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

Mecanismo de parada síncrona para comparação paralela de sequências biológicas longas em ambientes heterogêneos

Eduardo Shindi Nanami
Jadiel Teófilo Amorim de Oliveira

Monografia apresentada como requisito parcial
para conclusão do Bacharelado em Ciência da Computação

Orientadora
Prof.a Dr.a Alba Cristina Magalhães Alves de Melo

Brasília
2018

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

Mecanismo de parada síncrona para comparação paralela de sequências biológicas longas em ambientes heterogêneos

Eduardo Shindi Nanami
Jadiel Teófilo Amorim de Oliveira

Monografia apresentada como requisito parcial
para conclusão do Bacharelado em Ciência da Computação

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

Prof. Dr. George Luiz Medeiros Teodoro
Universidade de Brasília

Prof. Dr. Edison Ishikawa
Universidade de Brasília

Prof. Dr. Edison Ishikawa
Coordenador do Bacharelado em Ciência da Computação

Brasília, 6 de julho de 2018

Dedicatória

Dedicamos esse trabalho de graduação para todos que nos apoiaram durante os anos de graduação.

Agradecimentos

Agradecemos a todos que nos apoiaram nessa jornada e em especial a Professora Alba que nos orientou na realização desse trabalho.

Resumo

Na Bioinformática, uma das operações mais básicas para a análise de sequências biológicas é o alinhamento das sequências. Essa operação, ao ser realizada utilizando algoritmos exatos, leva tempo quadrático, e por isso pode levar muitas horas no alinhamento de sequências muito longas. O MASA é uma das ferramentas que realiza tal operação, e para isso pode utilizar vários nós de processamento, para os quais, atualmente, distribui estaticamente a carga. Fatores como a realização da operação de *Block Pruning* e a utilização concorrente de recursos com outros processos nos nós podem levar a um desbalanceamento de carga ao longo da execução, o que resulta em um congestionamento dos *buffers* entre os nós e consequentemente a um processamento mais lento. Para que seja possível o balanceamento dinâmico de carga, é necessária a parada da operação de alinhamento de todos os nós. No presente trabalho de graduação foi projetado, implementado e avaliado um mecanismo de parada sincronizada dos nós da arquitetura MASA. O mecanismo projetado consiste de um módulo de *checkpoint* que estabelece a conexão inicial entre os nós e em um determinado momento da execução para os nós de forma sincronizada. Nessa operação os nós decidem uma linha em comum da matriz de alinhamento, a partir da qual a operação será retornada na reinicialização. O *overhead* introduzido à arquitetura MASA pela adição do módulo de *checkpoint* foi avaliado em um ambiente controlado composto por três nós com uma GPU cada e se mostrou bastante pequeno em relação ao tempo total de execução da aplicação, com tempos variando de 1 a 2 segundos em execuções de até 1 hora e 9 minutos.

Palavras-chave: Bioinformática, Alinhamento de Sequências, CUDAlign, MASA

Abstract

One of the most basic operations in Bioinformatics for the biological sequence analysis is the alignment of sequences. This operation, if executed using exact algorithms, takes a quadratic execution time, and hence, can take many hours on the alignment of very large sequences. MASA is one of the tools that are able to perform this operation and, in order to achieve that, it may utilize of many processing nodes, statically assigning the load. Factors like the utilization of the *Block Pruning* operation and the concurrent use of resources with other processes on the processing nodes may lead to a unbalanced distribution of loads, resulting on a buffer overload and consequently to a worsened performance. To enable the execution of a load balancing a alignment stop operation in all nodes is needed. In the current undergraduate thesis, a mechanism of synchronous stop for the MASA architecture was designed, implemented and evaluated. The designed mechanism consists of a checkpoint module which lays down a connection between the nodes and, when needed, stops all nodes in a synchronized manner. In this stopping operation, the nodes decide on a unique a row on the alignment matrix from which the operation will resume. The overhead of the implemented module on the MASA architecture was evaluated on a controlled environment formed by three nodes with one GPU each and turned out to be rather small compared to the full execution time of the application.

Keywords: Bioinformatics, Sequence Alignment, CUDAlign, MASA

Sumário

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

A Bioinformática é um campo de estudos interdisciplinar que envolve a Biologia, Ciência da Computação, Matemática e Estatística. Neste campo são desenvolvidos ferramentas computacionais para análise de dados biológicos, como sequências de DNA, RNA e proteínas, com o intuito de auxiliar os biólogos a entenderem as funções e a estrutura dos organismos. Nos últimos anos, a quantidade de sequências armazenadas em bancos de dados biológicos cresceu exponencialmente [4], o que mostra a necessidade da criação de ferramentas capazes de analisar esses dados de forma rápida e eficiente.

Uma das operações mais básicas na análise das sequências biológicas é a realização do alinhamento entre elas, comumente feita através de programas computacionais. Essa tarefa de comparação representa uma importante operação na Bioinformática e permite atribuir um grau de similaridade entre as sequências. Isso auxilia os biólogos na identificação de aspectos evolutivos entre as espécies e na predição da função e estrutura de proteínas, dentre outros [5].

Vários algoritmos exatos projetados para esse fim obtém o *score* e/ou alinhamento ótimo [5], porém esses demandam um alto custo computacional, pois em geral, apresentam complexidade de tempo de execução e memória quadrática. Tal fato invisibiliza a utilização desse tipo de algoritmo para o alinhamento de sequencias muito longas. Como exemplos desses algoritmos podemos mencionar o de Needleman-Wunsch [6] e Smith-Waterman [7].

Para lidar com o alto custo computacional do alinhamento de sequências muito longas dos algoritmos exatos, surgiram os algoritmos heurísticos como o FASTA [8] e BLAST [9]. Esses algoritmos, utilizando de determinadas heurísticas, apresentam uma solução consideravelmente mais rápida para o alinhamento de sequências. O lado negativo deles é, por usarem de heurísticas, a sua execução não resulta, em geral, em um alinhamento ótimo.

Uma das ferramentas para a realização do alinhamento de sequências é o CUDAlign

[10][3] implementado utilizando de programação paralela e da arquitetura CUDA. Essa ferramenta, através de uma combinação de algoritmos como os propostos em [7] e [11], é capaz de obter o alinhamento ótimo entre sequências longas de forma bastante eficiente.

Presente a partir da versão 2.1 do CUDAlign, a otimização *Block Pruning* permite a não realização do cálculo de parte da matriz de alinhamento, resultando em uma melhora significativa de desempenho. Essa otimização porém, não se encontra implementada para múltiplas GPUs até o momento. Para isso será exigido um particionamento dinâmico de cargas entre as GPUs pois com o descarte de parte do que seria calculado, um grande desbalanceamento pode ser gerado.

A partir da versão 3.0 do CUDAlign é possível realizar o alinhamento de sequências com múltiplas GPUs com distribuição estática de carga. Nessa execução multi-GPUs, caso se tenha um ambiente heterogêneo, podem surgir variações no processamento que levam ao desbalanceamento e fazem com que os nodos avancem de maneira mais lenta.

A arquitetura MASA apresentada em [3] facilitou o desenvolvimento de ferramentas de alinhamento de sequências, para diversas plataformas, como por exemplo OpenMP, OmpSs, Intel Phi e também CUDA (MASA-CUDAlign).

O objetivo do presente trabalho de graduação é propor, implementar e avaliar um mecanismo de parada sincronizada dos nós para a arquitetura MASA [3]. O mecanismo proposto consiste de um módulo de *checkpoint* adicionado à arquitetura MASA. Por meio desse módulo e da utilização de *sockets*, os nós realizam a parada sincronizada da operação de alinhamento e decidem uma linha de *checkpoint* em comum. Essas operações são realizadas com o intuito de permitir aos nós redistribuírem a carga de processamento e reinicializarem a execução em seguida.

O presente documento está organizado como se segue. No capítulo 2 descrevemos a operação de alinhamento de sequências, a sua importância e os conceitos fundamentais. Neste mesmo capítulo são apresentados diversos algoritmos existentes na literatura e utilizados neste trabalho. No capítulo 3 são apresentados as interfaces de programação OpenMP e CUDA para programação paralela em plataformas híbridas utilizadas pela arquitetura MASA. No capítulo 4 descrevemos as versões 1.0, 2.0, 2.1, 3.0 e 4.0 do CUDAlign e suas respectivas otimizações. Neste mesmo capítulo apresentamos a arquitetura MASA, à qual neste trabalho adicionamos o módulo de *checkpoint*. No capítulo 5 apresentamos o projeto do módulo de *checkpoint*, sua arquitetura e funcionamento. No capítulo 6 avaliamos o *overhead* causado pela introdução do módulo de *checkpoint* à execução da operação de alinhamento. Por fim, no capítulo 7 apresentamos as conclusões deste trabalho e descrevemos os trabalhos futuros.

Capítulo 2

Comparação de sequências biológicas

Sequências biológicas são formadas por de nucleotídeos ou aminoácidos e constituem importantes estruturas para o funcionamento do organismo dos seres vivos [12]. O seu estudo se faz presente como um importante ramo da biologia e o avanço das técnicas de análise da estrutura e das informações contidas nessas sequências auxiliam pesquisas em diversas áreas. Pode-se citar, por exemplo, a área farmacêutica, onde a comparação das sequências de aminoácidos permite aos pesquisadores preverem a função e estrutura de proteínas, o que os ajuda a projetar novos medicamentos. Outro exemplo é o campo da genética, onde a comparação entre sequências de DNA possibilita identificar aspectos evolutivos das espécies bem como mutações que contribuem para manifestações de doenças [5].

2.1 DNA, RNA e Proteínas

O ácido desoxirribonucleico, ou DNA, é uma macromolécula que está presente nas células de quase todos os seres vivos, estando nela contidas as informações genéticas que definem o desenvolvimento e funcionamento dos organismos. O DNA é composto por duas longas fitas de nucleotídeos estruturadas lado-a-lado em forma de uma dupla hélice. Os nucleotídeos que a formam, por sua vez, são compostos por um ácido fosfórico, uma pentose e uma das quatro bases nitrogenadas, Adenina (A), Guanina (G), Citosina (C) ou Timina (T) [12]. Tais bases são emparelhadas de uma fita à outra, de forma que a Adenina geralmente se une à Timina e a Guanina geralmente se une à Citosina. Em razão disso, ambas cadeias de nucleotídeos guardam informações redundantes, sendo geralmente possível deduzir as bases nitrogenadas que formam uma das cadeias a partir da outra.

O RNA, ou ácido ribonucleico, é uma macromolécula de composição semelhante ao DNA, diferindo, porém, por possuir uma única fita de nucleotídeos, e pela presença da base nitrogenada Uracila (U), ao invés da Adenina [12]. As suas funções, entretanto são bastantes distintas, e são realizadas por diferentes tipos de RNA. O RNA mensageiro, por

exemplo, recolhe informações do DNA para posteriormente serem convertidas em proteínas pela célula. Já o RNA ribossômico consiste do principal constituinte dos ribossomos, parte importante das células relacionados à síntese das proteínas. Por fim pode-se citar o RNA transportador, cuja função é transportar aminácidos para a síntese proteica [12].

