Soluções em GPU para o Problema do Alinhamento *Spliced*

Anisio Vitorino Nolasco

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 2014

Resumo

As GPUs (*Graphics Processing Units* – Unidades de Processamento Gráfico) têm se mostrado uma boa plataforma de computação paralela devido à sua grande capacidade de processamento, que evolui muito a cada ano, e seu bom custo-benefício. Algumas áreas apresentam grande potencial de aplicação da computação paralela, como por exemplo a análise de sequências biológicas, uma importante área da Bioinformática. Devido aos avanços nas técnicas de sequenciamento de DNA, o tamanho das bases de dados biológicos vem crescendo muito nos últimos anos, motivando as pesquisas de soluções de alto desempenho para os problemas da área. Assim, este trabalho tem como objetivo desenvolver soluções em GPU para o algoritmo de Gelfand para o problema do alinhamento *spliced*, e estudar formas de explorar paralelismo na execução do algoritmo nesse dispositivo.

Palavras-chave: Alinhamento *spliced*, algoritmo de Gelfand, GPU.

Abstract

GPUs (Graphics Processing Units) have shown to be a good platform for parallel computing because of their large processing capacity that evolves every year and their good cost-benefit ratio. Some areas present great potential for the use of parallel computing, as the analysis of biological sequences, an important area in Bioinformatics. Due to advances in DNA sequencing techniques, the size of the biological databases has been increasing drastically in recent years, motivating the research of high performance solutions to the problems in this area. Therefore, this work aims to develop GPU-based solutions to the Gelfand algorithm for the spliced alignment problem, and investigate ways of exploiting parallelism in the algorithm execution on this device.

Keywords: Spliced alignment, Gelfand algorithm, GPU.

Sumário

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

Atualmente as arquiteturas dos microprocessadores têm como característica comum possuírem mais de um núcleo por *chip*. Isto mostra que a computação paralela é de suma importância para o melhor aproveitamento dos recursos oferecidos pelo hardware. Construir ou comprar um *cluster* de computadores para o desenvolvimento de soluções paralelas para diversos problemas pode ser custoso em termos financeiros, de espaço físico, entre outros. Uma alternativa interessante são as GPUs, que apresentam grande potencial de desempenho, aliado a um baixo custo e baixa demanda de espaço, energia e refrigeração [22].

Criadas com o intuito acelerar o processamento gráfico, programar as GPUs para problemas de propósito geral era difícil pois era necessário o entendimento das interfaces de programação de aplicações (APIs) gráficas e modelar o problema para utilizá-las. Recentemente, os fabricantes começaram a desenvolver facilidades tanto na arquitetura das GPUs como nos modelos de programação. Assim, a cada nova geração de GPUs, suas capacidades computacionais são aumentadas e seu modelo de programação é melhorado, permitindo sua aplicação em outras áreas e não apenas em processamento gráfico [40].

Algumas áreas apresentam grande potencial de aplicação da computação paralela, como por exemplo, a análise de sequências biológicas, uma importante área da Bioinformática. A Figura 1.1 mostra o crescimento nos últimos anos, praticamente exponencial, do número de pares de bases de DNA e do número de sequências, do banco GenBank [26] de sequências de DNA e aminoácidos, mantido pelo NCBI (*National Center for Biotechnology Information*). Esse crescimento é devido aos grandes avanços nas técnicas de sequenciamento de moléculas de DNA, gerando um grande volume de dados e tornando expressivo o número de sequências genômicas a serem analisadas. Tal realidade motiva as pesquisas de soluções de alto desempenho para os problemas da área.

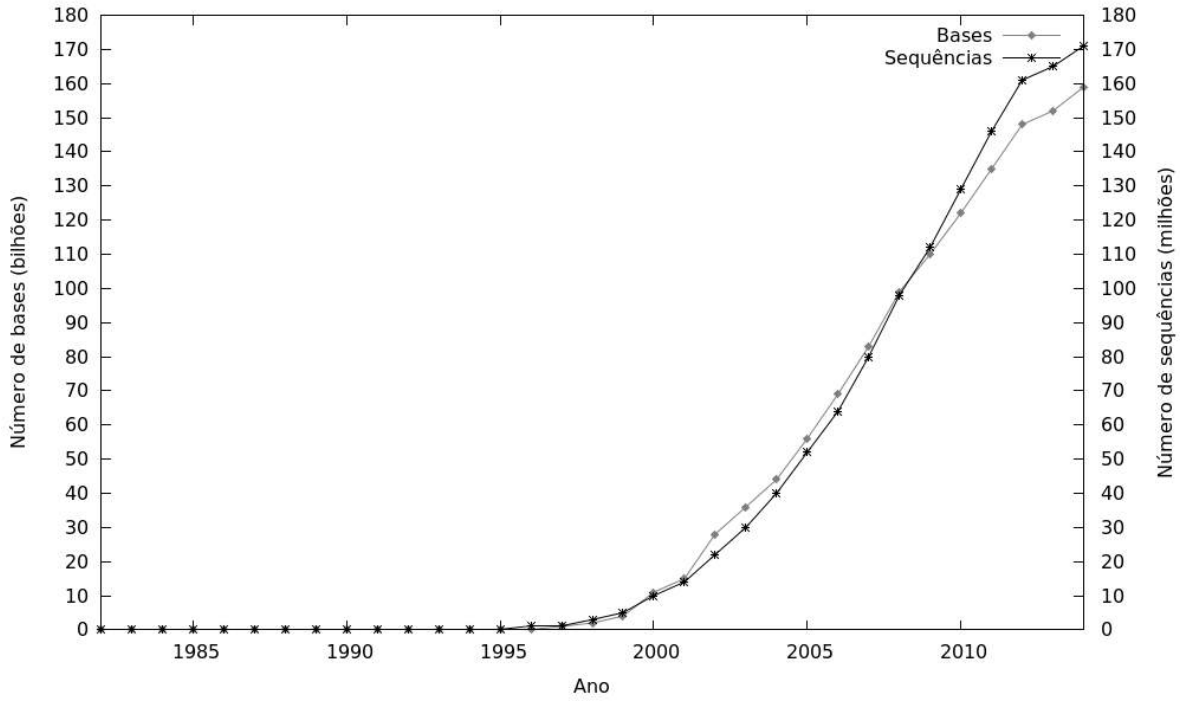

Figura 1.1: Crescimento da base de dados biológicos GenBank entre os anos de 1982 e 2014

1.1 Objetivos do Trabalho

Este trabalho visa inicialmente ao estudo das GPUs como plataforma de computação paralela e sua utilização para o desenvolvimento de soluções de alto desempenho para um problema fora da área de processamento gráfico. Para a implementação das soluções em GPU, o modelo de programação CUDA (Compute Unified Device Architecture) [27] será adotado.

O estudo das características da plataforma de computação paralela é de suma importância para um melhor aproveitamento do potencial da mesma e assim obter um desempenho satisfatório. Este estudo deve abordar questões da arquitetura da plataforma, tais como seu modelo de execução, aspectos de sincronização e hierarquia de memórias, assim como técnicas de otimização e ferramentas de avaliação de desempenho. Portanto, é necessário que os recursos da GPU sejam utilizados de forma eficiente, otimizando ao máximo as soluções desenvolvidas, tanto no uso de memória como no tempo de execução.

Mais especificamente, este trabalho tem como objetivo desenvolver soluções em GPU para o algoritmo de Gelfand [14, 15] para o problema do alinhamento *spliced* aplicado à identificação de genes em sequências eucarióticas. O entendimento do problema é fundamental, assim como a compreensão dos conceitos e do algoritmo sequencial relacionados ao problema. O estudo das dependências de dados existentes na resolução do problema permite a análise do potencial de paralelismo desta resolução e o projeto de

possíveis formas de explorá-lo.

Finalmente, pretende-se realizar uma extensa avaliação experimental de desempenho usando dados biológicos reais e representativa para analisar as soluções em GPU desenvolvidas, em relação ao desempenho e à escalabilidade.

1.2 Contribuições

De forma mais detalhada, as principais contribuições deste trabalho são:

- A identificação de duas formas de explorar paralelismo no algoritmo de Gelfand para o alinhamento *spliced*;
- O desenvolvimento de dois aceleradores em GPU para o algoritmo de Gelfand para o alinhamento *spliced* que atingem *speedups* de até 7,89 e 31,51, respectivamente, quando comparados com uma implementação sequencial executada em um processador convencional;
- Uma forma de organização particular para as estruturas de dados dos aceleradores, de modo a otimizar a sua eficiência;
- Uma análise do potencial de paralelismo do conjunto de dados biológicos empregado nos experimentos, com o objetivo de medir a quantidade de paralelismo que, de fato, está disponível para ser explorada por uma implementação paralela; e
- Um modelo preciso de estimativa de desempenho que permite estimar o desempenho dos aceleradores em GPU antes de implementá-los, e avaliar se haverá ganhos de desempenho, em relação a uma solução sequencial.

1.3 Organização do Texto

Este texto está organizado em oito capítulos. O Capítulo 2 apresenta a GPU como uma plataforma de computação paralela, e descreve sua arquitetura, a hierarquia de memórias, o modelo de programação CUDA e os mecanismos de sincronização. Um exemplo de programa usando o modelo de programação CUDA é fornecido.

No Capítulo 3, o problema do alinhamento *spliced* aplicado à identificação de genes é apresentado, assim como o algoritmo de Gelfand que o resolve. As dependências de dados presentes neste algoritmo e um exemplo de execução do mesmo também são apresentados.

Em seguida, o Capítulo 4 descreve trabalhos relacionados a soluções paralelas para o problema do alinhamento *spliced* encontrados na literatura, detalhando suas propostas e os resultados obtidos.

Duas formas de exploração de paralelismo na execução do algoritmo de Gelfand para o problema do alinhamento *spliced* são identificadas no Capítulo 5: paralelismo intra-éxon e inter-éxon. O conjunto de dados biológicos utilizado nos experimentos é descrito, assim como uma análise do potencial de paralelismo nestes dados. Um modelo de estimativa de desempenho é proposto para as soluções paralelas do problema de alinhamento *spliced*.

O Capítulo 6 apresenta o desenvolvimento de dois aceleradores em GPU para o problema do alinhamento *spliced*, descrevendo as abordagens adotadas nestas soluções e a organização das estruturas de dados utilizada. A plataforma de desenvolvimento e execução utilizada e as ferramentas adotadas são descritas no Capítulo 7. Os resultados obtidos na avaliação experimental são apresentados, sendo analisados o desempenho e a escalabilidade dos aceleradores e a precisão do modelo de estimativa de desempenho.

Por fim, o Capítulo 8 apresenta a conclusão deste trabalho, fazendo uma análise geral sobre o que foi desenvolvido e apresentando ideias para possíveis trabalhos futuros.

Capítulo 2

Computação de Alto Desempenho em GPU

Microprocessadores baseados em um único núcleo, com tecnologia de fabricação que permitia frequências do *clock* cada vez maiores, foram responsáveis por ganhos de desempenho por vários anos. Entretanto, o consumo de energia e a dissipação de calor limitaram o aumento continuado da frequência de *clock*, o que levou os fabricantes de microprocessadores a adotarem modelos de vários núcleos, aumentando o poder de processamento de cada *chip* [23].

Desde então, a indústria segue duas trajetórias principais para o projeto de microprocessadores: processadores *multi-core* e *many-core*. Os processadores *multi-core* começaram com dois núcleos, sendo que o número de núcleos praticamente dobra a cada nova geração, e seu foco é reduzir o tempo de execução de programas sequenciais. Já os processadores *many-core*, que têm seu foco na execução de aplicações paralelas, começaram com um grande número de núcleos que também dobra a cada geração, porém estes núcleos possuem frequências de *clock* bem menores [23].

Um exemplo atual de processadores *many-core* são as GPUs, dispositivos com grande capacidade de processamento paralelo, que avançaram muito nos últimos anos em desempenho e em facilidade de programação. Por exemplo, a GPU GeForce GTX 690 da NVIDIA possui 3072 núcleos, alcançando 5621 GFLOPS (*Giga Floating-point Operations Per Second* – 10^9 operações de ponto flutuante por segundo) [30]. O segundo supercomputador mais rápido do mundo utiliza GPUs como co-processadores, segundo o *ranking* Top 500 (em outubro de 2014) [39].

Para permitir a execução de programas em suas GPUs sem a utilização de APIs gráficas, a NVIDIA criou em 2007 o modelo de programação CUDA (*Compute Unified Device Architecture*) [27]. Além da mudança do modelo de programação, modificações também foram necessárias na arquitetura das GPUs, adicionando novos componentes de

hardware e levando à criação de uma nova linha de GPUs destinadas à programação paralela.

2.1 Modelo de Programação CUDA

O modelo de programação CUDA é uma extensão das linguagens de programação C e C++ [23]. O programador escreve programas que invocam *kernels*, que são as funções criadas pelo programador especificamente para executar na GPU. Um *kernel* é executado em paralelo através de um conjunto de *threads* paralelas. Essas *threads* são organizadas em uma hierarquia, na forma de um *grid* de blocos de *threads*.

Quando um *kernel* é invocado, um *grid* é criado para a execução das instruções do *kernel*. Um *grid* é uma estrutura de até três dimensões de blocos que podem ser executados concorrentemente e assim explorar paralelismo, ou seja, não há dependências entre os mesmos. Um bloco é uma estrutura de até três dimensões de *threads* concorrentes que podem cooperar entre si, sincronizar-se através de uma barreira de sincronização e comunicar-se através da memória compartilhada pertencente ao bloco. As *threads* são a unidade básica de execução e executam as instruções do *kernel*.

A Figura 2.1 mostra a estrutura de um *grid* de duas dimensões, com 3×2 blocos, blocos estes de duas dimensões, com 2×2 *threads*. Cada *thread* de um bloco e cada bloco de um *grid* possuem identificações únicas que permitem diferenciá-los dos outros *threads* e blocos.

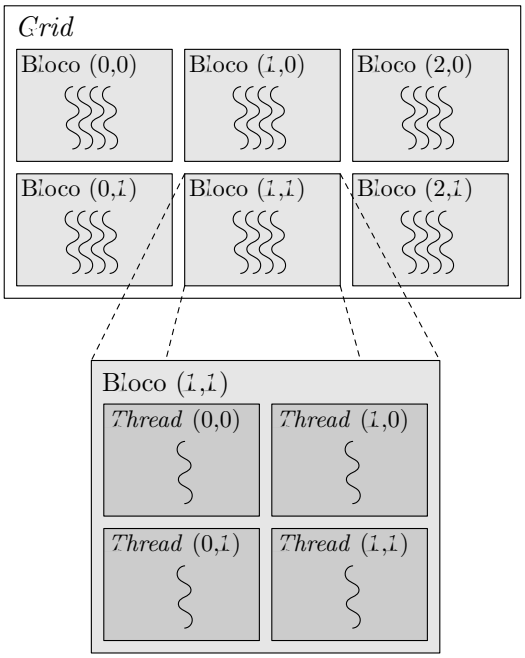

Figura 2.1: *Grid* com 3×2 blocos de 2×2 *threads* [31]

Como será descrito mais adiante, a GPU é acoplada a um computador, denominado *host*, através de um barramento [23]. As memórias da GPU são organizadas na forma de uma hierarquia: memória global, memória constante, memória de textura, memória compartilhada, memória local e registradores.

A Figura 2.2 mostra o fluxo de execução de um programa baseado no modelo de programação CUDA. O código principal é executado pelo *host* e invoca uma função *kernel* que é executada pela GPU. Na fase de inicialização ocorre a preparação de variáveis, a alocação de estruturas de dados na memória do *host* e da GPU e a preparação da GPU. Na fase de transferência *host*-GPU (1) ocorre a passagem de dados da memória do *host* para a memória da GPU, pois no modelo de programação CUDA as *threads* executadas na GPU não têm acesso à memória do *host*, então todos os dados necessários para execução do *kernel* e que estão na memória do *host* devem ser transferidos para memória da GPU, podendo ser para memória global ou memória constante.

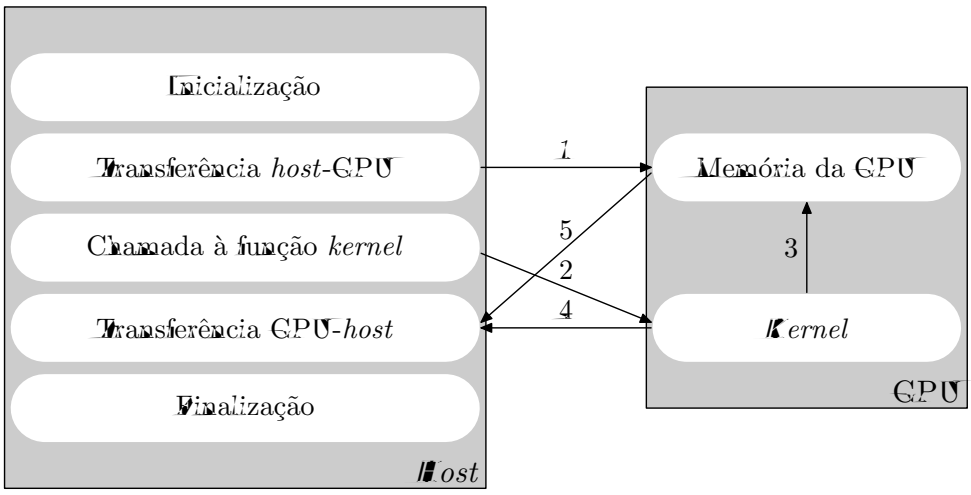

Figura 2.2: Fluxo de execução de um programa no modelo de programação CUDA

Em seguida, a função *kernel* é invocada (2), causando a criação do *grid* de blocos de *threads*. Esta mesma função *kernel* é executada por todas as *threads* do *grid*, porém executando sobre dados diferentes, mantidos nas memórias da GPU (3). Após o término da função *kernel* (4), a transferência *GPU-host* é realizada, quando o resultado produzido que se encontra na memória da GPU é transferido para a memória do *host* (5). Finalmente, realiza-se a fase de finalização, liberando as estruturas de dados das memórias e reportando os resultados [23].

2.1.1 Exemplo de Programa em CUDA

Para um melhor entendimento do modelo de programação CUDA, esta subseção apresenta uma possível solução para soma de dois vetores. A Figura 2.3 mostra a função

```
1  // kernel: cada thread calcula um elemento do vetor C
2  __global__ void vectorSumKernel(int A[], int B[], int C[], int N) {
3     // A: primeiro vetor a ser somado
4     // B: segundo vetor a ser somado
5     // C: vetor com soma de A + B
6     // N: tamanho dos vetores
7
8     // Calcula id da thread e
9     // índice do elemento do vetor que será calculado pela thread
10    int id = threadIdx.x + blockIdx.x * blockDim.x;
11
12    // Verifica se limite do vetor não foi ultrapassado
13    if (id < N)
14       C[id] = A[id] + B[id];
15 }
```

Figura 2.3: Função *kernel* que soma dois vetores, executada na GPU

kernel que é executada por todas as *threads* criadas na invocação do *kernel*. A função recebe como parâmetros os vetores A e B a serem somados, o vetor C que armazenará a soma dos vetores A e B e o tamanho N dos vetores.

Na linha 10 do *kernel*, calcula-se a variável *id* que identifica cada *thread* de forma única no *grid*, combinando-se as variáveis pré-inicializadas *threadIdx.x* (índice da *thread* no bloco), *blockIdx.x* (índice do bloco no *grid*) e *blockDim.x* (dimensão do bloco, isto é, número de *threads* por bloco). A variável *id* também é usada como índice dos vetores. A verificação da linha 13 é necessária pois N não é necessariamente múltiplo do número de *threads* por bloco, e com isso pode haver mais *threads* do que o necessário. Por fim, na linha 14 cada *thread* faz uma soma, sendo que todas as somas podem, em teoria, ser feitas em paralelo, uma vez que essas *threads* são independentes entre si.

A Figura 2.4 contém a função executada no *host*. Esta função recebe como parâmetros os vetores *host_A* e *host_B* a serem somados, o vetor *host_C* que armazenará a soma e o tamanho N dos vetores. Nas linhas 9 a 13 são realizadas as declarações e inicializações de variáveis necessárias para invocar o *kernel*. O número de *threads* por bloco (*blockDim*) é inicializado com 1024. O número de blocos (*gridDim*) é inicializado em função do tamanho N dos vetores, com $\lceil \frac{N}{1024} \rceil$. Nas linhas 16 a 18 ocorre a alocação de espaço na memória global da GPU para os três vetores. A cópia dos vetores a serem somados da memória do *host* para a memória global da GPU ocorre nas linhas 21 e 22.

Após as fases de inicialização e transferência *host*-GPU, ocorre na linha 25 a invocação da função *kernel*, apresentada na Figura 2.3, causando a criação do *grid* de $\lceil \frac{N}{1024} \rceil$ blocos de 1024 *threads*. Na fase de transferência GPU-*host* ocorre a cópia do vetor com o resultado da soma da memória global da GPU para a memória do *host*, mostrada na linha 28. Ao final, nas linhas 31 a 33, libera-se o espaço alocado para os vetores na

```
1  // Função executada no host
2  void vectorSum(int host_A[], int host_B[], int host_C[], int N) {
3      // host_A: primeiro vetor na memoria do host
4      // host_B: segundo vetor na memoria do host
5      // host_C: terceiro vetor na memoria do host
6      // N: tamanho dos vetores
7
8      // Declaração e inicialização de variáveis
9      int blockDim = 1024     // Número de threads por bloco
10     int gridDim = (N + (blockDim - 1)) / blockDim;  // Número de blocos
11     int* device_A = NULL;  // Primeiro vetor na memória da GPU
12     int* device_B = NULL;  // Segundo vetor na memória da GPU
13     int* device_C = NULL;  // Terceiro vetor na memória da GPU
14
15     // Alocação de memória para vetores na GPU
16     cudaMalloc((void**) &device_A, N * sizeof(int));
17     cudaMalloc((void**) &device_B, N * sizeof(int));
18     cudaMalloc((void**) &device_C, N * sizeof(int));
19
20     // Cópia de vetores do host para GPU
21     cudaMemcpy(device_A, host_A, N * sizeof(int), cudaMemcpyHostToDevice);
22     cudaMemcpy(device_B, host_B, N * sizeof(int), cudaMemcpyHostToDevice);
23
24     // Invocação do kernel
25     vectorSumKernel<<< gridDim, blockDim >>>(device_A, device_B, device_C, N);
26
27     // Cópia de vetor da GPU para host
28     cudaMemcpy(host_C, device_C, N * sizeof(int), cudaMemcpyDeviceToHost);
29
30     // Liberação de memória de vetores na GPU
31     cudaFree(device_A);
32     cudaFree(device_B);
33     cudaFree(device_C);
34 }
```

Figura 2.4: Função, executada no *host*, responsável por invocar o *kernel* e realizar transferências de dados para e da memória da GPU

memória da GPU.

2.2 Arquitetura CUDA

A Figura 2.5 mostra de forma simplificada a arquitetura de uma GPU típica e adequada para o modelo de programação CUDA. Conforme descrito, a GPU é acoplada a um computador *host* através de um barramento. Atualmente, o barramento utilizado é o PCI-Express [32].

