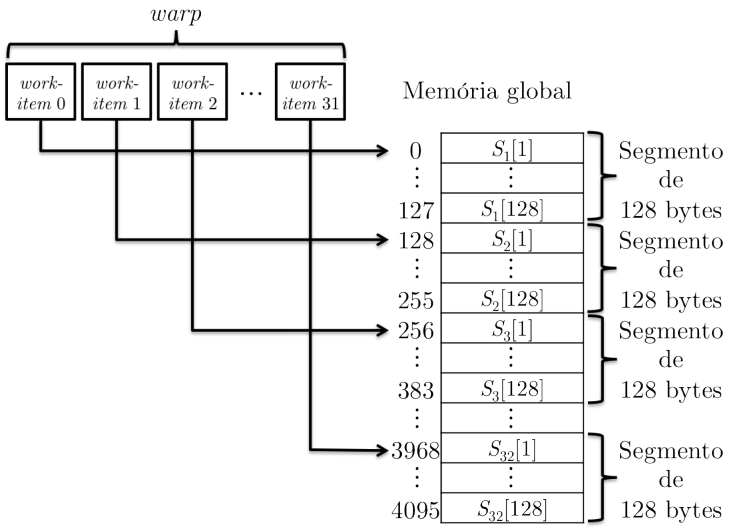


Figura 9.1: Acessos de um *warp* ao primeiro símbolo de $S = \{S_1, ..., S_{32}\}$, tal que $|S_k| = 128$ para $1 \leq k \leq 32$

Na Figura 9.1, todas as requisições de memória do *warp* pertencem a segmentos de memória de

128 bytes diferentes, portanto, são necessárias 32 transações de memória para atendê-las, dado que as transações são sempre de 128 bytes. Dos 4096 bytes trazidos, apenas 32 são utilizados pelo *warp*, pois cada posição da sequência ocupa um byte. Assim, para cada *warp*, 4064 bytes transferidos não são utilizados, levando a um desperdício de largura de banda.

Para reduzir esse desperdício, os símbolos das sequências podem ser intercalados, de forma que símbolos de uma mesma posição das diferentes sequências ocupem um segmento de memória de 128 bytes e, portanto, seus acessos sejam coalescidos em apenas uma transação, como mostra o padrão de acessos da Figura 9.2. Assim, o desperdício de largura de banda cai para 96 bytes por *warp*. Esse procedimento não altera a quantidade de memória necessária para o armazenamento da estrutura de sequências S.

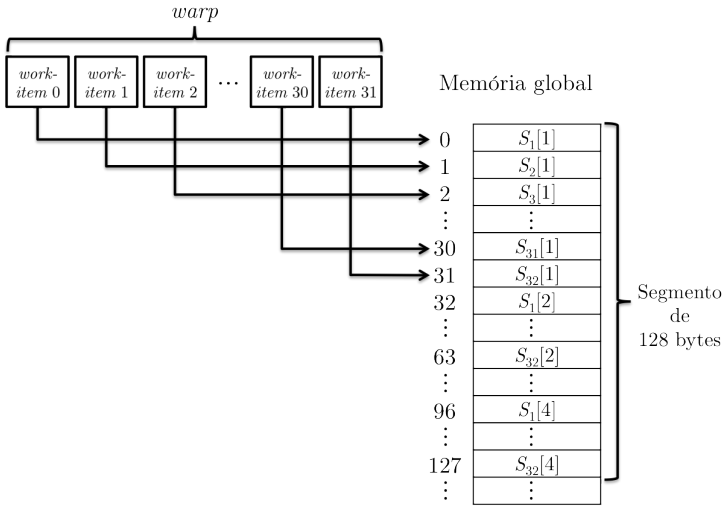


Figura 9.2: Acessos coalescidos de um *warp* ao primeiro símbolo de $S = \{S_1, ..., S_{32}\}$, tal que $|S_k| = 128$ para $1 \leq k \leq 32$

Para um conjunto de sequências $S = \{S_1, S_2, ..., S_{64}\}$ tal que $|S_k| = 1$ se k for par e $|S_k| = 32$ se k for ímpar, para $1 \leq k \leq 64$, dois *warps* são criados para o acelerador em GPU. É necessária uma transação de memória para atender a requisição de cada *warp*, pois cada transação realiza acessos de apenas um *warp*.

A partir do acesso ao segundo elemento das sequências, o padrão de acessos se altera, pois as sequências têm tamanhos diferentes e metade delas já terminaram. A Figura 9.3 mostra o padrão de acessos dos *work-items* dos *warps* 1 e 2 ao segundo símbolo do conjunto de sequências S.

Novamente, uma transação é necessária para atender a requisição de cada *warp*. Entretanto, metade dos *work-items* estão ociosos em cada *warp*, pois correspondem a sequências que já terminaram. É possível diminuir a quantidade total de transações, atribuindo sequências de tamanhos próximos a *work-items* de um mesmo *warp*.

As 32 sequências de tamanho 32 podem ser atribuídas aos 32 primeiros *work-items* e as sequências de tamanho 1 aos 32 *work-items* restantes. Nos acessos ao segundo símbolo das sequências, o *warp* 1 precisa de uma transação para acessar os 32 símbolos, mantendo o padrão apresentado na Figura 9.2, enquanto os *work-items* do *warp* 2 ficam ociosos e não geram transação de memória. Portanto, ao invés de duas transações, é necessária apenas uma transação.

A redução na quantidade total de transações com o agrupamento de sequências com tamanhos próximos no mesmo *warp* pode ser alcançada através da ordenação das sequências, de forma crescente ou decrescente do seu tamanho. Por facilitar o acesso às estruturas de dados, a ordenação decrescente das sequências é escolhida. Esse procedimento também não altera a quantidade de memória necessária para o armazenamento da estrutura de sequências S.

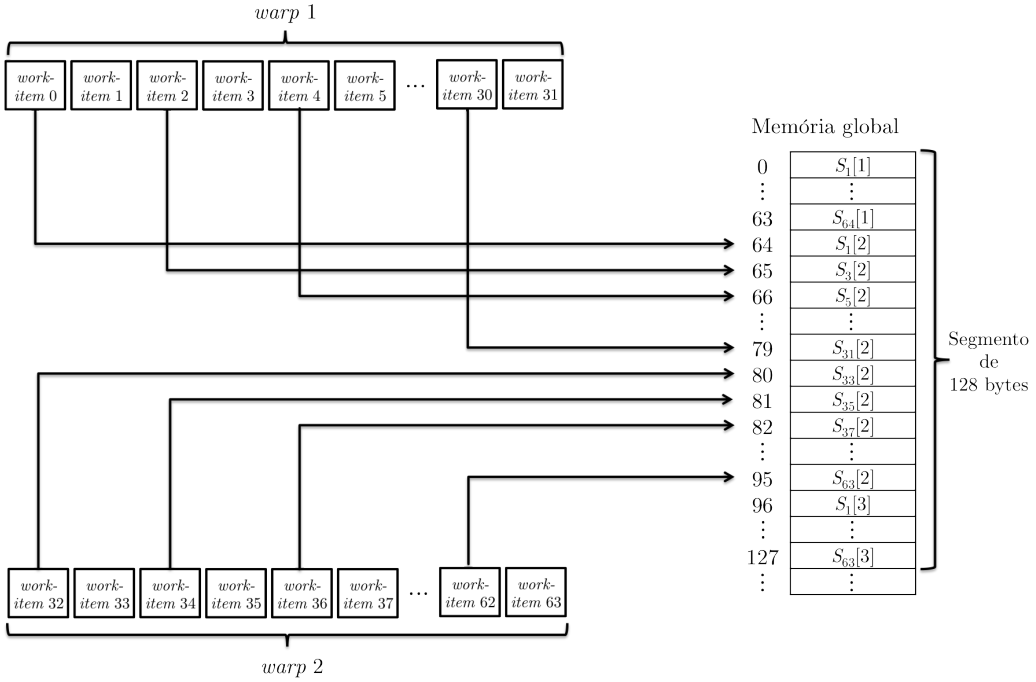


Figura 9.3: Acessos de dois *warps* ao segundo símbolo de $S = \{S_1, ..., S_{64}\}$, tal que $|S_k| = 1$ se k for par e $|S_k| = 32$ se k for ímpar, para $1 \leq k \leq 64$

A Tabela 9.1 apresenta os resultados da execução do acelerador em GPU que utiliza a coalescência de memória no acesso à estrutura S, para o conjunto de 70.000 sequências da base Swiss-Prot, ordenadas de forma decrescente do tamanho, e os HMMs *Top Twenty*. A coalescência proporciona, em média, uma redução de 8% no tempo de execução em relação ao acelerador básico.

Mesmo com a coalescência dos acessos dos 32 *work-items* de um *warp* em apenas uma transação de memória, há ainda desperdício de largura de banda, pois dos 128 bytes trazidos, apenas 32 bytes são utilizados. Isso acontece pois o tamanho das transações é sempre 128 bytes e cada um dos 32 *work-items* do *warp* precisa de apenas um símbolo de um byte por vez.

Esse desperdício pode ser eliminado se, ao invés de trazer apenas um símbolo de cada sequência em cada acesso, quatro símbolos sejam agrupados em um dado de 4 bytes e trazidos, como mostra a Figura 9.4, fazendo com que os 128 bytes transferidos sejam utilizados pelo *warp*.

A cada quatro iterações do laço externo (que itera sobre os símbolos da sequência) do algoritmo de Viterbi, um dado de 4 bytes é lido por cada *work-item* de um *warp* e convertido em 4 dados de 1 byte que são armazenados em uma variável local, cada byte correspondendo a um símbolo da sequência. Nas próximas três iterações do laço externo, os símbolos da sequência são lidos diretamente dessa variável, que provavelmente é mapeada pelo compilador em registradores.

O outro problema relativo ao armazenamento e acesso às sequências é a quantidade de transações de memória geradas por um *warp* para trazer os símbolos das sequências, mesmo quando os acessos são coalescidos. Para o conjunto de sequências $S = \{S_1, S_2, ..., S_{32}\}$, tal que $|S_k| = 32$ para $1 \leq k \leq 31$ e $|S_{32}| = 3$, armazenadas com os símbolos intercalados na memória global, há apenas um *warp* em execução.

Os acessos aos três primeiros símbolos das sequências feitos pelo *warp* são realizados com uma transação de memória cada um. No acesso ao quarto símbolo, o *work-item* 31 do *warp* está ocioso, pois a sequência S_{32} já terminou, mas ainda assim o acesso aos 31 símbolos restantes caem no mesmo segmento e são atendidos com uma única transação.

Entretanto, no acesso ao quinto símbolo das sequências, mostrado na Figura 9.5, o *work-item* 0 acessa o símbolo $S_1[5]$, que ainda está no primeiro segmento de memória, enquanto o *work-item* 1

Tabela 9.1: Tempo de execução dos aceleradores básico e com coalescência no acesso a S e ordenação decrescente das sequências

Família	Tempo de execução (s)	
	Acelerador básico	Com coalescência no acesso a S
ABC_tran	97,34	88,95
Acetyltransf_1	68,15	62,86
adh_short	138,78	126,70
BPD_transp_1	154,34	142,19
COX1	373,26	341,63
Cytochrom_B_C	84,19	77,13
Cytochrom_B_N	155,96	142,79
GP120	414,40	377,69
HATPase_c	91,27	83,68
Helicase_C	63,80	58,61
HTH_1	48,99	45,05
MFS_1	295,91	270,61
Oxidored_q1	221,92	202,77
Pkinase	217,05	199,90
Response_reg	91,88	84,56
RVP	82,33	75,42
RVT_1	176,21	161,40
RVT_thumb	57,34	52,74
WD40	31,59	28,92
zf-C2H2	17,80	16,40
Média	144,13	132,00

acessa o símbolo $S_2[5]$, que está no próximo segmento, assim como os símbolos acessados pelos demais *work-items*. Ou seja, os *work-items* do mesmo *warp* acessam símbolos em dois diferentes segmentos, gerando duas transações para atender aos acessos, apesar da quantidade de dados solicitada ser apenas 31 bytes. Nesse caso, há desperdício de banda, mesmo quando as sequências estão ordenadas pelo tamanho, que é o caso desse conjunto S.