As proteínas constituem uma outra importante sequência biológica. São compostas de uma longa cadeia de vinte possíveis tipos naturais de aminoácidos interligados entre si. São eles, Valina(V), Leucina(L), Isoleucina(I), Alanina(A), Arginina(R), Glutamina(Q), Lisina(K), Ácido aspártico(D), Ácido glutâmico(E), Prolina(P), Cisteína(C), Treonina(T), Metionina(M), Histidina(H), Fenilalanina(F), Tirosina(Y), Triptofano(W), Asparagina(N), Glicina(G) e Serina(S). As incumbências das proteínas vão de catalisadora de reações químicas ao exercício de funções motoras no auxílio do movimento de células e tecidos [12].

2.2 Alinhamento das Sequências

Na natureza tem-se a ocorrência de uma grande quantidade de sequências biológicas. Dentre essas sequências é possível notar semelhanças. Como se veio a conhecer na biologia, essas semelhanças entre as sequências está estreitamente relacionada com o processo de evolução dos seres vivos. Por meio desse, o conteúdo genético dos organismos foi passado de progenitores para a prole inúmeras vezes, acrescido de possíveis mutações. Esse contínuo processo fez com que organismos simples, como bactérias, possuíssem genes semelhantes a organismos mais complexos. Em virtude disso é possível estabelecer a hipótese de que as sequências biológicas que compartilham de regiões semelhantes provavelmente exercem funções similares e/ou foram herdadas de ancestrais comuns [5].

A fim de melhor compreender esses relacionamentos entre as sequências, é comumente realizado o alinhamento entre as mesmas. Tal procedimento consiste em comparar duas ou mais sequências com o objetivo de determinar um valor de similaridade entre elas e, na maioria dos casos, também identificar regiões similares. Em geral, as sequências são escritas em linhas, uma abaixo da outra, e entre os seus caracteres são acrescentados quantidades arbitrárias de espaços. Por conseguinte, em quaisquer das colunas podem ser alinhados caracteres idênticos, constituindo um *match*, caracteres diferentes, constituindo um *mismatch*, ou um caractere junto a um espaço, constituindo um *gap* [5]. Objetivando encontrar uma grandeza tangível, é atribuída para cada um destes casos, uma pontuação de similaridade, positiva ou negativa, onde o somatório destas pontuações é chamado de *score* de alinhamento [5].

O alinhamento que resulta no maior *score* dentre todos os alinhamentos possíveis é o alinhamento ótimo. Este alinhamento fornece as informações mais relevantes para o es-

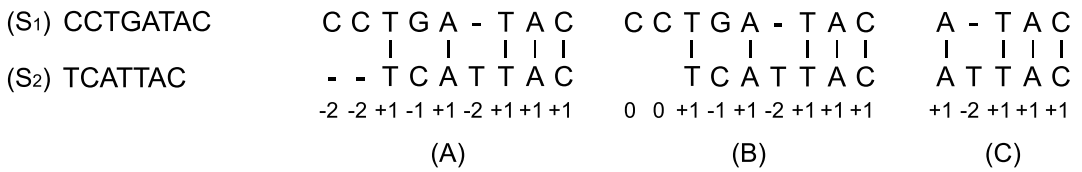

Figura 2.1: Exemplo de Alinhamento global (A), semi-global (B) e local (C).

tudo das sequências, pois caso existam semelhanças entre as duas sequências é muito provável que se encontrem ilustradas no mesmo. Pode-se dividir as técnicas de alinhamento em três tipos principais: alinhamento global, local ou semi-global, cada um adequado para situações distintas, com suas vantagens e desvantagens [5].

O alinhamento global é aquele feito entre todos os caracteres das sequências envolvidas, do começo ao fim. Por isso, neste tipo de alinhamento é esperado que as sequências analisadas sejam semelhantes em conteúdo e tamanho, visto que o alinhamento de cadeias fora deste domínio apresentaria *gaps* muitos longos e *score* muito baixo. O alinhamento semi-global é semelhante ao alinhamento global, porém não penaliza o *score* devido a *gaps* no início e no fim de uma das sequências. O alinhamento local é feito apenas entre partes das sequências. Ante a isso, esse alinhamento se torna mais adequado para casos em que as sequências biológicas não compartilham de semelhanças em sua cadeia como um todo, mas sim em regiões isoladas. A Figura 2.1 apresenta um exemplo de alinhamento global, semi-global e local, com *scores* -2, 2 e 2 respectivamente.

2.3 Algoritmos Exatos para Alinhamento de Sequências

Para a realização do alinhamento entre sequências, as abordagens de programação dinâmica são muito utilizadas. Essa área da computação possui seu alicerce na resolução de subproblemas de forma ótima, evitando retrabalho, de forma a alcançar a melhor solução para o problema em si [13].

A parte às conveniências, o que de fato permite tal método ser aplicado no alinhamento de sequências é a presença de duas características, uma grande quantidade de subproblemas semelhantes e o fato da resolução ótima de subproblemas levar à do problema em si. No ramo, essas são conhecidos como, superposição de subproblemas e subestrutura ótima [13].

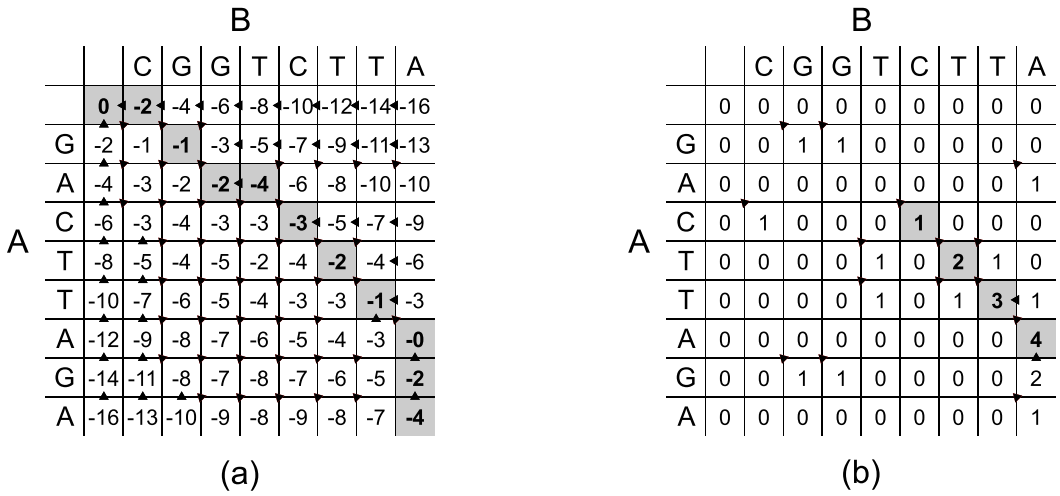

Figura 2.2: Matrizes utilizadas no algoritmo de Needleman-Wunsch (a) e Smith-Waterman (b), os elementos destacados indicam o trajeto percorrido na operação de *traceback*.

2.3.1 Algoritmo de Needleman-Wunsch (NW)

Como uma das primeiras aplicações da programação dinâmica na comparação de sequências biológicas, o algoritmo de Needleman-Wunsch (NW) [6], apresentado em 1970, foi, na época, uma das melhores opções para se obter o alinhamento global ótimo. Consiste em um algoritmo simples que sempre fornece a solução ótima do alinhamento e do *score*, e para isso utiliza de tempo e espaço quadrático.

Nesse algoritmo, o alinhamento de duas sequências A e B de comprimento m e n, respectivamente, é realizado a partir da construção de uma matriz H de dimensões $m+1$ por $n+1$, na qual elementos $H_{i,j}$ armazenam o *score* ótimo do alinhamento entre os prefixos das sequências A e B de comprimento i e j, respectivamente. Essa memorização dos resultados de subproblemas é um dos fatores que caracterizam o algoritmo como uma técnica de programação dinâmica.

Inicialmente, são decididos os *scores* para os casos de *match*, *mismatch* e *gap*, tendo em mente que os valores atribuídos para cada um deles afetam notavelmente o resultado final do alinhamento e do respectivo *score*. No exemplo ilustrado na Figura 2.2 foram utilizados os valores 1, -1 e -2 para o *match*, *mismatch* e *gap*, respectivamente.

O primeiro elemento a ser preenchido na matriz é o $H_{0,0}$, ao qual é atribuído o valor 0. A partir dele a coluna $H_{0,j}$ é preenchida seguindo a fórmula $H_{0,j} = j * gap$ e a linha $H_{i,0}$ seguindo $H_{i,0} = i * gap$, assim como o visto na Figura 2.2(a). Em seguida os demais valores da matriz são calculados utilizando-se das pontuações de cada caso de emparelhamento entre caracteres e se valendo da Fórmula 2.1, onde *p(i, j)* representa a pontuação de

mismatch ou *match*.

$$H_{i,j} = max \begin{cases} H_{i-1,j-1} + p(i,j) \\ H_{i-1,j} + penalidade\ de\ gap \\ H_{i,j-1} + penalidade\ de\ gap \end{cases} \tag{2.1}$$

Ao finalizar o preenchimento da matriz, é encontrado no elemento $H_{m,n}$, o *score* do alinhamento ótimo entre as sequências envolvidas.

Para que seja possível obter o alinhamento ótimo, porém, uma nova etapa deve ser executada, o *traceback*. Essa etapa leva em conta quais posições da matriz foram utilizadas no cálculo do *score* ótimo de cada um dos subproblemas adjacentes para montar a resolução do problema maior, o alinhamento ótimo. Consiste em percorrer a matriz do elemento da extrema direita inferior até o elemento inicial, $H_{0,0}$, seguindo, para cada posição, as origens dos valores que foram utilizados na solução de cada subproblema, representados na Figura 2.2 pelas setas. Valores originados à diagonal representam um alinhamento de A_i e de B_j, valores originados à esquerda, um alinhamento de B_j com um gap em A e valor originados à cima, um alinhamento de A_i com um gap em B.

2.3.2 Algoritmo de Smith-Waterman (SW)

Diferentemente da abordagem de NW (Seção 2.3.1), o Smith-Waterman (SW) [7] se mostra como uma solução para a busca de alinhamentos locais. A primordial diferença entre os dois se encontra nas pontuações guardadas na matriz. No algoritmo de NW, todos os valores são armazenados, independente de serem positivos ou negativos; já no SW apenas valores positivos são aceitos, sendo os demais zerados, incluindo os da primeira linha e coluna. A primeira linha e coluna da matriz são inicializadas com zero, i.e., $H_{0,j} = 0$ onde $(0 \leq j \leq m)$ e $H_{i,0} = 0$ onde $(0 \leq i \leq n)$. O resto da matriz é preenchida seguindo a fórmula 2.2. Esta característica de restrição aos valores negativos é o que possibilita o algoritmo SW fornecer alinhamentos locais e com essa abordagem, uma melhor observação de padrões em áreas isoladas é possível.

$$H_{i,j} = max \begin{cases} H_{i-1,j-1} + p(i,j) \\ H_{i-1,j} + penalidade\ de\ gap \\ H_{i,j-1} + penalidade\ de\ gap \\ 0\ se\ os\ três\ casos\ acima\ são\ negativos \end{cases} \tag{2.2}$$

No algoritmo SW, o *score* do alinhamento ótimo é o maior *score* encontrado na matriz e o alinhamento correspondente é obtido realizando a operação de *traceback*, partindo do elemento de maior valor, e percorrendo a matriz, da mesma forma feita no algoritmo NW, porém, com finalização no primeiro elemento de valor 0 alcançado, Figura 2.2(b).