A GPU é organizada como uma coleção de multiprocessadores de *streaming* (SMs – *Streaming Multiprocessors*) altamente escaláveis, isto é, quanto maior sua quantidade, maior a capacidade de processamento paralelo da GPU. O número de SMs varia de uma geração para outra de GPUs e de versões de uma mesma geração.

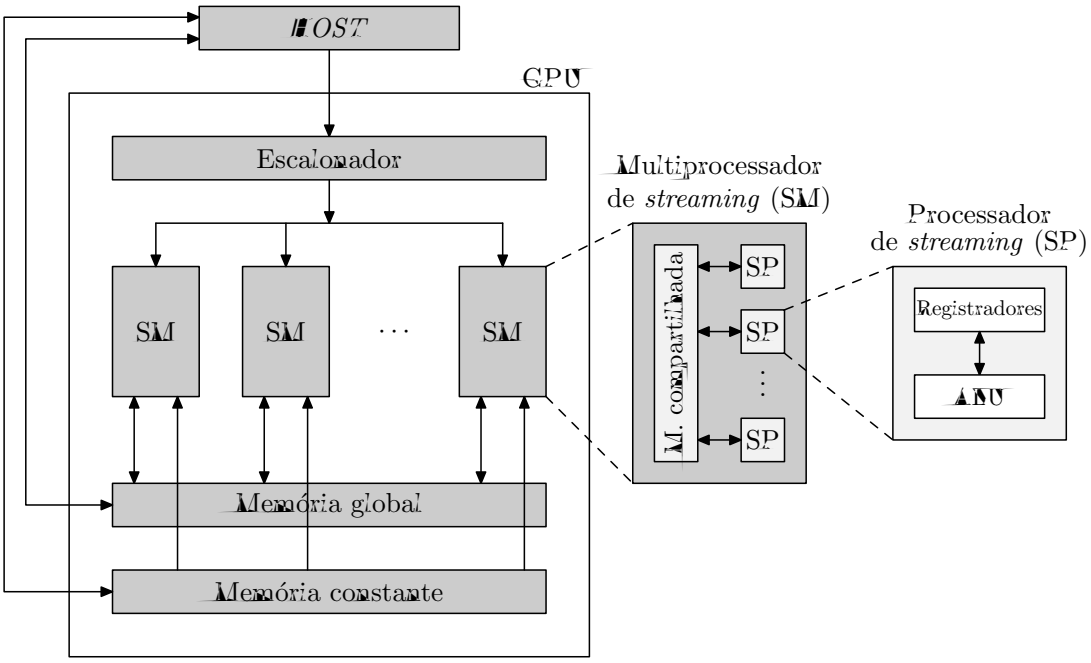

Figura 2.5: Arquitetura típica de uma GPU

Cada SM possui um certo número de processadores de *streaming* (SPs – *Streaming Processors*), também chamados de CUDA *cores*, que compartilham a lógica de controle e a cache de instruções. O número de SPs por SM é o mesmo para diferentes versões de GPUs de uma mesma geração, e todos os SMs de uma GPU têm o mesmo número de SPs. Os SPs de um SM podem ser divididos em grupos, onde todos os SPs de um mesmo grupo executam a mesma instrução, contudo, cada SP pode operar dados diferentes segundo o modelo SIMD (*Single Instruction stream, Multiple Data streams*) de execução [13].

Um *warp* é a unidade básica para o escalonamento de trabalho dentro de um SM e é, na maioria das GPUs, um grupo de 32 *threads* consecutivas de um bloco alocado a um grupo de 32 SPs de um SM, operando de forma SIMD.

2.2.1 Hierarquia de Memórias

A GPU possui várias memórias, cada uma com características diferentes, formando uma hierarquia. Na Figura 2.5 também é possível visualizar a hierarquia de memórias da GPU. As *threads* podem acessar dados de vários espaços de memória durante o seu tempo de execução. Cada SM possui um barramento próprio de acesso à memória global e, consequentemente, à memória constante que é uma forma de endereçamento virtual dentro da memória global [8]. A memória constante é acessível, apenas para leitura, a todas as *threads* de todos os blocos. Ela é acessível ao *host* para leitura e escrita, para passagem de dados constantes para a GPU [23].

Todas as *threads* de todos os blocos têm acesso à memória global, tanto para leitura

como para escrita. A memória global é utilizada para transferência de dados do *host* para a GPU e vice-versa. Portanto, o *host* pode acessá-la para leitura e escrita. Ela é a memória mais lenta e também a de maior capacidade de armazenamento dentro da GPU.

Cada bloco possui uma memória compartilhada, acessível para leitura e escrita a todas as *threads* do bloco, e que tem o mesmo tempo de vida que o bloco. A memória compartilhada possui uma largura de banda muito superior a da memória global, associada a uma baixa latência. Portanto, ela pode prover um acesso rápido e o compartilhamento de dados entre as *threads* de um mesmo bloco.

Cada *thread* também possui acesso exclusivo, de leitura e escrita, a uma memória local. Essa memória é utilizada, no modelo de programação CUDA, para as variáveis privadas da *thread* que não cabem nos registradores. Os registradores são o tipo de memória com acesso mais rápido em uma GPU. Cada SP possui um número limitado de registradores, aos quais possui acesso exclusivo, de leitura e escrita.

Como geralmente a maior parte dos dados de uma aplicação é armazenada na memória global, o seu acesso de forma desorganizada pode anular o potencial de desempenho da execução do programa na GPU. Os acessos à memória global devem ocorrer de forma coalescida, combinando múltiplas requisições de memória em uma única grande transferência. De forma simplificada, a **coalescência de memória** é obtida quando os acessos de leitura ou escrita são feitos a posições consecutivas de um segmento de 128 bytes da memória global [13].

2.2.2 Transferência *Host*-GPU e Memória Não-Paginada

Os dados do programa são alocados na memória do *host* em áreas paginadas, por padrão. Logo, quando há uma transferência de dados entre o *host* e a GPU, a cópia dos dados não é totalmente direta, isto é, o *host* copia os dados da sua área de memória paginada para uma área de memória que não sofre paginação (*pinned memory*), e só então ocorre a cópia dos dados desta área para a memória da GPU [8].

Uma forma de otimizar essa transferência é fazer o uso da memória não-paginada (*pinned memory*), alocando-se os dados diretamente nesta área de memória do *host*. Assim, na transferência de dados entre o *host* e a GPU a cópia dos dados é direta da memória não-paginada do *host* para a memória da GPU. Isto garante transferências mais rápidas e é uma maneira de se aproximar do pico da largura de banda do barramento nas transferências de dados [41].

A Figura 2.6 ilustra como funciona a transferência de dados usando uma alocação convencional na memória paginada e usando uma alocação na memória não-paginada. As setas com linhas contínuas representam o caminho percorrido ao se realizar uma

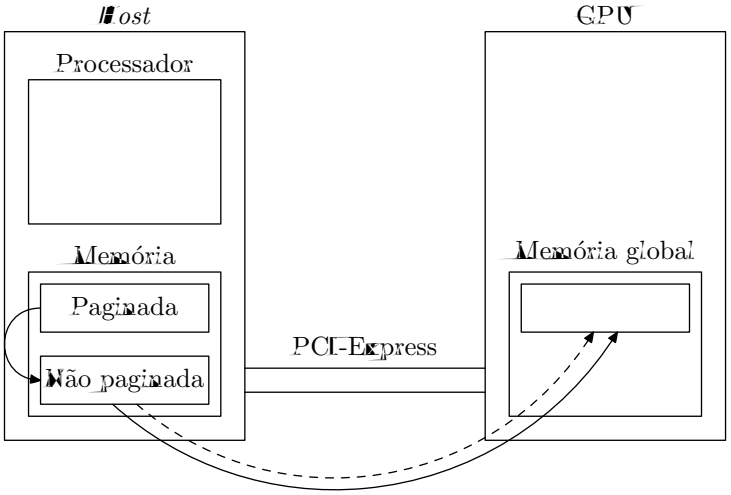

Figura 2.6: Transferência *host*-GPU e memória não-paginada

transferência de dados do *host* para a GPU, de dados alocados de forma convencional na memória paginada do *host*, mostrando as duas cópias necessárias para a transferência. Já a seta com a linha tracejada mostra a única cópia necessária para realizar uma transferência de dados do *host* para a GPU, de dados alocados na memória não-paginada do *host*.

2.3 Escalonamento e Sincronização

Cada *thread* é executada em um SP e cada bloco de *threads* é executado em um único SM da GPU. Um *warp* é um conjunto de 32 *threads* consecutivas de um mesmo bloco, que são alocadas aos SPs do SM e executam em paralelo de forma síncrona, isto é, em *lock-step*.

Quando as *threads* de um mesmo *warp* tomam diferentes caminhos de execução dentro de um *kernel* (devido a comandos condicionais como *if*), criam-se **divergências** entre as *threads*, e esses caminhos são executados sequencialmente até que todas as *threads* juntem-se novamente em um único caminho de execução. Por isso, divergências podem causar perda de desempenho [8].

A GPU possui um escalonador de blocos que escalona os blocos do *grid* nos SMs da GPU. Há também, em todo SM, escalonadores de *warps* que escalonam *warps* nos SPs do SM [23].

Ao resolver um problema em paralelo usando o modelo de programação CUDA, quanto maior a independência entre os dados, melhor, pois assim não será necessária a sincronização entre as *threads*. Porém na maioria das vezes há dependências de dados e com isso, em algum momento pode ser necessário que as *threads* se sincronizem, garantindo a integridade dos dados.

A sincronização é oferecida no modelo de programação CUDA de duas formas. A função $\_\_syncthreads()$ permite a sincronização de *threads* de um mesmo bloco e funciona como uma barreira. Quando uma *thread* chama essa função, ela é bloqueada até que todas as *threads* do bloco executem a mesma chamada. Isto garante que todas as *threads* do bloco tenham completado uma fase do programa antes de prosseguirem para a fase seguinte. Não há funções disponíveis no modelo CUDA para sincronizar *threads* de blocos distintos. A outra forma de sincronização é através da função $cudaDeviceSynchronize()$ que, quando executada no *host*, bloqueia o fluxo de execução até que todas as chamadas de *kernels* anteriores a ela sejam completadas [8].

2.3.1 *Streams*

No modelo de programação CUDA, um *stream* é uma fila de operações que podem ser a invocação de um *kernel* ou a transferência de dados entre *host* e GPU. A ordem na qual as operações são adicionadas no *stream* é a ordem na qual elas são executadas pela GPU. Uma operação de um *stream* só é iniciada após todas as operações anteriores daquele *stream* terem terminado. Entretanto, múltiplos *streams* podem ser executados em paralelo [8].

O uso de múltiplos *streams* pode melhorar o desempenho de um programa. Algumas GPUs mais novas suportam a execução em paralelo de *kernels*, de até duas transferências (uma do *host* para GPU e outra da GPU para o *host*) e a execução de *kernels* e transferências entre *host* e GPU. Sobrepor essas operações pode reduzir o tempo total de execução do programa.

Capítulo 3

Alinhamento *Spliced*

Uma sequência biológica pode ser definida como uma cadeia finita de símbolos que carregam informações biológicas. Estes símbolos pertencem a um alfabeto formado por um conjunto finito de símbolos [11]. Essa sequência pode ser modelada como um vetor de caracteres, como mostra a Figura 3.1.

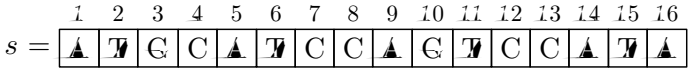

Figura 3.1: Sequência biológica s

Um dos tipos de sequências biológicas presentes nos organismos vivos são os ácidos nucleicos, cadeias formadas por bases nitrogenadas e que podem ser de dois tipos: DNA (ácido desoxirribonucleico) e RNA (ácido ribonucleico) [38].

Um gene corresponde a um fragmento de DNA que pode ser transcrito em um pré-RNA mensageiro no processo de síntese de proteínas que ocorre dentro das células [2]. Os genes em sequências eucarióticas são constituídos de éxons e íntrons. De forma simplificada, um éxon é um trecho contíguo da sequência de DNA que pode ser usado na síntese do RNA mensageiro. As regiões correspondentes aos éxons de uma sequência de DNA são chamadas de regiões codificantes [2].

3.1 Identificação de Genes

A identificação ou predição de genes é uma das etapas mais importantes no processo de compreender o genoma de um organismo. Ela consiste em identificar os trechos biologicamente funcionais (genes) em uma sequência de DNA. O problema da identificação de genes em sequências eucarióticas consiste em, dada uma sequência de DNA, determinar a posição inicial e final de cada um dos genes desta sequência, assim como, dentro deles,

determinar a posição inicial e final dos éxons que constituem estes genes [21]. A Figura 3.2 mostra um exemplo de uma sequência de DNA, seus genes e éxons.

Figura 3.2: Sequência de DNA com genes e éxons

Para resolver esse problema de identificação de genes, existem duas categorias principais de métodos: métodos intrínsecos e extrínsecos. Nos métodos intrínsecos utilizam-se apenas informações presentes na própria sequência para a identificação de seus genes. Existem ainda dois tipos de métodos intrínsecos: métodos estatísticos, que levam em consideração características estatísticas presentes na molécula de DNA, e métodos de busca por sinais, que tentam localizar sinais específicos dentro da sequência [2].

Métodos extrínsecos utilizam informações presentes em outras sequências já conhecidas, ou seja, informações extrínsecas à sequência sendo investigada, e também são conhecidos como métodos de busca por similaridades. Esse tipo de método realiza comparações entre a sequência sendo investigada e sequências anotadas, identificando regiões significantemente parecidas e criando um alinhamento ótimo entre elas [2].

3.2 Alinhamento *Spliced*

Gelfand e colegas modelaram o problema de identificação de genes em sequências eucarióticas através do problema do **alinhamento *spliced*** e propuseram um método extrínseco para este problema [14, 15]. Neste método, utiliza-se uma proteína alvo relacionada a um genoma já conhecido para reconstruir a estrutura dos éxons de um ou mais genes na sequência investigada. O método começa pela seleção dos possíveis éxons, ao filtrar este conjunto de maneira a não perder éxons verdadeiros, ficando com um conjunto de éxons candidatos que pode conter éxons falsos, mas seguramente contém todos os verdadeiros. Embora seja difícil distinguir éxons verdadeiros de falsos através apenas de um procedimento estatístico, pode-se usar o alinhamento com a proteína alvo. Em teoria, apenas os éxons verdadeiros formarão uma representação coerente de uma proteína [21].

Dados uma sequência base s sendo investigada, um conjunto ordenado β de éxons candidatos de s e uma sequência alvo t, o problema do alinhamento *spliced* consiste em encontrar um subconjunto de β que tenha maior similaridade com t (considerando uma função de pontuação de similaridade) e de tal forma que os éxons desse subconjunto não tenham sobreposição entre si.

A Figura 3.3 mostra um exemplo de alinhamento *spliced*, sendo s a sequência base sendo investigada, $\beta = \{b_1, b_2, b_3, b_4, b_5\}$ o conjunto de éxons candidatos de s e t a sequência alvo. Neste exemplo, o subconjunto de β de maior similaridade com t é formado pelos éxons b_1, b_4 e b_5.

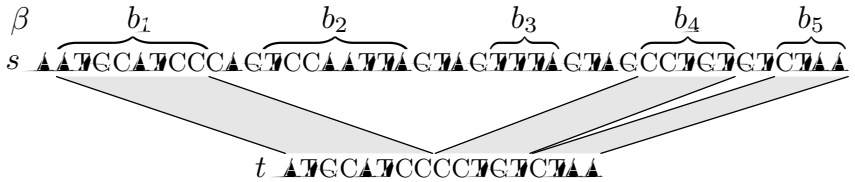

Figura 3.3: Alinhamento *spliced* entre sequência base s, com conjunto de éxons candidatos β, e sequência alvo t

3.3 Algoritmo de Gelfand

O número de diferentes subconjuntos de β pode ser enorme, porém o algoritmo de alinhamento *spliced* é capaz de encontrar, dentre todos eles, aquele que produz o alinhamento de maior similaridade com a sequência alvo t, em tempo polinomial usando uma estratégia de programação dinâmica. Nesta estratégia, um problema é decomposto em subproblemas menores e as soluções destes são combinadas para formar a solução do problema original [9].

O algoritmo definido por Gelfand para o problema do alinhamento *spliced* utiliza as seguintes estruturas de dados:

- Um vetor de símbolos s que representa a sequência base sendo investigada, onde $m = |s|$ é o tamanho de s;
- Um vetor de símbolos t que representa a sequência alvo, com $|t| = n$, onde $m > n$;
- Um conjunto ordenado β, com todos os éxons candidatos, $\beta = \{b_1, b_2, ..., b_k, b_{k+1}, ..., b_{|\beta|}\}$;
- Uma função de pontuação ω para a comparação entre dois símbolos u e v, tal que:

$$\omega(u,v) = \begin{cases} 1, \text{ se } u = v \ (match) \\ -1, \text{ se } u \neq v \ (mismatch) \\ -2, \text{ se } u = \text{`–'} \text{ ou } v = \text{`–'} \ (linear\ gap\ penalty) \end{cases} \tag{3.1}$$

Define-se também:

- $first(k)$: índice do primeiro símbolo do éxon b_k em s;

- $last(k)$: índice do último símbolo do éxon b_k em s;
- $\beta(i)$: conjunto de todos os éxons candidatos que pertencem a β, mas terminam antes da posição i de s, isto é, $\beta(i) = \{k : last(k) < i\}$.

As pontuações de similaridade são mantidas em uma estrutura tridimensional S, que é calculada pela relação de recorrência das Equações 3.2 e 3.3. Se $s[i]$ não é o primeiro símbolo do éxon b_k, então:

$$S(i,j,k) = \max \begin{cases} S(i-1, j-1, k) + \omega(s[i], t[j]) \\ S(i-1, j, k) + \omega(s[i], \text{`-'}) \\ S(i, j-1, k) + \omega(\text{`-'}, t[j]) \end{cases} \text{, se } i \neq first(k) \tag{3.2}$$

Por outro lado, se $s[i]$ é o primeiro símbolo do éxon b_k, então:

$$S(i,j,k) = \max \begin{cases} \max_{l \in \beta(first(k))} S(last(l), j-1, l) + \omega(s[i], t[j]) \\ \max_{l \in \beta(first(k))} S(last(l), j, l) + \omega(s[last(l)], \text{`-'}) \\ S(i, j-1, k) + \omega(\text{`-'}, t[j]) \end{cases} \text{, se } i = first(k) \tag{3.3}$$

Nestas equações, o índice i corresponde à posição na sequência base s (e por consequência no éxon b_k), $1 \leq i \leq m$, o índice j corresponde à posição na sequência alvo t, $1 \leq j \leq n$, e k é o índice do éxon b_k sendo analisado, $1 \leq k \leq |\beta|$. Desta forma, $S(i,j,k)$ representa a similaridade entre a concatenação dos elementos de um subconjunto de β até o éxon b_k inclusive (até a posição i de b_k) e a sequência alvo t, até sua posição j. A pontuação do alinhamento *spliced* ótimo entre s e t, considerando β, é obtida ao final do algoritmo através da Equação 3.4.

$$Sim(s,t,\beta) = \max_{1 \leq k \leq |\beta|} S(last(k), n, k) \tag{3.4}$$

Como a relação de recorrência das Equações 3.2 e 3.3 calcula a estrutura S tridimensional, um projeto ingênuo do algoritmo que calcula a pontuação do alinhamento *spliced* ótimo levaria a um algoritmo com complexidades de tempo e espaço $O(m \times n \times |\beta|)$. Entretanto, nem todas as células da estrutura S precisam ser calculadas, apenas as faixas referentes ao alinhamento entre os éxons e a sequência alvo. Com isso, a complexidade de espaço depende apenas do tamanho da sequência alvo e do tamanho dos éxons, e é dada por $O(n \times \sum_{k=1}^{|\beta|} |b_k|)$. A complexidade de tempo é $O(n \times \sum_{k=1}^{|\beta|} |b_k| + |\beta|^2 \times n)$ [1].

Após o cálculo da pontuação do alinhamento *spliced* ótimo entre os éxons candidatos e a sequência alvo, é necessário encontrar o alinhamento ótimo propriamente dito e determinar o subconjunto dos éxons candidatos de maior similaridade com a sequência alvo. Isto pode ser realizado percorrendo-se de forma inversa a estrutura S de similaridade a partir da pontuação do alinhamento ótimo dada pelas Equações 3.4, 3.2 e 3.3.

3.4 Dependências de Dados

Apesar do tempo de execução polinomial, devido ao crescimento exponencial das bases de dados biológicos, o algoritmo de alinhamento *spliced* pode ser bastante custoso computacionalmente, tanto em termos de tempo de execução quanto em termos de espaço em memória. Uma ideia para reduzir o tempo de execução do algoritmo é calcular células da estrutura S em paralelo. Para investigar formas de explorar paralelismo na execução do algoritmo, as dependências de dados no cálculo destas células são analisadas.

De acordo com a relação de recorrência das Equações 3.2 e 3.3 que calculam a estrutura S, existem dois casos principais no cálculo de $S(i, j, k)$. O primeiro caso, $i \neq first(k)$, ocorre quando a célula sendo calculada não corresponde ao primeiro símbolo do éxon b_k e é dado pela Equação 3.2. Neste caso, três dependências de dados entre células da estrutura S são identificadas, como mostra a Figura 3.4. Nesta figura, representa-se como uma matriz bidimensional S_k, a seção da estrutura tridimensional S correspondente ao éxon b_k. Esta seção armazena a pontuação do melhor alinhamento entre um subconjunto de β até b_k, inclusive, e a sequência alvo t. A região sombreada representa as células de S_k relativas a b_k que são calculadas pelo algoritmo. As setas representam as dependências de dados, isto é, indicam quais células são utilizadas para calcular o valor da célula $S(i, j, k)$: são as células $S(i, j-1, k)$, $S(i-1, j, k)$ e $S(i-1, j-1, k)$.