Esse desperdício de banda é eliminado se todas as sequências atribuídas a um mesmo *warp* tiverem o mesmo tamanho, pois assim os acessos dos *work-items* desse *warp* ao mesmo elemento das sequências caem no mesmo segmento de memória. Para isso, a cada subconjunto de 32 sequências, o tamanho da maior sequência do subconjunto é determinado e símbolos nulos são inseridos no final das sequências até que todas fiquem com o mesmo tamanho que a maior delas.

Essa técnica, denominada *array padding* [37], aumenta a quantidade de memória necessária para o armazenamento da estrutura de sequências S. Para o conjunto de sequências $S = \{S_1, ..., S_N\}$, a demanda de memória passa de $\sum_{k=1}^{N} |S_k|$ bytes, como apresentado na Tabela 7.3, para:

$$32 \times \max_{1 \leq k \leq 32} |S_k| \; + \; 32 \times \max_{33 \leq k \leq 64} |S_k| \; + \; ... \; \leq \; 32 \times \sum_{w=1}^{\lceil \frac{N}{32} \rceil} \max_{1 \leq k \leq 32} |S_{32 \times (w-1)+k}| \quad \text{bytes.}$$

A ordenação das sequências pelo tamanho faz com que o aumento da demanda de memória causado pela inserção de símbolos nulos nas sequências não seja tão grande. Entretanto, o ganho de desempenho proporcionado por essa técnica também é reduzido quando as sequências estão ordenadas, pois sequências do mesmo tamanho ou tamanhos próximos provavelmente são atribuídas a *work-items*

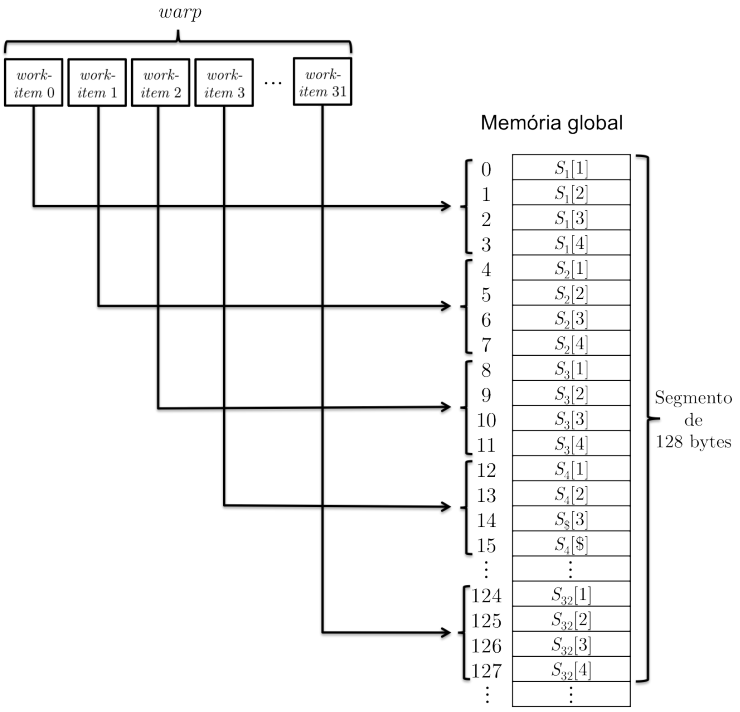


Figura 9.4: Acessos coalescidos de um *warp* aos quatro primeiros símbolos de $S = \{S_1, ..., S_{32}\}$, tal que $|S_k| \geq 4$ para $1 \leq k \leq 32$, agrupando quatro símbolos em um dado de 32 bits

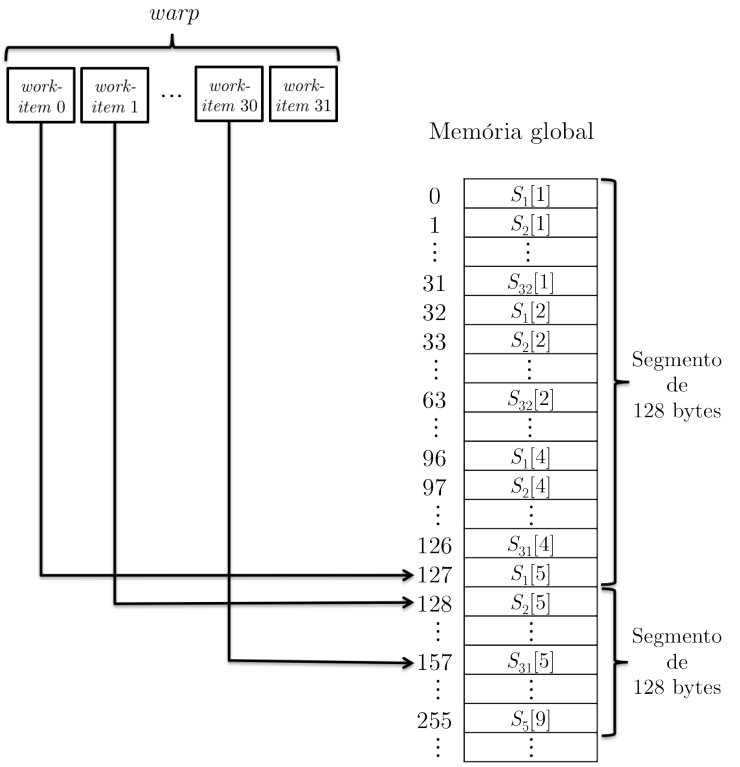


Figura 9.5: Acessos de um *warp* ao quinto símbolo de $S = \{S_1, ..., S_{32}\}$, tal que $|S_k| = 32$ para $1 \leq k \leq 31$ e $|S_{32}| = 3$

de um mesmo *warp*. Por essas razões a otimização de *padding* não é aplicada à estrutura de sequências do acelerador em GPU.

9.2 Estruturas de *Scores*

As estruturas de *scores* são muito acessadas, tanto para leitura quanto para escrita, na execução do algoritmo de Viterbi apresentado no Algoritmo 7.1, portanto esses acessos devem ser otimizados para que o algoritmo tenha um bom desempenho. Para um conjunto de 32 sequências $S = \{S_1, S_2, ..., S_{32}\}$, tal que $|S_k| = 8$ para $1 \leq k \leq 32$, e um HMM com $Q = 8$ nós, a Figura 9.6 mostra o padrão de acessos dos *work-items* de um *warp* ao primeiro elemento da estrutura de *scores* M, armazenada na memória global da GPU de acordo com a organização da Figura 7.3(a). Na figura, M_k representa a estrutura de *scores* M relativa à sequência S_k. Como um valor de *score* é representado por um inteiro de 4 bytes, cada elemento dessa estrutura ocupa 4 bytes.

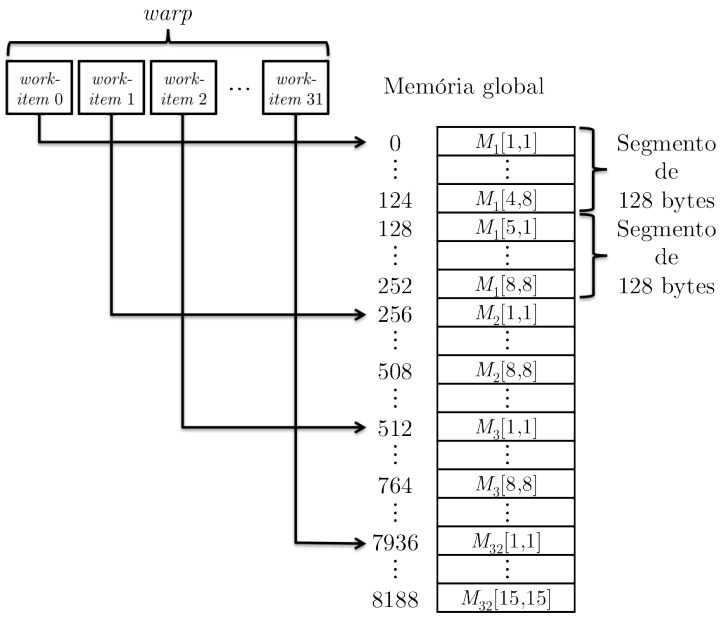


Figura 9.6: Acessos de um *warp* ao primeiro elemento de M, para $S = \{S_1, ..., S_{32}\}$, tal que $|S_k| = 8$ para $1 \leq k \leq 32$, e $Q = 8$

A distância entre os elementos $[i, j]$ das estruturas M, I e D acessados por quaisquer dois *work-items* de um *warp* é maior que 128 bytes, portanto esses acessos caem em segmentos de memória diferentes e uma transação de memória é gerada para cada acesso. Dos 128 bytes transferidos, apenas 4 bytes são utilizados e 124 bytes representam desperdício de banda. Portanto, para cada acesso feito por um *warp*, há um desperdício de largura de banda de 3.968 bytes. Situação semelhante ocorre nos acessos às estruturas I e D .

Para permitir a coalescência de memória no acesso às estruturas M, I e D, elas são armazenadas de acordo com a organização da Figura 9.7, que mostra os acessos dos *work-items* de um *warp* ao primeiro elemento de M para um conjunto de 32 sequências. A ideia é similar àquela utilizada com os símbolos das sequências (Seção 9.1), e consiste na intercalação dos elementos de *scores* dos diferentes *work-items*. Dessa maneira, todos os acessos feitos por um *warp* caem no mesmo segmento de 128 bytes e podem portanto ser realizados com apenas uma transação, sem que haja desperdício de largura de banda.

As estruturas de *scores* B, C e J possuem duas posições por *work-item* e são armazenadas na memória global da GPU com a organização mostrada na Figura 7.3(b). A Figura 9.8 mostra o padrão de acessos dos *work-items* de um *warp* ao primeiro elemento de J para um conjunto de 32 sequências. Na figura, J_k representa a estrutura de *scores* J relativa à sequência S_k. Para atender essa solicitação, são necessárias duas transações, pois metade dos acessos caem em um segmento e a outra metade em outro. Entretanto, dos 256 bytes transferidos, apenas 128 bytes são utilizados e o restante corresponde a um desperdício de banda. Situação semelhante ocorre nos acessos às estruturas B e C.

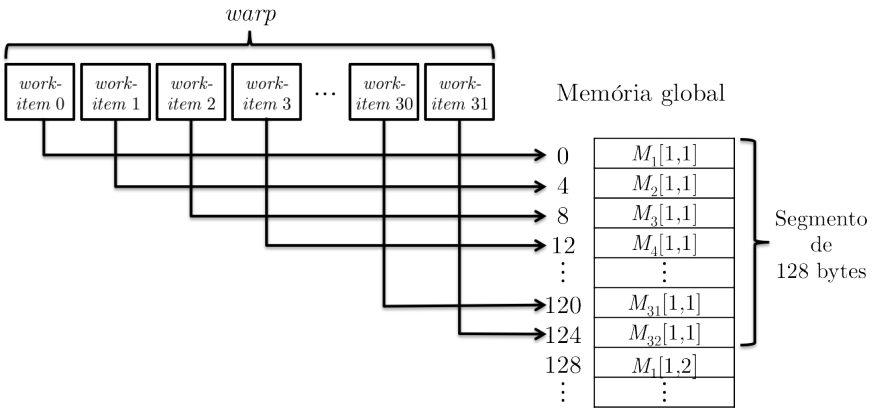


Figura 9.7: Acessos coalescidos de um *warp* ao primeiro elemento de M, para $S = \{S_1, ..., S_{32}\}$

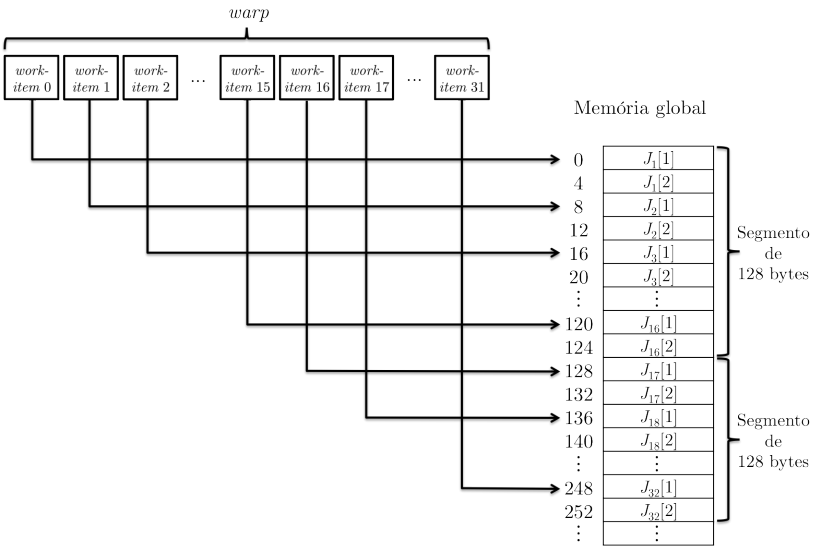


Figura 9.8: Acessos de um *warp* ao primeiro elemento de J, para $S = \{S_1, ..., S_{32}\}$

Esse desperdício também é evitado intercalando-se os *scores* referentes aos diferentes *work-items*. Dessa maneira, o padrão de acessos dos *work-items* de um *warp* ao primeiro elemento de J fica semelhante ao acesso a M mostrado na Figura 9.7, gerando apenas uma transação.