2.3.3 Algoritmo de Gotoh

Sabe-se que na natureza os eventos de mutações das sequências biológicas tendem a alterar conjuntos de elementos consecutivos e não elementos isolados. Por este motivo, os métodos de penalização linear de *gaps* utilizados até então, com pontuações iguais para todos os *gaps*, não são o melhor modelo. O algoritmo apresentado por Gotoh implementa o modelo *affine gap*, que representa melhor os eventos biológicos [11]. Este algoritmo, de tempo e espaço quadrático, fornece como resultado o alinhamento ótimo global e o score ótimo entre duas sequências, diferenciando a penalização de extensão de *gap*, G_{ext}, da penalização de abertura de *gap*, G_{open}. No *affine gap*, toda sequência de *gaps* consecutivos resulta em uma penalidade de pontuação total G_{open} + G_{ext} * k onde k é o comprimento da sequência de *gaps*.

No algoritmo de Gotoh são utilizadas três matrizes, H, E e F, para realizar o alinhamento entre duas sequências, A e B. Os elementos destas matrizes, assim como no algoritmo de NW, armazenam os *scores* dos alinhamentos entre os prefixos das sequências em questão. O elemento $H_{i,j}$ da matriz H guarda o *score* do alinhamento ótimo que finaliza em um *match* ou *mismatch* nos elementos A_i e B_j. Por sua vez, os elementos $E_{i,j}$ e $F_{i,j}$ armazenam o *score* do alinhamento ótimo que finaliza com o emparelhamento do elemento B_j com um *gap* ou A_i com um *gap*, respectivamente. Estas matrizes são preenchidas conforme as fórmulas 2.3, 2.4 e 2.5, onde G_{first} é a soma de G_{open} e G_{ext}.

$$H_{i,j} = p(i,j) + max \begin{cases} H_{i-1,j-1} \\ E_{i-1,j-1} \\ F_{i-1,j-1} \end{cases} \tag{2.3}$$

$$E_{i,j} = max \begin{cases} H_{i,j-1} + G_{\text{first}} \\ E_{i,j-1} + G_{\text{ext}} \\ F_{i,j-1} + G_{\text{first}} \end{cases} \tag{2.4}$$

$$F_{i,j} = max \begin{cases} H_{i-1,j} + G_{\text{first}} \\ E_{i-1,j} + G_{\text{first}} \\ F_{i-1,j} + G_{\text{ext}} \end{cases} \tag{2.5}$$

2.3.4 Algoritmo de Hirschberg

Um problema apresentado pelos algoritmos discutidos nas seções 2.3.1 a 2.3.3 é a quantidade de memória utilizada. Por se tratarem de soluções que utilizam espaço quadrático, as suas aplicações no alinhamento de sequências muito grandes se tornam inviáveis. Como exemplo, para alinhar as sequências do cromossomo 22 do homem (*Homo sapiens*) e do

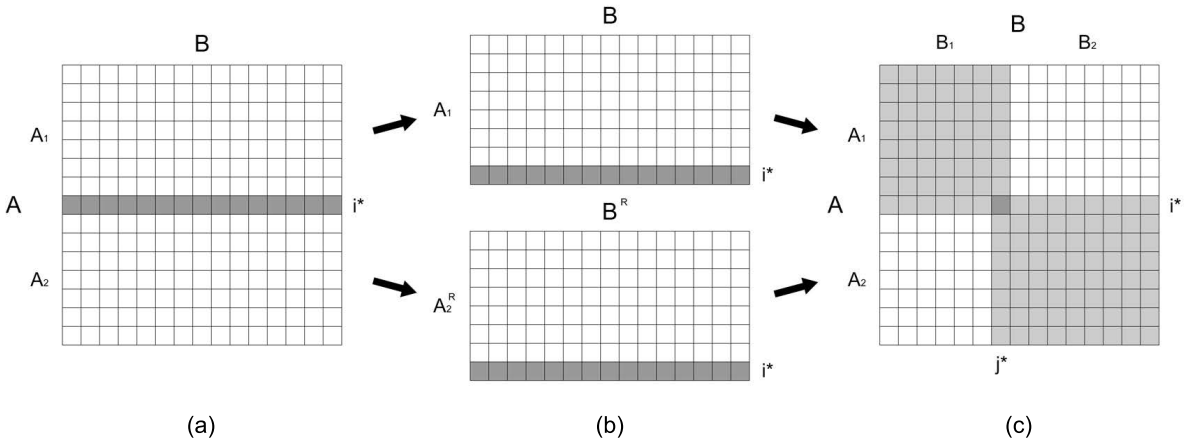

Figura 2.3: Uma iteração do algoritmo de Hirschberg no alinhamento de duas sequências A e B.

chimpanzé (*Pan troglodites*), que tem 33 MBP (Milhões de pares de bases) cada uma, seria necessário pelo menos 4.3 Petabytes de memória.

O algoritmo de Hirschberg [14] é uma versão alterada do algoritmo de NW (Seção 2.3.1) que permite uma utilização linear da memória. Com ele, o espaço ocupado se torna linearmente proporcional à menor das duas sequências, possibilitando o alinhamento de cadeias substancialmente maiores, antes inviável.

Considerando duas sequências, A e B, de comprimento m e n respectivamente, tal algoritmo segue o ilustrado na Figura 2.3. O primeiro passo a ser feito é o processamento da matriz H_1, que alinha as sequências B e A_1, onde A_1 é prefixo de A composto por $i^* = m/2$ elementos. Posteriormente, tem-se o cálculo da matriz H_2, que alinha as sequências B^R e $A_2{}^R$, onde B^R representa o reverso da sequência B, e $A_2{}^R$, o reverso do sufixo de A com comprimento $m/2 + 1$. Nesses processamentos, diferentemente dos algoritmos vistos anteriormente, apenas os valores de duas linhas da matriz são armazenados, a linha que contém os valores que estão sendo calculados e a linha imediatamente anterior. Daí vem a possibilidade de um espaço linear.

Consequente ao cálculo das linhas finais L_1 e L_2, das matrizes H_1 e H_2, respectivamente, é realizada a soma dos elementos de L_1 com os elementos em posições equivalentes no reverso de L_2. Por fim, através da posição do maior valor da soma dos elementos, encontra-se um índice j^*, a coluna do ponto intermediário, posição de um dos elementos do alinhamento ótimo.

Em posse do ponto intermediário (i^*, j^*), a matriz é dividida em duas submatrizes (Figura 2.3(c)), onde, para cada uma delas, o algoritmo se repete recursivamente até que a solução se torne trivial e se tenha todos os pontos do alinhamento global. Vale mencionar

que devido ao constante descarte dos valores calculados pelo algoritmo, muitos desses devem ser recalculados na operação de recursão.

2.3.5 Algoritmo de Myers and Miller

O algoritmo de Myers and Miller [15] se baseia no algoritmo de Hirschberg (Seção 2.3.4) para apresentar uma nova versão do algoritmo de Gotoh (Seção 2.3.3). Com ele, pode-se obter o alinhamento global entre duas sequências com os benefícios da pontuação *affine gap* e com a conveniência da utilização de espaço linear.

Assim como em Hirschberg (Seção 2.3.4), para obter o alinhamento entre duas sequências A e B, de tamanho m e n, respectivamente, deve-se primeiramente obter um ponto intermediário na linha $i^* = m/2$. Como visto anteriormente, este cálculo é feito armazenando apenas a linha da matriz que é necessária para o cálculo da linha seguinte. Esta técnica possibilita dividir o problema de alinhamento em dois subproblemas menores, que podem ser divididos sucessivamente de forma semelhante, até obter-se um problema com solução trivial.

A diferença desse algoritmo para o algoritmo de Hirschberg (Seção 2.3.4) está na maneira de como é escolhido o ponto intermediário. Nesse, é requerido que os valores das soluções dos subproblemas sejam calculados da mesma maneira feita no algoritmo de Gotoh (Seção 2.3.3). Ou seja, são necessárias três matrizes, H_1, E_1 e F_1, para o alinhamento de B e A_1, e três matrizes H_2, E_2 e F_2 para o alinhamento no sentido reverso das sequências B e A_2.

Tendo calculados os valores das matrizes H_1, E_1, F_1, H_2, E_2 e F_2 , consideram-se quatro vetores para a escolha do ponto intermediário: o vetor $CC(j)$ que contém o score ótimo do alinhamento que finaliza sem um *gap* das sequências A_1 e o prefixo de B de comprimento j, denotado por $B[1..j]$, o vetor $DD(j)$ que contém o *score* ótimo do alinhamento que finaliza com um *gap* das sequências A_1 e $B[1..j]$, o vetor $RR(n - j)$ que contém o score ótimo do alinhamento que começa sem um *gap* das sequências A_2 e $B[j..n]$ e o vetor $DD(j)$ que contém o score ótimo do alinhamento que começa com um *gap* das sequências A_2 e $B[j..n]$.

A coordenada j^* do ponto intermediário é equivalente ao valor j que maximiza o valor da fórmula 2.6.

$$max_{j\in[0..n]} \begin{cases} max \begin{cases} CC(j) + RR(n-j) \\ DD(j) + SS(n-j) - G_{\text{open}} \end{cases} \end{cases} \tag{2.6}$$

2.4 Algoritmos Heurísticos para Alinhamento de Sequências

Apesar dos diversos avanços alcançados pelos algoritmos citados na Seção 2.3, todos esses gastam no mínimo tempo quadrático para obterem o alinhamento ótimo. Tal fato faz com que seja geralmente inviável a utilização desses em aplicações que precisam realizar comparações entre milhões de sequências diariamente, como por exemplo, no banco de sequências biológicas *GenBank*, que armazena mais de 200 milhões de sequências, que somam em um total de mais de 230 bilhões de bases [16]. Direcionados a esse problema surgiram diversos algoritmos que utilizam heurísticas para encontrar bons alinhamentos entre grandes quantidades de sequências [17].

Esses algoritmos executam muito mais rapidamente se comparados com os algoritmos exatos. Porém, o uso de métodos heurísticos leva a uma potencial perda da qualidade do resultado final obtido pelo algoritmo. Não se pode afirmar que o alinhamento obtido por tais algoritmos é garantidamente o alinhamento ótimo. Por isso, os algoritmos heurísticos são utilizados apenas nos casos em que tais resultados aproximados são aceitáveis.

2.4.1 FASTA

O algoritmo FASTA [8] foi apresentado por David J. Lipman e William R. Pearson em 1988 como uma melhoria do algoritmo FASTP [18]. Surgiu como uma das primeiras alternativas aos algoritmos exatos para comparações em bases de dados de sequências biológicas. Trata-se de um algoritmo que parte da heurística de que bons alinhamentos devem conter várias subsequências curtas e idênticas. Essas subsequências de tamanho k são localizadas nas duas sequências sendo alinhadas e posteriormente selecionadas e unidas para formar o alinhamento [5].