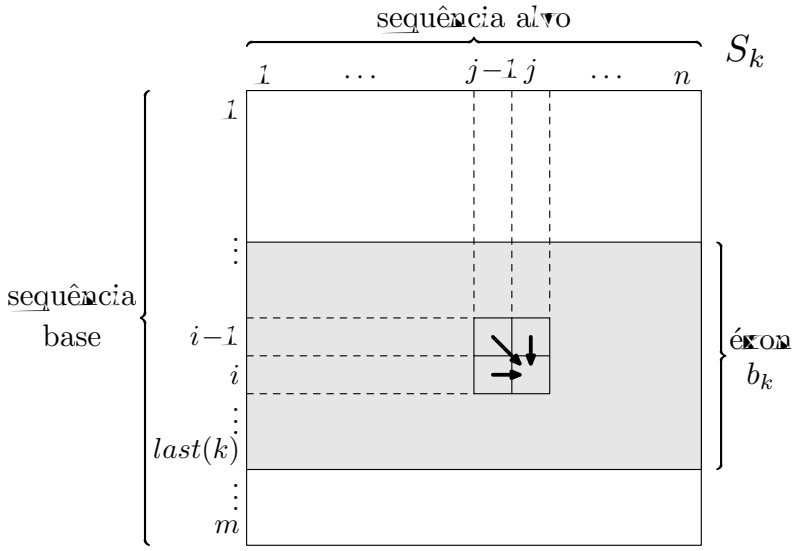

Figura 3.4: Dependências de dados no cálculo de $S(i, j, k)$, se $i \neq first(k)$

O segundo caso, $i = first(k)$, ocorre quando a célula sendo calculada corresponde ao primeiro símbolo do éxon b_k e é dado pela Equação 3.3. Neste caso, mais dependências de dados entre células da estrutura S são identificadas, ilustradas na Figura 3.5. Nesta figura, há a matriz bidimensional S_k correspondente ao éxon b_k e há também a matriz bidimensional S_l correspondente ao éxon b_l, que aqui representa todos os éxons b_l tais que $last(l) < first(k)$, isto é, todos os éxons que terminam antes do início do éxon b_k. As

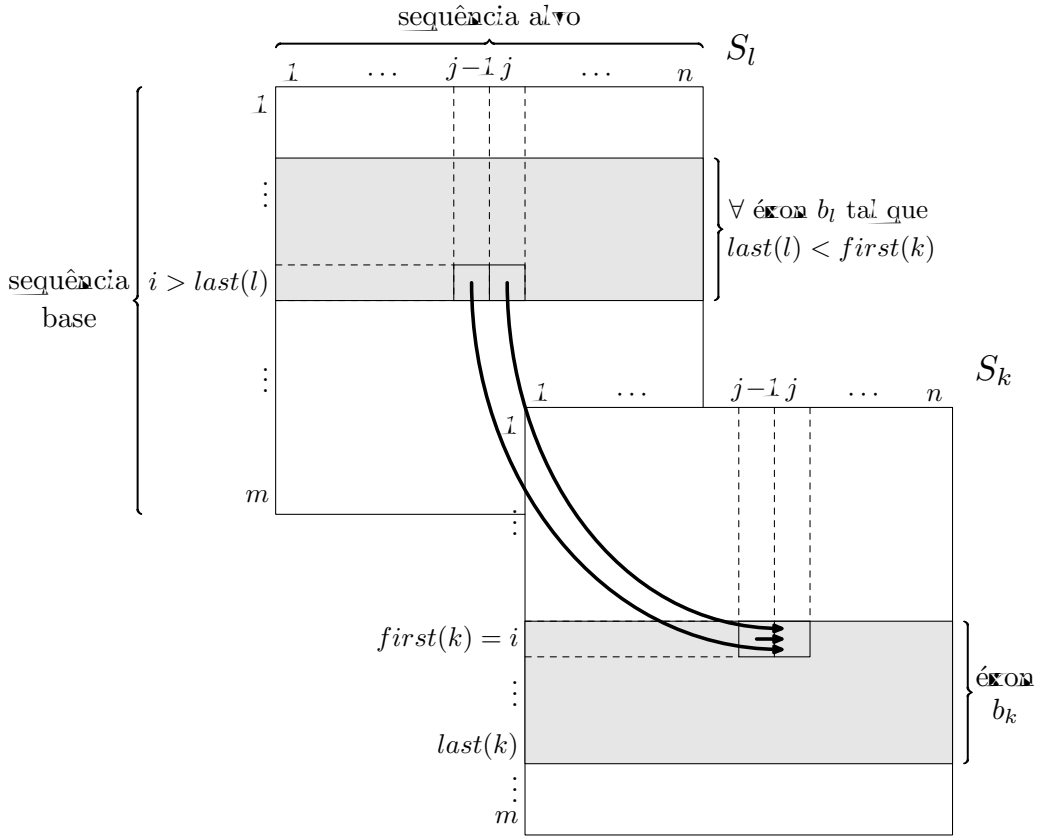

Figura 3.5: Dependências de dados no cálculo de $S(i, j, k)$, se $i = first(k)$

dependências identificadas indicam que, para calcular a célula $S(i, j, k)$ correspondente ao primeiro símbolo do éxon b_k, utiliza-se a célula $S(i, j-1, k)$, relativa a este mesmo éxon, além das células $S(last(l), j-1, l)$ e $S(last(l), j, l)$ de todo éxon b_l que termina antes do início do éxon b_k.

Neste caso, informações relativas aos éxons anteriores também são usadas, uma vez que o objetivo é encontrar o subconjunto dos éxons candidatos com maior similaridade com a sequencia alvo.

3.5 Exemplo de Execução do Algoritmo de Gelfand

Buscando um melhor entendimento do algoritmo de Gelfand esta seção apresenta um exemplo da sua execução. Sejam s uma sequência base, s = ACCGTATGT, e $\beta = \{b_1, b_2, b_3, b_4, b_5\}$, um conjunto de éxons candidatos de s, onde $b_1 = 1..2$, $b_2 = 3..4$, $b_3 = 4..5$, $b_4 = 6..8$ e $b_5 = 8..9$. Seja também t a sequência alvo, t = CCGGT. A Figura 3.6 ilustra o exemplo apresentado.

As Equações 3.2 e 3.3 são usadas para calcular as pontuações de similaridade. A estrutura tridimensional S de pontuações de similaridade é representada por $|\beta|$ matrizes

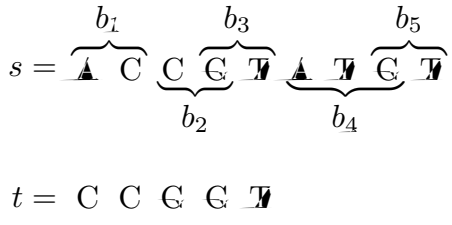

Figura 3.6: Sequência base s, éxons candidatos b_1, b_2, b_3, b_4, b_5 e sequência alvo t

bidimensionais, onde cada matriz bidimensional S_k armazena a pontuação do melhor alinhamento entre um subconjunto de β até o éxon b_k, inclusive, e a sequência alvo t.

A Figura 3.7 mostra a matriz de pontuação S_1 do éxon b_1. No cálculo da primeira coluna da matriz, realiza-se o alinhamento de um símbolo com um *gap* ('–') na primeira linha e de dois símbolos com dois *gaps* na segunda linha, então os valores para essa inicialização são dados de acordo com a função de pontuação ω. Esta condição é utilizada somente na inicialização e isso também ocorre no cálculo da primeira coluna das matrizes das Figuras 3.8, 3.9, 3.10 e 3.11, correspondentes aos éxons b_2, b_3, b_4 e b_5, respectivamente.

No cálculo da primeira linha de S_1 também realiza-se o alinhamento de um ou mais símbolos com um ou mais *gaps*, obtendo-se os valores de inicialização. Isto também ocorre no cálculo da primeira linha das matrizes das Figuras 3.8, 3.9, 3.10 e 3.11.

Figura 3.7: Matriz de pontuação S_1 do melhor alinhamento entre um subconjunto de β até o éxon b_1, inclusive, e t

	–	C	C	G	G	T
–	0	−2	−4	−6	−8	−10
A	−2	−1	−3	−5	−7	−9
C	−4	−1	0	−2	−4	−6

Para calcular a segunda linha de S_1 na Figura 3.7, utiliza-se a Equação 3.3, pois a segunda linha corresponde ao $first(1)$, o índice do primeiro símbolo de b_1 em s. Porém, como b_1 é o primeiro éxon a ser calculado, não há éxon anterior a ser considerado, isto é, que termine antes do início de b_1. Para calcular a terceira linha de S_1 a Equação 3.2 é utilizada, já que nesta linha $i \neq first(1)$.

A Figura 3.8 mostra a matriz de pontuação S_2 do éxon b_2, e assim como na matriz S_1, utiliza-se a Equação 3.3 para o cálculo da segunda linha e a Equação 3.2 para o cálculo da terceira linha. Porém neste caso, como b_1 termina antes do início de b_2, isto é, atende à condição $last(1) < first(2)$, as funções $\max_{l \in \beta(first(k))} S(last(l), j-1, l)$ e $\max_{l \in \beta(first(k))} S(last(l), j, l)$ são calculadas considerando o éxon b_1. Para calcular estas funções, basta encontrar os éxons que terminam antes da posição $first(k)$, e determinar o valor máximo entre eles, de acordo com a coluna que está sendo calculada.

A Figura 3.9 mostra a matriz de pontuação S_3 do éxon b_3 e seu cálculo é feito

Figura 3.8: Matriz de pontuação S_2 do melhor alinhamento entre um subconjunto de β até o éxon b_2, inclusive, e t

	–	C	C	G	G	T
–	0	−2	−4	−6	−8	−10
C	−2	−1	0	−1	−3	−5
G	−4	−3	−2	1	0	−2

de forma similar ao da matriz S_2. Inclusive no cálculo da segunda linha, utilizando a Equação 3.3, somente o éxon b_1 é considerado, pois embora a matriz S_2 já tenha sido calculada, o éxon b_2 não termina antes do início de b_3, isto é, não atende à condição $last(2) < first(3)$, o que o deixa fora do cálculo de S_3.

Figura 3.9: Matriz de pontuação S_3 do melhor alinhamento entre um subconjunto de β até o éxon b_3, inclusive, e t

	–	C	C	G	G	T
–	0	−2	−4	−6	−8	−10
G	−2	−1	−2	1	−1	−3
T	−4	−3	−3	−1	0	0

A Figura 3.10 mostra a matriz de pontuação S_4 do éxon b_4, e para calcular sua segunda linha, todos os éxons anteriores são considerados, visto que eles terminam antes do início de b_4, ou seja $last(1) < first(4)$, $last(2) < first(4)$ e $last(3) < first(4)$. No restante, os cálculos são similares aos das matrizes anteriores.

Figura 3.10: Matriz de pontuação S_4 do melhor alinhamento entre um subconjunto de β até o éxon b_4, inclusive, e t

	–	C	C	G	G	T
–	0	−2	−4	−6	−8	−10
A	−2	−1	0	1	0	0
T	−4	−3	−2	−3	−2	1
G	−6	−5	−4	−1	−2	−1

A Figura 3.11 mostra a matriz de pontuação S_5 do éxon b_5. O éxon b_4 não termina antes do início de b_5, logo, no cálculo da segunda linha de S_5, o éxon b_4 não é considerado, tornando o cálculo similar ao da matriz S_4.

Figura 3.11: Matriz de pontuação S_5 do melhor alinhamento entre um subconjunto de β até o éxon b_5, inclusive, e t

	–	C	C	G	G	T
–	0	−2	−4	−6	−8	−10
G	−2	−1	−2	1	2	0
T	−4	−3	−3	−1	0	3

Após calcular as pontuações de similaridade, a pontuação do alinhamento ótimo entre a sequência base s e a sequência alvo t, considerando o conjunto de éxons

candidatos β, é obtida através da Equação 3.4. No caso deste exemplo, $Sim(s, t, \beta) = \max\{-6, -2, 0, -1, 3\} = 3$.

A partir da estrutura de similaridade S é possível construir o alinhamento ótimo formado pelas escolhas feitas ao preencher cada célula da estrutura através das Equações 3.2 e 3.3. A Figura 3.12 ilustra o alinhamento *spliced* ótimo encontrado para o exemplo da Figura 3.6.

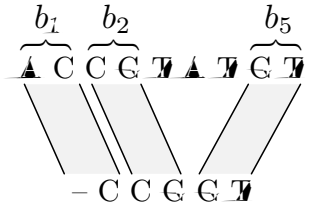

Figura 3.12: Alinhamento *spliced* ótimo para o exemplo da Figura 3.6: entre sequências base s e alvo t, considerando conjunto de éxons candidatos β

Capítulo 4

Trabalhos Relacionados

Algumas soluções paralelas para o problema da predição de genes são apresentadas na literatura. No entanto, a maioria dessas soluções não se baseia no problema do alinhamento *spliced*, e sim em métodos intrínsecos baseados em estatísticas e com aplicação a organismos procariontes. Métodos de predição de genes para organismos eucariontes tendem a ser mais complexos do que para organismos procariontes, devido à estrutura de éxons e íntrons de um gene em eucariotos [2].

Diferentemente das abordagens baseadas em similaridade, que usam sequências de proteínas relacionadas a genes codificados em uma sequência de DNA previamente anotada como um modelo para o reconhecimento de genes desconhecidos na sequência de DNA investigada, os métodos estatísticos procuram por características que aparecem com frequência nos genes e raramente aparecem fora deles, e baseam-se na detecção de variações estatísticas entre regiões codificantes e não codificantes da sequência investigada [21].

Os trabalhos em [5, 6, 7, 33] descrevem soluções paralelas para a predição de genes utilizando métodos estatísticos. O sistema paralelo de predição de genes descrito em [5] executa em um cluster e combina abordagens estatísticas com abordagens baseadas em homologia, para organismos procariontes. Um sistema de hardware para a predição de genes é proposto em [7], baseado em uma abordagem estatística para procariotos, e implementado em uma FPGA. Um algoritmo paralelo para predição de genes é apresentado em [33], utilizando uma abordagem estatística para eucariotos e executando em um *cluster*. Dois aceleradores diferentes para predição de genes, sendo um executado em FPGA e outro em GPU, são apresentados em [6] e baseados em uma abordagem estatística para eucariotos.

Em [20] os autores apresentam uma revisão sobre ferramentas para o alinhamento *spliced* que aplicam soluções heurísticas baseadas em várias abordagens. Algumas dessas ferramentas podem ser executadas em processadores *multi-core* para obter um melhor

desempenho.

As soluções paralelas exatas para o problema do alinhamento *spliced* encontradas na literatura são apresentadas a seguir. Entretanto, esses trabalhos não são baseados no algoritmo de Gelfand e/ou não fornecem informações suficientes sobre seu desenvolvimento.

4.1 Soluções para Alinhamentos no Cell BE

O trabalho de Sarje e Aluru [34, 35, 36] tem como proposta criar soluções paralelas para os problemas de alinhamento global, alinhamento local, alinhamento sintênico e alinhamento *spliced* usando como plataforma o *Cell Broadband Engine* (Cell BE), microprocessador desenvolvido em conjunto pela Sony, Toshiba e IBM [18].

Entretanto, a abordagem usada para o alinhamento *spliced* neste trabalho é diferente da abordagem definida por Gelfand [14], tratando o alinhamento *spliced* como um caso especial do alinhamento sintênico, onde a principal diferença é que não existe a informação de éxons candidatos.

No alinhamento sintênico a entrada de dados é formada por apenas duas sequências s e t, e deseja-se obter uma série ordenada de subsequências de s que esteja alinhada com outra série ordenada de subsequências de t [17]. A Figura 4.1 mostra a ideia do alinhamento sintênico e do alinhamento *spliced* tratado como caso especial do primeiro. No alinhamento sintênico, as subsequências alinhadas das duas sequências de entrada são representadas pelas linhas grossas.

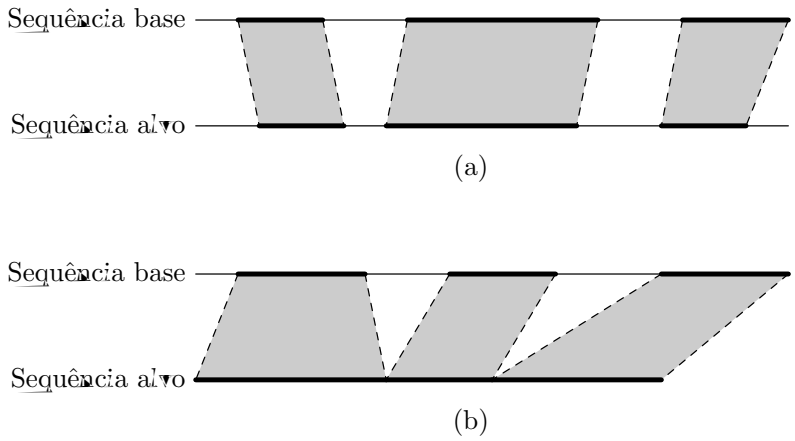

Figura 4.1: (a) Alinhamento sintênico; (b) Alinhamento *spliced* tratado como caso especial do alinhamento sintênico

No problema do alinhamento *spliced* proposto originalmente, o conjunto de éxons candidatos é fornecido como entrada e os éxons são alinhados com a sequência alvo. Entretanto, na abordagem usada em [34, 35], a entrada de dados é formada apenas pelas

sequências base e alvo. O alinhamento sintênico é então aplicado, com a condição adicional de que as subsequências obtidas da sequência alvo, se concatenadas, formam a sequência completa. A Figura 4.1(b) ilustra este alinhamento, onde as subsequências alinhadas estão representadas com linhas grossas.

A forma de paralelismo utilizada é baseada na técnica de *wave-front*, ilustrada na Figura 4.2. A matriz de programação dinâmica é dividida em blocos e cada processador é responsável por calcular uma coluna de blocos. O cálculo dos blocos segue em anti-diagonais, com os blocos de uma mesma anti-diagonal sendo calculados em paralelo.

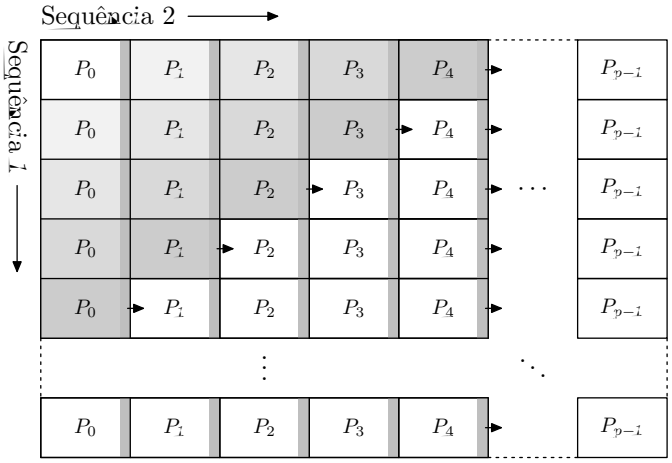

Figura 4.2: Técnica de *wave-front* usada em [35]

Outra estratégia adotada em [34, 35] é o método de alinhamento usando espaço linear, um algoritmo proposto por Hirschberg [16] que, ao invés de armazenar a matriz inteira de programação dinâmica, armazena somente três linhas da matriz. A Figura 4.3 mostra as dependências de dados usualmente encontradas no cálculo da matriz de similaridades. O cálculo da célula $[i, j]$ depende da célula $[i-1, j]$ acima, da célula $[i, j-1]$ do lado esquerdo, e da célula $[i-1, j-1]$ anterior na diagonal.

Portanto, para o cálculo de uma célula, precisa-se apenas da linha atual e da linha anterior. Desta forma, a pontuação do alinhamento ótimo pode ser calculada utilizando espaço em memória linear em relação ao tamanho das entradas, ao invés de quadrático. Para tanto, a matriz de programação dinâmica não é armazenada, durante seu cálculo. Apenas a linha corrente sendo calculada e a linha anteriormente calculada são armazenadas.

Entretanto, além de calcular a pontuação do alinhamento ótimo, também é necessário determinar o alinhamento ótimo encontrado, isto é, quais símbolos de uma sequência estão alinhados com quais símbolos de outra sequência. Isto corresponde a determinar quais são os pontos dentro da matriz de programação dinâmica que descrevem este alinhamento.

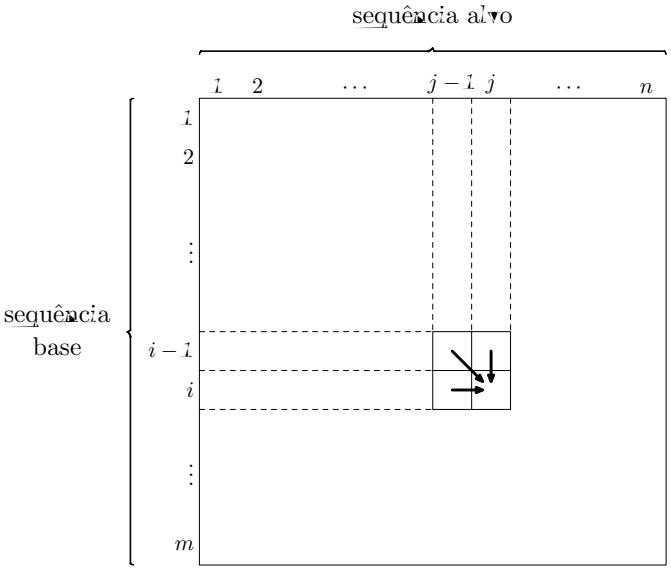

Figura 4.3: Dependências de dados usuais no cálculo da matriz de similaridades

O algoritmo de Hirschberg para encontrar um alinhamento ótimo entre duas sequências s de tamanho m e t de tamanho n usando espaço linear usa a técnica de divisão e conquista. Inicialmente, calcula-se a pontuação do alinhamento ótimo entre s e t em espaço linear e armazena-se a linha $\frac{m}{2}$ desta matriz de programação dinâmica. Em seguida, calcula-se a pontuação do alinhamento ótimo entre o reverso de s e o reverso de t e armazena-se a linha $\frac{m}{2}$ desta nova matriz de programação dinâmica.

Usando as duas linhas armazenadas, encontra-se a coluna onde o alinhamento ótimo entre s e t passa pela linha $\frac{m}{2}$ da matriz de similaridade. Este ponto é usado para dividir o problema em dois subproblemas, resolvidos recursivamente, até que todos os pontos do alinhamento sejam encontrados. A Figura 4.4 ilustra essa estratégia.

Dessa forma, a matriz de programação dinâmica nunca estará inteiramente em memória, no entanto troca-se espaço em memória por mais processamento, dado que ocorre o cálculo repetido de algumas partes da matriz. Mesmo assim, a complexidade de tempo quadrática do algoritmo é mantida. Este método é interessante quando as sequências a serem comparadas são muito grandes para serem armazenadas completamente em memória.

O método para encontrar um alinhamento ótimo em espaço linear também é realizado em paralelo em [34, 35]. Para realizar esta tarefa em paralelo, cada processador fica responsável por um bloco de colunas e por encontrar a parte do alinhamento ótimo neste bloco de colunas, usando somente uma parte das linhas e a última coluna atribuídas ao processador anterior, como mostra a Figura 4.5.