A Tabela 9.2 apresenta, na terceira coluna, os resultados da execução do acelerador com coalescência de memória no acesso às estruturas de *scores*, armazenadas na memória global da GPU, para o conjunto de 70.000 sequências da base Swiss-Prot e os HMMs *Top Twenty*. Em média, essa otimização reduz em 86% o tempo de execução em relação ao acelerador básico, confirmando que a coalescência de memória é uma otimização de alta prioridade quando se trabalha com GPUs [49].

Para o conjunto de sequências $S = \{S_1, S_2, ..., S_{31}\}$ e um HMM com $Q \geq 4$ nós, a Figura 9.9 mostra o padrão de acessos ao segundo elemento da estrutura M pelos *work-items* de um *warp* (linhas contínuas na figura). O último *work-item* do *warp* permanece inativo durante a execução. O acesso ao elemento $M_1[1, 2]$ cai no primeiro segmento de memória enquanto os demais acessos caem no segundo segmento. Portanto, para atender essas requisições são necessárias duas transações de memória. O mesmo acontece nos acessos pelos *work-items* ao terceiro elemento de M (linhas tracejadas na figura), que caem nos segundo e terceiro segmentos, gerando duas transações.

Na Figura 9.9 existem 31 sequências, mas se M fosse capaz de armazenar os *scores* de 32 sequências, que é o número de *work-items* de um *warp*, apenas uma transação seria suficiente para atender os acessos dos *work-items* a qualquer elemento de M, bem como o acesso a todas as outras posições de

Tabela 9.2: Tempo de execução dos aceleradores básico e com estruturas de *scores* na memória global com coalescência e *padding* e na memória exclusiva

Família	Tempo de execução (s)			
	Acelerador básico	Com *scores* na memória		
		global		exclusiva
		Com coalescência	Com coalescência e *padding*	
ABC_tran	97,34	13,44	12,03	4,48
Cytochrom_B_C	84,19	11,59	10,32	3,90
Cytochrom_B_N	155,96	22,26	19,58	7,17
Pkinase	217,05	30,82	27,30	9,93
Response_reg	91,88	12,78	11,32	4,26
RVP	82,33	11,51	10,20	3,83
RVT_1	176,21	25,12	22,20	8,20
zf-C2H2	17,80	2,58	2,27	0,93
adh_short	138,78	19,61	17,28	6,38
COX1	373,26	53,91	47,76	17,85
HTH_1	48,99	6,75	5,99	2,30
Helicase_C	63,80	8,90	7,83	2,99
Oxidored_q1	221,92	31,89	28,26	10,39
WD40	31,59	4,43	3,87	1,53
BPD_transp_1	154,34	21,73	19,07	7,08
Acetyltransf_1	68,15	9,47	8,35	3,21
HATPase_c	91,27	12,73	11,26	4,28
RVT_thumb	57,34	7,96	7,07	2,72
MFS_1	295,91	42,26	37,45	13,66
GP120	414,40	58,95	52,55	18,88
Média	144,13	20,43	18,10	6,70

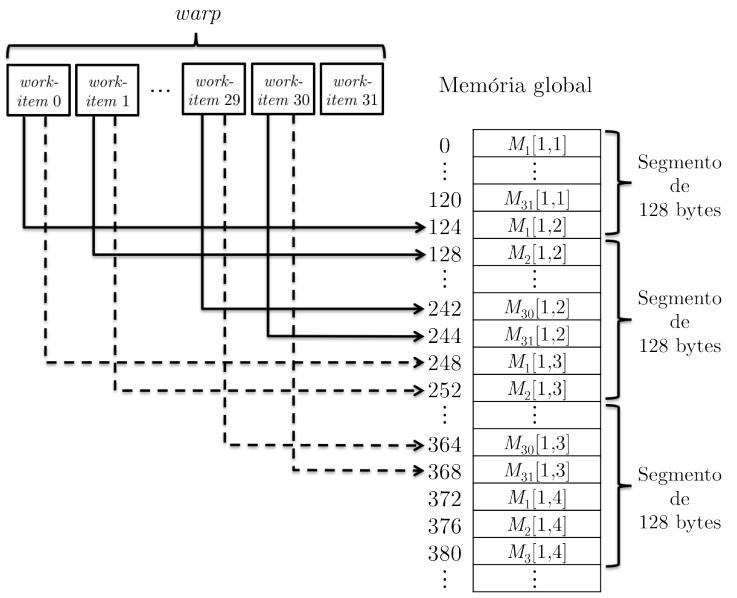


Figura 9.9: Acessos de um *warp* aos segundo e terceiro elementos de M, para $S = \{S_1, ..., S_{31}\}$ e $Q \geq 4$

M. A Figura 9.10 exemplifica a aplicação da técnica de *array padding*, através da inserção de elementos nulos em M (representados pelas células hachuradas), com o intuito de evitar transações de memória desnecessárias. Com *padding*, os acessos dos *work-items* 1 a 30 ao segundo elemento de M caem em um mesmo segmento, gerando uma única transação.

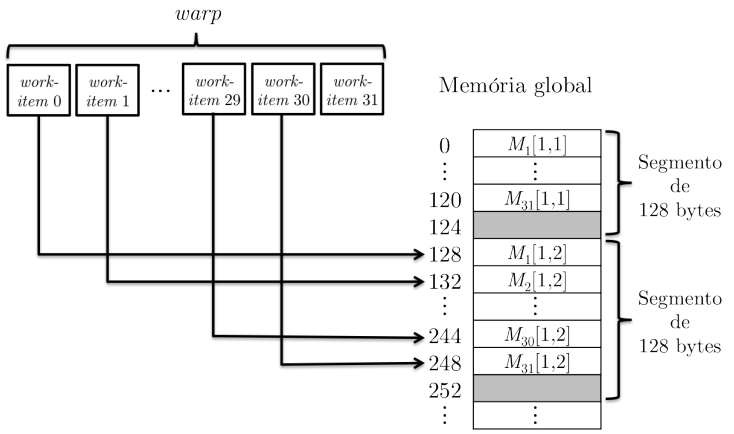


Figura 9.10: Acessos de um *warp* ao segundo elemento de M, com inserção de elementos nulos, para $S = \{S_1, ..., S_{31}\}$

Portanto, na execução do acelerador para N sequências, transações de memória desnecessárias no acesso à estrutura M são evitadas se ela for organizada para armazenar os *scores* de $\lceil N/32 \rceil \times 32$ sequências, através da inserção de elementos nulos. O mesmo pode ser feito com as estruturas I, D, B, C e J, ao custo de aumentar a quantidade de memória necessária para armazená-las.

A quarta coluna da Tabela 9.2 apresenta os resultados da execução do acelerador com coalescência de memória e *padding* no acesso às estruturas de *scores* na memória global, para o conjunto de 70.000 sequências da base Swiss-Prot e os HMMs *Top Twenty*. Essa otimização reduz, em média, 87,5% o tempo de execução, em relação ao acelerador básico.

Mesmo com *padding*, dependendo do tamanho e da distribuição das sequências pelo conjunto, pode haver desperdício de banda nos acessos às estruturas de *scores*. Esse desperdício é amenizado se o conjunto de sequências de entrada estiver ordenado de forma decrescente no tamanho das sequências. Isto é, a combinação da otimização de *padding* nas estruturas de *scores* com a ordenação das sequências, apresentada na Seção 9.1, pode trazer um ganho de desempenho ainda maior.

Outra possibilidade de armazenamento das estruturas de *scores* é na memória exclusiva da GPU, que embora seja mapeada na memória global, possui uma organização diferente. Para isso, para cada *work-item* são alocadas na memória exclusiva as estruturas M, I e D e B, C e J. Em todas elas, os *work-items* de um mesmo *warp* acessam o mesmo endereço relativo, portanto, os acessos são naturalmente coalescidos na memória exclusiva, sem necessidade da intercalação manual dos dados e simplificando os cálculos de índices das estruturas no acelerador.

A Tabela 9.2 também apresenta, na última coluna, os resultados da execução do acelerador com as estruturas de *scores* armazenadas na memória exclusiva, para o conjunto de 70.000 sequências da base Swiss-Prot e os HMMs *Top Twenty*. Essa otimização produz, em média, uma redução de 95% em relação ao acelerador básico, que é superior ao ganho obtido quando os *scores* são armazenados e acessados de maneira coalescida diretamente na memória global.

É possível combinar a otimização de armazenar as estruturas de *scores* na memória exclusiva com a ordenação das sequências de forma decrescente do tamanho, apresentada na Seção 9.1, obtendo um ganho de desempenho ainda maior.

9.3 Probabilidades de Transições Regulares

A Figura 9.11 mostra o padrão de acessos dos *work-items* de um *warp* ao elemento Pt_{M_1,M_2} da estrutura de probabilidade de transições regulares $Pt_{regulares}$, armazenada na memória global da GPU de acordo com a organização da Figura 7.3(d). Na execução do algoritmo de Viterbi apresentado no Algoritmo 7.1, essas probabilidades são tão acessadas quanto as estruturas de *scores*, porém apenas para leitura e com um padrão de acessos diferente, que facilita a aplicação de uma otimização de memória.

A Figura 9.11 mostra que todos os *work-items* do *warp* acessam a mesma posição da memória global referente à probabilidade de transição desejada. Ler da memória constante é tão rápido quanto ler de um registrador, contanto que todos os *work-items* leiam do mesmo endereço [49], que é exatamente o padrão de acessos a essa estrutura.

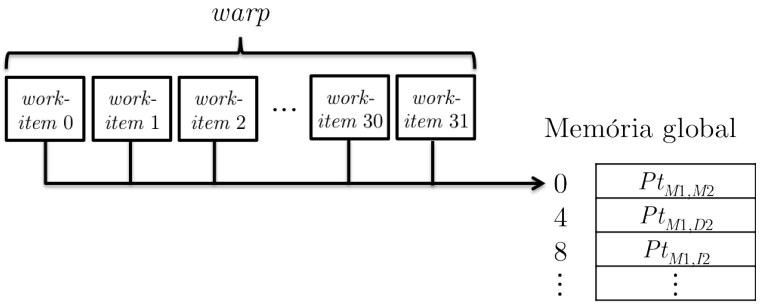


Figura 9.11: Acessos de um *warp* ao primeiro elemento de $Pt_{regulares}$

A Tabela 9.3 apresenta, na terceira coluna, os resultados da execução do acelerador com as probabilidades de transições regulares $Pt_{regulares}$ armazenadas na memória constante da GPU, para o conjunto de 70.000 sequências da base Swiss-Prot e os HMMs *Top Twenty*. Em média, há uma pequena melhora de desempenho em relação ao acelerador básico.

Outra possibilidade de alocação da estrutura $Pt_{regulares}$ é na memória local da GPU, compartilhada entre os *work-items* de um *work-group*. Além de ter um tempo de acesso menor que a memória global, quando os *work-items* de um *warp* acessam uma mesma posição da memória local, é realizado um *broadcast* do dado para os *work-items* [49].