Esse algoritmo se constrói em quatro passos, sendo o primeiro passo aquele que viabiliza o seu alto desempenho. Nele é feita uma busca de todas as palavras idênticas de tamanho maior ou igual a *ktup* em uma sequência *query* e em uma sequência do banco. Todos esses *matchs* de palavras são marcados em uma matriz. Em geral *ktup* tem valor 2 no alinhamento de aminoácidos e 4 ou 6 no alinhamento de nucleotídeos [5]. A partir dessa matriz e de uma fórmula que leva em consideração a quantidade de pareamentos de palavras e não pareamentos, são identificadas as 10 melhores regiões locais, que são representadas por longas diagonais na matriz.

Na segunda etapa, as regiões encontradas na primeira etapa são reavaliadas utilizando um sistema de pontuação que fornece uma métrica melhor de similaridade. Nessa reavaliação são considerados palavras menores que *ktup* e substituições conservativas de

aminoácidos. Feito isso, para cada uma das regiões é identificada uma sub-região de maior *score*, aqui chamada de região inicial (*init1*).

Na terceira etapa, tenta-se juntar as regiões iniciais com *gaps* para formar regiões com *score* maiores, denominados *score* inicial de similaridade (*initn*). Regiões iniciais com *score* que não superam um determinado valor de corte são desconsideradas nessa etapa. Essas três primeiras etapas são realizada para cada sequência do banco.

Por fim, na quarto etapa, é realizado o alinhamento apenas das sequências com os maiores valores *initn*. Esse alinhamento consiste em uma modificação dos já citados algoritmos da programação dinâmica, o NW (seção 2.3.1) e o SW (seção 2.3.2). Entretanto, no FASTA o alinhamento é restringido a uma banda diagonal centrada na região *init1* obtida na etapa 2.

2.4.2 BLAST

O BLAST (*Basic Local Alignment Search Tool*) [9] foi desenvolvido em 1990 por Altschul e passou a ser uma alternativa ao FASTA (Seção 2.4.1), com maior performance e sensibilidade na detecção de similaridades entre sequências. O algoritmo parte de uma heurística semelhante, que considera que alinhamentos estatisticamente significativos tendem a conter subsequências idênticas alinhadas com *scores* altos [19]. Para computar a similaridade entre uma sequência *query* e um conjunto de sequências em um banco, gera-se inicialmente todas as possíveis subsequências de comprimento k da sequência *query*. Geralmente tem se $k = 3$ para aminoácidos e $k = 11$ para nucleotídeos [19].

Em seguida, cada subsequência gerada na etapa anterior é comparada com todas as possíveis palavras de comprimento k (20^3 possíveis palavras de aminoácidos e 4^{11} de nucleotídeos), utilizando um determinado sistema de pontuação. As palavras que geram um score maior que um valor de *threshold* T são armazenadas para serem utilizadas na etapa seguinte.

Na etapa de escaneamento, é feito uma procura de pareamentos exatos entre as palavras armazenadas e subsequências das sequências do banco. Quando um pareamento exato é encontrado, formando um alinhamento de comprimento k, é realizado a extensão de um elemento de cada vez em ambas as extremidades do alinhamento até o *score* acumulado fique menor que um valor X, em comparação ao maior valor alcançado. A etapa de extensão do alinhamento chega a consumir mais de 90% do tempo de execução total do algoritmo [20]. Por esse motivo, a performance e a precisão do algoritmo é extremamente sensível às variáveis T e X.

Ao fim, os alinhamentos estendidos, denominados *HSP* (*High Scoring Segment Pairs*), são filtrados por um valor de corte S. Os alinhamentos com score maior que S são selecionados para compor o relatório final gerado pelo algoritmo. O valor de S é determinado

empiricamente, analisando-se o *score* de alinhamentos entre sequências aleatórias e escolhendo um valor significantemente maior que o *p value*.

Capítulo 3

Programação Paralela em Plataformas Heterogêneas

É muito comum se deparar com problemas computacionais onde uma decomposição em subproblemas é possível. Em problemas como esses, pode-se notar que as abordagens sequenciais, onde o fluxo de execução percorre um caminho linear, não representa a forma mais eficiente de resolvê-los, o ideal aqui seria um método que permitisse várias instruções rodarem concomitantemente [21].

Esse método é o utilizado pela programação paralela, um ramo da computação que se vale de execuções simultâneas de instruções para alcançar melhores performances [21].

A utilização de programação paralela é muito comum na operação de alinhamento de sequências. Uma das plataformas que permitem tal forma de programação é o CUDA, que possibilita a utilização de GPUs para propósito geral.

3.1 CUDA

As unidades gráficas de processamento (GPU) surgiram como um importante dispositivo para a computação, passando a representar uma solução bastante eficiente para muitos dos problemas da mesma. O seu diferencial está no enfoque da arquitetura, pois as comumente utilizadas unidades centrais de processamento (CPU) tem enfoque na minimização da latência. Já a GPU aposta em uma quantidade maior de núcleos, na casa dos milhares, trabalhando de forma paralela no enfoque de maximizar a vazão, o *throughput* (Figura 3.1) [22]. Aplicações que exigem uma grande quantidade de cálculos simultâneos são os grandes beneficiados por essa abordagem, especialmente aplicações gráficas onde a utilização dos diversos processadores é facilitada pelo paralelismo intrínseco no tratamento da enorme quantidade de *pixels*.

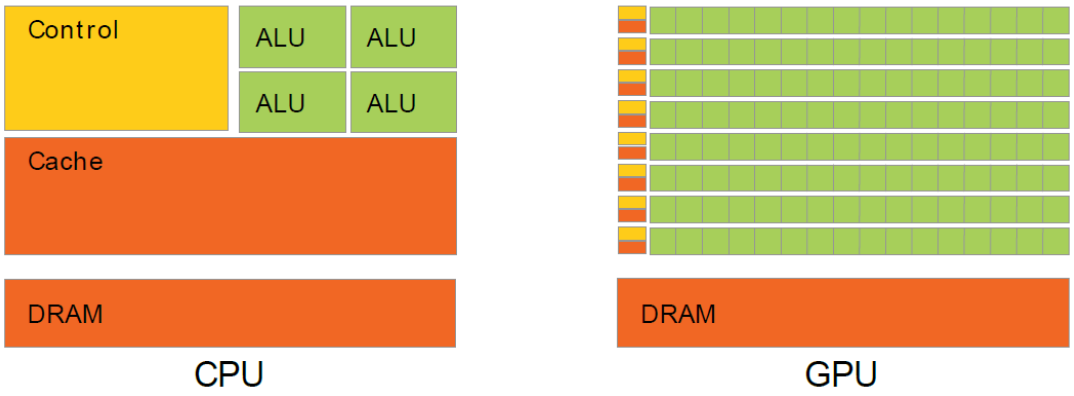

Figura 3.1: Comparação entre a CPU e GPU [1]

Diferentemente do uso para processamento gráfico, a utilização de GPUs para problemas fora deste domínio, se apresentou como um desafio na década de 1990. Até ocorreram tentativas de empregar ferramentas gráficas como o Direct3D e OpenGL para tal, porém no fim, a busca de um mecanismo para simplificação de tal utilização se mostrou necessária. [22]

Um dos mais notáveis mecanismos para programação de propósito geral em GPUs surgiu na NVIDIA em um projeto liderado por Ian Bucks, e teve o nome de CUDA (Compute Unified Device Architecture). CUDA é uma arquitetura de *hardware* e *software* que permite programadores C, C++ e Fortran [1] utilizarem de um pequeno conjunto de extensões de fácil usabilidade para escreverem pedaços de código para GPU de alto desempenho, ampla portabilidade e grande potencial de escalabilidade [22].

CUDA se apresenta como uma API com um conjunto de diretivas e funções, que coordenam a execução de diversas *threads* utilizando de recursos como memória compartilhada e sincronizações, por meio de operações atômicas e barreiras, para que, trabalhando paralelamente, um conjunto de *threads* possa resolver um dado problema comum de forma eficiente [22].

3.1.1 Constituição básica de um programa em CUDA

A programação em CUDA consiste em um modelo heterogêneo de programação onde parte do código executa na CPU (*host*) e outra parte em GPUs (*device*). Um programa em CUDA inicia sua execução no *host* e a partir de comandos de configuração de execução (linha 3 no Algoritmo 1), são inicializadas *threads* nos *devices*, que executam concorrentemente funções conhecidas como *kernels* [1].

Os parâmetros de configuração de execução entre os símbolos <<< e >>> definem a quantidade de *threads* que serão executadas paralelamente. O primeiro parâmetro define

Algoritmo 1 Configuração de execução de um *kernel*.

```
1: dim3 dimencaoGrid(2, 2, 1);
2: dim3 dimencaoBloco(16, 16);
3: funcaoKernel<<<dimencaoGrid, dimencaoBloco>>>();
```

a quantidade de blocos em um *grid*, o segundo define a quantidade de *threads* por bloco. Um bloco é conjunto de *threads*, que em geral são responsáveis por processar um mesmo conjunto de dados, a sua utilização e aplicação será explicada posteriormente [1].

A estrutura de dado *dim3* (linhas 1 e 2 no algoritmo 1) é um vetor de três inteiros definida pela API do CUDA e é utilizada para especificar a quantidade de blocos ou *threads* e como elas serão organizadas logicamente em três dimensões, x, y e z. No algoritmo 1, a chamada de função da linha 3 inicializa a execução de 4 blocos (2x2x1) que executam 256 *threads* (16x16x1) cada, totalizando 1024 *threads* [1]. É possível também realizar a chamada do *kernel* sem a utilização da estrutura dim3, deve-se passar em seu lugar um inteiro. Essa operação equivale a utilização da estrutura dim3 com passagem de apenas um argumento, nesse caso as dimensões y e z são inicializadas com 1 e a dimensão x recebe o argumento de entrada.

Todas as *threads* inicializadas por essa instrução executam um mesmo *kernel*, e portanto, um mesmo código, em consequência disso, é necessário uma maneira que permita diferentes *threads* realizarem diferentes fluxos de execução e/ou processarem diferentes dados. Para isso, são utilizados os *Ids* das *threads* e dos blocos, que são números exclusivos que identificam um bloco dentro de um *grid* ou uma *thread* em um bloco [1].

As coordenadas de uma *thread* e de um bloco em conjunto com as dimensões dos blocos e dos *grids* podem ser utilizadas para gerar um identificador único para cada *thread* (Algoritmo 2), essas podem ser acessadas por meio das estruturas *threadIdx*, *blockIdx*, *blockDim* e *gridDim*, respectivamente [1].

Algoritmo 2 Cálculo do identificador do bloco e da thread.

```
1: int blockId = blockIdx.x + (blockIdx.y * gridDim.x)
2:                 + (blockIdx.z * gridDim.x * gridDim.y);
3: int threadId = (blockId * blockDim.x * blockDim.y * blockDim.z)
4:                  + (threadIdx.z * blockDim.x * blockDim.y)
5:                  + (threadIdx.y * blockDim.x)
6:                  + threadIdx.x;
```

3.1.2 Hierarquia de Memória

Um dos aspectos mais importantes que deve ser levado em consideração ao implementar um programa em CUDA é a utilização correta da hierarquia de memória da GPU e da

CPU. A *device memory* é a hierarquia de memória da GPU, que pode ser alocada e acessada de diversas formas. Os seus tipos são: a memória global, memória constante, memória local e memória de textura [22]. Um exemplo da distribuição lógica de memória na GPU se encontra na figura 3.2.