A plataforma utilizada, o Cell BE, é um microprocessador heterogêneo com oito núcleos de co-processamento com a arquitetura SIMD denominados SPEs (*Synergistic*

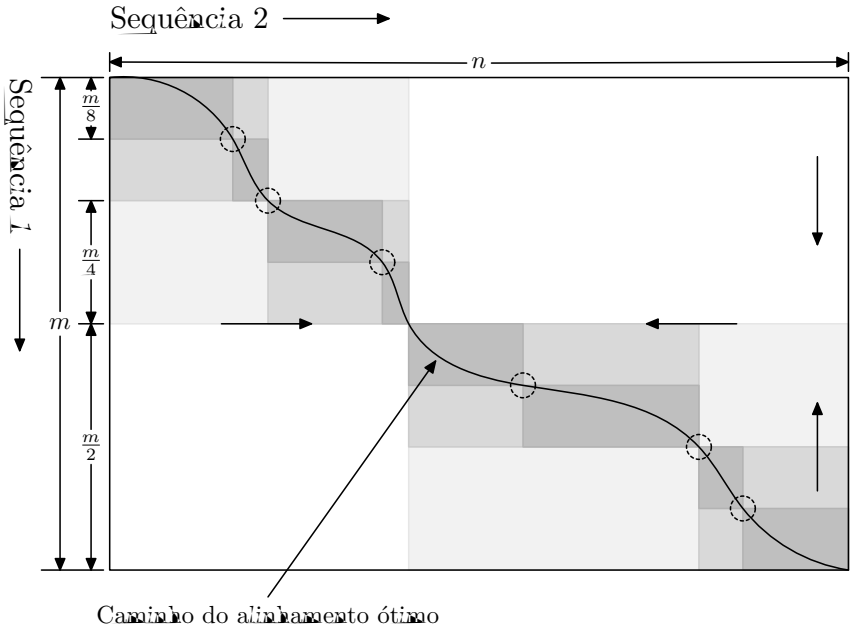

Figura 4.4: Método de espaço linear de Hirschberg

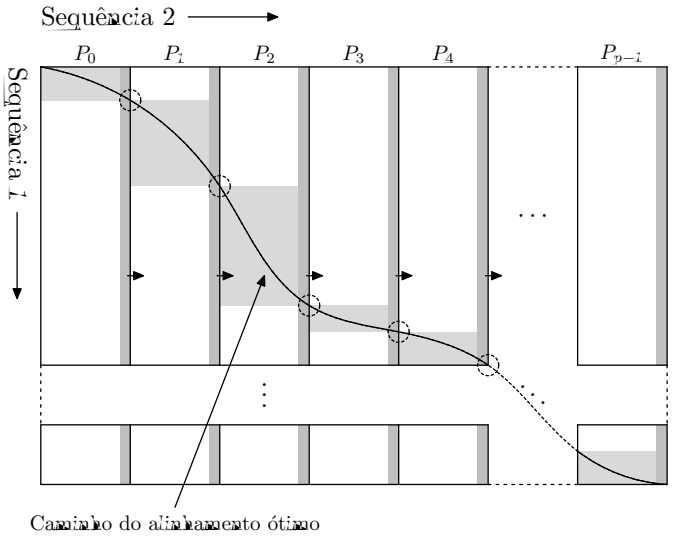

Figura 4.5: Obtenção dos pontos do alinhamento ótimo em paralelo aplicada em [35]

Processing Units), mais um núcleo principal com arquitetura SMT (*Simultaneous Multithreading*). Foram utilizados dois microprocessadores Cell BE, explorando apenas o paralelismo de dados com um total de 16 SPEs. Os *speedups* obtidos foram 8 para o alinhamento global, 7 para o alinhamento *spliced* e 6 para o alinhamento sintênico, em relação às suas respectivas implementações sequenciais.

4.2 *Framework* Baseado em Programação Dinâmica para *Cluster*

O trabalho desenvolvido por Liu e Schmidt [25] propõe um *framework* para resolver problemas de Bioinformática que usam soluções baseadas em programação dinâmica. Apesar de não especificar como foi desenvolvido esse *framework*, o trabalho cita o uso do método de espaço linear de Hirschberg [16] e a criação de classes genéricas relacionadas a três aspectos dos algoritmos de programação dinâmica aplicados à comparação de sequências. Os três aspectos são:

- os valores iniciais para a matriz de programação dinâmica devem ser especificados;
- as relações de dependência de dados entre as células da matriz devem ser especificadas;
- a ordem para calcular as células da matriz deve ser especificada.

A plataforma utilizada foi um *cluster* de memória distribuída com oito nós, cada nó com quatro processadores de 1GHz e 1GB de RAM para cada processador. A Tabela 4.1 mostra os *speedups* obtidos na implementação de alguns algoritmos de alinhamento usando o *framework* proposto em relação à execução em apenas um processador. Alguns *speedups* superlineares são reportados, isto é, *speedups* maiores que o número de processadores utilizados, devido ao aumento da quantidade de dados nas memórias caches com o aumento do número de processadores. Detalhes sobre a solução adotada para o alinhamento *spliced* não são fornecidos.

Tabela 4.1: *Speedups* obtidos em [25] para alguns algoritmos de Bioinformática em *cluster*

Algoritmo e tamanho dos dados de entrada	Número de processadores				
	2	4	8	16	32
Smith-Waterman com penalidade de *gap* linear (90000×90000)	1,87	3,7	7,37	14,74	28,25
Smith-Waterman com penalidade de *gap* afim (70000×70000)	1,85	3,68	7,31	14,13	27,65
Alinhamento sintênico (60000×60000)	2	4,25	8,31	16,84	33,68
Nussinov (5000×5000)	2,04	4,07	8,03	15,96	29,87
Alinhamento múltiplo para três sequências (3000×3000×3000)	2	4,04	8,72	17,7	33,37
Alinhamento *spliced* (4000×4000×1000)	2,12	4,55	9,19	18,2	36

Capítulo 5

Exploração de Paralelismo no Alinhamento *Spliced*

A partir da análise das dependências de dados do problema do alinhamento *spliced*, são identificadas formas de calcular os dados explorando paralelismo, porém mantendo a solução correta. O potencial de paralelismo do problema é investigado e dimensionado para o conjunto de dados biológicos utilizado e um modelo de estimativa de desempenho de soluções paralelas para este problema é proposto.

5.1 Formas de Paralelismo no Alinhamento *Spliced*

As Figuras 3.4 e 3.5 mostram as dependências de dados presentes no cálculo da célula $S(i, j, k)$ da estrutura de similaridade, através do algoritmo de Gelfand para o problema do alinhamento *spliced*. A partir da análise dessas dependências, duas formas de explorar paralelismo na execução do algoritmo de Gelfand são encontradas:

- Paralelismo intra-éxon, que calcula em paralelo as células de uma anti-diagonal da matriz de pontuação S_k de um éxon candidato b_k;
- Paralelismo inter-éxon, que calcula em paralelo as células da matriz S_k de dois ou mais éxons b_k, desde que estes éxons possuam elementos da sequência base em comum.

5.1.1 Paralelismo Intra-éxon

A Figura 5.1 mostra apenas a parte da matriz S_k correspondente ao éxon b_k. A região sombreada representa células que já foram calculadas e a linha pontilhada indica

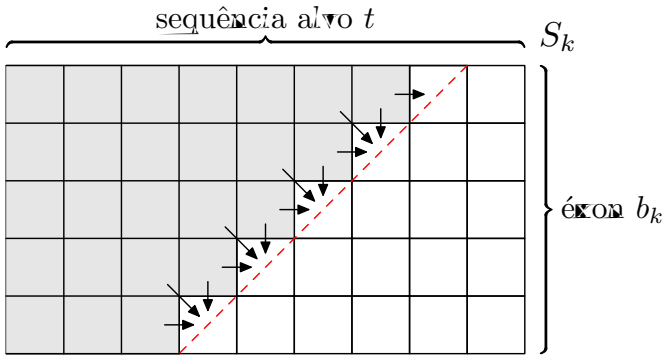

Figura 5.1: Paralelismo intra-éxon: células em uma anti-diagonal da matriz de pontuação S_k, do éxon b_k, podem ser calculadas em paralelo

células em uma anti-diagonal. Dadas as dependências para calcular $S(i, j, k)$ quando i não é o primeiro símbolo de b_k (mostradas na Figura 3.4), as dependências para calcular as células da anti-diagonal são como indicadas pelas setas na Figura 5.1.

Claramente todas as células nesta anti-diagonal podem ser calculadas em paralelo, pois não há dependências entre elas. Denomina-se **paralelismo intra-éxon** o cálculo em paralelo das células em uma anti-diagonal da matriz S_k correspondente ao éxon b_k. Esta forma de paralelismo é encontrada em alguns outros problemas de alinhamento em Bioinformática. Todavia, a dificuldade é explorar paralelismo além das fronteiras dos éxons, isto é, lidar com as dependências de dados mostradas na Figura 3.5.

5.1.2 Paralelismo Inter-éxon

As dependências de dados da Figura 3.5, para calcular $S(i, j, k)$ quando i é o primeiro símbolo de b_k, impedem que as matrizes S_l e S_k sejam calculadas em paralelo, para todos os éxons b_l que terminam antes do início de b_k, isto é, tal que $last(l) < first(k)$. Neste caso, os éxons b_l e b_k não possuem nenhum trecho da sequência base s em comum. Todavia, se b_l e b_k possuem algum trecho da s em comum, isto é, se $last(l) \geq first(k)$, as dependências da Figura 3.5 não se aplicam.

A Figura 5.2 mostra apenas as partes das matrizes S_l e S_k correspondentes aos éxons b_l e b_k, respectivamente. Se b_l e b_k possuem algum trecho da sequência base s em comum, S_l e S_k podem ser calculadas em paralelo. Denomina-se **paralelismo inter-éxon** o cálculo em paralelo das células das matrizes S_l e S_k de diferentes éxons b_l e b_k que possuem algum trecho da sequência base s em comum.

Ambas as formas de paralelismo, paralelismo intra- e inter-éxon, podem ser combinadas e usadas em conjunto, como ilustrado na Figura 5.2. Portanto, uma anti-diagonal de S_l pode ser calculada ao mesmo tempo que uma anti-diagonal de S_k, como representado pelas linhas tracejadas na figura.

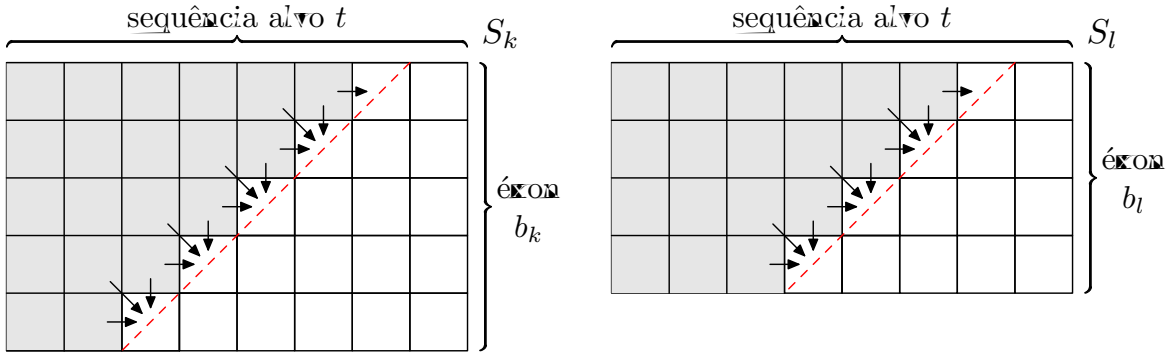

Figura 5.2: Paralelismo inter-éxon: anti-diagonais das matrizes de pontuação S_k e S_l, dos éxons independentes b_k e b_l, respectivamente, podem ser calculadas em paralelo

5.2 Dados Biológicos Utilizados

Uma vez identificadas as formas de exploração de paralelismo no problema do alinhamento *spliced*, a análise do potencial de paralelismo do problema é de suma importância para o planejamento e desenvolvimento de soluções paralelas. Mesmo que, em teoria, os paralelismos intra- e inter-éxon possam ser explorados, é necessário verificar se dados biológicos reais exibem potencial de paralelismo, isto é, se surgem oportunidades de explorar estas formas de paralelismo quando dados reais são usados.

O conjunto de dados biológicos utilizado neste trabalho foi construído em [24]. Este conjunto contém 240 sequências base, obtidas a partir do projeto ENCODE [12] que tem como objetivo identificar todos os elementos funcionais do genoma humano. Cada sequência base corresponde a um gene do organismo *Homo sapiens*, juntamente com as 1000 bases que o antecedem e sucedem.

Para cada sequência base, até cinco sequências alvo são usadas. Estas sequências são sequências de cDNA de genes homólogos ao gene da sequência base, obtidas a partir da base de dados Homologene [37], disponível no site do NCBI. A ferramenta GENSCAN [3] de identificação de genes foi usada para construir o conjunto de éxons candidatos das sequências base, com éxons verdadeiros e falsos. Mais detalhes sobre a construção do conjunto de dados podem ser encontrados em [24].

A Tabela 5.1 mostra informações das sequências base do conjunto de dados empregado, e dos éxons candidatos destas sequências. A Tabela 5.2 mostra algumas informações das sequências alvo do conjunto de dados empregado.

Cada execução do algoritmo de alinhamento *spliced* usa como entrada uma sequência base, seu conjunto de éxons candidatos e uma sequência alvo. O conjunto de dados biológicos descrito produz um total de 1.010 casos de teste diferentes e representativos e provê uma base sólida para a realização de uma avaliação experimental abrangente.

Tabela 5.1: Sequências base e éxons do conjunto de dados biológicos empregado

Informação	Mínimo	Médio	Máximo
Tamanho da sequência base	2848	42181,10	575746
Número de éxons de uma sequência base	2	55,08	622
Tamanho dos éxons de uma sequência base	1	149,35	4778
Soma do tamanho dos éxons de uma sequência base	442	7711,73	71873
Número total de sequências base	240		

Tabela 5.2: Sequências alvo do conjunto de dados biológicos empregado

Informação	Mínimo	Médio	Máximo
Número de sequências alvo por sequência base	1	4,20	5
Tamanho da sequência alvo	189	1460,48	8652

5.3 Potencial de Paralelismo nos Dados Biológicos

Deseja-se dimensionar o volume de paralelismo que estaria de fato disponível para ser explorado por uma implementação paralela, usando o conjunto de dados biológicos. Para isso, dois experimentos são realizados para dimensionar separadamente o volume de paralelismo intra-éxon e inter-éxon disponível.

O volume de paralelismo intra-éxon disponível para ser explorado em uma execução depende do tamanho das anti-diagonais (do número de células que podem ser calculadas em paralelo) da matriz S_k, para cada éxon b_k da sequência base. O tamanho das anti-diagonais de S_k é definido pelo mínimo entre o comprimento da sequência alvo e o comprimento do éxon b_k, como mostrado na Figura 5.1. Como éxons representam subsequências da sequência base e o alinhamento será entre um subconjunto de éxons e a sequência alvo, supõe-se, sem perda de generalidade, que o comprimento do éxon é sempre menor que o comprimento da sequência alvo, excluindo-se portanto comparações entre genes muito distintos, onde a sequência alvo seria menor que os possíveis éxons que a formariam. Portanto, o volume de paralelismo intra-éxon disponível para ser explorado depende do comprimento dos éxons da sequência base.

Para cada caso de teste do conjunto de dados, calcula-se a porcentagem de éxons com um certo comprimento em relação ao número total de éxons, para cada comprimento de éxon. Os resultados de todos os casos de teste são sumarizados através de uma média, produzindo o resultado do que denomina-se **caso de teste médio**. A Figura 5.3 mostra estes resultados de potencial de paralelismo intra-éxon para o caso de teste médio. Este histograma representa a porcentagem de éxons em relação ao número total de éxons, para cada comprimento de éxon, no caso de teste médio. O comprimento do éxon corresponde ao número de células que podem ser calculadas em paralelo, representadas em faixas de 32 células. Por exemplo, a primeira barra no histograma indica que, no caso de teste

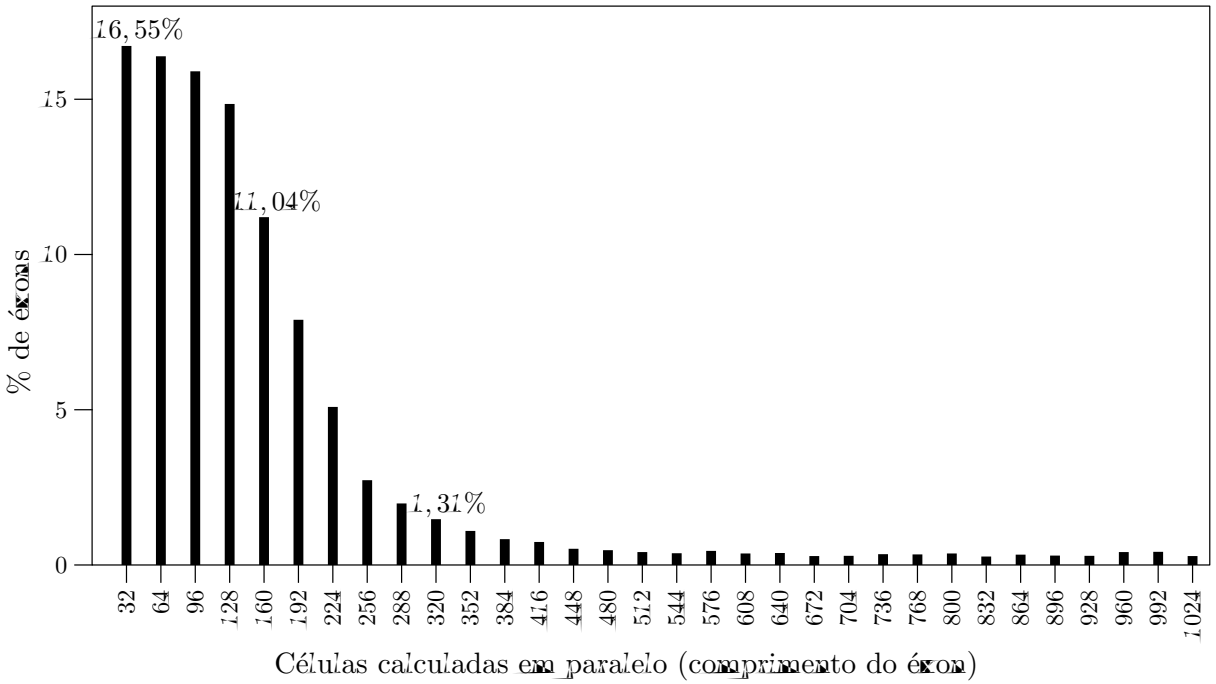

Figura 5.3: Potencial de paralelismo intra-éxon no caso de teste médio

médio, há 16,55% de éxons com apenas 32 células para serem calculadas em paralelo. Por clareza, barras com porcentagem de éxons inferior 0,1% (que correspondem a éxons com comprimentos maiores que 1024) são omitidas e somam 1,14% do número total de éxons.

Ao analisar os resultados obtidos com o experimento, observa-se que em média mais de 50% dos éxons possuem pelo menos 128 células que podem ser calculadas em paralelo, concluindo que o paralelismo intra-éxon é uma boa forma de paralelismo a ser explorada. Isso também motiva a procurar por outras formas de paralelismo que possam ser usadas em conjunto.

O volume de paralelismo inter-éxon disponível para ser explorado em uma execução depende do número de diferentes éxons b_k que tenham algum trecho da sequência base s em comum, pois assim as matrizes S_k correspondentes a estes éxons podem ser calculadas em paralelo. Em um caso de teste é possível haver vários grupos de éxons cujas matrizes podem ser calculadas em paralelo, porém sequencialmente em relação aos outros grupos.

Por exemplo, a Figura 5.4 mostra uma sequência base s com os éxons candidatos $b_1, ..., b_5$. Os éxons b_2 e b_3 possuem um símbolo em comum e b_4 e b_5 possuem dois símbolos em comum. Portanto, para calcular as pontuações de similaridade da estrutura S, respeitando as dependências de dados, inicialmente a matriz S_1 correspondente a b_1 deve ser calculada sozinha (um grupo de um éxon), pois todas as outras matrizes dependem dela. Em seguida, S_2 e S_3 podem ser calculadas em paralelo (um grupo de dois éxons), explorando paralelismo inter-éxons, porém S_4 e S_5 dependem delas e por isso não podem ser calculadas neste mesmo grupo. Finalmente, S_4 e S_5 também podem ser calculadas em

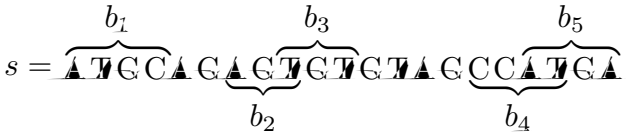

Figura 5.4: Sequência base s e éxons candidatos $b_1, ..., b_5$: matrizes S_2 e S_3 podem ser calculadas em paralelo, assim como S_4 e S_5

paralelo (outro grupo de dois éxons).

Para cada caso de teste do conjunto de dados, calcula-se a porcentagem de grupos com um determinado número de éxons em relação ao número total de grupos executados, para cada número de éxons. Novamente, os resultados de todos os casos de teste são sumarizados através de uma média, produzindo o resultado do caso de teste médio. A Figura 5.5 mostra os resultados de potencial de paralelismo inter-éxon para o caso de teste médio. Este histograma representa a porcentagem de grupos de éxons em relação ao número total de grupos executados, no caso de teste médio. O número de éxons em um grupo corresponde ao número de matrizes que podem ser calculadas em paralelo. Por exemplo, a segunda barra no histograma indica que, em média, 21,35% dos grupos são formados por dois éxons, logo duas matrizes podem ser calculadas em paralelo.

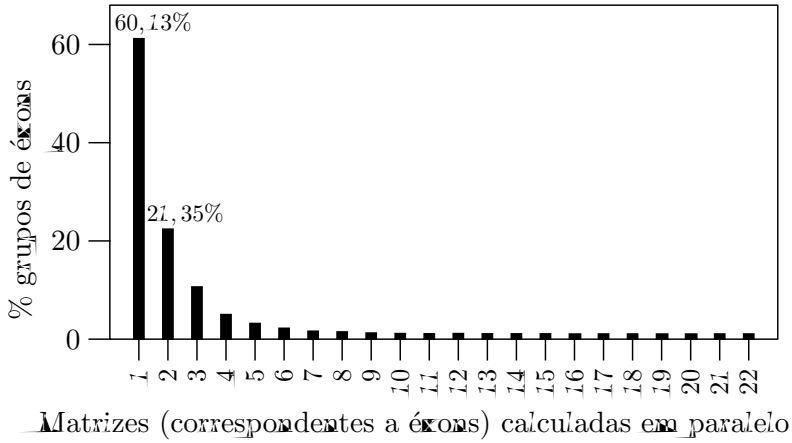

Figura 5.5: Potencial de paralelismo inter-éxon no caso de teste médio

Analisando os resultados, conclui-se que, em média, quase 40% dos grupos de éxons são formados por pelo menos dois éxons, quando o paralelismo inter-éxon pode ser explorado. Assim, combinar essa forma de paralelismo ao paralelismo intra-éxon pode ser uma estratégia promissora para obter um bom desempenho neste problema, pois o estudo também mostra que problema do alinhamento *spliced* não é de fácil paralelização, motivando ainda mais a pesquisa e o desenvolvimento de soluções paralelas para ele.

5.4 Estimativa do Desempenho das Soluções Paralelas

Antes de desenvolver e implementar soluções paralelas para o algoritmo de alinhamento *spliced*, deseja-se estimar o seu tempo de execução. A ideia é ter uma estimativa do ganho de desempenho que as soluções paralelas serão capazes de obter, em relação a uma solução sequencial, para decidir se é vantajoso desenvolvê-las ou não.

Usando o tempo para calcular uma célula da estrutura de similaridade S como a unidade de tempo, o tempo de execução estimado de uma solução é calculado considerando quais células podem ser calculadas em paralelo e quais células precisam ser calculadas sequencialmente pela solução, considerando as dependências de dados.