A cópia de dados para a memória local deve ser feita pelos próprios *work-items*, a partir da memória global, no início da execução do *kernel*, da seguinte maneira:

- Para cada *work-group* haverá uma cópia de $Pt_{regulares}$ na memória local correspondente;
- Para evitar a serialização dos acessos dos diferentes *work-groups* à mesma posição da memória global, a estrutura $Pt_{regulares}$ é replicada também na memória global, com uma cópia para cada *work-group*;
- Cada cópia é organizada na memória global com um tamanho múltiplo de 128 bytes, para evitar transações de memória desnecessárias;
- A cópia é realizada com os *work-items* lendo da memória global posições consecutivas de $Pt_{regulares}$, para que os acessos sejam coalescidos;
- É necessária uma barreira de sincronização após a cópia, para garantir que a execução do *kernel* prossiga somente após os *warps* de um *work-group* terem copiado toda a estrutura para a memória local.

A Tabela 9.3 apresenta, na última coluna, os resultados da execução do acelerador com a estrutura $Pt_{regulares}$ armazenada na memória local da GPU, para o conjunto de 70.000 sequências

Tabela 9.3: Tempo de execução dos aceleradores básico e com $Pt_{regulares}$ na memória constante e na memória local

Família	Tempo de execução (s)		
	Acelerador básico	Com $Pt_{regulares}$ na memória	
		constante	local
ABC_tran	97,34	96,91	96,96
Acetyltransf_1	68,15	68,49	68,27
adh_short	138,78	137,63	137,86
BPD_transp_1	154,34	154,56	153,90
COX1	373,26	369,81	369,46
Cytochrom_B_C	84,19	84,32	84,22
Cytochrom_B_N	155,96	156,23	154,64
GP120	414,40	409,09	397,83
HATPase_c	91,27	90,99	90,99
Helicase_C	63,80	63,90	63,62
HTH_1	48,99	49,23	49,00
MFS_1	295,91	293,04	292,26
Oxidored_q1	221,92	220,45	219,53
Pkinase	217,05	216,64	216,34
Response_reg	91,88	91,76	91,91
RVP	82,33	82,28	82,39
RVT_1	176,21	175,41	174,81
RVT_thumb	57,34	57,70	57,49
WD40	31,59	31,61	31,47
zf-C2H2	17,80	17,89	17,77
Média	144,13	143,40	142,54

da base Swiss-Prot e os HMMs *Top Twenty*. Essa otimização também traz um pequeno ganho de desempenho em relação ao acelerador básico. Entretanto, ela ocasiona um maior uso da memória global (devido à replicação de $Pt_{regulares}$), além de inibir outras otimizações que utilizem a memória local e proporcionem um maior ganho de desempenho, dado que esta memória é muito pequena.

9.4 Probabilidades de Transições Especiais

O padrão de acessos dos *work-items* de um *warp* à estrutura de probabilidades de transição especiais $Pt_{especiais}$ é análogo ao da estrutura $Pt_{regulares}$, apresentado na Figura 9.11, entretanto, $Pt_{especiais}$ é menos acessada. No Algoritmo 7.1, $Pt_{especiais}$ é acessada 10 vezes a cada iteração do laço externo (que itera sobre os símbolos da sequência) do algoritmo de Viterbi, por cada *work-item*.

Uma otimização natural é armazenar $Pt_{especiais}$ na memória constante da GPU. No entanto, os resultados da execução do acelerador com essa otimização não apresentam, na média, ganho de desempenho em relação ao acelerador básico, possivelmente por a estrutura ser pouco acessada.

É possível ainda armazenar $Pt_{especiais}$ na memória local da GPU, com algumas diferenças em relação à estrutura $Pt_{regulares}$:

- Os dados de $Pt_{especiais}$ são copiados para a memória local de maneira coalescida pelos *work-items*. Como essa estrutura contém apenas oito posições, alguns *work-items* do mesmo *work-group* permanecem inativos.
- Não há replicação de $Pt_{especiais}$ na memória global para cada *work-group*, uma vez que $Pt_{especiais}$

é muito pequena e a cópia gera poucas leituras na memória global;

- Há uma cópia de $Pt_{especiais}$ na memória local para cada *warp*. Dessa maneira, não há necessidade de barreiras de sincronização entre os *work-items* de um *work-group*;
- Podem existir conflitos de banco de memória nos acessos às posições de $Pt_{especiais}$ na memória local pelos *work-items* de um *warp*. No entanto esses conflitos não devem ter um impacto muito negativo no desempenho e evitá-los não é uma otimização de alta prioridade [49].

A Tabela 9.4 apresenta, na terceira coluna, os resultados da execução do acelerador com as probabilidades de transições especiais $Pt_{especiais}$ armazenadas na memória local da GPU, para o conjunto de 70.000 sequências da base Swiss-Prot e os HMMs *Top Twenty*. Há um pequeno ganho de desempenho em todas as execuções em relação ao acelerador básico. Em média, o ganho de desempenho dessa otimização foi melhor que o obtido com o armazenamento de $Pt_{regulares}$ na memória local, embora $Pt_{regulares}$ seja muito mais acessada que $Pt_{especiais}$. Esse fato se deve, principalmente, à necessidade da barreira de sincronização ao fim da cópia de $Pt_{regulares}$ da memória global para a memória local, o que não acontece com $Pt_{especiais}$, que por ser muito pequeno, pode ser replicado por *warp* na memória local.

Tabela 9.4: Tempo de execução dos aceleradores básico, com $Pt_{especiais}$ na memória local e na global, replicado e com coalescência

Família	Tempo de execução (s)		
	Acelerador básico	Com $Pt_{especiais}$ na memória	
		local	global, replicado e intercalado
ABC_tran	97,34	94,63	94,85
Acetyltransf_1	68,15	67,03	66,85
adh_short	138,78	135,56	135,62
BPD_transp_1	154,34	151,60	151,77
COX1	373,26	366,17	363,70
Cytochrom_B_C	84,19	82,40	82,49
Cytochrom_B_N	155,96	152,76	152,83
GP120	414,40	403,31	403,84
HATPase_c	91,27	89,01	80,09
Helicase_C	63,80	62,45	62,47
HTH_1	48,99	47,71	47,80
MFS_1	295,91	288,70	288,81
Oxidored_q1	221,92	217,45	217,00
Pkinase	217,05	214,07	213,74
Response_reg	91,88	89,82	89,82
RVP	82,33	80,79	80,74
RVT_1	176,21	172,41	172,21
RVT_thumb	57,34	56,08	56,28
WD40	31,59	30,74	30,74
zf-C2H2	17,80	17,04	17,05
Média	144,13	140,99	140,44

No acelerador básico, $Pt_{especiais}$ está alocada na memória global e não há coalescência de memória nos acessos às suas posicões, pois todos os *work-items* acessam a mesma posição em um mesmo instante de tempo. Como $Pt_{especiais}$ é bastante pequena, a coalescência de memória pode ser obtida da seguinte forma:

- $Pt_{especiais}$ é replicada na memória global com cópia para cada *work-item*;

- Os elementos de $Pt_{especiais}$ são intercalados na memória global de acordo com a Figura 9.12, permitindo a coalescência dos acessos pelos *work-items* de um *warp* a cada elemento.

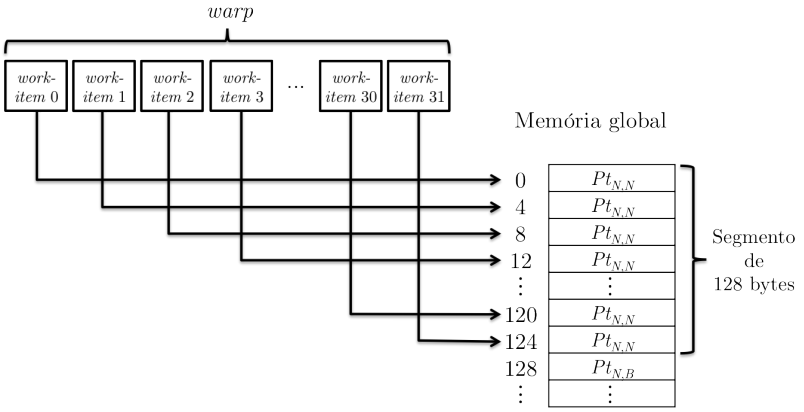


Figura 9.12: Acessos de um *warp* ao primeiro elemento de $Pt_{especiais}$ replicado e intercalado na memória global, para $S = \{S1, ..., S_{32}\}$

A última coluna da Tabela 9.4 apresenta os resultados da execução do acelerador com a estrutura $Pt_{especiais}$ armazenada na memória global, replicada e com acessos coalescidos, para o conjunto de 70.000 sequências da base Swiss-Prot e os HMMs *Top Twenty*. A replicação e intercalação de $Pt_{especiais}$ na memória global traz um ganho de desempenho similar ao armazenamento de $Pt_{especiais}$ na memória local, mostrando que, remover acessos concorrentes dos *work-items* de um mesmo *warp* a uma mesma posição da memória global, através de replicação e coalescência dos acessos, pode trazer ganhos de desempenho.

9.5 Probabilidades de Emissão

No laço interno (que itera sobre os nós do HMM) do algoritmo de Viterbi, as probabilidades de emissão Pe_M e Pe_I do símbolo s_i da sequência são acessadas sequencialmente para cada nó j do HMM, com $1 \leq j \leq Q$. Com essas estruturas armazenadas na memória global com a organização apresentada na Figura 3.4, dois acessos consecutivos feitos por um *work-item* a uma dessas estruturas estão distantes 20 posições (80 bytes) um do outro, pois $|\Sigma| = 20$, onde Σ é o alfabeto dos símbolos das sequências de proteínas.

Esse padrão de acessos à memória global não apresenta localidade espacial, causando mais falhas nos acessos a *cache* da memória global. Portanto, as estruturas Pe_M e Pe_I são armazenadas de maneira transposta na memória global, como mostra a Figura 9.13. Assim, dois acessos consecutivos feitos por um *work-item* caem em posições consecutivas da memória. Essa organização apresenta mais localidade espacial na memória global, porém não há ganho de desempenho significativo do acelerador com essa otimização.

No Algoritmo 7.1, as estruturas de probabilidades de emissão Pe_M e Pe_I são acessadas uma vez cada uma a cada iteração do laço interno do algoritmo de Viterbi, por cada *work-item*. Entretanto, a posição na estrutura acessada por cada *work-item* de um *warp* depende dos símbolos da sequência de entrada atribuída ao *work-item*.

Portanto, se Pe_M e Pe_I são armazenados na memória global, não há como prever se as posições acessadas pelos *work-items* de um *warp* caem ou não no mesmo segmento de memória e geram ou não acessos coalescidos à memória global. A memória local da GPU é a mais adequada para o padrão de acessos de Pe_M e Pe_I, pois nela não há necessidade de coalescer os acessos.

Embora não tenha trazido ganho de desempenho na memória global, a organização transposta de

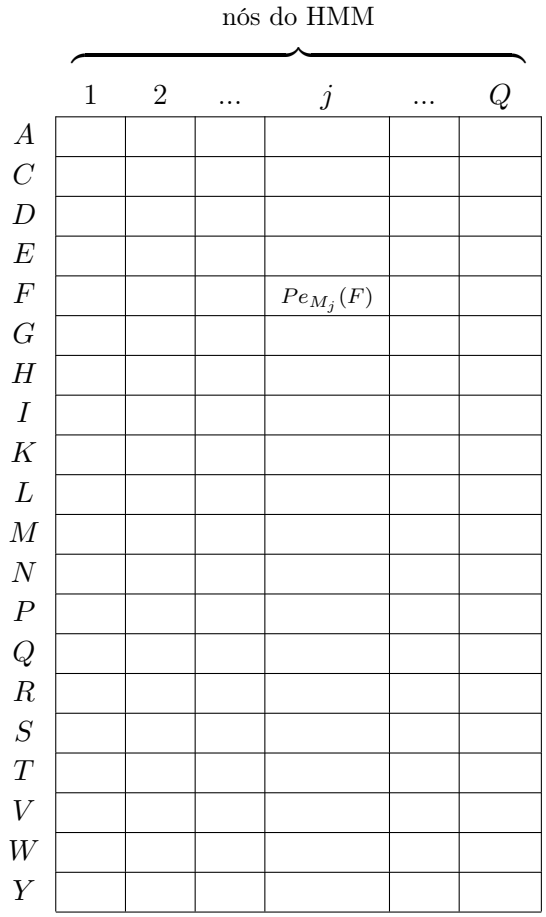


Figura 9.13: Probabilidades de emissão Pe_M e Pe_I organizadas para apresentar localidade espacial nos acessos

Pe_M e Pe_I é mantida para que os *work-items* copiem essas estruturas de maneira coalescida da memória global para a memória local. Durante a execução do acelerador, todos os acessos às probabilidades de emissão são realizados na memória local. A técnica de *array padding* é aplicada de forma que a alocação de Pe_M e Pe_I na memória local é feita arredondando o número de nós Q para $\lceil Q/33 \rceil \times 33$. Assim, não ocorrem conflitos nos 32 bancos da memória local, nos acessos feitos por quaisquer dois *work-items* k e r de um mesmo *warp* às probabilidades de emissão $Pe_{M_j}(S_k[i])$ e $Pe_{M_j}(S_r[i])$ de diferentes símbolos $S_k[i]$ e $S_r[i]$ para um mesmo nó j do HMM.