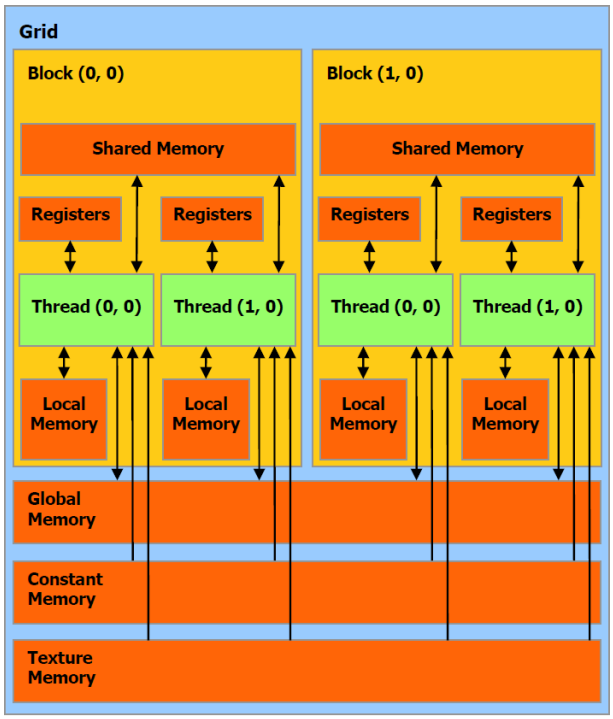

Figura 3.2: Organização lógica das diferentes memórias da GPU [1]

Memória do *Host*

A memória associada à CPU, em CUDA chamada de *host memory*, é normalmente paginada. Apesar dessa técnica possibilitar a execução de programas com necessidades acima das capacidades de memória, a utilização de um endereçamento virtual com valores que não necessariamente correspondem a posições reais na memória dificulta o acesso direto da memória por outros periféricos, dentre eles a GPU [22].

O acesso direto de memória (DMA) pelo device porém, é possibilitado por uma particularidade do sistema operacional que permite certas partes da *host memory* serem alocadas de forma não paginada (*page-locked*) e posteriormente mapeadas na GPU, esses trechos de memória são conhecidos como *pinned memory*. Tal método traz grandes vantagens na comunicação *host-device* e *device-host*. Além do acesso direto à memória ele permite altas velocidades de transferência e cópias assíncronas de memória [22].

Para essa utilização otimizada de memória do *host* a API disponibiliza as funções para alocação e liberação de memória descritas no algoritmo 3. Vale notar que a utilização

excessiva deste método pode degradar a performance do sistema dada a exaustão de espaços de memória paginável [22].

Algoritmo 3 Interface para as funções de alocação de memória *page-locked* no *host*.

```
1: cudaError_t cudaHostAlloc(void ** pHost, size_t size, unsigned int flags);
2: cudaError_t cudaFreeHost(void * ptr);
```

Memória Global

A memória global é uma das principais abstrações de memória da GPU, podendo ser acessada por todas as *threads* de todos os blocos assim como o ilustrado na figura 3.2. Estando diretamente ligada ao hardware do *device*, esse tipo de memória é a mais lenta dentre as memórias da GPU [22].

O uso desse tipo de memória pelo *kernel* pode ocorrer através dos *device pointers*, ponteiros que possuem um endereço pertencente a área de memória do *device*. O trecho de algoritmo 4 ilustra essa utilização em um *kernel* [22].

Algoritmo 4 Trecho de código da definição de um *kernel*, as *d_in* e *d_out* são os *device pointers* que apontam para trechos de memória global.

```
1: ___device___ int d_out[10];
2: ___device___ int d_in[10];
3: ___global___
4: void decrementa() {
5:     int id = threadId.x;
6:     float aux = d_in[id];
7:     d_out[id] = aux - 1;
8: }
```

A maior parte da memória global é alocada dinamicamente e para tais eventos de alocação e liberação de memória o CUDA possui funções específicas, são elas as descritas pelo algoritmo 5. Alocações estáticas são possíveis utilizando do prefixo ___*device*___ na frente das declarações de variáveis.

Algoritmo 5 Funções para alocação de liberação de memória no *device*.

```
1: cudaError_t cudaMalloc(void **, size_t);
2: cudaError_t cudaFree(void);
```

Memória Constante

A memória constante, como o nome infere, consiste de uma área de memória definida como apenas para leitura e é otimizada para esse tipo de operação. Quanto ao seu escopo, assim como o visto na figura 3.2, essa área de memória é compartilhada entre todas as *threads*. Para a sua utilização é necessária a adição do termo ___*constant*___ na frente de cada declaração assim como o visto no trecho de código 6. A sua velocidade de acesso se equipara com a da memória global.

Algoritmo 6 Código *kernel* de declaração e definição de uma constante.

```
1: ___constant___ int array[2] =  0, 1 ;
```

Memória de Textura

Assim como a memória constante a memória de textura é apenas para leitura. Esse tipo de memória é segmentada em duas formas, o CUDA *array* que contém a alocação física da memória e a *texture reference* que contém a abstração de acesso da memória, possuindo informações sobre como o CUDA *array* deve ser endereçado e como o seu conteúdo deve ser interpretado, permitindo assim a sua leitura e escrita. [22]

Memória Local

A memória local reside na *device memory* e apresenta a mesma banda e latência da memória global. Sua organização porém, ocorre de forma que cada *thread* possui acesso para uma parte específica da memória de acordo com seu *id*, e logo é individual a cada uma delas [1]. Normalmente ela é utilizada para armazenar variáveis do *device* que não cabem nos registradores, como nos casos de grandes estruturas de dados e vetores que não são declarados com tamanhos constantes. [1]

Memória Compartilhada

A *shared memory* ou memória compartilhada, é acessível à todas as *threads* de um determinado bloco e possui o ciclo de vida do mesmo, ou seja, após declarada deixa de existir no instante que todas as *threads* do bloco finalizam a execução. Esse tipo de memória possui latência e banda muito maiores do que as do memória global, sendo 10 vezes mais rápida que essa [22]. A sua utilização é feita através da aplicação da diretiva ___*shared*___ antes de declarações das variáveis. Um exemplo se encontra no trecho de código 7.

Algoritmo 7 Código ilustrando a utilização de memória compartilhada.

```
1: /** Utilizando 128 threads e 1 bloco, e um ponteiro
2: para um array de 128 inteiros na memória global **/
3: ___global___
4: void usar_mem_compartilhada(int* vetor) {
5:     int id = threadId.x;
6:     ___shared___ int aux[128];
7:     aux[id] = vetor[id];
8:     ___syncthreads();
9:     aux[id] = aux[(id+1) % blockDim.x]*10;
10:    ___syncthreads();
11:    vetor[id] = aux[id];
12: }
```

Dada a diferença significante entre a velocidade da memória compartilhada e da memória global, é comum que programas que buscam atingir altos níveis de performance sigam um fluxo de execução semelhante ao seguinte :

1. Dados na host memory passam à memória global do device;
2. Informação é carregada na memória compartilhada e o comando ___syncthreads() é utilizado, fazendo com que as threads do bloco esperem no ponto da chamada até que todas se encontrem no mesmo lugar, evitando assim que alguma thread prossiga sem que a atribuição tenha sido toda completada;
3. O processamento é feito e o comando ___syncthreads() é novamente chamado;
4. Os resultados são escritos na memória global e passados à host memory.

Desta maneira, o processamento dos dados é feito majoritariamente na memória compartilhada.

Cópia de Memória

A GPU não possui acesso direto às memórias do *host* e vice versa, para comunicação entre os dois é necessária uma função específica. Para tal o CUDA implementa uma rotina que contempla as comunicações *HostToHost*, *HostToDevice*, *DeviceToHost* e *DeviceToDevice*.

A função em questão tem sua interface ilustrada no trecho de código 8. Nela *dst* se refere a um ponteiro de destino, *src* um ponteiro de origem, *count* o número de bytes a ser copiado e *kind*, o tipo de operação a ser realizado, podendo ser *cudaMemcpyHostToHost*, *cudaMemcpyHostToDevice*, *cudaMemcpyDeviceToHost*, *cudaMemcpyDeviceToDevice* ou *cudaMemcpyDefault*.

Um exemplo real da aplicação dessa rotina se encontra no algoritmo 8 que ilustra a sua utilização passando primeiro os dados do *host* para o *device* e, após feito o processamento, realizando a passagem inversa dos dados [22].

É importante perceber que na utilização do CUDA os pontos do código que mais demandam tempo em sua execução são justamente os referentes a essas cópias de memória e portanto a construção de programas utilizando dessa API devem ter em mente a quantidade de transferências de dados deve ser mínima em comparação com o número de cálculos a serem feitos.

3.1.3 Organização das Threads, Streaming Multiprocessors e SIMT

As GPUs feitas pela NVIDIA possuem múltiplas unidades de multiprocessamento chamadas de *Streaming Multiprocessor* (SM), que são os componentes de hardware responsáveis por executar os kernels. Cada um destes multiprocessadores são compostos por milhares de registradores, memória compartilhada, cache das memórias constantes, local, global e de textura, escalonadores de *warps* e centenas de unidades de lógica e aritmética para cálculo de operações de inteiros e de ponto flutuante (CUDA *cores*)[22]. Como exemplo, as GPUs da arquitetura Pascal (GP100) contém 6 GPC (*Graphics Procesing Clusters*) cada qual com 10 SMs (*Stream Multiprocessors*), que possuem 64 cores cada um, totalizando 3840 *cores* para a placa com um todo, conforme a Figura 3.3 [2].

Quando a chamada para execução de um *kernel* é realizada, as *threads* deste são escolhidas em unidades de blocos para executar nos SMs. Dentro de cada SM os blocos são subdivididos em conjuntos de 32 *threads* de *Ids* consecutivos e crescentes chamados de *warps*. Os *warps* constituem um conjunto mínimo de execução dentro de um SM, e são as unidades escalonadas pelo escalonador de *warps* para executarem paralelamente nos CUDA *cores*.

A execução de um *warp* segue o modelo *Single Instruction Multiple Thread* (SIMT), no qual todas as *threads* que fazem parte de um mesmo *warp* sempre executam uma mesma instrução em um dado momento. No entanto, caso seja executada uma instrução de desvio (*branch*), a condição pode ser avaliada verdadeira para algumas *threads* e falsa para outras. Nesse caso, as *threads* cuja condição foi avaliada verdadeira seguem pelo caminho "*Then*"enquanto as outras *threads* esperam. Ao terminar a execução do caminho "*Then*", o caminho "*Else*"é executado pelas outras *threads*, enquanto as primeiras esperam. Tal fenômeno chama-se divergência de desvios (*Branch divergence*) e essa divergência acarreta em uma perda de performance e um mal aproveitamento dos recurso da GPU, que só é evitado quando todas as *threads* de um *warp* sempre executam a mesma instrução.