Neste modelo de estimativa de desempenho proposto, dada uma sequência base s, com um conjunto β de éxons candidatos, e uma sequência alvo t de comprimento n, o tempo de execução estimado de uma solução sequencial é simplesmente o número total de células de S, como indicado na Equação 5.1, porque a solução sequencial calcula as células uma de cada vez.

$$\text{tempo estimado}_{\text{seq}} = n \times \sum_{k=1}^{|\beta|} |b_k| \tag{5.1}$$

Em uma solução paralela que explora apenas o paralelismo intra-éxon, as matrizes S_k correspondentes aos éxons são calculadas sequencialmente em relação umas às outras. Entretanto, cada matriz S_k tem células de uma mesma anti-diagonal calculadas em paralelo, enquanto sucessivas anti-diagonais são calculadas uma após a outra. Desta forma, o tempo estimado para calcular S_k é o número de anti-diagonais que ela possui, que é a soma das suas dimensões (o comprimento do éxon e o comprimento da sequência alvo), menos um, isto é:

$$|b_k| + n - 1$$

Assim, o tempo de execução estimado de uma solução paralela que explora o paralelismo intra-éxon é obtido como indicado na Equação 5.2.

$$\text{tempo estimado}_{\text{par intra-éxon}} = \sum_{k=1}^{|\beta|} (|b_k| + n - 1) \tag{5.2}$$

Já em uma solução paralela que explora tanto o paralelismo intra- como inter-éxon, há grupos de éxons cujas matrizes são calculadas em paralelo umas com as outras, porém sequencialmente em relação aos outros grupos.

Dado um grupo $\mathcal{G}$ de éxons b_k, correspondentes a matrizes S_k que podem ser calculadas em paralelo, as anti-diagonais destas diferentes matrizes podem ser calculadas

em paralelo, e o tempo estimado para calcular o grupo $\mathcal{G}$ completo é determinado pela matriz que possui mais anti-diagonais para calcular. Portanto, o tempo estimado para calcular as matrizes de todos os éxons em $\mathcal{G}$ é:

$$\max_{\forall b_k \in \mathcal{G}} |b_k| + n - 1$$

Finalmente, o tempo de execução estimado de uma solução paralela que explora paralelismo intra- e inter-éxon é obtido considerando todos os grupos de éxons, como indicado na Equação 5.3.

$$\text{tempo estimado}_{\substack{\text{intra-éxon} \\ \text{inter-éxon}}}^{\text{par}} = \sum_{\forall \mathcal{G}} \max_{\forall b_k \in \mathcal{G}} |b_k| + n - 1 \tag{5.3}$$

Usando as Equações 5.1, 5.2 e 5.3, foram calculados os tempos de execução estimados para a solução sequencial e para as soluções paralelas intra-éxon e intra- e inter-éxon para todos os casos de teste do conjunto de dados biológicos. A Figura 5.6 compara os resultados, onde os casos de teste estão ordenados pelo tempo de execução estimado da solução sequencial e todos os tempos estimados são mostrados usando escala logarítmica.

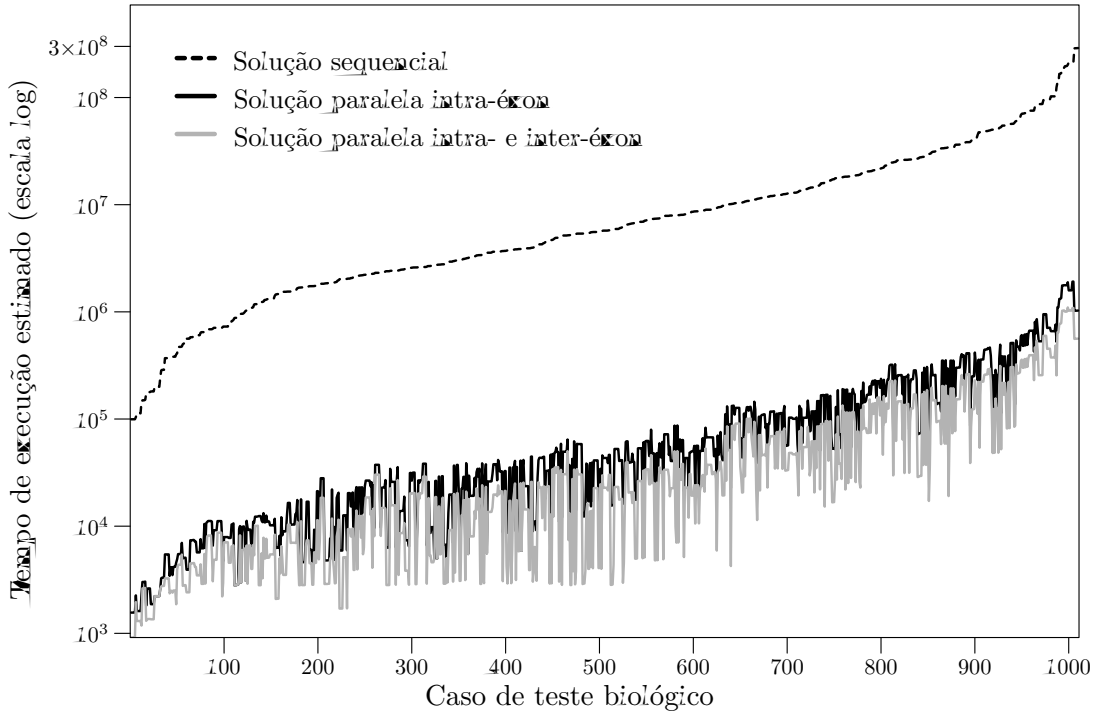

Figura 5.6: Tempos de execução estimados: soluções sequencial e paralelas

Os tempos estimados para a solução que explora apenas o paralelismo intra-éxon mostram uma melhoria de aproximadamente 100 vezes em relação aos tempos estimados para a solução sequencial. Os tempos estimados para a solução que explora ambas as formas de paralelismo são sempre melhores ou iguais que aqueles obtidos para a solução

que explora apenas paralelismo intra-éxon, com uma melhoria de aproximadamente 10 vezes em muitos casos de teste. Portanto, conclui-se que é vantajoso desenvolver soluções paralelas para o algoritmo de alinhamento *spliced* explorando paralelismo intra- e inter-éxon. Todavia, ao implementar as soluções paralelas, não se pode esperar obter ganhos de desempenho na mesma proporção das estimativas desta análise, pois a plataforma de implementação pode não ser capaz de explorar todo o paralelismo disponível nos dados biológicos e este modelo de estimativa de desempenho não considera qualquer custo adicional de controle, comunicação ou sincronização.

Capítulo 6

Soluções em GPU para o Alinhamento *Spliced*

Com o objetivo de avaliar separadamente o impacto dos paralelismos intra- e inter-éxon no desempenho, dois aceleradores em GPU foram desenvolvidos para o algoritmo de alinhamento *spliced*. O primeiro acelerador explora apenas o paralelismo intra-éxon, enquanto o segundo explora ambas as formas de paralelismo. Os aceleradores recebem como entrada um caso de teste, isto é, uma sequência base s, o seu conjunto β de éxons candidatos e uma sequência alvo t, e calcula a pontuação do alinhamento *spliced* ótimo. Os aceleradores foram desenvolvidos usando o modelo de programação CUDA.

6.1 Primeiro Acelerador: Exploração de Paralelismo Intra-éxon

Este primeiro acelerador em GPU implementa o paralelismo intra-éxon descrito na Subseção 5.1.1, onde apenas uma matriz de pontuação S_k, correspondente ao éxon b_k, é calculada por vez na GPU, respeitando a ordem dos éxons. O paralelismo ocorre no cálculo das anti-diagonais da matriz, de modo que cada célula de uma mesma anti-diagonal é calculada por uma *thread*, enquanto sucessivas anti-diagonais são calculadas sequencialmente, respeitando as dependências de dados vistas na Seção 3.4.

A Figura 6.1 apresenta a função, executada no *host*, responsável pelas invocações do *kernel*, para o primeiro acelerador em GPU. Visando um melhor entendimento, alguns detalhes são omitidos. O *kernel* é responsável por calcular a matriz de pontuação S_k correspondente a um éxon b_k. Para cada éxon do conjunto β de éxons candidatos, uma nova invocação do *kernel* é feita, porém somente após o fim da execução da invocação anterior, como mostrado no laço das linha 23 a 25.

```
1  void aceleradorParalIntra(int n, Exons b, int L, S[], base[], target[]) {
2     // n: comprimento da sequência alvo
3     // b: objeto da classe Exons, com informações dos éxons
4     // L: número de éxons candidatos
5     // S: estrutura de pontuação
6     // base: sequência base
7     // target: sequência alvo
8
9     // Declaração e inicialização de variáveis
10    ...
11
12    int nBlocos = 1;  // Número de blocos
13    int nThreads;     // Número de threads por bloco
14    if (n > 1024)
15       nThreads = 1024 ;
16    else
17       nThreads = n ;
18
19    // Transferência de dados host-GPU
20    ...
21
22    // Para cada éxon candidato bk
23    for (int k = 0; k < L; k++)
24       // Invoca kernel para calcular matriz Sk de éxon bk
25       kernelCalcExon<<<nBlocos, nThreads>>>(k, n, S, b->last, base, target);
26
27    // Transferência de dados GPU-host
28    ...
29
30    // Calcula pontuação do alinhamento spliced ótimo
31    ...
32 }
```

Figura 6.1: Função responsável por invocar o *kernel*, para o primeiro acelerador em GPU para o alinhamento *spliced*

Cada invocação do *kernel* cria um *grid* com apenas um bloco, para calcular a matriz S_k correspondente ao éxon b_k, e este bloco tem 1024 *threads* ou menos, dependendo do comprimento da sequência alvo. Isto é determinado nas linhas 12 a 17 da função da Figura 6.1.

Na Seção 5.3 concluiu-se que o tamanho da anti-diagonal da matriz S_k é definido pelo comprimento do éxon b_k. Portanto, na fase do *kernel* em que as células de uma anti-diagonal de S_k são calculadas em paralelo pelas diferentes *threads* do bloco, o número de *threads* necessárias depende do comprimento do éxon b_k. Entretanto, conforme será apresentado a seguir, o *kernel* possui outras duas fases em que o número de *threads* depende do comprimento da sequência alvo, que é maior que o comprimento do éxon. Por isto, esta dimensão maior é utilizada na invocação.

Após a invocação do *kernel* para o cálculo de todas as matrizes S_k, a pontuação do alinhamento *spliced* ótimo entre as sequências base e alvo, considerando o conjunto de

éxons candidatos, é calculada, implementando-se a Equação 3.4, o que é realizado no final da função executada no *host* apresentada na Figura 6.1.

A Figura 6.2 apresenta o *kernel* do primeiro acelerador em GPU que é responsável por calcular a matriz de pontuação S_k correspondente a um éxon b_k. Novamente, visando um melhor entendimento, alguns detalhes são omitidos. A execução do *kernel* pode ser dividida basicamente em três fases.

Na primeira fase, que se inicia na linha 20 do *kernel* da Figura 6.2, obtém-se as pontuações dos éxons b_l anteriores a b_k, necessárias para calcular a matriz S_k, como indicado nas duas primeiras linhas da Equação 3.3. Nesta fase, cada *thread* está associada a um símbolo da sequência alvo t. Há um laço sequencial para percorrer todos os éxons b_l anteriores a b_k (linhas 21 a 26), procurando as maiores pontuações. Em cada iteração do laço, uma matriz S_l (correspondente ao éxon b_l) é acessada a partir da memória global da GPU, e a linha $last(l)$ de S_l (última posição de b_l) é lida. Cada *thread* lê um elemento diferente da linha, de forma que todas as *threads* acessam a linha inteira em paralelo, obtendo um acesso coalescido na memória global. Este acesso é realizado na linha 23 do *kernel*. Cada *thread* compara, na linha 24, a pontuação lida com a pontuação máxima acumulada nas iterações anteriores do laço, de modo que ao final da execução do laço, o valor máximo dentre os valores lidos é obtido. Estas pontuações máximas determinadas pelas *threads* são armazenadas em uma estrutura alocada na memória compartilhada da GPU, na linha 27 do *kernel*.

Calcular as pontuações máximas dos éxons b_l anteriores a b_k e armazená-las na memória compartilhada evita a realização do mesmo cálculo repetidas vezes, pois diferentes células de S_k dependem das mesmas células das matrizes S_l.

A Figura 6.3 ilustra de forma gráfica o paralelismo explorado pelas *threads* na primeira fase do *kernel* do acelerador em GPU.

Na segunda fase do *kernel*, que se inicia na linha 33 do *kernel* da Figura 6.2, a matriz S_k é calculada. Cada *thread* é associada a um símbolo do éxon b_k. Há um laço sequencial (linhas 33 a 48) para percorrer todas as anti-diagonais de S_k. Em cada iteração do laço, uma anti-diagonal de S_k é calculada, com cada *thread* computando uma célula diferente da anti-diagonal (linhas 36 a 41), de tal forma que as *threads* calculam a anti-diagonal completa em paralelo.

Nesta segunda fase, todas as pontuações são acessadas na estrutura alocada na memória compartilhada (nas linhas 37 a 41), tanto para leitura como para escrita, não havendo portanto acessos à estrutura S alocada na memória global, como será descrito na Seção 6.3.

Na realidade, apesar de não estar indicado no *kernel* da Figura 6.2, a técnica de *loop unrolling* é aplicada no laço da segunda fase do *kernel*, de forma que três anti-diagonais são

```
1  __global__ void kernelCalcExon(int k, int n, int *S, int *last, char *base,
2                                 char *target) {
3     // k: índice do éxon bk, n: comprimento da sequência alvo
4     // S: matriz de pontuação Sk do éxon bk na memória global
5     // last: índice da última posição do éxon
6     // base: sequência base, target: sequência alvo
7
8     __shared__ unsigned int indKS; // índice da linha do éxon bk em S
9     __shared__ short maxAnt[sizeMaxAnt]; // vetor com pontuações máximas
10                                         // dos éxons anteriores ao éxon bk
11    __shared__ short SShrd[sizeSShrd]; // matriz de pontuação Sk do
12                                       // éxon bk na memória compartilhada
13    int first; // índice da primeira posição do éxon bk
14    unsigned short kLen; // comprimento do éxon bk
15    int th; // índice da thread
16
17    // Declaração e inicialização de variáveis ...
18
19    // Fase 1: Procura por scores máximos dos éxons anteriores ao éxon bk
20    if (th <= n) {
21       for (j = k - 1; j >= 0; j--)
22          if (last[j] < first) {
23             r1 = S[(j * (n + 1)) + th]; // Lê valor da memória global
24             if (max < r1)
25                max = r1;
26          }
27       maxAnt[th] = max; // Salva valor na memória compartilhada
28    }
29    __syncthreads();
30
31    // Fase 2: Calcula matriz Sk
32    // Para cada anti-diagonal de Sk
33    for (i = 2; i < n + kLen; i++) {
34       if (th < (kLen - 1)) {
35          if ((th < (i - 1)) && ((i - 2) < th + n)) { // Calcula célula
36             if (th == 0)
37                SShrd[0] = maxAnt[i - 1];
38             r1 = SShrd[prev2Line + th] + w(base[first-1+th], target[i-2-th]);
39             r2 = SShrd[prev1Line + th] - 2;
40             r3 = SShrd[prev1Line + th + 1] - 2;
41             SShrd[curLine + th + 1] = MAX(r1,r2,r3);
42          }
43       }
44       __syncthreads();
45       // Salva valor na memória compartilhada
46       if ((th == 0) && ((i+1) > kLen))
47          maxAnt[i - kLen + 1] = Sshared[curLine + kLen - 1];
48    }
49
50    //  Fase 3: Salva valor na memória global
51    if (th < n + 1)
52       S[indKS + th] = maxAnt[th];
53 }
```

Figura 6.2: Função *kernel* do primeiro acelerador em GPU

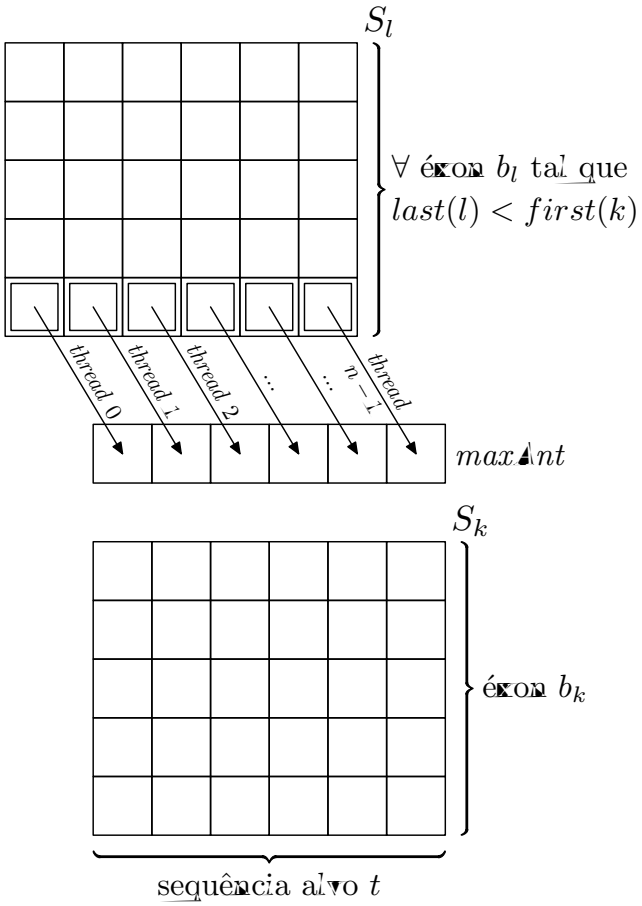

Figura 6.3: Primeira fase do *kernel*: *threads* associadas aos símbolos da sequência alvo

calculadas a cada iteração do laço, através da replicação do corpo do laço três vezes. Esta técnica traz ganhos de desempenho pois, além de reduzir o *overhead* do laço, expõe mais paralelismo para o escalonador de *warps*, reduzindo os conflitos entre as instruções [41].

A Figura 6.4 ilustra de forma gráfica o paralelismo explorado pelas *threads* na segunda fase do *kernel* do acelerador em GPU.

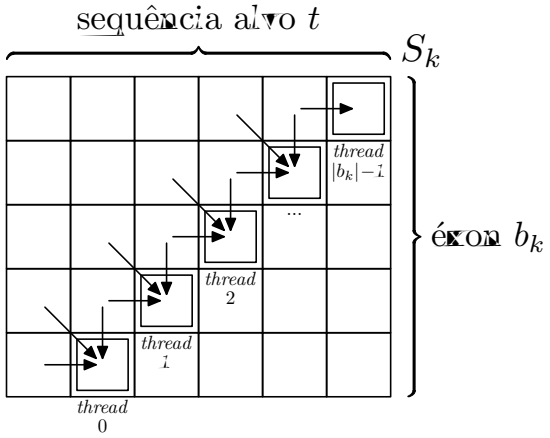

Figura 6.4: Segunda fase do *kernel*: *threads* associadas aos símbolos da sequência alvo

Finalmente, na terceira e última fase da execução do *kernel*, a última linha da matriz

S_k (correspondente ao éxon b_k) que está alocada na memória compartilhada é copiada para a estrutura da memória global (na linha 52). Mais uma vez, cada *thread* é associada a um símbolo da sequência alvo t. Cada *thread* escreve um elemento diferente da linha, de forma que todas as *threads* acessam a linha completa em paralelo, obtendo um acesso coalescido na memória global. A Figura 6.5 ilustra de forma gráfica o paralelismo explorado pelas *threads* na terceira fase do *kernel* do acelerador em GPU.

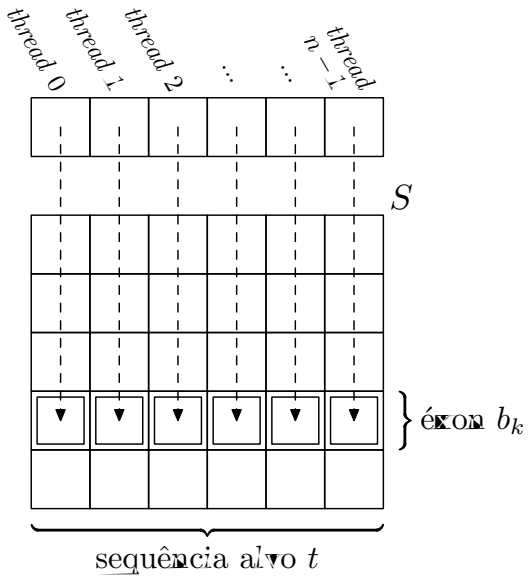

Figura 6.5: Terceira fase do *kernel*: *threads* associadas aos símbolos do éxon b_k

Algumas sincronizações entre as *threads* são necessárias em certos pontos do *kernel*. Após o fim da primeira fase e antes do início da segunda fase, na linha 29, uma sincronização é necessária, a fim de garantir que todas as pontuações máximas necessárias para segunda fase tenham sido encontradas. Dentro da segunda fase, ao fim de cada iteração do laço, na linha 44, há uma sincronização para garantir que uma anti-diagonal tenha sido calculada inteira, antes de iniciar o cálculo da próxima. Esta sincronização garante também que, ao se iniciar a terceira fase, todas as células da matriz S_k tenham sido calculadas. Todas essas sincronizações são realizadas através da função $\_\_syncthreads()$.

Esse primeiro acelerador em GPU ainda é capaz de lidar com dados de entrada com comprimentos de éxons e/ou comprimento da sequência alvo maiores que o número máximo de *threads* da GPU utilizada, cujo valor é 1024. Para que isso seja possível, as três fases do *kernel* foram implementadas usando a técnica de *tiling* [23]. Assim, dependendo do tamanho das entradas de dados, cada *thread* pode ser associada a várias células, ao invés de uma única. No *kernel* da Figura 6.2, a implementação desta técnica não é mostrada afim de deixá-lo mais simples e facilitar sua compreensão.

6.2 Segundo Acelerador: Exploração de Paralelismo Intra- e Inter-éxon

O primeiro acelerador não usa todos os recursos de hardware da GPU. Como ele faz invocações do *kernel* criando *grids* com apenas um bloco, este bloco é escalonado em apenas um SM (*Streaming Multiprocessor*) da GPU. Além disso, o primeiro acelerador não explora todo o paralelismo disponível na aplicação e nos dados de entrada, pois explora apenas o paralelismo intra-éxon. Portanto, um segundo acelerador é desenvolvido para superar estas limitações. O segundo acelerador em GPU explora, além do paralelismo intra-éxon, o paralelismo inter-éxon descrito na Subseção 5.1.2.

As duas formas de paralelismo são exploradas de forma conjunta. Cada matriz S_k tem as células de uma anti-diagonal calculadas em paralelo. Além disso, para um grupo de matrizes que podem ser calculadas em paralelo umas com as outras, porém sequencialmente em relação a outros grupos, as anti-diagonais das diferentes matrizes deste grupo são calculadas em paralelo.

A Figura 6.6 apresenta a função, executada no *host*, responsável pelas invocações do *kernel*, para o segundo acelerador em GPU. Novamente alguns detalhes são omitidos, para um melhor entendimento.