A Tabela 9.5 apresenta os resultados da execução do acelerador com as probabilidades de emissão Pe_M e Pe_I armazenadas na memória local, para o conjunto de 70.000 sequências da base Swiss-Prot e os HMMs *Top Twenty*. Quanto maior o número de nós do HMM, mais acessos ocorrem às probabilidades de emissão, e em consequência, maior o ganho de desempenho ao utilizar a memória local sem conflitos de banco para essas estruturas ao invés da memória global sem acessos coalescidos (utilizada pelo acelerador básico).

Os valores não disponíveis indicam que não foi possível executar o acelerador devido à falta de recursos, nesse caso, causada pelo tamanho das estruturas das probabilidades de emissão dos HMMs, que superam o tamanho da memória local, o que restringe a aplicação dessa otimização. Os HMMs que não puderam ser executados foram desconsiderandos no cálculo da média do acelerador básico e do acelerador com Pe_M e Pe_I na memória local.

Tabela 9.5: Tempo de execução dos aceleradores básico e com Pe_M e Pe_I na memória local

Família	Tempo de execução (s)	
	Acelerador básico	Com Pe_M e Pe_I na memória local
ABC_tran	97,34	94,05
Acetyltransf_1	68,15	68,03
adh_short	138,78	127,28
BPD_transp_1	154,34	138,76
COX1	373,26	– *
Cytochrom_B_C	84,19	81,88
Cytochrom_B_N	155,96	141,91
GP120	414,40	– *
HATPase_c	91,27	88,37
Helicase_C	63,80	63,35
HTH_1	48,99	48,56
MFS_1	295,91	– *
Oxidored_q1	221,92	199,67
Pkinase	217,05	201,31
Response_reg	91,88	88,94
RVP	82,33	81,49
RVT_1	176,21	157,78
RVT_thumb	57,34	57,00
WD40	31,59	31,36
zf-C2H2	17,80	17,81
Média	105,82	99,27

* Recursos da GPU insuficientes para a execução.

Capítulo 10

Acelerador Otimizado em GPU

Este capítulo apresenta o acelerador em GPU final, criado a partir das otimizações apresentadas anteriormente, e realiza avaliações experimentais com o mesmo. Para esse acelerador otimizado, a influência da escolha do tamanho dos *work-groups* na ocupação da GPU e o no desempenho é analisada. Por último, para investigar a influência da granularidade do *kernel* do algoritmo de Viterbi, dois outros aceleradores com abordagens de menor granularidade são propostos e avaliados.

10.1 Combinação das Otimizações

A Tabela 10.1 mostra o tempo médio de execução dos aceleradores com as otimizações propostas nos Capítulos 8 e 9, para o conjunto de 70.000 sequências da base Swiss-Prot e os HMMs *Top Twenty*, assim como o *speedup* obtido por eles em relação ao acelerador básico.

Tabela 10.1: Tempo médio de execução dos aceleradores com otimizações e *speedup* em relação ao acelerador básico

Acelerador / Otimização	Seção	Tempo médio de execução (s)	*Speedup*
Básico	7.3	144,13	1,0
Escalonamento de instruções	8.1	131,98	1,09
Loop unrolling com fator 8	8.2	139,92	1,03
Overlapping	8.3	145,47	0,99
Sequências na memória global, com coalescência e ordenação	9.1	132,00	1,09
Scores na memória global, com coalescência e *padding*	9.2	18,10	7,96
Scores na memória exclusiva	9.2	6,70	21,51
$Pt_{regulares}$ na memória constante	9.3	143,40	1,01
$Pt_{regulares}$ na memória local, replicado por *work-group*	9.3	142,54	1,01
$Pt_{especiais}$ na memória local, replicado por *warp*	9.4	140,99	1,02
$Pt_{especiais}$ na memória global, replicado por *work-item* e com coalescência	9.4	140,44	1,03
Pe_M e Pe_I na memória local, com transposição e *padding*	9.5	99,27	1,45

Para construir o acelerador otimizado final, as otimizações que produzem ganho de desempenho são acrescentadas uma a uma, a partir do acelerador básico, e avaliadas. Nem todas as otimizações são aplicadas, dado que algumas são excludentes entre si, pois atuam sobre a mesma estrutura de dados.

A ordem adotada para inclusão das otimizações, baseada no ganho de desempenho proporcionado por elas, é:

1. Estruturas de *scores* armazenadas na memória exclusiva, com acessos naturalmente coalescidos;
2. *Loop unrolling* com fator 8;
3. Escalonamento de instruções;
4. Sequências armazenadas na memória global, com acessos coalescidos e ordenação das sequências de forma descrescente pelo tamanho;
5. Probabilidades de emissão Pe_M e Pe_I armazenadas na memória local, de forma transposta e com *padding*;
6. Probabilidades de transições especiais $Pt_{especiais}$ armazenadas na memória local, replicadas para cada *warp*;
7. Probabilidades de transições regulares $Pt_{regulares}$ armazenadas na memória constante.

A Tabela 10.2 mostra os resultados da execução dos aceleradores criados a partir da aplicação das otimizações no acelerador básico, para o conjunto de 70.000 sequências da base Swiss-Prot e o HMM correspondente à família Oxidored_q1 da Pfam. As versões são identificadas pelos números dos itens apresentados na lista anterior.

Tabela 10.2: Tempo de execução do acelerador com a inclusão das otimizações a partir do acelerador básico

Acelerador / Otimizações	Tempo de execução (s)
Básico	220,85
1	10,60
1 + 2	11,94
1 + 2 + 3	8,04
1 + 3	10,40
1 + 2 + 3 + 4	5,20
1 + 2 + 3 + 4 + 5	6,25
1 + 2 + 3 + 4 + 6	5,17
1 + 2 + 3 + 4 + 6 + 7	4,36

Algumas conclusões importantes podem ser extraídas dessa tabela. Primeiramente, o ganho de desempenho obtido com a otimização 1 é muito significativo, pois envolve a coalescência nos acessos às estruturas de *scores*, que são as mais acessadas pelo *kernel*.

A inserção da otimização 2, de *loop unrolling*, causa uma pequena perda de desempenho, que é eliminada com a inserção da otimização 3, escalonamento de instruções. Para descobrir se o ganho da otimização 3 poderia ser maior em um acelerador sem a otimização 2, a versão 1 + 3 é criada, e o desempenho obtido é pior que aquele da versão 1 + 2 + 3. Isso mostra que, embora a otimização 2 cause perda de desempenho quando implementada sem a otimização 3, ela potencializa o ganho de desempenho desta otimização. Ou seja, o *loop unrolling* permite que o escalonamento de instruções aumente a distância entre as instruções dependentes entre si.

A otimização 4, que muito embora trate apenas dos acessos aos símbolos das sequências, causa um bom ganho de desempenho, decorrente também da ordenação das sequências, que reduz o desperdício de banda no acesso coalescido às matrizes de *score*.

A otimização 5, relativa às probabilidades de emissão, melhora o desempenho quando aplicada diretamente no acelerador básico, mas quando inserida na versão 1 + 2 + 3 + 4 piora o desempenho.

A explicação para este fato é que a otimização 5 utiliza a memória local, o que influencia diretamente a ocupação da GPU.

A Tabela 10.3 mostra a influência na ocupação da GPU causada pela inserção da otimização 5 no acelerador básico e na versão 1 + 2 + 3 + 4 para o conjunto de 70.000 sequências da base Swiss-Prot e o HMM correspondente à família Oxidored_q1 da Pfam. O acelerador básico tem ocupação de 50% que, após a aplicação da otimização 5, cai para 17%, mas ainda assim há melhora no tempo de execução. Ou seja, a piora na ocupação não causa piora de desempenho, mesmo quando a ocupação fica abaixo dos 18.75%, mínimo sugerido nas orientações da NVIDIA [49]. Durante os experimentos, notou-se que isso geralmente acontece quando uma otimização de memória é inserida em uma versão com pouca ou nenhuma otimização de memória, fazendo com que o tempo de execução melhore mesmo quando a ocupação da GPU diminui.

Tabela 10.3: Mudança de ocupação causada pela inserção da otimização 5

Acelerador / Otimizações	Número de registradores	Quantidade de memória local (KB)	Ocupação da GPU	Tempo de execução (s)
Básico	37	0	50%	221,92
5	37	47520	17%	217,00
1 + 2 + 3 + 4	62	0	33%	5,20
1 + 2 + 3 + 4 + 5	63	47520	17%	6,25

Quando a otimização 5 é inserida na versão 1 + 2 + 3 + 4, cuja ocupação é de 33%, a ocupação cai para 17% e o tempo de execução piora. Na versão 1 + 2 + 3 + 4, o acesso às principais estruturas de dados já está otimizado, demonstrando que, nessas situações, a diminuição da ocupação pode implicar em uma piora de desempenho.

A inserção da otimização 6, que trata das probabilidades de transições especiais, na versão 1 + 2 + 3 + 4 causa um pequeno ganho, próximo proporcionalmente do ganho obtido na aplicação dessa otimização no acelerador básico. Já a inserção da otimização 7, relativa às probabilidades de transições regulares, na versão 1 + 2 + 3 + 4 + 6 proporciona um ganho muito mais significativo do que quando aplicada ao acelerador básico, mostrando que otimizações envolvendo o uso da memória local podem ser mais impactantes quando boa parte dos acessos à memória já estão otimizados.

10.2 Resultados Finais

O acelerador otimizado final contém as otimizações 1 + 2 + 3 + 4 + 6 + 7 e é executado para o conjunto de 70.000 sequências da base Swiss-Prot, ordenadas de forma decrescente no tamanho, e todos os HMMs *Top Twenty*. A ordenação das sequências executada no *host* consome 0.44 segundos e é realizada apenas uma vez, antes da execução do acelerador para todos os HMMs. A Tabela 10.4 mostra, nas quatro primeiras colunas, os resultados obtidos. O tempo de execução do acelerador otimizado é comparado ao tempo consumido pela ferramenta HMMer2 quando executada no *host*. O *speedup* do acelerador em relação ao HMMer2 varia entre 46,47 e 115,64, com média de 52,77.

Para uma avaliação mais completa do acelerador, a base de sequências Swiss-Prot inteira é utilizada. A base é ordenada em 4,63 segundos e dividida em lotes (*batches*) de 70.000 sequências, que são executados no acelerador um após o outro, para todos os HMMs *Top Twenty*. A divisão em lotes é necessária devido às restrições de tamanho da memória da GPU utilizada, que impedem que o acelerador seja executado para todas as sequências dessa base de uma só vez. Os tempos de execução para cada lote são somados para compor o tempo final para cada HMM. A Tabela 10.4 mostra nas cinco últimas colunas os resultados das execuções do acelerador otimizado e do HMMer2 (executado no *host*) para a base Swiss-Prot completa, comparando os tempos de execução obtidos e a quantidade de GCUPS (10^9 *Cell Updates Per Second*) atingida por cada um.