Algoritmo 8 Código que ilustra a utilização da cópia de memória. Aqui é criado um vetor que tem em cada posição o seu índice e é feita uma cópia sua para um vetor presente no *device*. O *kernel* utilizado incrementa em um cada posição do vetor. Ao fim do processamento é feita a cópia do vetor resultado presente no *device* para um *array* no *host*.

```
 1: ___global___
 2: void incrementar(float* d_out, float* d_in) {
 3:     int id = threadIdx.x;
 4:     float aux = d_in[id];
 5:     d_out[id] = aux + 1;
 6: }
 7:
 8: int main(int argc, char ** argv) {
 9:     const int ARRAY_NUM_BYTES = 128 * sizeof(float);
10:     // gera o array de entrada e saida no host
11:     float h_in[128];
12:     float h_out[128];
13:     for (int i = 0; i < 128; i++) {
14:         h_in[i] = float(i);
15:     }
16:     // declara os ponteiros para o device
17:     float * d_in;
18:     float * d_out;
19:     // aloca a memória na GPU
20:     cudaMalloc((void**) &d_in, ARRAY_NUM_BYTES);
21:     cudaMalloc((void**) &d_out, ARRAY_NUM_BYTES);
22:     // realiza a cópia de dados do host para o device
23:     cudaMemcpy(d_in, h_in, ARRAY_NUM_BYTES,
24:                     cudaMemcpyHostToDevice);
25:     // faz a chamada do kernel
26:     incrementar<<<1, 128>>>(d_out, d_in);
27:     // realiza a cópia de dados do device para o host
28:     cudaMemcpy(h_in, d_out, ARRAY_NUM_BYTES,
29:                     cudaMemcpyDeviceToHost);
30:     // print out the resulting array
31:     for (int i =0; i < 128; i++) {
32:         printf("
33:     }
34:     // libera a memória global na gpu
35:     cudaFree(d_in);
36:     cudaFree(d_out);
37:     return 0;
38: }
```

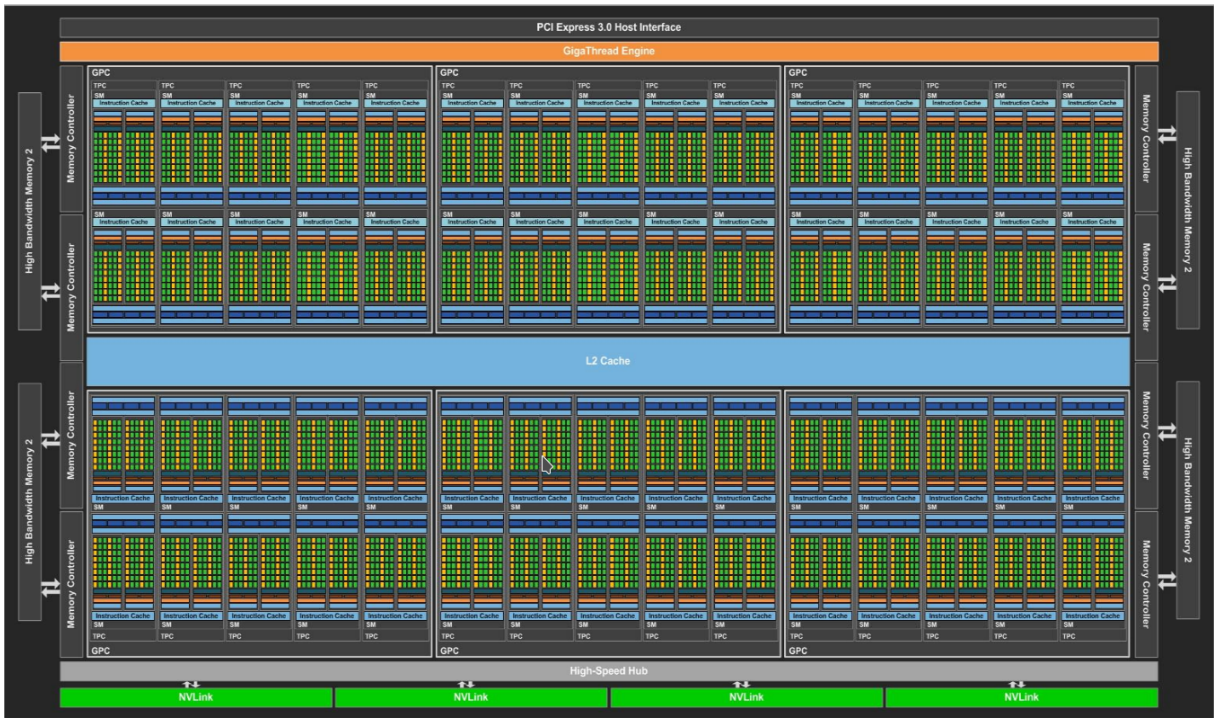

Figura 3.3: GPU Pascal GP100 completa [2].

Portanto, ao implementar um programa em CUDA, o programador pode levar em consideração os *Ids* que serão atribuídos às *threads* para prever quais *threads* constituirão um mesmo *warp*, e assim, evitar cuidadosamente a ocorrência da divergência de controle. A mesma consideração não deve ser utilizada como garantia de sincronismo de execução entre as *threads* em um *warp*, pois levaria a possíveis erros ao executar o código em futuras arquiteturas que utilizam *warps* de tamanho diferente.

Capítulo 4

CUDAlign

Desenvolvido na Universidade de Brasília, o CUDAlign [10][3] é uma ferramenta que surgiu inicialmente com o objetivo de encontrar o *score* ótimo do alinhamento entre duas sequências de DNA utilizando memória linear. Atualmente o CUDAlign é uma das poucas ferramentas capazes de obter eficientemente o alinhamento e *score* ótimos entre sequências muito longas. Para isso, o CUDAlign utiliza diversos algoritmos, dentre eles o algoritmo de SW (Seção 2.3.2) e o algoritmo de Gotoh (Seção 2.3.3), implementados seguindo a abordagem de programação paralela da arquitetura CUDA (Seção 3.1).

O CUDAlign foi se aprimorando ao longo dos anos, apresentando as versões 1.0 [10], 2.0, 2.1, 3.0 e 4.0 [3], que serão explicadas nas seções seguintes.

4.1 CUDAlign 1.0

A primeira versão do CUDAlign [10] utiliza o algoritmo de SW com *affine gap* (Seção 2.3.3) e memória linear para obter o *score* ótimo do alinhamento entre duas sequências em uma GPU. Essa ferramenta foi a primeira capaz de comparar, em tempo viável, cromossomos muito grandes utilizando métodos exatos. Grande parte desse alto desempenho vem do paralelismo obtido ao aplicar a técnica do *wavefront* entre blocos (paralelismo externo) e entre *threads* dentro de um bloco (paralelismo interno).

4.1.1 *Wavefront*

O método *wavefront* é uma técnica que pode ser utilizada no cálculo das matrizes dos algoritmos da seção 2.3. Essa técnica consiste dividir a matriz em grupos de células em uma mesma anti-diagonal e processá-las em ordem, começando da anti-diagonal esquerda superior e finalizando na direita inferior. Esse agrupamento de células é possível devido ao padrão de dependências do cálculo de cada célula, no qual o valor $H[i, j]$ depende dos

Figura 4.1: Execução em *wavefront* [3].

valores das células $H[i-1, j]$, $H[i, j-1]$ e $H[i-1, j-1]$. Portanto, células em uma mesma anti-diagonal, por exemplo $H[i-1, j+1]$, $H[i, j]$ e $H[i+1, j-1]$, podem ser processadas paralelamente pois não dependem uma das outras.

Como ilustrado na Figura 4.1, nas primeiras anti-diagonais processadas no *wavefront* o paralelismo máximo não é obtido, visto que as anti-diagonais não tem o tamanho máximo, porém, o paralelismo cresce a cada anti-diagonal processada, até atingir um valor máximo na anti-diagonal *d5*. Na anti-diagonal *d10* o paralelismo passa a decrescer até o fim do processamento da matriz.

4.1.2 Paralelismo Externo

O método de *wavefront* é aplicado primeiramente entre blocos de células. Uma matriz, que alinha duas sequências de comprimento m e n, é dividida em m/R * n/C blocos de células, onde R é a quantidade de linhas e C é a quantidade de colunas de cada bloco. Esses valores são relacionados com o comando de configuração de execução do *kernel* $<<<$ B, T $>>>$ (Seção 3.1), de forma que $C = n/B$ e $R = \alpha T$, onde α é a quantidade de linhas que cada *thread* ficará responsável por processar, e B e T são as quantidades de blocos e *threads* por bloco respectivamente. Em seguida esses blocos de células são agrupados em anti-diagonais D_k, chamadas de diagonais externas, seguindo a fórmula $D_{\mathrm{k}} = \{G_{\mathrm{i,j}} | i + j = k\}$.

Como o processamento das células de um bloco depende de valores de células do mesmo bloco ou de valores de blocos da diagonal externa imediatamente anterior, o processamento dos blocos de uma diagonal externa pode ocorrer paralelamente e em qualquer ordem. Desta forma, a CPU invoca uma sequência de *kernels*, um de cada vez, para processarem cada uma das diagonais externas seguindo a ordem de processamento do *wavefront*, onde o primeiro *kernel* processa a diagonal externa da esquerda superior e o último a da direita inferior.

4.1.3 Paralelismo Interno

Dentro de cada bloco o processamento também é realizado utilizando o método *wavefront*. As células do bloco são agrupadas em diagonais internas de acordo com a fórmula $d_{\mathrm{k}} = \{(i,j)|\ \lfloor i/\alpha \rfloor + j = k\}$, onde i e j são índices relativos a célula superior esquerda do mesmo bloco. Cada *thread* T_k processa as células das linhas αk até αk+α-*1*, da esquerda para a direita, sendo sincronizadas entre elas por meio da diretiva *__syncthreads()* (Seção 3.1), para calcularem sempre a mesma diagonal interna.

Como o processamento das células de uma diagonal interna que uma *thread* é responsável por processar depende de valores de células calculadas na mesma diagonal interna pela mesma *thread* ou da diagonal interna imediatamente anterior, o processamento da diagonal interna pode ocorrer paralelamente entre as *threads*.

4.1.4 Estruturas em Memória

Como mencionado na seção 3.1.2, os tipos de memória da arquitetura CUDA diferem significantemente em velocidade de acesso, tamanho e escopo. Em consequência disso, as estruturas de dados utilizadas pelo CUDAlign 1.0 foram escolhidas para serem armazenadas nas diferentes memórias da GPU levando-se em consideração a compatibilidade das características de cada estrutura de dados com a memória. As estruturas de dados utilizadas no CUDAlign 1.0 armazenadas em memória da GPU são listadas a seguir:

- Sequências : As sequências são armazenadas na memória de textura.
- Memória Compartilhada : Dentro de um mesmo bloco, as *threads* enviam os valores da última linha pela qual são responsáveis, para a próxima *thread* no mesmo bloco pela memória compartilhada.
- Barramento Horizontal : Os valores da última linha da última *thread* de cada bloco são armazenados no barramento horizontal, que fica na memória global, para serem utilizadas pelo bloco imediatamente inferior na invocação do próximo *kernel*. A leitura pelo bloco inferior é feita pela memória de textura.
- Barramento Vertical : Os valores da última coluna de cada *thread* são armazenados no barramento vertical, que fica na memória global, para serem utilizadas pelo o bloco imediatamente à direita na invocação do próximo *kernel*.

4.1.5 Otimizações

Com o objetivo de aumentar o paralelismo do algoritmo foram introduzidas algumas otimizações importantes no CUDAlign 1.0.