O mesmo *kernel* da Figura 6.2 usado para o primeiro acelerador, para calcular a matriz de pontuação S_k correspondente a um éxon b_k, também é usado para o segundo acelerador. A diferença está na forma da invocação deste *kernel*, isto é, no espaço de execução criado para ele. No primeiro acelerador, cada invocação do *kernel* cria um *grid* com apenas um bloco de *threads* para calcular a matriz S_k correspondente ao éxon b_k. No segundo acelerador, para cada grupo $\mathcal{G}$ de éxons correspondentes a matrizes S_k que podem ser calculadas em paralelo, uma nova invocação do *kernel* é feita. Cada invocação do *kernel* cria um *grid* com $|\mathcal{G}|$ blocos de *threads*, cada bloco associado a um éxon $b_k \in \mathcal{G}$. Uma nova invocação, relativa a um novo grupo, só é realizada após a invocação anterior terminar sua execução.

Nas linhas 14 a 17 da função da Figura 6.6 determina-se o número de *threads* dos blocos do *grid*, que é 1024 ou menos, dependendo do comprimento da sequência alvo, da mesma forma que no primeiro acelerador. O laço que inicia na linha 28 da Figura 6.6 percorre os éxons candidatos da sequência base. Ao analisar o éxon b_k, tenta-se formar um grupo de éxons cujas matrizes possam ser calculadas em paralelo à matriz S_k (laço das linhas 34 a 40). Assim é determinado o número de blocos do *grid* da próxima invocação do *kernel*, que ocorre na linha 44.

Novamente, após o cálculo de todas as matrizes S_k, a pontuação do alinhamento *spliced* ótimo entre as sequências base e alvo, considerando o conjunto de éxons candidatos,

```
1  void aceleradorParalIntraInter(int n, Exons b, int L, S[], base[], target[]) {
2     // n: comprimento da sequência alvo
3     // b: objeto da classe Exons, com informações dos éxons
4     // L: número de éxons
5     // S: estrutura de pontuação
6     // base: sequência base
7     // target: sequência alvo
8
9     // Declaração e inicialização de variáveis
10    ...
11
12    int nBlocos;    // Número de blocos
13    int nThreads;   // Número de threads por bloco
14    if (n > 1024)
15       nThreads = 1024 ;
16    else
17       nThreads = n ;
18
19    // Transferência de dados host-GPU
20    ...
21
22    // count: número de éxons do grupo que será calculado em paralelo
23    // k: primeiro éxon do grupo que será calculado
24    // max: fronteira máxima na procura por éxons de um mesmo grupo
25
26    // Percorre éxons candidatos
27    i = 0;
28    while (i < L) {
29       k = i;
30       count = 1;
31       max = b->last[i];
32       i++;
33       // Encontra éxons do mesmo grupo que bk (p/ calcular matrizes em paralelo)
34       while ((i < L) && (b->first[i] < max)) {
35          count++;
36
37          if (b->last[i] < max)
38             max = b->last[i];
39          i++;
40       }
41
42       nBlocos = cont;
43       // Invoca kernel para calcular matrizes de grupo de éxons
44       kernelCalcExon<<<nBlocos, nThreads>>>(k, n, S, b->last, base, target);
45    }
46
47    // Transferência de dados GPU-host
48    ...
49
50    // Calcula pontuação do alinhamento spliced ótimo
51    ...
52 }
```

Figura 6.6: Função responsável por invocar o *kernel*, para o segundo acelerador em GPU para o alinhamento *spliced*

é calculada, implementando-se a Equação 3.4, o que é realizado no final da função executada no *host* pelo segundo acelerador, apresentada na Figura 6.6.

Outra diferença importante entre o primeiro e o segundo acelerador é que este último explora o paralelismo entre o processador *host* e a GPU. O *host* faz uma invocação não bloqueante do *kernel* para um grupo $\mathcal{G}$ de éxons (na linha 44 da função da Figura 6.6). Enquanto a GPU calcula as matrizes de $\mathcal{G}$, o *host* identifica o próximo grupo $\mathcal{G}'$ (no laço das linhas 34 a 40). Assim, quando a GPU terminar de executar a primeira invocação, o *host* está pronto para realizar a invocação do *kernel* para o grupo $\mathcal{G}'$. Dessa forma, a GPU não fica ociosa entre as invocações de *kernel*.

Da mesma forma que o primeiro acelerador, este segundo também é capaz de lidar com dados de entrada com comprimento de éxons e/ou comprimento da sequência alvo maiores que o número máximo de *threads* da GPU utilizada, dado que o mesmo *kernel* é usado e a técnica de *tiling* é aplicada. Entretanto, como o segundo acelerador faz invocações do *kernel* criando *grids* com vários blocos, estes blocos podem ser escalonados em vários SMs da GPU, aproveitando melhor os recursos de hardware da mesma. Além disso, o segundo acelerador também explora melhor o potencial de paralelismo dos dados de entrada do que o primeiro.

6.3 Organização das Estruturas de Dados

De acordo com as Equações 3.2 e 3.3, é necessário calcular um total de $n \times \sum_{k=1}^{|\beta|} |b_k|$ células, referentes às pontuações de similaridade da estrutura tridimensional S, onde n é o comprimento da sequência alvo e β é o conjunto de éxons candidatos da sequência base. Armazenar esta estrutura completamente na memória global da GPU limitaria o tamanho dos dados de entrada que os aceleradores seriam capazes de tratar. Um caso de teste com uma sequência alvo longa e/ou uma sequência base com muitos éxons e/ou éxons longos poderia produzir uma estrutura S cujo tamanho excedesse a capacidade da memória global da GPU.

Para transpor esta limitação, apenas a última linha (linha $last(l)$) de cada matriz S_l correspondente ao éxon b_l é armazenada na memória global da GPU. Como mostrado na Figura 3.5, esta é a única linha de S_l que pode ser necessária quando for realizado o cálculo da matriz S_k, correspondente a um éxon b_k que sucede b_l.

Como descrito anteriormente, uma estrutura é alocada na memória compartilhada da GPU para armazenar a matriz S_k correspondente ao éxon b_k, enquanto ela é calculada pelo *kernel*. Para cada célula calculada, quatro acessos são realizados a S_k: três células são lidas e a célula calculada é escrita. Além disso, ocorre reutilização de dados, isto é, uma mesma célula é usada no cálculo de outras três células. Tais características de

acesso, juntamente com o fato de que o acesso à memória global é bem mais lento que o acesso à memória compartilhada, fazem com que a alocação da matriz S_k na memória compartilhada seja vantajosa em termos de desempenho.

Entretanto, se matriz S_k for armazenada completamente na memória compartilhada, como mostra a Figura 6.7, isso novamente limitaria o tamanho dos dados de entrada dos aceleradores, visto que a memória compartilhada da GPU é muito menor que a memória global. Um caso de teste com uma sequência alvo longa e/ou uma sequência base com éxons longos poderia produzir uma matriz S_k cujo tamanho excedesse a capacidade da memória compartilhada.

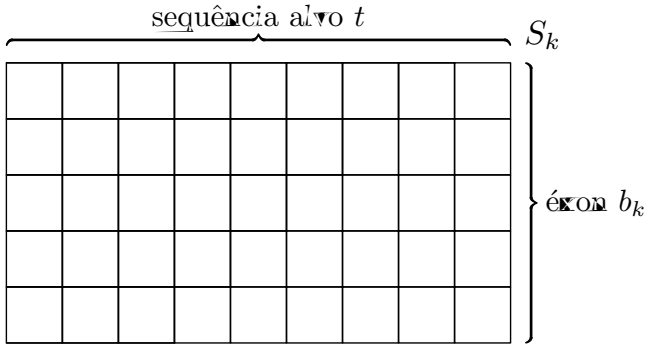

Figura 6.7: Matriz de pontuação S_k para o éxon b_k e sequência alvo t: $|b_k| \times n$ células

Observando a segunda fase do *kernel*, em que as células de uma anti-diagonal são calculadas em paralelo e analisando a Figura 5.1, conclui-se que, para calcular as células da anti-diagonal corrente, apenas células das duas anti-diagonais imediatamente anteriores são utilizadas. Desta forma, a estrutura alocada na memória compartilhada não precisa armazenar a matriz S_k completamente, e sim apenas três anti-diagonais da matriz S_k, aquela que está sendo calculada e as duas anteriores. Esta estrutura é então reutilizada para calcular as anti-diagonais subsequentes da matriz.

A Figura 6.8(a) mostra a matriz S_k correspondente ao éxon b_k organizada na sua forma convencional. As células com a mesma tonalidade de cinza pertencem à mesma anti-diagonal e são calculadas em paralelo por diferentes *threads*. Usar esta organização da matriz S_k, combinada com a alocação de espaço para apenas três anti-diagonais na memória compartilhada, faz com que as *threads* realizem cálculos de índices relativamente complexos para acessar as células de uma anti-diagonal, além de gerar divergências entre as *threads*, o que pode prejudicar o desempenho.

A solução é organizar as células da matriz S_k de uma forma inclinada, como mostra a Figura 6.8(b). Assim, as células de uma mesma anti-diagonal são tratadas como células de uma mesma linha, ocupando posições contíguas na memória. Esta forma de organização é apresentada em [10] para outro problema de Bioinformática, o alinhamento de sequência-*profile* baseado no algoritmo de Viterbi, e é aqui adaptada e aplicada a esta

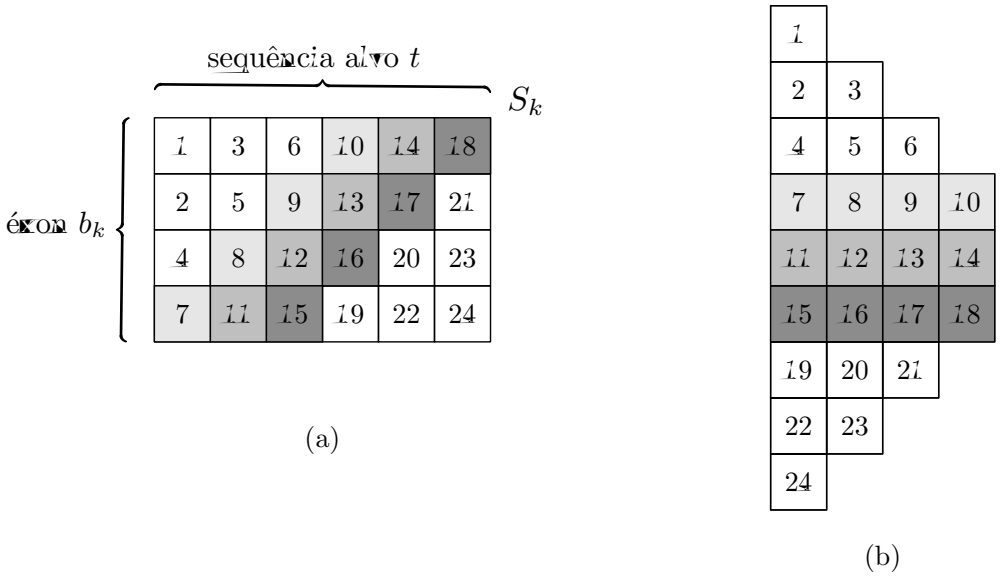

Figura 6.8: Organização da matriz de pontuação S_k do éxon b_k: (a) convencional e (b) inclinada. Células com mesma tonalidade de cinza pertencem à mesma anti-diagonal e são calculadas em paralelo

solução em GPU para o alinhamento *spliced*. Na Figura 6.8, para ambas as organizações de S_k, as células da matriz estão numeradas sequencialmente com base na sua ordem na organização inclinada de S_k.

A Figura 6.9(a) mostra a matriz S_k com sua organização convencional e identifica (com setas) as dependências de dados para o cálculo de algumas células, com base na Equação 3.2. Por exemplo, para calcular a célula 5 as células 1, 2 e 3 são utilizadas. A Figura 6.9(c) destaca como essas dependências de dados aparecem na organização inclinada da matriz S_k, mostrada na Figura 6.9(b). Observa-se que, na organização convencional, as dependências de dados possuem um padrão de acesso uniforme para as células da matriz. Porém, na organização inclinada, estas mesmas dependências não apresentam mais o padrão uniforme de acesso. Assim, quando as *threads* acessam as células da matriz em paralelo, este padrão não uniforme de acesso produz cálculos complexos de índices e divergências entre as *threads*, podendo ter um impacto negativo no desempenho.

Mais uma vez, a solução é reorganizar as células da matriz S_k, desta vez modificando a organização inclinada de S_k, alinhando suas células à direita, como mostrado na Figura 6.10(a). A Figura 6.10(b) destaca como as dependências de dados aparecem na organização inclinada e alinhada à direita de S_k. Assim, um padrão de acesso uniforme para todas as células de S_k é obtido novamente, eliminando os cálculos de índices complexos e as divergências entre as *threads*.

A partir dessa organização inclinada e alinhada à direita da matriz de pontuação S_k, as *threads* calculam em paralelo todas as células em uma linha e cada *thread* é associada a

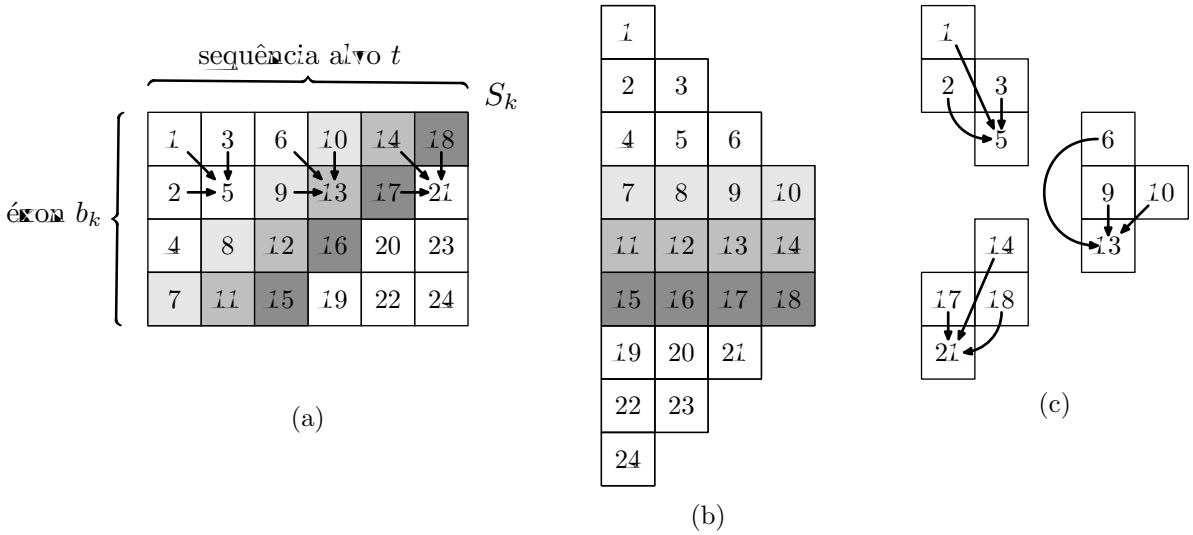

Figura 6.9: (a) Organização convencional de S_k e dependências de dados com padrão de acessos uniforme; (b) Organização inclinada de S_k; (c) Dependências de dados com padrão de acessos não uniforme, da organização inclinada de S_k

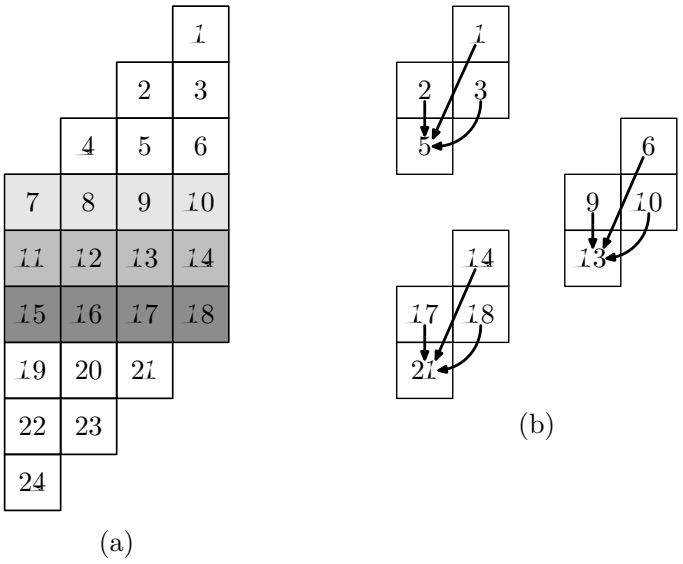

Figura 6.10: (a) Organização inclinada e alinhada à direita de S_k; (b) Dependências de dados com padrão de acessos uniforme, desta organização

uma coluna diferente da matriz, como mostra a Figura 6.11. Desta forma, as células acessadas em paralelo ocupam posições contíguas da memória e as *threads* possuem um padrão de acesso uniforme a elas. Apenas três linhas são alocadas na memória compartilhada da GPU para a matriz S_k e usadas para armazenar a linha sendo calculada correntemente e as duas linhas anteriores. Estas três linhas são então reutilizadas para o cálculo das linhas seguintes.

Para todas as estruturas de dados de entrada alocadas na memória global da GPU, foi utilizado o mecanismo de memória não paginada para otimizar as transferências das estruturas de dados, entre a memória do *host* e a memória global da GPU através do

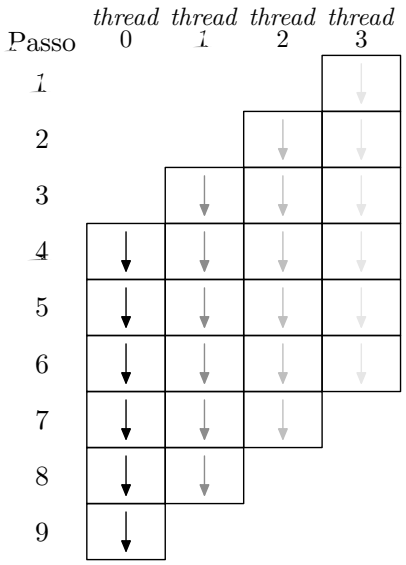

Figura 6.11: Modo de execução: a cada passo, diferentes *threads* calculam em paralelo as células de uma mesma linha (originalmente, uma anti-diagonal). Uma *thread* calcula as células de uma coluna em diferentes passos

barramento PCI-Express, como visto na Subseção 2.2.2.

Capítulo 7

Resultados e Análise

Os aceleradores em GPU desenvolvidos foram avaliados usando o mesmo conjunto de dados biológicos descrito na Seção 5.2 e que foi utilizado para a análise do potencial de paralelismo e estimativa de desempenho. Agora, o objetivo é avaliar o ganho de desempenho real obtido com os aceleradores.

7.1 Plataformas e Ferramentas Utilizadas

A GPU utilizada para executar as soluções paralelas desenvolvidas é uma NVIDIA GeForce GTX 460 [29], com 1 GB de memória RAM e 336 CUDA *cores*. Ela é conectada a um computador *host* com processador Intel Core 2 *Quad Processor* Q9550 [19] de 2.83 GHz e 4 GB de memória RAM, executando o sistema operacional Ubuntu 11.10 [4]. A conexão entre a GPU e o *host* é através da interface padrão PCI-Express [32]. Essa GPU possui capacidade de computação 2.1. A capacidade de computação de uma GPU (fabricada pela NVIDIA) indica a arquitetura utilizada no projeto da GPU e as melhorias realizadas nesta arquitetura [8].

O mesmo computador *host* também é utilizado para execução da implementação sequencial do algoritmo de alinhamento *spliced* de Gelfand, usado para comparação com as soluções em GPU. Nesta implementação sequencial, apenas as células calculadas da estrutura de similaridade S são armazenadas em memória. Para o seu desenvolvimento foi utilizada a linguagem de programação C e o compilador GCC 4.6.1 com opção de otimização -O2.

O CUDA *Toolkit* e SDK versão 5.0 [28] e *driver* versão 304.54 fornecidos pela NVIDIA foram utilizados, como também o compilador GCC 4.6.1 com a opção de otimização -O2, permitindo a execução de programas escritos em CUDA C. A versão utilizada do CUDA foi a 5.0. O *toolkit* também fornece a ferramenta NVIDIA Visual

Profiler, que retorna informações estatísticas sobre o desempenho do programa executado na GPU, permitindo analisar a execução dos *kernels*, a eficiência de operações de acesso à memória global, etc. Os dados fornecidos por esta ferramenta foram aplicados na avaliação experimental de desempenho das soluções desenvolvidas.

7.2 Desempenho dos Aceleradores

A Figura 7.1 compara o tempo de execução real da implementação sequencial do algoritmo de Gelfand para o alinhamento *spliced* (linha tracejada) e do primeiro acelerador em GPU (linha contínua), para todos os 1.010 casos de teste do conjunto de dados biológicos. Os casos de teste estão ordenados pelo tempo de execução da implementação sequencial e os tempos de execução estão mostrados em escala logarítmica. No último caso de teste, por exemplo, o tempo de execução da implementação sequencial é de 14.078,961 ms e do primeiro acelerador é 1.782,647 ms.

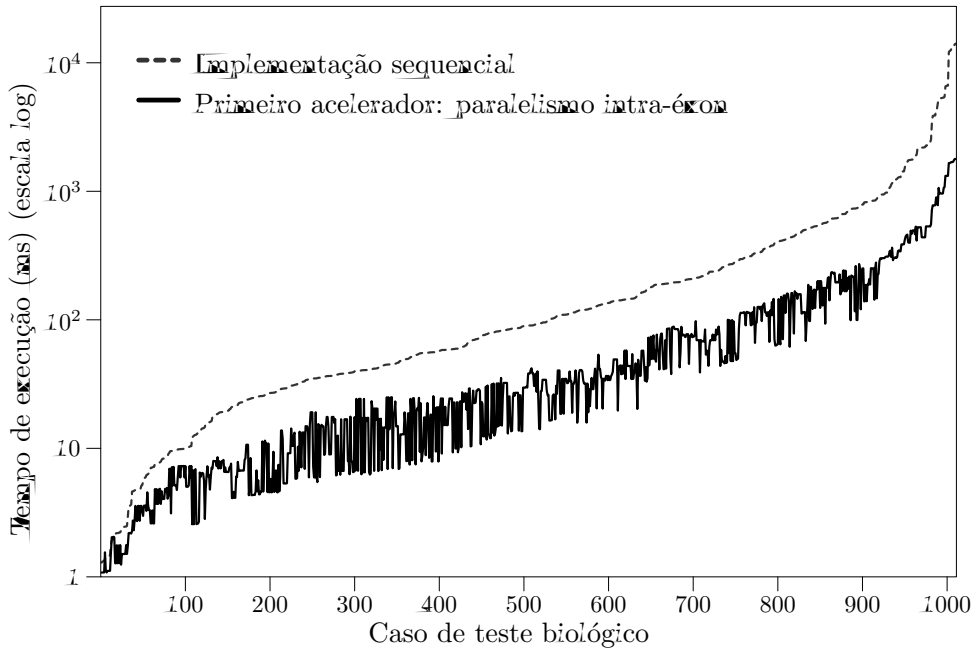

Figura 7.1: Tempo de execução real da implementação sequencial e do primeiro acelerador em GPU

O tempo de execução da implementação sequencial depende basicamente do número total de células calculadas. Como o primeiro acelerador explora o paralelismo intra-éxon, seu tempo de execução depende também do comprimento dos éxons, além do número de células calculadas. Isto explica porque a curva do tempo de execução do primeiro acelerador é irregular, com pequenas subidas e descidas.