O *speedup* do acelerador em relação ao HMMer2 varia entre 40,35 e 102,83, com média de 48,82,

Tabela 10.4: Tempo de execução e CUPS do acelerador otimizado e do HMMer2 e *speedup* do acelerador em relação ao HMMer2

Família	Conjunto de 70.000 sequências			Base Swiss-Prot completa				
	Tempo de execução (s)		*Speedup*	Tempo de execução (s)		*Speedup*	GCUPS	
	Acelerador otimizado	HMMer2		Acelerador otimizado	HMMer2		Acelerador otimizado	HMMer2
ABC_tran	1,90	95,15	49,99	7,52	324,45	43,14	9,07	0,21
Cytochrom_B_C	1,64	81,60	49,51	6,52	280,32	43,02	9,06	0,21
Cytochrom_B_N	3,03	156,57	51,60	12,01	512,98	42,71	9,00	0,21
Pkinase	4,19	205,73	49,03	16,62	1691,02	101,75	8,97	0,09
Response_reg	1,79	90,63	50,57	7,05	725,42	102,83	9,18	0,09
RVP	1,60	80,07	49,79	6,36	286,24	44,97	9,11	0,20
RVT_1	3,45	170,44	49,32	13,69	588,88	43,01	8,97	0,21
zf-C2H2	0,42	19,57	46,47	1,68	67,78	40,35	8,36	0,21
adh_short	2,71	140,04	51,54	10,81	457,07	42,28	8,89	0,21
COX1	7,30	356,08	48,76	28,98	1242,28	42,86	8,82	0,21
HTH_1	0,98	48,48	49,10	3,89	165,17	42,42	9,03	0,21
Helicase_C	1,27	147,45	115,64	5,04	213,58	42,40	9,00	0,21
Oxidored_q1	4,36	214,72	49,17	17,28	736,80	42,65	8,96	0,21
WD40	0,67	31,88	47,00	2,68	109,72	40,91	8,64	0,21
BPD_transp_1	3,00	146,53	48,78	11,90	504,79	42,42	8,94	0,21
Acetyltransf_1	1,35	66,78	49,33	4,80	227,72	47,45	10,05	0,21
HATPase_c	1,80	90,36	50,10	7,13	303,50	42,55	9,00	0,21
RVT_thumb	1,15	59,40	51,62	4,54	191,90	42,23	8,99	0,21
MFS_1	5,74	280,46	48,81	22,74	987,44	43,42	8,91	0,21
GP120	7,90	389,84	49,31	31,41	1351,80	43,04	8,88	0,21
Mínimo	0,42	19,57	46,47	1,68	67,78	40,35	8,36	0,09
Média	2,81	143,59	52,77	11,13	548,44	48,82	8,99	0,20
Máximo	7,90	389,84	115,64	31,41	1691,02	102,83	10,05	0,21

que é um pouco menor que a média obtida para o conjunto de apenas 70.000 sequências. Na base Swiss-Prot completa existem sequências de tamanho muito pequeno que, mesmo em um computador convencional, executam o algoritmo de Viterbi rapidamente para os HMMs utilizados, diminuindo o ganho do acelerador em GPU para algumas execuções.

O CUPS do acelerador mantém-se regular em todas das execuções, com média de 8,99 GCUPS e variando entre 8,36 e 10,05 GCUPS, enquanto o HMMer2 obtém em média 0,20 GCUPS. O CUPS atingido pelo HMMer2 para os HMMs das famílias Pkinase e Response_reg é muito distante do CUPS das demais execuções, ambos com 0,09 GCUPS. Provavelmente a piora nessas duas execuções é causada por um mau desempenho nos acessos à hierarquia de memórias do *host*. Os resultados do HMMer para essas duas famílias explicam os *speedups* anômalos de 101,75 e 102,83 obtidos pelo acelerador.

A Tabela 10.5 mostra o uso de cada uma das memórias da GPU pelo acelerador executando com o HMM da família GP120, que é o maior (com maior número de nós) dentre os *Top Twenty* da Pfam. A memória exclusiva é mapeada na memória global e, somando o uso de ambas, 839,06 MB são alocados, representando um uso de 82% do total disponível. Dentre todas as estruturas de dados, as que mais demandam memória são as estruturas de *scores* M, I e D.

A memória local é pouco utilizada pelo acelerador otimizado. A otimização 5, que a utiliza para alocar as probabilidades de emissão, não é inserida no acelerador otimizado, pois o tamanho dessa memória é insuficiente para os HMMs maiores e a inserção da otimização influencia negativamente na ocupação da GPU e no desempenho do acelerador. Em relação a memória de constantes, há um total de 64 KB disponíveis, dos quais 17,19 KB são usados, representando um uso de 27%.

A Tabela 10.6 compara o acelerador em GPU desenvolvido neste trabalho com as principais implementações do algoritmo de Viterbi em GPU, descritas no Capítulo 4. As implementações

Tabela 10.5: Uso das memórias da GPU pelo acelerador otimizado

Memória	Estrutura	Tamanho	Total
Global	Sequências	53,05 MB	53,39 MB
	$Pt_{especiais}$	0,03 KB	
	Pe_M e Pe_I	76,40 KB	
	Scores finais	273,49 KB	
Constante	$Pt_{regulares}$	17,19 KB	17,19 KB
Local	$Pt_{especiais}$	0,25 KB	0,25 KB
Exclusiva	*Scores* M, I e D	783,81 MB	785,67 MB
	Scores B, C e J	1,86 MB	

propostas por ClawHmmer [29] e Du *et al.* [14] não implementam a arquitetura Plan7 completa, e ainda assim ambas alcançam um *speedup* máximo inferior ao obtido pelo acelerador otimizado desenvolvido neste trabalho.

Tabela 10.6: Principais implementações do algoritmo de Viterbi em GPU e acelerador otimizado desenvolvido

Implementação	Arquitetura do HMM	GPU	Processador base	*Speedup*
[29]	M, I e D	ATI R520	Intel Pentium 4	36 (máximo)
[75]	Plan7	NVIDIA GeForce GTX 8800 Ultra	AMD Athlon 275	12 a 38,6
[69]	Plan7	NVIDIA GeForce GTX 280	Intel Q8200	12 (médio)
[84]	Plan7	NVIDIA GeForce GTX 8800 e *host* Intel Core2 E7200	AMD Athlon64 X2 Dual Core 3800+	13 a 45
[14]	M, I e D	NVIDIA Geforce 9800 GTX	Intel Dual Core	1,97 a 72,21
[24]	Plan7	4 NVIDIA Tesla C1060	AMD Opteron	100
Acelerador otimizado	Plan7	NVIDIA GeForce GTX 460	AMD Athlon II X3	40,35 (mínimo) 48,82 (médio) 102,83 (máximo)

GPU-HMMER [75] e CuHMMer [84] propõem soluções que implementam a arquitetura Plan7 completa e alcançam um *speedup* máximo inferior *speedup* médio obtido pelo acelerador proposto neste trabalho. Ganesan *et al.* [24] também propõem uma solução que implementa a arquitetura Plan7 completa e que alcança um *speedup* superior ao *speedup* médio do acelerador aqui proposto. Entretanto, sua execução utiliza quatro GPUs NVIDIA Tesla C1060, cada uma com 240 núcleos operando a 1,3 GHz e 4 GB de memória, enquanto este trabalho utiliza apenas uma GPU com 336 núcleos operando a 1,35 GHz e 1 GB de memória.

10.3 Ocupação da GPU

Para analisar a influência da ocupação da GPU no desempenho da solução em GPU, o acelerador otimizado é executado para o conjunto de 70.000 sequências da base Swiss-Prot e o HMM da família GP120, para diferentes valores para o número de *work-items* por *work-group*. Como a ocupação da GPU é a razão entre o número de *warps* ativos (disponíveis para execução) e o número máximo possível de *warps* ativos, em um *Streaming Multiprocessor*, é razoável supor que quanto maior o número de *work-items* por *work-group*, mais *warps* ativos haverá e, em consequência, maior será a ocupação.

A Figura 10.1 mostra o tempo de execução do acelerador e a ocupação da GPU obtida, variando-se o número de *work-items* por *work-group* de 32 em 32, até 512. A ocupação é calculada utilizando a planilha de ocupação da NVIDIA. O aumento da ocupação causa, em geral, melhora no desempenho do acelerador, exceto quando o número de *work-items* por *work-group* aumenta de 160 para 192, pois a ocupação aumenta mas ainda assim há uma piora no desempenho. Esse resultado ratifica o

que experimentos anteriores mostraram, que o aumento da ocupação da GPU não necessariamente é acompanhado pela melhora de desempenho. Portanto, a medida ocupação deve ser utilizada com cuidado na avaliação de soluções em GPU.

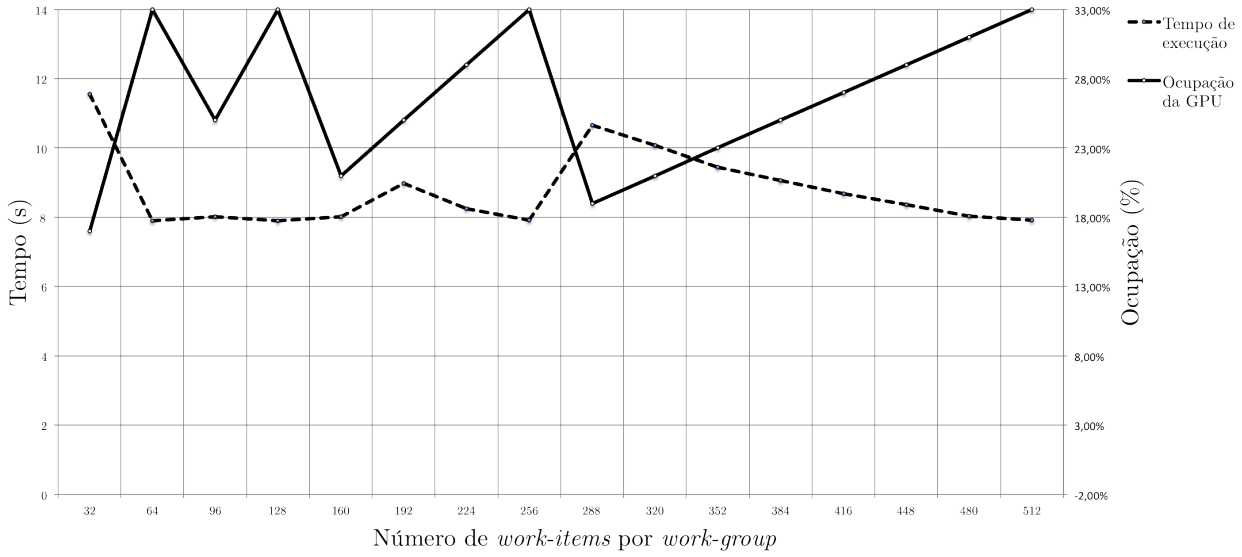


Figura 10.1: Tempo de execução do acelerador otimizado e ocupação da GPU obtida

A figura também mostra que o número de 256 *work-items* por *work-group*, utilizado nos experimentos anteriores para a execução do acelerador otimizado, produz a melhor ocupação da GPU (33%) e também o melhor desempenho.

10.4 Variação da Granularidade do *Kernel*

Para avaliar o impacto da granularidade do *kernel* no desempenho de implementações do algoritmo de Viterbi em GPU, outros dois aceleradores de menor granularidade, chamados de *acelerador de granularidade média* e *acelerador de granularidade fina*, são desenvolvidos, e as otimizações analisadas para o acelerador de granularidade grossa são neles aplicadas.

Dados um conjunto de sequências $S = \{S_1, S_2, ..., S_N\}$ e um HMM, no acelerador de granularidade média, *grids* com N *work-items* são instanciados sucessivamente, cada *work-item* k calculando r linhas das matrizes M, I e D e r elementos dos vetores B, C e J para a sequência $S_k \in S$, onde $1 \leq r \leq \max_{1 \leq k \leq N} |S_k|$. *Grids* são invocados sequencialmente para calcular as linhas de M, I e D e as posições de B, C e J das sequências, até que todos os *scores* sejam calculados.

A intenção dessa abordagem é fazer com que *work-items* de um *grid* calculem as mesmas quantidades de elementos e analisar se essa distribuição homogênea de carga melhora o desempenho. Em termos de memória, ela demanda a mesma quantidade que o acelerador de granularidade grossa.