Delegação de Células

A delegação de células [10] é uma técnica que otimiza o paralelismo do processamento das diagonais internas. Na seção 4.1.1 foi mencionado que no início e no fim do processamento em *wavefront* o nível de paralelismo obtido é baixo, devido ao fato de que a quantidade de células a serem processadas nas anti-diagonais iniciais e finais não é máxima. Essa situação de baixo nível de paralelismo ocorre no processamento de todos os bloco de todas as invocações de *kernel*.

Para contornar esse problema, foi proposta a delegação de células. Nessa técnica as *threads* de um bloco processam até a última diagonal interna em que o paralelismo é máximo, e as células pendentes de processamento no mesmo bloco tornam-se responsabilidade de *threads* de blocos seguintes. Desta forma, as etapas finais de baixo paralelismo não ocorrem, e o baixo paralelismo das etapas iniciais é evitado com a adição do processamento de células pendentes de blocos anteriores.

Com a aplicação de delegação de células, os blocos deixam de ser retangulares e passam a ter formato de paralelogramo não retangular.

Divisão de Fases

A delegação de células introduz um novo problema no processamento em *wavefront*, pois o formato de paralelogramo faz com que o processamento de algumas células dependa de valores de células de outros blocos contidos na mesma diagonal externa, o que impede que os blocos de um mesmo *kernel* executem paralelamente e em qualquer ordem. Essa dependência é evitada dividindo-se o processamento de diagonais externas em duas fases [10], a fase curta e a fase longa.

Inicialmente a CPU invoca o *kernel* da fase curta, em que todos blocos processam as $T - 1$ diagonais internas na parte esquerda do paralelogramo. Essas diagonais contém células pendentes que antes eram de responsabilidade de outro bloco. Ao o fim da execução do *kernel* da fase curta o controle é retornado à CPU, que invoca o *kernel* da fase longa, no qual os blocos processam as diagonais internas restantes compostas por células que eram originalmente do bloco.

O sincronismo do processamento dos blocos é feito pelo retorno de controle à CPU, que garante que as células que geram dependência entre blocos da mesma diagonal externa já estejam processadas quando forem necessárias na fase longa.

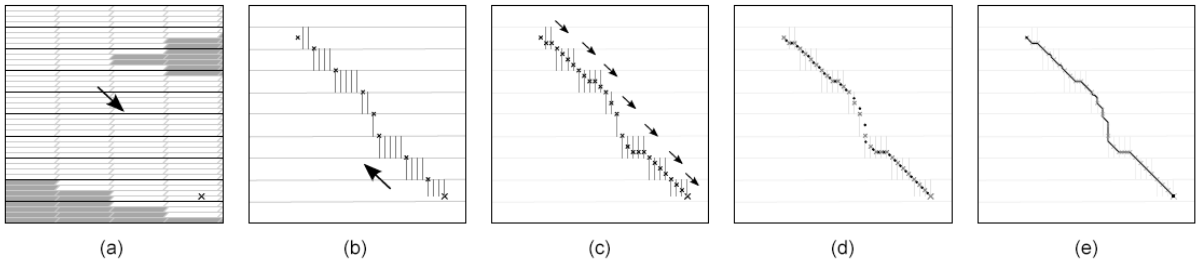

Figura 4.2: Representação gráfica dos diversos estágios do CUDAlign 2.0

4.2 CUDAlign 2.0

O CUDAlign 2.0, proposto em [3], tem por objetivo encontrar, além do *score* ótimo, o alinhamento local ótimo das sequências de entrada, em memória linear e em tempo reduzido. A ideia aqui consiste em descobrir alguns dos pontos do alinhamento desejado e, a partir dos mesmos, realizar procedimentos incrementais de forma a encontrar uma quantidade suficiente de pontos que permita a construção do alinhamento completo.

Como base para a construção dessa versão foram utilizadas as ideias propostas nos algoritmos de Myers e Miller (Seção 2.3.5) e FastLSA [23]. Além disso foram propostas 3 otimizações referentes à área processada da matriz: o *matching baseado em objetivo*, a *execução ortogonal* e a *divisão balanceada*.

A execução do CUDAlign foi dividida em 6 estágios: Obtenção do Escore Ótimo, Traceback Parcial, Divisão de Partições, Myers-Miller otimizado, Obtenção do alinhamento completo e Visualização.

4.2.1 Estágio 1: Obtenção do Score Ótimo

Este primeiro estágio (Figura 4.2 (a)) do algoritmo calcula as matrizes de programação dinâmica na sua totalidade, o score ótimo e a coordenada final do alinhamento ótimo. Para tal, ele utiliza o algoritmo do CUDAlign 1.0 (seção 4.1) com uma modificação específica: através do barramento horizontal, em intervalos de x em x blocos, são salvas *linhas especiais* da matriz. Essas linhas são utilizadas posteriormente no estágio 2 e servem também como pontos de retomada da execução para casos de interrupção do algoritmo.

Para guardar as *linhas especiais* é utilizada uma área no disco chamada de *special rows area* (SRA) que possui |SRA| bytes, quantidade definida em tempo de execução. Cada célula da linha a ser armazenada ocupa 8 bytes, 4 para o valor referente a matriz H e 4 à matriz F. A quantidade de linhas que pode ser salva é, portanto, |SRA|/(8n) e o intervalo entre blocos os quais as linhas são armazenadas é de $8mn/(\alpha * T * |SRA|)$ onde $\alpha * T$ representa o número de linhas calculado por bloco.

4.2.2 Estágio 2: Traceback Parcial

O estágio 2 recebe como entrada os dados resultantes do primeiro estágio e, através da execução de um alinhamento semi-global na direção reversa, encontra todos os pontos do alinhamento ótimo que se encontram nas linhas especiais, a posição inicial do mesmo e salva algumas *colunas especiais* (Figura 4.2 (b)). As otimizações *matching* baseado em objetivo e execução ortogonal são utilizadas nessa fase.

Nesse estágio uma versão do algoritmo de Myers-Miller (Seção 2.3.5) é executada para achar o ponto do alinhamento ótimo em cada linha especial. Aqui a otimização chamada de *matching baseado em objetivo* é utilizada. Nesta versão de Myers-Miller (seção 2.3.5), diferentemente da original, o algoritmo não precisa percorrer toda a linha em busca da dupla de células que geram o maior *score*, pois tem-se conhecimento do *score* máximo que pode ser gerado. Esse valor é conhecido como *escore-alvo* e inicializado com o *score* ótimo do alinhamento encontrado no estágio 1. Sua atualização é feita com o valor do *score* encontrado na coordenada da linha especial, sempre que novas coordenadas são descobertas.

Para que se tenha ganho com essa otimização porém, é necessária uma modificação na forma com que as threads são executadas. As threads no estágio 1 são executadas na direção horizontal, já no estágio 2 a execução deve ocorrer na vertical, direção ortogonal à do estágio 1, para que a área de processamento de fato diminua. Essa alteração é a otimização *execução ortogonal*[3].

4.2.3 Estágio 3: Divisão de Partições

Diferentemente do estágio 2, o estágio 3 trabalha com partições bem definidas montadas a partir das coordenadas obtidas no estágio anterior. O seu objetivo é encontrar ainda mais coordenadas que pertencem ao alinhamento ótimo (Figura 4.2 (c)).

Para encontrar essas coordenadas o algoritmo executado no estágio 2 é utilizado, desta vez com as buscas voltadas para as *colunas especiais*. Como cada execução ocorre dentro de partições que não dependem de valores externos para chegar a um resultado, esse estágio permite um amplo paralelismo no que compete ao cálculo das coordenadas.

4.2.4 Estágio 4: Myers-Miller otimizado

Por meio de sucessivas iterações o estágio 4 executa-se em CPU de modo a aumentar o número de coordenadas encontradas do alinhamento (Figura 4.2 (d)). Para isso, ele executa o algoritmo de Myers-Miller (Seção 2.3.5) nos intervalos entre as coordenadas extraídas do estágio 3 e se prolonga até que o tamanho de todas as partições entre as coordenadas encontradas seja menor do que uma certa constante: o *tamanho máximo de*

partição [3] que, escolhida empiricamente, possui o valor de 16 bases. Assim como visto no estágio 3 o trabalho distribuído nas diversas partições facilita o paralelismo.

O algoritmo Myers-Miller, descrito em 2.3.5, possui seu funcionamento baseado na divisão sucessiva da matriz na linha central. No estágio 4 uma abordagem diferente é tomada, nele a matriz passa a poder ser dividida também em sua coluna central. Essa otimização é conhecida como *divisão balanceada* [3] e previne a existência de partições com lados excessivamente desproporcionais. A *execução ortogonal*[3] é também utilizada nesse estágio.

4.2.5 Estágio 5: Obtenção do alinhamento completo

O estágio 5 recebe como entrada as coordenadas do alinhamento ótimo com partições entre si de tamanho fixo definido pelo *tamanho máximo de partição*. Utilizando desses dados, o objetivo desse estágio é realizar o alinhamento completo das sequências (Figura 4.2 (e)). Para tal, o algoritmo NW (seção 2.3.1) é executado em CPU para cada uma das partições, resultando na descoberta de todas as coordenadas remanescentes do alinhamento. Devido ao tamanho constante das partições, esse procedimento pode ser realizado com utilização linear de memória.

4.2.6 Estágio 6: Visualização

O estágio 5 apresenta como saída um arquivo binário que permite a representação otimizada do alinhamento, sem o armazenamento das bases das sequências. O estágio 6 é opcionalmente utilizado para a visualização textual e gráfica desse arquivo binário.

4.3 CUDAlign 2.1

No CUDAlign 2.1 [24], foi introduzida a otimização *Block Pruning* no estágio 1 (Seção 4.2.1), técnica por meio da qual o processamento de alguns blocos de células são evitados pois pode-se garantir que estes não contribuem para o alinhamento local ótimo.

4.3.1 *Block Pruning*

O *Block Pruning* utiliza a ideia de que o maior score obtido em um dado instante no processamento da matriz, o $best(i)$, pode ser utilizado para classificar células como *prunable*. A classificação de uma células como *prunable*, indica que essa garantidamente não contribui para o alinhamento local ótimo.

Uma célula (i, j) é *prunable* quando o seu score, $H(i, j)$, somado a pontuação de um alinhamento local perfeito das subsequências restantes após a mesma célula, é menor que o score $best(i)$. Ou seja uma célula é *prunable* quando $H(i, j)+min(\Delta i, \Delta j)*ma <= best(i)$, onde Δi é a distância entre a linha i e a última linha da matriz, Δj é a distância entre a coluna j e a última coluna da matriz e *ma* é a pontuação de *match*.

A partir das células *prunables* pode-se classificar células ainda não processadas também como *prunable* da seguinte forma: Se as células $(i, j-1)$, $(i-1, j)$ e $(i-1, j-1)$ são *prunable* então (i, j) também é *prunable*. Assim, o processamento destas células pode ser dispensado sem impactar o resultado do score final.

Para amenizar o *overhead* causado pela aplicação dessa técnica, as classificações são feitas considerando-se blocos inteiros, e não células. A classificação de um bloco como *prunable* é feita levando em consideração que o maior score obtido no bloco está no canto esquerdo superior do bloco, e portanto, o mais distante possível da última linha e coluna.

No alinhamento de sequências similares, o *Block Pruning* possibilita descartar até mais de 50% do total de células [24], e nos casos em que as sequências são pouco similares o ganho em desempenho é baixo, porém, o *overhead* causado pela aplicação do *Block Pruning* também é baixo.