Os tempos de execução produzidos pelo primeiro acelerador são quase sempre bem menores que os tempos de execução da implementação sequencial. O primeiro acelerador

produziu um tempo de execução mais longo que a implementação sequencial em apenas 0,4% dos casos de teste. Analisando estes casos de teste, observa-se que eles possuem o comprimento médio de éxon mínimo, dentre todos os casos de teste, e consequentemente, pouco paralelismo intra-éxon disponível para ser explorado.

A Figura 7.2 mostra o *speedup* obtido com o primeiro acelerador em GPU em relação a implementação sequencial do algoritmo de Gelfand para o alinhamento *spliced*. A ordenação dos casos de teste é a mesma da Figura 7.1. O primeiro acelerador produziu um *speedup* máximo de 7,89 e 3,57 na média.

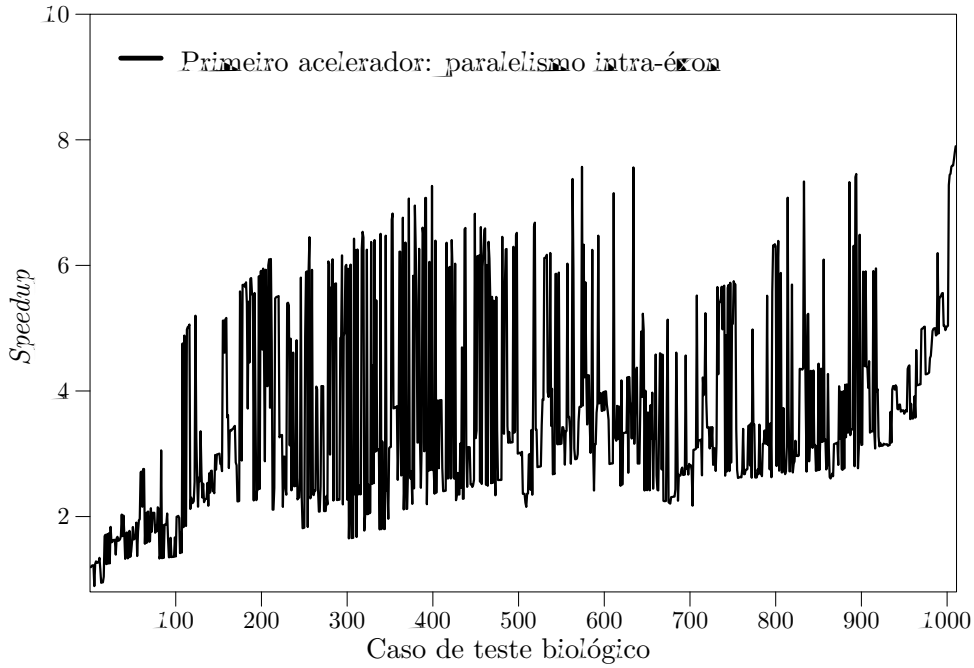

Figura 7.2: *Speedup* do primeiro acelerador em GPU em relação à implementação sequencial

Investigando os casos de testes com *speedups* entre 5 e 8, eles representam as instâncias que possuem um grande número de éxons longos e sequências alvo longas, ou seja, instâncias com bastante paralelismo intra-éxon para ser explorado, pois as matrizes de pontuação S_k terão muitas anti-diagonais longas para serem calculadas. Ao analisar todos os resultados, conclui-se que em mais de 50% dos casos de teste, o primeiro acelerador é pelo menos três vezes mais rápido que a implementação sequencial, e em 89,6% dos casos de teste ele é pelo menos duas vezes mais rápido que a implementação sequencial.

A Figura 7.3 compara o tempo de execução da implementação sequencial (linha tracejada) e dos primeiro e segundo aceleradores em GPU (linhas contínuas preta e cinza, respectivamente), para todos os 1.010 casos de teste. Novamente, os casos de teste estão ordenados pelo tempo de execução da implementação sequencial e os tempos de execução estão mostrados em escala logarítmica. No último caso de teste, por exemplo, o tempo

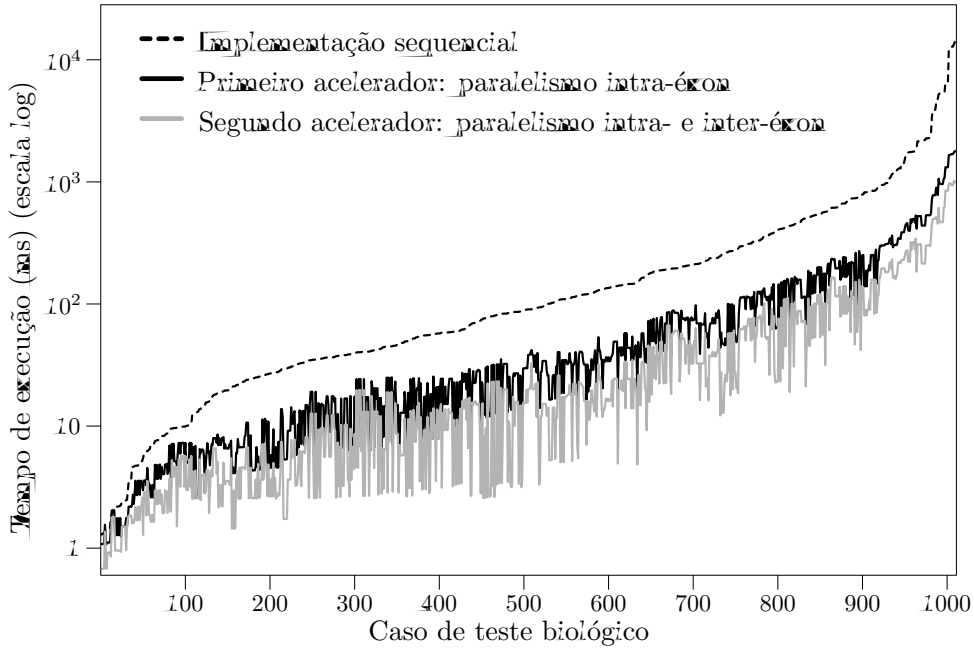

Figura 7.3: Tempo de execução real da implementação sequencial e dos aceleradores em GPU

de execução do segundo acelerador é 992,646 ms.

O segundo acelerador em GPU explora tanto intra- quanto inter-éxon paralelismo. Portanto, seu tempo de execução depende, além do número de células a serem calculadas, do comprimento dos éxons e do número e cardinalidade de grupos de matrizes S_k que podem ser calculadas em paralelo. Isto explica porque a curva do tempo de execução do segundo acelerador é bastante irregular, com subidas e descidas mais acentuadas.

O segundo acelerador foi mais rápido que a implementação sequencial para todos os casos de teste, até mesmo naqueles em que o primeiro acelerador era mais lento que a implementação sequencial, por haver pouco paralelismo intra-éxon disponível para ser explorado. Ele também produziu tempos de execução quase sempre muito menores que o primeiro acelerador. Em apenas 0,4% dos casos de teste o segundo acelerador produziu um tempo de execução mais longo que o primeiro acelerador. Analisando estes casos de testes, observa-se que eles não possuem nenhum paralelismo inter-éxon disponível para ser explorado, isto é, todas as matrizes S_k precisam ser calculadas sequencialmente. Portanto, para estes casos de teste, ambos os aceleradores exploraram apenas o paralelismo intra-éxon e o segundo acelerador possui um pequeno *overhead* por procurar por oportunidades de paralelismo inter-éxon. Mesmo assim, a diferença nos tempos de execução dos aceleradores foi insignificante e em média 0,002 ms.

A Figura 7.4 compara os *speedups* obtidos com o primeiro e o segundo aceleradores em GPU (linhas preta e cinza, respectivamente) em relação à implementação sequencial. A ordenação dos casos de teste é a mesma da Figura 7.3. No último caso de teste, por

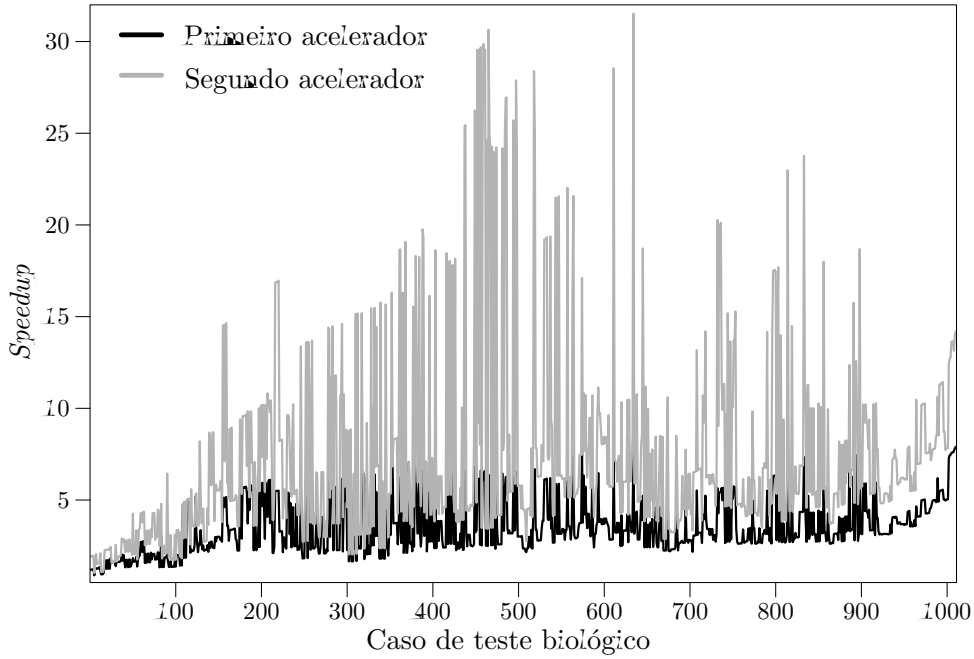

Figura 7.4: *Speedup* dos aceleradores em GPU em relação à implementação sequencial

exemplo, o primeiro acelerador foi 7,90 vezes mais rápido que a implementação sequencial, enquanto que o segundo foi 14,18 vezes mais rápido. O segundo acelerador produziu um *speedup* máximo de 31,51 e 7,28 na média.

Em quase 50% dos casos de teste, o segundo acelerador foi pelo menos seis vezes mais rápido que a implementação sequencial. Em alguns casos de testes o segundo acelerador foi até 30 vezes mais rápido que a implementação sequencial. Analisando estes casos, observa-se que eles possuem éxons longos, sequência alvo longa e um bom número de éxons independentes, consequentemente, bastante paralelismo intra- e inter-éxon para ser explorado.

Também foi avaliada a medida de desempenho de *throughput* CUPS (*Cell Updates per Second*), que indica quantas células da matriz de programação dinâmica (neste caso, a estrutura de similaridade S) são atualizadas em um segundo. A Tabela 7.1 mostra os valores de MCUPS (10^6 CUPS) médio e máximo obtidos pela implementação sequencial, pelo primeiro e segundo aceleradores em GPU, dentre todos os 1010 casos de teste. O caso de teste que atingiu o CUPS máximo quando executado no segundo acelerador é o mesmo que obteve o *speedup* máximo neste mesmo acelerador. Analisando este caso de teste, observa-se que 90% dos éxons têm suas matrizes calculadas com apenas uma invocação do *kernel*, apresentando portanto bastante paralelismo inter-éxon para ser explorado.

O ganho de desempenho produzido pelo primeiro acelerador é limitado pelo comprimento dos éxons, dado que ele explora apenas o paralelismo intra-éxon. O segundo acelerador combina esta forma de paralelismo com o paralelismo inter-éxon, explorando melhor o potencial de paralelismo dos casos de teste. Portanto, o *speedup* do segundo

Tabela 7.1: MCUPS médio e máximo para implementação sequencial e aceleradores em GPU

Solução	MCUPS Médio	MCUPS Máximo
Implementação sequencial	62,01	83,90
Primeiro acelerador	219,83	506,71
Segundo acelerador	452,73	2083,81

acelerador é sempre maior ou igual que o *speedup* do primeiro acelerador, exceto para casos de teste em que não há paralelismo inter-éxon disponível.

7.3 Comparação com Trabalho Relacionado

O segundo acelerador, que explora intra- e inter-éxon paralelismo, teve seu desempenho comparado com outra solução para o alinhamento *spliced* dentre os trabalhos relacionados. A Figura 7.5 mostra o tempo de execução do segundo acelerador em GPU, para todos os 1.010 casos de teste do conjunto de dados biológicos. Entretanto, aqui os resultados estão representados por pontos e os casos de teste estão ordenados pelo produto $m \times n$ do comprimento das sequências base e alvo. Tanto os tempos de execução quanto os produtos $m \times n$ estão mostrados em escala logarítmica. Observa-se que o desempenho do acelerador cresce linearmente com o produto do comprimento das sequências de entrada, como era esperado.

Na Figura 7.6 os resultados do trabalho de Sarje e Aluru [35], descrito na Seção 4.1, são reproduzidos. Estes resultados são obtidos para o alinhamento *spliced* executando em um Cell Broadband Engine com 16 SPEs usando um conjunto de dados sintético. A figura mostra o tempo de execução em relação ao produto $m \times n$ do comprimento das duas sequências de entrada.

Mesmo considerando que a comparação destes resultados com aqueles da Figura 7.5 não é precisa, dadas as diferenças nos experimentos, é possível observar que o segundo acelerador em GPU tem desempenho melhor que os resultados da Figura 7.6. Por exemplo, a maior entrada de dados na Figura 7.6 é 2816×2816, que tem um tempo de execução de aproximadamente 140 ms. Para casos de teste com tamanhos similares, como 6092×1585, o segundo acelerador consome um tempo de execução muito menor, de aproximadamente 10 ms. Além disso, o segundo acelerador em GPU é capaz de tratar entradas de dados de tamanho muito maior.

Por fim, os autores em [35] relatam obterem o *throughput* máximo de 755 MCUPS para o alinhamento *spliced*, enquanto o segundo acelerador em GPU alcançou um máximo de 2083,81 MCUPS, um resultado quase três vezes maior.

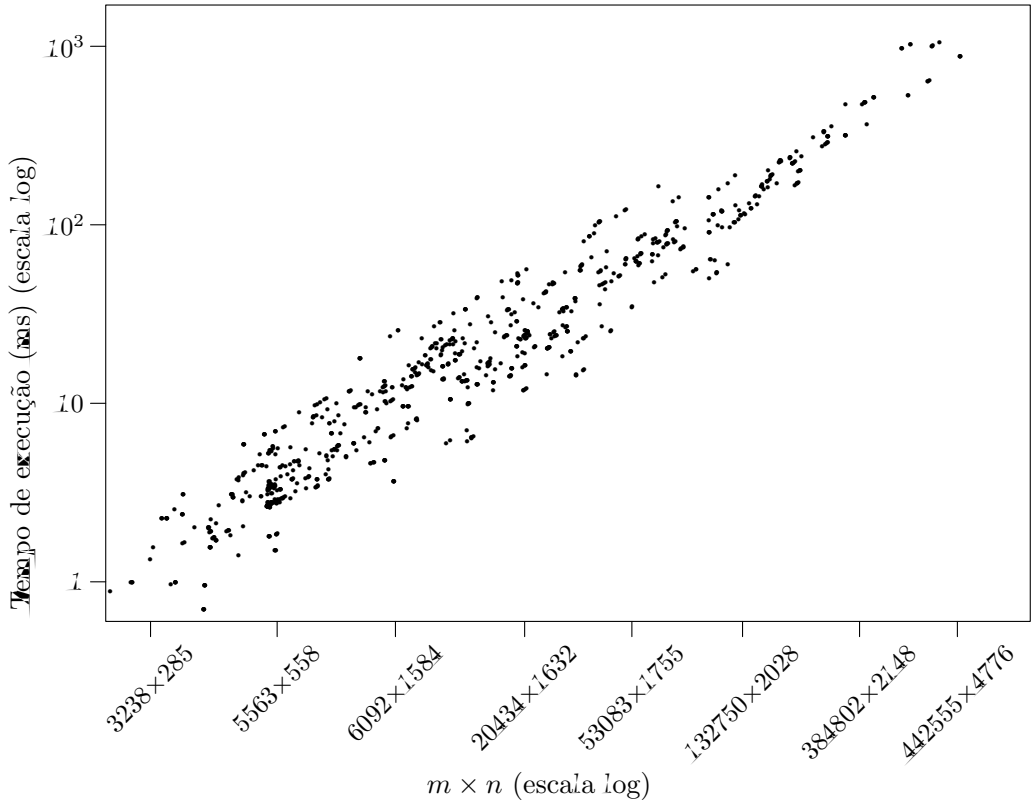

Figura 7.5: Tempo de execução do segundo acelerador em GPU, com casos de teste ordenados por $m \times n$: m e n são o comprimento das sequências base e alvo, respectivamente

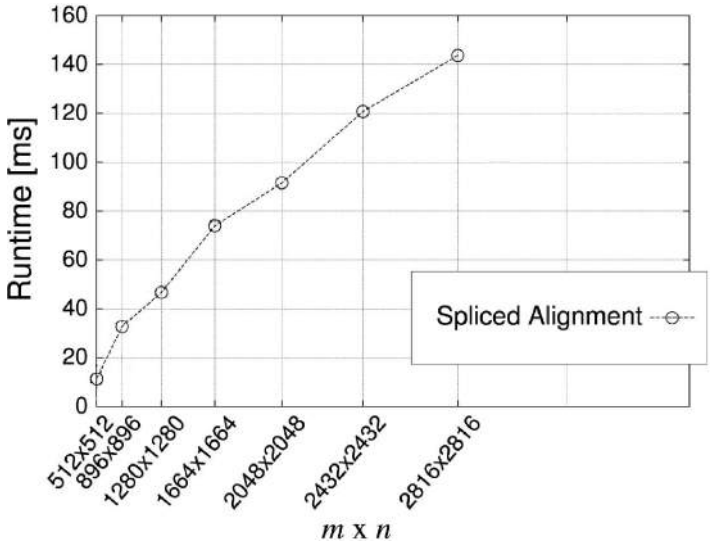

Figura 7.6: Resultados do trabalho de Sarje e Aluru [35]: tempo de execução na plataforma Cell BE com 16 SPEs (m e n são os comprimentos das sequências de entrada)

7.4 Escalabilidade dos Aceleradores

Uma avaliação da escalabilidade dos aceleradores em GPU foi realizada com o objetivo de analisar o desempenho deles para entradas de dados maiores que aquelas dos 1.010 casos de teste do conjunto de dados biológicos utilizado até agora. Um novo caso de teste foi construído, a partir de dados obtidos do NCBI. A sequência base contém o gene IL1RAPL1, juntamente com as 400.000 bases que o antecedem e sucedem, do cromossomo X do organismo *Homo sapiens*. A sequência alvo é uma sequência de cDNA de um gene homólogo ao gene IL1RAPL1, do organismo *Canis lupus familiaris*. As sequências base e alvo possuem comprimento 2.168.787 e 2.355, respectivamente. Usando a ferramenta GENSCAN, um conjunto de 1.878 éxons candidatos foi gerado. Esta entrada de dados constitui um caso de teste muito maior que aqueles do conjunto de dados inicial, com um total de 596.182.380 células a serem calculadas na estrutura S de similaridade.

O novo caso de teste foi submetido à implementação sequencial do algoritmo de alinhamento *spliced* e aos aceleradores em GPU. Os tempos de execução da implementação sequencial, do primeiro e do segundo aceleradores foram 357.506,67 ms, 6.794,17 ms e 3.934,57 ms, respectivamente. O primeiro e segundo aceleradores atingiram *speedups* de 52,62 e 90,86, respectivamente, em relação à implementação sequencial. Ambos os aceleradores foram capazes de tratar esta entrada de dados maior e alcançaram *speedups* bem mais altos do que aqueles obtidos com o conjunto de dados inicial, como era desejado.

Comparando os *speedups* máximos obtidos para o conjunto de dados inicial, que foram 7,89 e 31,51, para o primeiro e segundo aceleradores, respectivamente, com os resultados obtidos para este novo caso de teste, o primeiro acelerador teve o seu *speedup* aumentado em quase sete vezes, enquanto o segundo acelerador em quase três vezes. Analisando o novo caso de teste, verifica-se que, na execução do segundo acelerador, quase 40% dos éxons têm sua matriz de pontuação calculada isoladamente e a cardinalidade de grupos de matrizes que podem ser calculadas em paralelo é em média 1,7. Portanto, o paralelismo inter-éxon disponível para ser explorado no caso de teste era razoavelmente limitado.

Também foi realizada a tentativa de executar este novo caso de teste contendo o gene IL1RAPL1 usando a ferramenta heurística spaln2 [20], executando em um processador *multi-core*, com o objetivo de comparar o desempenho obtido com os aceleradores em GPU que implementam soluções exatas para o alinhamento *spliced*. Entretanto, esta ferramenta não é capaz de tratar sequências base com comprimentos maiores que 100.000.

7.5 Precisão da Estimativa de Desempenho

Na Seção 5.4 um modelo de estimativa de desempenho foi proposto, com o objetivo de estimar os tempos de execução dos aceleradores em GPU e da solução sequencial, antes de implementá-los. Naquele estudo, concluiu-se que era vantajoso implementar as soluções paralelas, pois os resultados indicaram ganhos de desempenho com soluções explorando os paralelismos intra- e inter-éxon. Agora, tendo os resultados reais de desempenho dos aceleradores em GPU, é possível comparar os tempos de execução estimados e reais e avaliar a precisão do modelo de estimativa de desempenho adotado.

A Figura 7.7 mostra na parte superior o tempo de execução estimado para a solução sequencial, tomado a partir da Figura 5.6, para todos os casos de teste do conjunto de dados biológicos. Na parte inferior, a figura mostra o tempo de execução real da implementação sequencial, tomado a partir da Figura 7.3, para todos os casos de teste. Em ambos os gráficos, a ordenação dos casos de teste é a mesma da Figura 5.6. Os tempos de execução em ambos os gráficos estão mostrados em escala logarítmica, porém usam diferentes unidades de medida para o tempo. No modelo de estimativa de desempenho proposto na Seção 5.4 foi utilizado o tempo para computar uma célula da estrutura de similaridade S como unidade de tempo, enquanto o tempo de execução real é dado em milissegundos. Isso explica por que os gráficos têm diferentes intervalos no seu eixo y.

Na Figura 7.8 as duas curvas da Figura 7.7 são sobrepostas, para salientar o quão preciso o modelo de estimativa de desempenho foi. A curva que corresponde ao tempo de execução real da implementação sequencial (curva na cor cinza), mesmo mais irregular, está alinhada à curva que corresponde ao tempo estimado para a solução sequencial (curva na cor preta). Conclui-se portanto que o modelo de estimativa de desempenho previu com precisão o tempo da solução sequencial.