Apenas a otimização que armazena as probabilidades de transições especiais na memória local não é aplicada, pois as probabilidades precisam ser copiadas da memória global para a local a cada criação de *grid*, o que inviabiliza essa otimização, dado que as probabilidades são acessadas poucas vezes no ciclo de vida de um *work-item*.

Uma otimização adicional é aplicada com o intuito de minimizar as transferências de dados entre o *host* e a GPU. Como cada *work-item* instanciado calcula r linhas das estruturas de *scores*, é necessário que ele saiba a partir de qual linha deve realizar o cálculo. Essa informação pode ser armazenada de três maneiras diferentes:

1. Antes da criação de cada *grid*, o *host* copia para a memória global da GPU o índice da linha inicial

das estruturas de *scores* e, no decorrer da execução, todos os *work-items* lêem essa informação;

2. Cada *work-item* possui uma posição na memória global da GPU, que armazena o índice da linha sendo calculada atualmente. Antes de invocar o primeiro *grid*, o *host* escreve o valor 0 em todas essas posições e, no decorrer da execução do *kernel*, cada *work-item* acessa a sua posição e incrementa seu valor.

3. Cada *work-group* possui uma posição na memória global da GPU, que armazena o índice da linha sendo calculada atualmente. Antes de invocar o primeiro *grid*, o *host* escreve o valor 0 em todas essas posições e, no decorrer da execução do *kernel*, cada *work-item* acessa a posição referente ao seu *work-group* e apenas um *work-item* de cada *wok-group* é responsável por incrementar esse valor. Exige sincronização entre os *work-items* de um *work-group*.

A duas últimas abordagens têm melhor desempenho que a primeira pois evitam que o *host* copie informações para a GPU antes da criação de cada *grid*. Embora a segunda abordagem necessite de mais memória que terceira, ela obtém melhor desempenho pois os acessos são coalescidos e não há necessidade de sincronização.

A execução do acelerador de granularidade média para o conjunto de 70.000 sequências da base Swiss-Prot e os HMMs *Top Twenty* com $r = 1$, isto é, com cada *work-item* calculando uma linha das estruturas de *scores*, produz o *speedup* médio de 18,11 em relação ao HMMer2, inferior ao *speedup* obtido com o acelerador de granularidade grossa.

Esse mesmo acelerador de granularidade média é executado aumentando progressivamente o número r de linhas calculadas pelos *work-items*, e o melhor desempenho é obtido quando a quantidade de linhas se torna igual ao tamanho da maior sequência ($r = \max_{1 \leq k \leq N} |S_k|$), ou seja, quando o acelerador realiza seus cálculos praticamente do mesmo modo que o acelerador de granularidade grossa. Ainda assim, os melhores tempos obtidos com esse acelerador são um pouco piores que os tempos obtidos com o acelerador de granularidade grossa, provavelmente devido ao *overhead* causado pelas invocações sucessivas dos *grids*.

O acelerador de granularidade fina diminui ainda mais a granularidade do *kernel* e *grids* com N *work-items* são instanciados sucessivamente, cada *work-item* k calculando apenas uma posição das matrizes M, I e D e, ao final do cálculo de uma linha dessas matrizes, um elemento dos vetores B, C e J para uma sequência $S_k \in S$. *Grids* são invocados sucessivamente para calcular as posições de M, I, D, B, C e J das sequências, até que todos os *scores* sejam calculados.

As mesmas otimizações aplicadas no acelerador de granularidade média são aplicadas no acelerador de granularidade fina que, quando executado para o conjunto de 70.000 sequências da base Swiss-Prot e os HMMs *Top Twenty*, obtém o menor *speedup* em relação ao HMMer2 dentre todos os aceleradores desenvolvidos, com um *speedup* médio de 14,03.

Esses resultados mostram que, para obter um bom desempenho, é melhor que o volume de computação realizada na GPU pelo *kernel* seja grande. De qualquer forma, mesmo com muitas transferências de dados entre o *host* e a GPU, os aceleradores de granularidade média e fina ainda produzem ganhos de desempenho em relação ao HMMer2, demostrando a capacidade de computação de alto desempenho das GPUs.

Capítulo 11

Conclusão

Este capítulo sumariza os resultados alcançados neste trabalho e sugere experimentos, análises e linhas de pesquisa para trabalhos futuros na área.

11.1 Resultados

O estudo realizado durante este trabalho permitiu a criação de um acelerador em GPU para o problema da comparação sequência-*profile*, em que uma sequência biológica e uma família de sequências, representada por um *profile* HMM, são comparadas com o intuito de determinar se a sequência faz parte da família ou não. O acelerador desenvolvido foi avaliado utilizando bases de sequências e famílias reais e executando em uma GPU com capacidade de processamento intermediário.

O acelerador alcançou um ótimo desempenho, com um *speedup* médio de 48,82 e máximo de 102,83 em relação à ferramenta HMMer2, amplamente utilizada, executada em um computador convencional. Além disso, é a primeira implementação descrita na literatura a utilizar o modelo de programação OpenCL na comparação base de sequência-*profile*. O desempenho obtido também é superior ao alcançado por outros aceleradores em GPU descritos na literatura. Assim, o acelerador otimizado possibilita que grandes bases de sequências sejam comparadas com *profile* HMMs em poucos segundos. Utilizando uma GPU com maior capacidade de processamento e mais memória, o acelerador projetado pode alcançar um desempenho ainda melhor.

A partir desses resultados, conclui-se que a GPU é um dispositivo com grande potencial para ser utilizado como plataforma de computação de alto desempenho, para problemas tanto na área de Bioinformática quanto em outras áreas, trazendo ganhos de desempenho expressivos a um custo acessível. Entretanto o desenvolvimento de soluções utilizando esse dispositivo ainda apresenta alguns desafios.

Embora o acelerador otimizado final tenha apresentado um desempenho excelente, o primeiro acelerador (básico) desenvolvido neste trabalho não obteve ganhos de desempenho, apesar de explorar o mesmo paralelismo entre sequências que o acelerador otimizado. No acelerador básico, o algoritmo de Viterbi foi implementado sem uma análise das características específicas da plataforma alvo. Ou seja, o desenvolvimento de soluções para GPU, quando realizado da forma tradicional utilizada no desenvolvimento para processadores convencionais, pode não trazer nenhum ganho de desempenho.

A experiência do desenvolvimento deste trabalho mostrou que, para que se obtenha um bom desempenho, o desenvolvimento de soluções para GPU exige conhecimentos sobre sua organização, seus diferentes tipos de memória e as formas de acesso a cada uma delas. Os modelos de programação, como OpenCL e CUDA, tornam o desenvolvimento um pouco mais amigável, mas a programação ainda é em mais baixo nível do que a programação para processadores convencionais.

O desenvolvimento de soluções eficientes nessa plataforma também exige um estudo detalhado do

problema a ser tratado, de seus algoritmos e estruturas de dados, para que as dependências de dados presentes sejam identificadas e seja possível projetar as formas de paralelismo a serem exploradas e o uso eficiente das estruturas de dados. Assim, o estudo sobre *profile* HMMs, arquitetura Plan7 e o algoritmo de Viterbi permitiram o desenvolvimento do acelerador otimizado, bem como o levantamento bibliográfico das implementações do algoritmo de Viterbi em outras plataformas.

Pelos resultados dos experimentos, é possível concluir que as principais otimizações a serem exploradas no desenvolvimento de soluções em GPU envolvem a utilização eficiente da hierarquia de memória, com coalescência de acessos e redução de transações de memória desnecessárias, contribuindo com a diminuição do desperdício de largura de banda. Além disso, algumas otimizações simples como expor paralelismo entre as instruções do *kernel* e reduzir os conflitos de dados entre elas, através de *loop unrolling* e escalonamento de instruções, também trazem ganho de desempenho, mostrando que o compilador para GPU não realiza de forma adequada algumas otimizações comumente feitas pelos compiladores para outras plataformas.

Há um leque grande de opções de otimizações em GPU [49] e as soluções para o problema da comparação sequência-*profile* em GPU disponíveis na literatura não analisam a viabilidade de todas essas otimizações. A extensa avaliação de desempenho realizada neste trabalho e as conclusões dela tiradas podem auxiliar no desenvolvimento de soluções de outros problemas em GPU, que também sejam baseadas em programação dinâmica, contribuindo para a construção de soluções otimizadas.

11.2 Trabalhos Futuros

Diversas ideias de novos experimentos e análises surgiram ao longo do desenvolvimento deste trabalho. Algumas não puderam ser exploradas e são aqui relacionadas como possíveis trabalhos futuros na área.

Na Seção 9.1, o acesso aos símbolos das sequências é realizado de maneira coalescida, entretanto ainda há um pequeno desperdício de largura de banda, que pode ser eliminado através de leituras agrupadas de quatro símbolos por vez, que são utilizados em quatro iterações consecutivas do laço mais externo do algoritmo de Viterbi. Essa modificação pode trazer mais um pequeno ganho de desempenho ao acelerador otimizado.

As bases de dados utilizadas nos experimentos são compostas de sequências biológicas e famílias de sequências reais, apresentando uma variação bastante grande no tamanho das sequências e das famílias. A avaliação de se o acelerador desenvolvido se comporta melhor com conjuntos de sequências com características específicas, pode ser realizada, por exemplo, comparando o ganho de desempenho do acelerador para uma base de sequências de tamanho bastante variado, uma base grande de sequências pequenas e uma base pequena de sequências longas.

Uma das vantagens do OpenCL em relação a outros modelos de programação é sua portabilidade, podendo o mesmo código ser executado em diferentes plataformas. O acelerador criado neste trabalho foi executado apenas em GPUs NVIDIA. É interessante executá-lo também em outras plataformas que suportam OpenCL e realizar comparações de desempenho entre elas.

OpenCL e o modelo de programação da arquitetura CUDA têm muitas similaridades, mas OpenCL possui uma pequena perda de desempenho em relação a CUDA quando executado em GPUs NVIDIA. Uma possível estratégia para confirmar essa perda de desempenho é implementar o acelerador projetado utilizando o modelo de programação CUDA e realizar comparações de desempenho com o acelerador em OpenCL.

Alguns aceleradores em GPU, para o problema da comparação sequência-*profile*, disponibilizam o código-fonte, como é o caso do GPU-HMMER [77]. Uma maneira precisa de comparar diretamente os ganhos de desempenho obtidos por esses aceleradores e o desenvolvido neste trabalho é executá-los na mesma plataforma e com os mesmos dados de entrada.

A versão 3 do HMMer [18] utiliza o algoritmo de Viterbi em uma segunda fase da comparação sequência-*profile*. O acelerador otimizado pode ser adaptado para ser utilizado diretamente nessa fase do HMMer3, com pouquíssimas modificações. A primeira fase utiliza o algoritmo MSV (*Multiple Segment Viterbi*), que é baseado no algoritmo de Viterbi para uma arquitetura de HMM mais simples que a Plan7, apresentando menos dependências de dados. É possível implementar esse algoritmo em GPU com modificações que permitam explorar mais paralelismo.

No decorrer deste trabalho comprovou-se através de experimentos que o *Toolkit* [56] fornecido pela NVIDIA não aplica de forma adequada otimizações clássicas de compiladores, como escalonamento de instruções. É importante estudar o processo de compilação de códigos OpenCL na arquitetura CUDA e entender as razões pelas quais o compilador fornecido não realiza tais otimizações satisfatoriamente. Uma linha de pesquisa muito interessante é justamente a inclusão de novas etapas de otimização no compilador para GPU.

Referências Bibliográficas

[1] Advanced Micro Devices, Inc. AMD Athlon II Processors. http://www.amd.com/us/products/desktop/processors/athlon-ii-x2/Pages/amd-athlon-ii-x2-dual-core-processors-desktop.aspx, 2012. Acessado em julho de 2012.

[2] Advanced Micro Devices, Inc. ATI. http://www.amd.com, 2012. Acessado em julho de 2012.

[3] C. Barrett, R. Hughey, and K. Karplus. Scoring Hidden Markov Models. *Computer Applications in the Biosciences*, 13(2):191–199, 1997.

[4] K. Benkrid, P. Velentzas, and S. Kasap. A High Performance Reconfigurable Core for Motif Searching Using Profile HMM. In *Proceedings of the NASA/ESA Conference on Adaptive Hardware and Systems (AHS)*, pages 285–292, 2008.