A Figura 7.9 compara o tempo de execução estimado da solução que explora paralelismo intra-éxon e o tempo de execução real do primeiro acelerador em GPU. Por fim, a Figura 7.10 compara o tempo de execução estimado da solução que explora paralelismo intra- e inter-éxon e o tempo de execução real do segundo acelerador em GPU. Nas duas figuras, as curvas correspondentes ao tempo estimado (curva em preto) e ao tempo real (curva em cinza) são sobrepostas.

Nos dois gráficos, a forma irregular da curva do tempo real acompanha a forma irregular da curva do tempo estimado, ponto por ponto, em suas oscilações. Assim, conclui-se que o modelo de estimativa de desempenho também previu com precisão o desempenho dos primeiro e segundo aceleradores em GPU.

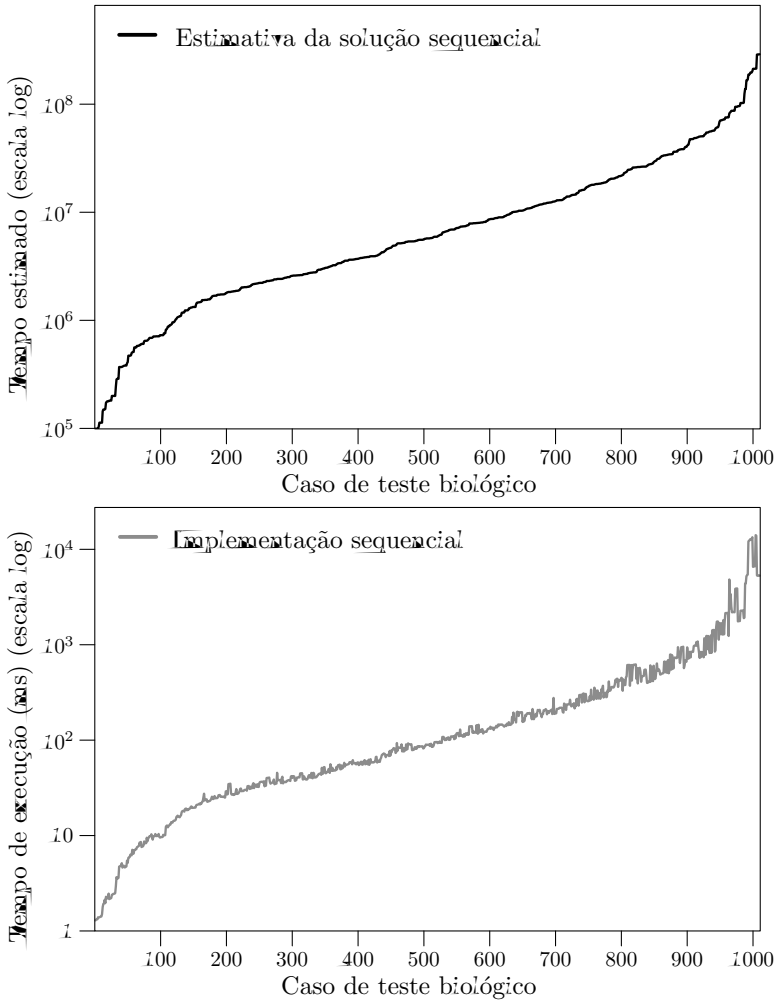

Figura 7.7: Comparação entre tempo de execução estimado e real, para implementação sequencial

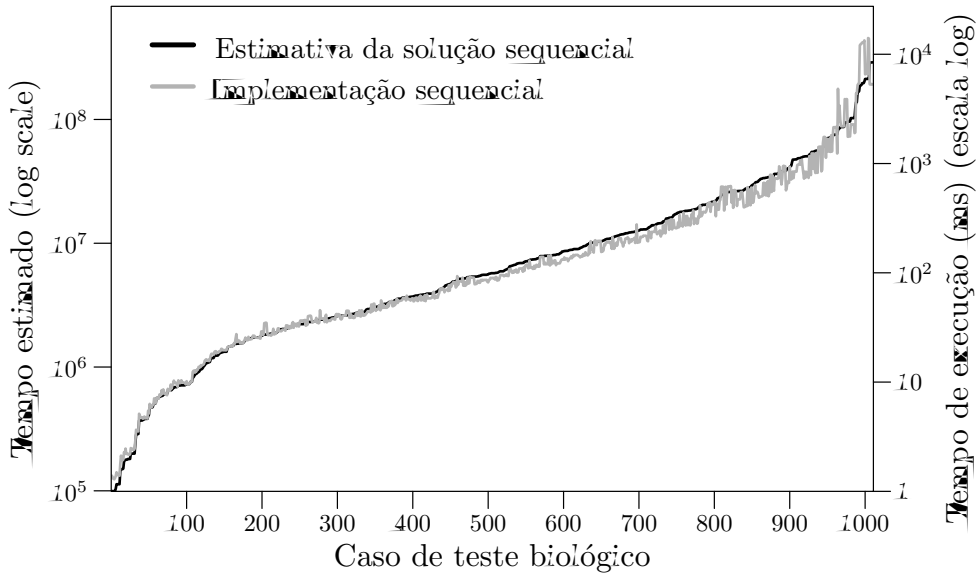

Figura 7.8: Tempos de execução estimado e real sobrepostos, para implementação sequencial

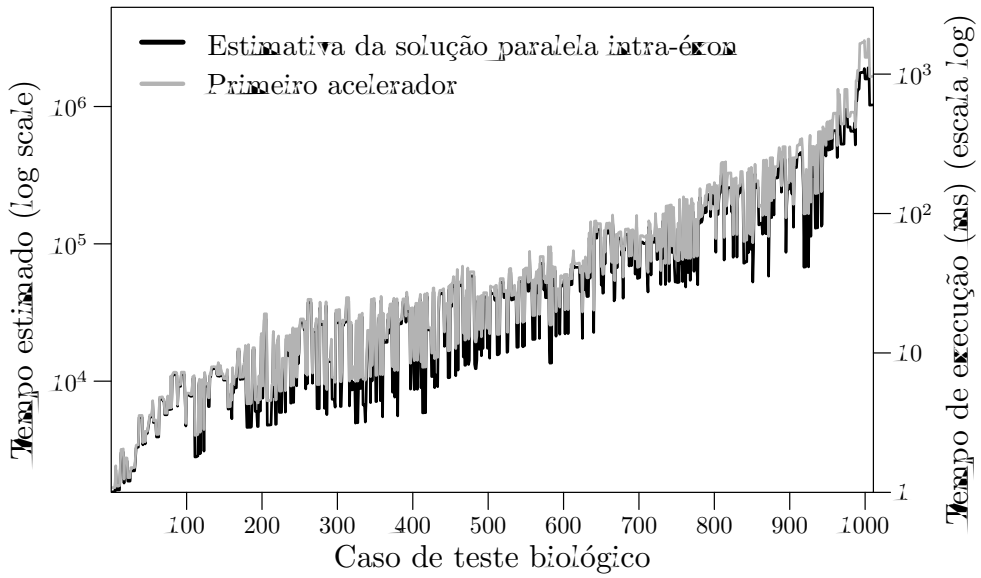

Figura 7.9: Tempos de execução estimado e real sobrepostos, para o primeiro acelerador

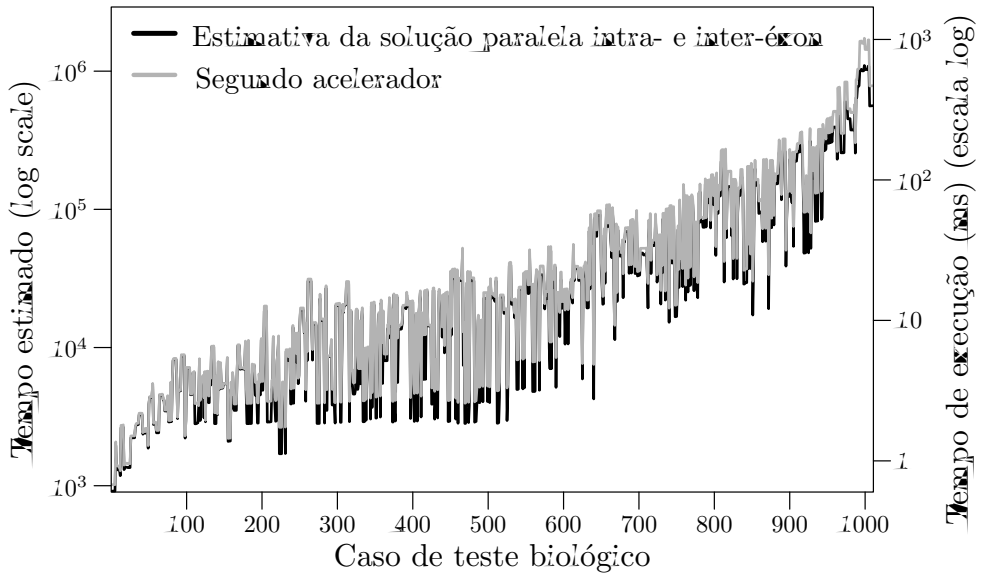

Figura 7.10: Tempos de execução estimado e real sobrepostos, para o segundo acelerador

Capítulo 8

Conclusão

Neste capítulo os resultados alcançados ao longo deste trabalho são descritos e propostas de melhorias e trabalhos futuros são elicitados.

8.1 Resultados

Neste trabalho nós apresentamos dois aceleradores em GPU para o problema do alinhamento *spliced* aplicado à identificação de genes em sequências eucarióticas. Até onde o autor tem conhecimento, estas são as primeiras soluções baseadas em GPU propostas para o problema do alinhamento *spliced*.

Nós identificamos duas formas para explorar paralelismo no algoritmo de alinhamento *spliced*, os paralelismos intra- e inter-éxon. Nosso primeiro acelerador em GPU explora apenas a primeira forma, enquanto ambas as formas são exploradas pelo segundo acelerador. Os resultados mostram que devemos explorar as duas formas de paralelismo a fim de ter ganhos mais significativos de desempenho.

Realizamos uma avaliação extensa de desempenho usando uma GPU modesta e um conjunto de dados biológico amplo, representativo e bem variado. Nossos aceleradores obtiveram excelentes resultados de desempenho. O primeiro acelerador produziu *speedups* médio e máximo de 3,57 e 7,89, respectivamente, enquanto que o segundo acelerador alcançou *speedups* médio e máximo de 7,28 e 31,51, respectivamente, quando comparados com uma implementação sequencial executando em um processador convencional. O primeiro acelerador atingiu um *throughput* médio de 219,83 MCUPS e máximo de 506,71 MCUPS. O *throughput* do segundo acelerador é em média 452,73 MCUPS e máximo de 2083,81 MCUPS.

Os experimentos mostraram que o desempenho dos aceleradores cresceu bem para um caso de teste maior, para o qual o primeiro e segundo aceleradores atingiram *speedups*

de 52,62 e 90,86, respectivamente, em relação à implementação sequencial. Em particular, o desempenho do segundo acelerador cresce linearmente com o produto do comprimento das sequências de entrada, superando os resultados de um trabalho relacionado. Com utilização de uma GPU mais poderosa, nossos aceleradores teriam atingido resultados de desempenho ainda melhores, dado que uma menor aplicação da técnica de *tiling* seria necessária.

Dado que as principais otimizações em GPU envolvem o uso eficiente da sua hierarquia de memórias, propusemos uma organização para as estruturas de dados dos aceleradores, a fim de maximizar a coalescência nos acessos à memória global, maximizar o uso da memória compartilhada, minimizar as divergências entre as *threads* e simplificar os cálculos dos índices de acesso. Além disso, a aplicação da técnica de *tiling* permitiu que nossos aceleradores tratassem entradas de dados com sequências alvo e éxons com comprimentos mais longos que o número máximo de *threads* da GPU utilizada.

Antes de implementar os aceleradores, nós realizamos uma análise do potencial de paralelismo do nosso conjunto dados biológicos, com o objetivo de dimensionar o volume de paralelismo que de fato estaria disponível para ser explorado por uma solução paralela. Também propusemos um modelo de estimativa de desempenho, que nos permitiu estimar o desempenho das soluções paralelas e avaliar se haveria ganhos de desempenho, em comparação com uma solução sequencial. Os resultados dos experimentos mostraram que este modelo previu com bastante precisão o desempenho dos aceleradores.

8.2 Trabalhos Futuros

Apesar dos excelentes resultados obtidos com a exploração dos paralelismos intra- e inter-éxon que encontramos no problema do alinhamento *spliced*, há algumas ideias interessantes para pesquisa em trabalhos futuros. Descrevemos algumas delas a seguir:

- Calcular o erro máximo da precisão das estimativas obtidas através do modelo de estimativa de desempenho proposto para estimar o desempenho das soluções paralelas.
- Otimizar a procura dos valores máximos nas matrizes de pontuação correspondentes aos éxons anteriores, na implementação da resolução da relação de recorrência da Equação 3.3, no *kernel* dos aceleradores em GPU.
- Otimizar o cálculo da pontuação do alinhamento *spliced* ótimo, na implementação da Equação 3.4 nos aceleradores em GPU.
- Experimentar otimizações no uso da memória compartilhada, para obter uma

melhor ocupação dos SMs e/ou aumentar a capacidade dos aceleradores em relação ao comprimento das sequências de entrada.

- Ao longo do desenvolvimento dos aceleradores, diferentes ideias surgiram para a implementação da técnica de *tiling*. Abordagens diferentes das adotadas aqui podem ser experimentadas e comparadas.

- Encontrar o alinhamento ótimo propriamente dito, isto é, o subconjunto dos éxons candidatos de maior similaridade com a sequência alvo. Uma possibilidade é adaptar o método de espaço linear, para não precisar armazenar a estrutura completa.

- PTX (*Parallel Thread eXecution*) é a representação intermediária de código compilado usada pelo compilador das GPUs NVIDIA [41]. Uma abordagem mais baixo nível é analisar o código PTX gerado para garantir que o compilador não está gerando código com alguma ineficiência. Se estiver, entender porque e tentar corrigi-las através de alterações no programa fonte [8].

- O segundo acelerador em GPU desenvolvido não explora totalmente o potencial de paralelismo inter-éxon. Quando o *kernel* é invocado para calcular em paralelo as matrizes correspondentes a um grupo de éxons, as matrizes dos éxons seguintes não podem ser calculadas ainda. Entretanto, à medida que a execução do *kernel* avança, o cálculo de algumas das matrizes do grupo termina e consequentemente, algumas das matrizes dos éxons seguintes poderiam começar a ser calculadas. É possível investigar o desenvolvimento de um novo acelerador em GPU que explore mais paralelismo inter-éxon, através da invocação de um novo *kernel* para calcular as matrizes de um novo grupo de éxons, quando apenas algumas matrizes do grupo anterior tenham terminado. Para isso, seria necessário utilizar GPUs da NVIDIA com capacidade de computação 3.5 ou superior, para aplicar o mecanismo de paralelismo dinâmico [28] que permite que um *kernel* invoque outro *kernel*.

- Investigar outras formas de exploração de paralelismo no algoritmo de alinhamento *spliced*, e como elas podem ser mapeadas em GPU ou em outras plataformas de alto desempenho.

- Estudar outros problemas similares ao alinhamento *spliced* e analisar se as soluções desenvolvidas neste trabalho podem ser estendidas para eles.

Referências Bibliográficas

[1] S. S. Adi. *Identificação de Genes por Comparação de Sequências*. PhD thesis, Instituto de Matemática e Estatística – Universidade de São Paulo, 2005.

[2] A. Baxevanis and B. Ouellette. *Bioinformatics – A Practical Guide to the Analysis of Genes and Proteins*. Wiley-Interscience, 3nd edition, 2004.

[3] C. Burge and S. Karlin. Prediction of complete gene structures in human genomic DNA. *Journal of molecular biology*, 268:78–94.

[4] Canonical Ltd. Ubuntu 11.10. http://releases.ubuntu.com/11.10. Acessado em 19 de maio de 2014.

[5] J. Chang, C. Park, D. S. Jung, M. Kim, J. Kim, S. Yoo, and H. G. Nam. Space-Gene: Microbial Gene Prediction System Based on Linux Clustering. *Genome Informatics Series*, 14:571–572, 2003.

[6] N. Chrysanthou, G. Chrysos, E. Sotiriades, and I. Papaefstathiou. Parallel Accelerators for GlimmerHMM Bioinformatics Algorithm. In *Proceedings of the Design, Automation & Test in Europe Conference & Exhibition*, pages 94–99, 2011.

[7] G. Chrysos, E. Sotiriades, I. Papaefstathiou, and A. Dollas. A FPGA Based Coprocessor for Gene Finding Using Interpolated Markov Model (IMM). In *Proceedings of the International Conference on Field Programmable Logic and Applications*, pages 683–686, 2009.

[8] S. Cook. *CUDA Programming – A Developer's Guide to Parallel Computing with GPUs*. Elsevier Science, 2013.

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

[10] 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 Distributed Processing Symposium*, pages 1–8, 2010.

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

[12] The ENCODE Project Consortium. The ENCODE (ENCyclopedia Of DNA Elements) Project. *Science*, 306(5696):636–640, 2004.

[13] R. Farber. *CUDA Application Design and Develoment*. Elsevier Publishing, 2011.

[14] M. S. Gelfand, A. A. Mironov, and P. A. Pevzner. Gene Recognition via Spliced Sequence Alignment. *Proceedings of the National Academy of Sciences*, 93:9061–9066, 1996.

[15] M. S. Gelfand, A. A. Mironov, and P. A. Pevzner. Spliced Alignment: A New Approach to Gene Recognition. In *Proceedings of the 7th Annual Symposium on Combinatorial Pattern Matching*, pages 141–158, 1996.

[16] D. H. Hirschberg. A Linear Space Algorithm for Computing Maximal Common Subsequences. *Communications of the ACM*, 18(6):341–343, 1975.

[17] X. Huang and K. Chao. A generalized global alignment algorithm. *Bioinformatics*, 19(2):228–233, 2003.

[18] IBM. The Cell Project. http://www.research.ibm.com/cell. Acessado em 19 de maio de 2014.

[19] Intel Corporation. Intel Core 2 Quad Processor Q9550. http://ark.intel.com/pt-BR/products/spec/slb8v. Acessado em 19 de maio de 2014.

[20] H. Iwata and O. Gotoh. Benchmarking spliced alignment programs including Spaln2, an extended version of Spaln that incorporates additional species-specific features. *Nucleic Acids Research*, 40(20):1–9, 2012.

[21] N. C. Jones and P. A. Pevzner. *An Introduction to Bioinformatics Algorithms*. The MIT Press, 2004.

[22] V. V. Kindratenko, J. J. Enos, G. Shi, M. T. Showerman, G. W. Arnold, J. E. Stone, J. C. Phillips, and W. Hwu. GPU Clusters for High-Performance Computing. In *Proceedings of the IEEE International Conference on Cluster Computing and Workshops*, pages 1–8, 2009.

[23] D. B. Kirk and W. W. Hwu. *Programming Massively Parallel Processors – A Hands-on Approach*. Elsevier Science, 2010.

[24] R. M. Kishi. Identificação de Genes e o Problema do Alinhamento Spliced Múltiplo. Master's thesis, Faculdade de Computação da Universidade Federal de Mato Grosso do Sul, 2010.

[25] W. Liu and B. Schmidt. A Generic Parallel Pattern-Based System for Bioinformatics. In *Proceedings of the European Conference on Parallel Processing*, pages 989–996, 2004.

[26] National Center for Biotechnology Information. GenBank. http://www.ncbi.nlm.nih.gov/genbank/. Acessado em 19 de maio de 2014.

[27] NVIDIA Corporation. CUDA Parallel Computing Platform. http://www.nvidia.com.br/object/cuda_home_new_br.html. Acessado em 19 de maio de 2014.

[28] NVIDIA Corporation. CUDA toolkit. http://docs.nvidia.com/cuda/cuda-toolkit-release-notes/index.html#more-information. Acessado em 19 de maio de 2014.

[29] NVIDIA Corporation. GeForce GTX 460 Specifications. http://www.geforce.com/hardware/desktop-gpus/geforce-gtx-460/specifications. Acessado em 19 de maio de 2014.

[30] NVIDIA Corporation. GeForce GTX 690 Specifications. http://www.geforce.com/hardware/desktop-gpus/geforce-gtx-690/specifications. Acessado em 19 de maio de 2014.

[31] NVIDIA Corporation. *CUDA C Programming Guide*, 2012. Version 4.1.

[32] PCI–SIG. Creating a PCI Express interconnect. http://www.pcisig.com/specifications/pciexpress/resources/PCI_Express_White_Paper.pdf. Acessado em 19 de maio de 2014.

[33] J. Puthukattukaran, S. Chalasani, and P. Senapathy. Design and Implementation of Parallel Algorithms for Gene-Finding. In *Proceedings of the 3rd IEEE International Symposium on High Performance Distributed Computing*, pages 186–193, 1994.

[34] A. Sarje and S. Aluru. Parallel Genomic Alignments on the Cell Broadband Engine. In *Proceedings of the IEEE International Symposium on Parallel and Distributed Processing*, pages 1–11, 2008.

[35] A. Sarje and S. Aluru. Parallel Genomic Alignments on the Cell Broadband Engine. *IEEE Transactions on Parallel and Distributed Systems*, 20(11):1600–1610, 2009.

[36] A. Sarje and S. Aluru. *Bioinformatics: High Performance Parallel Computer Architectures*, chapter Parallel Algorithms for Alignments on the Cell BE, pages 59–83. CRC Press, 2010.

[37] E. W. Sayers, T. Barrett, D. A. Benson, E. Bolton, S. H. Bryant, K. Canese, V. Chetvernin, D. M. Church, M. D. Cuccio, S. Federhen, M. Feolo, L. Y. Geer, W. Helmberg, Y. Kapustin, D. Landsman, D. J. Lipman, Z. Lu, T. L. Madden, T. Madej, D. R. Maglott, A. Marchler-Bauer, V. Miller, I. Mizrachi, J. Ostell, A. Panchenko, K. D. Pruitt, G. D. Schuler, E. Sequeira, S. T. Sherry, M. Shumway, K. Sirotkin, D. Slotta, A. Souvorov, G. Starchenko, T. A. Tatusova, L. Wagner, Y. Wang, W. J. Wilbur, E. Yaschenko, and J. Ye. Database resources of the National Center for Biotechnology Information. *Nucleic Acids Research*, 38:D5–D16, 2010.

[38] J. C. Setubal and J. Meidanis. *Introduction to Computational Molecular Biology*. PWS Publishing Company, 1997.

[39] Top500.Org. Top 500 – Supercomputer Sites. http://www.top500.org/. Acessado em 19 de maio de 2014.

[40] L. P. Veronese and R. A. Krohling. Differential Evolution Algorithm on the GPU with C-CUDA. In *Proceedings of the IEEE Congress on Evolutionary Computation*, pages 1–7, 2010.

[41] N. Wilt. *The CUDA Handbook – A Comprehensive Guide to GPU Programming*. Pearson Education, 2013.