[5] M. Brand, N. Oliver, and A. Pentland. Coupled Hidden Markov Models for Complex Action Recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 994–999, 1997.

[6] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan. Brook for GPUs: Stream Computing on Graphics Hardware. *ACM Transactions on Graphics*, 23(3):777–786, 2004.

[7] Canonical Ltd. Ubuntu 10.10. http://releases.ubuntu.com/10.10, 2012. Acessado em agosto de 2012.

[8] J. M. Chandonia, G. Hon, N. S. Walker, L. Lo Conte, P. Koehl, M. Levitt, and S. E. Brenner. The ASTRAL Compendium in 2004. *Nucleic Acids Research*, 32(D1):189–192, 2004.

[9] K. Chao and L. Zhang. *Sequence Comparison: Theory and Methods*. Springer, 2008.

[10] W. Ching and M. K. Ng. *Markov Chains: Models, Algorithms and Applications*. Springer, 2006. International Series in Operations Research and Management Science.

[11] P. Conway and B. Hughes. The AMD Opteron Northbridge Architecture. *IEEE Micro*, 27(2):10–21, 2007.

[12] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. McGraw-Hill, 3rd edition, 2009.

[13] S. Derrien and P. Quinton. Parallelizing HMMER for Hardware Acceleration on FPGAs. In *Proceedings of the IEEE International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, pages 10–17, 2007.

[14] 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 Parallel and Distributed Processing Symposium (IPDPS)*, pages 1–8, 2010.

[15] R. Durbin, S. Eddy, A. Krog, and G. Mitchison. *Biological Sequence Analysis – Probabilistic Models of Proteins and Nucleic Acids*. Cambridge University Press, 2008.

[16] S. Eddy. Profile Hidden Markov Models. *Bioinformatics*, 14(9):755–763, 1998.

[17] S. Eddy. *HMMER User's Guide – Biological Sequence Analysis Using Profile Hidden Markov Models – Version 2.3.2.* Howard Hughes Medical Institute, http://hmmer.janelia.org/, 2003. Acessado em julho de 2012.

[18] S. Eddy. *HMMER User's Guide – Biological Sequence Analysis Using Profile Hidden Markov Models – Version 3.0.* Howard Hughes Medical Institute, http://hmmer.janelia.org/, 2010. Acessado em julho de 2012.

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

[20] EMBL-EBI. UniProtKB/TrEMBL Protein Database Release 2012-07 Statistics. http://www.ebi.ac.uk/uniprot/TrEMBLstats/, 2012. Acessado em julho de 2012.

[21] G. A. Fink. *Markov Model for Pattern Recognition – From Theory to Applications.* Springer, 2008.

[22] L. Firoiu and P. R. Cohen. Segmenting Time Series with a Hybrid Neural Networks-Hidden Markov Model. In *Proceedings of the National Conference on Artificial Intelligence*, pages 247–252, 2002.

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

[24] 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 (BCB)*, pages 418–421, 2010.

[25] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam. *PVM: Parallel Virtual Machine – A User's Guide and Tutorial for Network Parallel Computing.* MIT Press, 1994.

[26] GNU. GCC, the GNU Compiler Collection. http://gcc.gnu.org/, 2012. Acessado em agosto de 2012.

[27] N. Govindaraju, J. Gray, R. Kumar, and D. Manocha. GPUTeraSort: High Performance Graphics Co-processor Sorting for Large Database Management. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*, pages 325–336, 2006.

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

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

[30] L. Hunter, editor. *Artificial Intelligence and Molecular Biology.* American Association for Artificial Intelligence Press, 1993.

[31] A. C. Jacob, J. M. Lancaster, J. D. Buhler, and R. D. Chamberlain. Preliminary Results in Accelerating Profile HMM Search on FPGAs. In *Proceedings of the IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 1–8, 2007.

[32] 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.

[33] J. Kahle. The Cell Processor Architecture. In *Proceedings of the Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, page 3, 2005.

[34] Khronos Group. OpenCL – The Open Standard for Parallel Programming of Heterogeneous Systems. http://www.khronos.org/opencl, 2012. Acessado em julho de 2012.

[35] Khronos Group. The Khronos Group. http://www.khronos.org/, 2012. Acessado em julho de 2012.

[36] G. Kimmel and R. Shamir. A Block-free Hidden Markov Model for Genotypes and its Application to Disease Association. *Journal of Computational Biology*, 12(10):1243–1260, 2005.

[37] D. B. Kirk and W. M. Hwu. *Programming Massively Parallel Processors: A Hands-on Approach*. Morgan Kaufmann Publishers Inc., 1st edition, 2010.

[38] H. T. Kung. Why Systolic Architectures? *IEEE Computer*, 15(1):37–46, 1982.

[39] J. Landman and J. Ray. Accelerating HMMer Searches on Opteron Processors with Minimally Invasive Recoding. In *Proceedings of the International Conference on Advanced Information Networking and Applications (AINA)*, pages 628–636, 2006.

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

[41] C. Maxfield. *The Design Warrior's Guide to FPGAs: Devices, Tools and Flows*. Newnes, 2004.

[42] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard – Version 2.2*. High Performance Computing Center Stuttgart, 2009.

[43] Microway, Inc. GPGPU Architecture Comparison of ATI and NVIDIA GPUs. http://www.microway.com/pdfs/GPGPU_Architecture_and_Performance_Comparison.pdf, 2010. Acessado em julho de 2012.

[44] D. R. Miller, T. Leek, and R. M. Schwartz. A Hidden Markov Model Information Retrieval System. In *Proceedings of the Annual International ACM Conference on Research and Development in Information Retrieval (SIGIR)*, pages 214–221, 1999.

[45] S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 2003.

[46] NCBI. National Center for Biotechnology Information. http://www.ncbi.nlm.nih.gov/. Acessado em julho de 2012.

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

[48] NVIDIA Corporation. *NVIDIA's Next Generation CUDA Compute Architecture: FERMI*. http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf, 2009. Acessado em julho de 2012.

[49] NVIDIA Corporation. *OpenCL Best Practices Guide*, 2010.

[50] NVIDIA Corporation. *OpenCL JumpStart Guide*, 2010.

[51] NVIDIA Corporation. *OpenCL Programming for the CUDA Architecture*, 2010.

[52] NVIDIA Corporation. The CUDA Compiler Driver NVCC, 2011. Acessado em agosto de 2012.

[53] NVIDIA Corporation. *CUDA C Programming Guide*. http://developer.download.nvidia.com/compute/DevZone/docs/html/C/doc/CUDA_C_Programming_Guide.pdf, 2012. Acessado em julho de 2012.

[54] NVIDIA Corporation. CUDA Occupancy Calculator, 2012. Acessado em agosto de 2012.

[55] NVIDIA Corporation. CUDA Toolkit, 2012. Acessado em agosto de 2012.

[56] NVIDIA Corporation. CUDA Toolkit Archive, 2012. Acessado em julho de 2011.

[57] NVIDIA Corporation. GeForce Graphics Cards. http://www.nvidia.com/object/geforce_family.html, 2012. Acessado em julho de 2012.

[58] NVIDIA Corporation. GPU Computing SDK, 2012. Acessado em agosto de 2012.

[59] NVIDIA Corporation. *NVIDIA's Next Generation CUDA Compute Architecture: Kepler GK110*. http://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf, 2012. Acessado em agosto de 2012.

[60] NVIDIA Corporation. *OpenCL 1.1 Pre-release drivers now available*. http://news.developer.nvidia.com/2010/06/opencl-11-prerelease-drivers-now-available.html, 2012. Acessado em dezembro de 2011.

[61] K. Oh and K. Jung. GPU Implementation of Neural Networks. *Pattern Recognition*, 37(6):1311–1314, 2004.

[62] J. D. Owens, M. Houston, D. Luebke, S. Green, J. E. Stone, and J. C. Phillips. GPU Computing. *Proceedings of the IEEE*, 96(5):879–899, 2008.

[63] PCI-SIG. Creating a PCI Express Interconnect. http://www.pcisig.com/specifications/pciexpress/resources/PCI_Express_White_Paper.pdf, 2004. Acessado em julho de 2012.

[64] A. B. Poritz. Hidden Markov Models: a Guided Tour. In *Proceedings of the International Conference on Acoustics, Speech, and Signal Processing (ICASSP)*, pages 7–13, 1988.

[65] M. Punta, P. C. Coggill, R. Y. Eberhardt, J. Mistry, J. Tate, C. Boursnell, N. Pang, K. Forslund, G. Ceric, J. Clements, A. Heger, L. Holm, E. L. Sonnhammer, S. Eddy, A. Bateman, and R. D. Finn. The Pfam Protein Families Database. *Nucleic Acids Research*, 40(D1):290–301, 2012.

[66] L. R. Rabiner. A Tutorial on Hidden Markov Models and Selected Applications in Speech Recognition. *Proceedings of the IEEE*, 77(2):257–286, 1989.

[67] J. C. Setubal and J. Meidanis. *Introduction to Computational Molecular Biology*. PWS Publishing, 1997.

[68] S. Siewert. Using Intel Streaming SIMD Extensions and Intel Integrated Performance Primitives to Accelerate Algorithms. http://software.intel.com/en-us/articles/using-intel-streaming-simd-extensions-and-intel-integrated-performance-primitives-to-accelerate-algorithms/, 2009. Acessado em julho de 2012.

[69] A. Stivala. Assessment of Graphics Processing Units for Acceleration of Sequence Database Search Programs. http://www.vpac.org/files/SeqAlignCaseStudy.pdf, 2009.

[70] 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 Parallel and Distributed Processing Symposium (IPDPS)*, pages 1–8, 2009.

[71] Swiss Institute of Bioinformatics. UniProtKB/Swiss-Prot Protein Knowledgebase Release 2012-07 Statistics. http://web.expasy.org/docs/relnotes/relstat.html, 2012. Acessado em julho de 2012.

[72] J. Tyler, J. Lent, A. Mather, and H. Nguyen. AltiVec: Bringing Vector Technology to the PowerPC Processor Family. In *Proceedings of the International Performance, Computing and Communications Conference (IPCCC)*, pages 437–444, 1999.

[73] UniProt Consortium. Universal Protein Resource. http://www.uniprot.org, 2012. Acessado em julho de 2012.

[74] P. D. Vouzis and N. V. Sahinidis. GPU-BLAST: Using Graphics Processors to Accelerate Protein Sequence Alignment. *Bioinformatics*, 27(2):182–188, 2011.

[75] 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 Parallel and Distributed Processing Symposium (IPDPS)*, pages 1–12, 2009.

[76] J. P. Walters, R. Darole, and V. Chaudhary. Improving MPI-HMMER's Scalability With Paralell I/O. In *Proceedings of the IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 1–11, 2009.

[77] J. P. Walters, J. Landman, and V. Chaudhary. GPU-HMMER. http://www.mpihmmer.org/. Acessado em julho de 2012.

[78] J. P. Walters, J. Landman, and V. Chaudhary. MPI-HMMER. http://www.mpihmmer.org/. Acessado em julho de 2012.

[79] J. P. Walters, J. Landman, and V. Chaudhary. *Grids for Bioinformatics and Computational Biology*, chapter Optimized Cluster-Enabled HMMER Searches. Wiley & Sons, 2007.

[80] J. P. Walters, X. Meng, V. Chaudhary, T. Oliver, L. Y. Yeow, B. Schmidt, D. Nathan, and J. Landman. MPI-HMMER-Boost: Distributed FPGA Acceleration. *Journal of VLSI Signal Processing Systems*, 48(3):223–238, 2007.

[81] R. Weber, A. Gothandaraman, R. J. Hinde, and G. D. Peterson. Comparing Hardware Accelerators in Scientific Applications: A Case Study. *IEEE Transactions on Parallel and Distributed Systems*, 22(1):58–68, 2011.

[82] Wellcome Trust Sanger Institute. Pfam Top Twenty Families. http://pfam.sanger.ac.uk/family/browse?browse=top twenty. Acessado em julho de 2012.

[83] S. Yamagiwa. Invitation to a standard programming interface for massively parallel computing environment: OpenCL. *International Journal of Networking and Computing*, pages 188–205, 2012.

[84] 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 (HPCS)*, pages 24–30, 2010.