4.4 CUDAlign 3.0

Mesmo com o aumento de velocidade obtido no CUDAlign 2.1 (seção 4.3), para sequências muito grandes o resultado ainda não é satisfatório em execuções com uma só GPU. Com uma entrada de 33 MBP o CUDAlign 2.1 ainda chega a demorar mais de 8 horas [3].

Com enfoque nesse cenário, o CUDAlign 3.0 [25] passou a apresentar suporte para execução em múltiplas GPUs através de uma nova arquitetura, capaz de compartilhar o processamento de uma comparação de sequências entre as várias GPUs simultaneamente. Em particular, é feita essa divisão no estágio 1 (Seção 4.2.1), a etapa mais demorada do CUDAlign.

No CUDAlign 3.0 cada GPU recebe uma parte da matriz de programação dinâmica para realizar o processamento, essa divisão é feita de forma a se ter um balanceamento de carga entre as GPUs e logo, entre as não homogêneas, colunas da matriz são distribuídas de forma desigual, de acordo com a capacidade de processamento de cada uma. Essa distribuição é crítica para o funcionamento eficiente do algoritmo, pois cada GPU depende dos resultados referentes às colunas anteriores feitos em outra GPU e caso a comunicação entre elas não seja bem projetada o desempenho pode ser degradado.

Para a comunicação entre as GPUs são utilizados um processo com três *threads* para cada uma delas. Uma *thread* gerente e duas outras voltadas para comunicação. A *thread*

gerente é responsável pela realização do cálculo das células da matriz e a passagem dos dados entre a GPU e as *threads* de comunicação. Já as *threads* de comunicação se ocupam com o transporte assíncrono de dados entre GPUs utilizando de *sockets*.

Essa interface entre as GPUs que permite a troca de dados utiliza de *buffers* circulares. Cada *thread* de comunicação possui um, que pode ser utilizado para entrada ou saída de dados, eles permitem que até certo nível o processamento seja desvinculado da comunicação, tornando imperceptíveis possíveis pequenos problemas na rede. Além disso, o nível de preenchimento do *buffer* é um bom indicador da qualidade do balanceamento, em um *wavefront* balanceado o *buffer* é constante, a entrada dos dados ocorre de forma equilibrada com a saída, pois a taxa de produção de uma GPU equivale à taxa de uso da outra, já em um *wavefront* desbalanceado o *buffer* de saída tenderá a, ou encher completamente, ou se esvaziar, em ambos os casos uma das GPUs fica em espera, prejudicando o desempenho da aplicação.

$$t(m,n) = c_1 + c_2 m + c_3 n + c_4 mn \tag{4.1}$$

$$CUPS(m,n) = mn/t(m,n) \tag{4.2}$$

Para a execução em *clusters*, que apresentam uma grande quantidade de GPUs, é necessário que se especifique o máximo tempo de execução das tarefas nas filas das mesmas. O tempo de execução de uma comparação de sequências S0 de tamanho m e S1 de tamanho n para uma GPU é dado pela equação 4.1, onde a constante c1 corresponde ao tempo gasto com inicializações, c2 e c3 ao gasto com operações de complexidade O(m) ou O(n) e a constante c4 corresponde ao tempo gasto com o cálculo da matriz. Essas constantes podem ser obtidas através de experimentos com tamanhos menores de sequências. Com a equação de tempo de execução para uma GPU pode-se derivar a fórmula para o número de células versus o tempo [3] como sendo a equação 4.2.

$$T_{\mathrm{p}}(m,n) = t(m,n_{\mathrm{p}}) + (p-1) * t(\beta,n_{\mathrm{p}})/2 \tag{4.3}$$

Considerando o tempo gasto para que o *wavefront* tenha se iniciado em todas as GPUs anteriores e o tempo gasto na última GPU pode-se chegar a fórmula 4.3 para a previsão do tempo de execução do algoritmo para múltiplas GPUs dedicadas e homogêneas. Nela, b é definido como $\alpha * B * T$, representando a quantidade de linhas para que o *wavefront* atinja seu paralelismo máximo, p representa o número de GPUs. Logo t(m,n/p) é o tempo previsto para a última GPU e t(β, n/p)/2 a estimativa do tempo necessário para o início do processamento pela última GPU.

4.5 CUDAlign 4.0

O CUDAlign 4.0 [26] é capaz de obter o score e o alinhamento ótimo de sequências muito longas em múltiplas GPUs. Para isso, foram realizadas duas mudanças no estágio 1 da versão 3.0 (Seção 4.4) e foram implementadas duas técnicas para a obtenção do alinhamento, o *Pipelined Traceback* (PT) e o *Incremental Speculative Traceback* (IST).

4.5.1 Modificações no Estágio 1

Como mencionado na Seção 4.1.4, as sequências são armazenadas na memória de textura, e este tipo de memória tem capacidade de armazenar no máximo 2^{27} elementos. Tal fato impõe uma limitação de 134 MBP para o tamanho máximo das sequências que podem ser alinhadas. Para possibilitar o alinhamento entre sequências de tamanho maior que esse limite, e ainda usufruir dos benefícios da velocidade de acesso proporcionada pela localidade espacial da memória de textura, o CUDAlign 4.0, nos casos necessários, realiza um subparticionamento da matriz em quadrantes que caibam dentro da memória de textura e as processa uma de cada vez.

Um outro mecanismo implementado no CUDAlign 4.0 no estágio 1, semelhante ao mecanismo implementado na versão 2.0 (Seção 4.2), é o armazenamento de linhas especiais em arquivos ou em memória RAM, para estas serem utilizadas nos estágios seguintes de *traceback* e de obtenção do alinhamento ótimo.

4.5.2 *Pipelined Traceback* (PT)

No *Pipelined Traceback* cada GPU executa os estágios de 2 a 4 em um pipeline. Quando uma GPU encerra o estágio 1 ela espera receber o *crosspoint* do estágio 2 da GPU à direita para iniciar o seu estágio 2. Essa dependência serial dos *crosspoint* das GPUs faz com que elas fiquem ociosas durante um longo período de tempo entre o fim do estágio 1 e o início do estágio 2. A única GPU que inicia o estágio 2 imediatamente após o fim do seu estágio 1 é a última GPU.

Ao finalizar o estágio 2, a GPU envia o *crosspoint* encontrado para a GPU à esquerda e em seguida executa os estágios 3 e 4 em *pipeline*. Ao finalizar o estágio 4 a GPU espera pelas coordenadas do estágio 4 da GPU à direita, e quando as recebe, concatena com as suas e envia para a GPU à esquerda. A GPU mais à esquerda ao receber as coordenadas da GPU à sua direita, que contém as coordenadas do estágio 4 de todas as GPUs, executa os estágios 5 e 6.

A avaliação do desempenho do *Pipelined Traceback* feita em [26], mostra que a escalabilidade do estágio 1 é muito boa, chegando a atingir *speedups* de até 97,3x na execução em

um ambiente com 128 nós de 3 GPUs cada. Em contrapartida, nos estágios de *traceback* o paralelismo não é muito bem aproveitado, por conta da dependência dos *crosspoints* entre as GPUs, sendo que na execução com 128 nós, a fase de *traceback* passou a representar 71% do tempo de execução total, sendo assim, a fase mais demorada.

4.5.3 *Incremental Speculative Traceback* (IST)

O *Incremental Speculative Traceback* é uma técnica por meio da qual as GPUs utilizam o tempo ocioso mencionado na Seção 4.5.2 para especular algumas possíveis coordenadas do *crosspoint* que será enviado pela GPU à direita, e assim, pré-processar o estágio 2 nesses *crosspoints* especulados. Isso é possível dada a observação de que as coordenadas do alinhamento ótimo nas colunas intermediárias entre as GPUs tende a coincidir com os *scores* máximos dessas colunas.

A GPU que receber o *crosspoint* real da GPU à direita e tiver especulado corretamente a coordenada do *crosspoint* real, poderá enviar os resultados pré-processados para a GPU à sua esquerda, caso contrário, o estágio 2 terá que ser realizado no *crosspoint* real recebido.

Para aumentar a probabilidade da especulação acertar as coordenadas do *crosspoint*, as GPUs consideram resultados de especulações de GPUs à direita que não coincidem com as especulações já realizadas, para fazerem especularem em outras coordenadas na coluna intermediária.

A avaliação do impacto do IST no desempenho feita em [26], mostra que o está técnica diminui consideravelmente o tempo da fase de *traceback* e não introduz *overhead* no tempo de execução, dado que o processamento do IST é feito nos momentos em que as GPUs estariam ociosas.

4.6 Balanceamento de *Wavefront* em múltiplas GPUs

A utilização de GPUs com desempenhos minimamente heterogêneas no *wavefront* pode levar a um desbalanceamento do mesmo, causando possíveis sobrecargas ou esvaziamentos dos *buffers* e consequentes problemas de performance. Em [3] é proposta uma solução para esse problema utilizando de agentes capazes de realizar dinamicamente o balanceamento de cargas do *wavefront* com base em uma série de métricas.

4.6.1 Projeto do sistema multiagente

No projeto de um agente existem quatro propriedades a serem consideradas, a PAGE, Percepções (*Percepts*), dados que o agente pode obter no ambiente, Ações (*Actions*), formas que o agente pode modificar o ambiente, Objetivos (*Goals*), motivações que dirigem

o agente, e Ambiente (*Environment*), onde se encontra o agente. O balanceamento proposto em [3] utiliza de dois agentes para cada nodo N, são eles o Agente Balanceador (*Balancer*) e o Agente Executor (Executor).

O agente executor possui como Ambiente o sistema operacional e a rede de comunicações e seu Objetivo é manter e realizar a avaliação do processo de seu nodo. As suas Percepções são: velocidade do algoritmo (linhas processadas por segundo), número de células presentes nos *buffers* e tempo de inatividade na espera por dados. Já quanto às Ações ele realiza a reinicialização do processo na ocorrência de uma redistribuição de colunas [3].

O agente balanceador tem como Objetivo otimizar a distribuição de colunas para diminuir o tempo de execução e seu Ambiente é o sistema multiagentes. As suas Percepções são: estatísticas provindas do agente executor e intenções dos agentes balanceadores presentes nos outros nodos.

Em [3] são propostos dois métodos a serem executados nos Agentes Balanceadores para o reconhecimento da necessidade de um rebalanceamento: A estratégia global que leva em conta os estados de todos os outros agentes nos diversos nodos, e a estratégia local leva em conta apenas os estados próprios.

4.6.2 Métricas da Execução

Tendo o número de colunas e linhas da matriz de programação dinâmica sendo representados respectivamente pelas letras W e H, em [3], as seguintes formalizações são feitas quanto às métricas observadas pelo agente executor:

- c_i — Número de colunas a ser processado por um dado nó N_i;
- s_i — Capacidade de processamento de um dado nó N_i em linhas por segundo;
- n_i — Capacidade de comunicação de um dado nó N_i em células por segundo;
- r_i — Índice correspondente à última linha completamente processada pelo nó N_i;
- $b_i^{<-}$ — Valor correspondente ao número de células presente no *buffer* de entrada de dado nó N_i;
- $b_i^{->}$ — Valor correspondente ao número de células presente no *buffer* de saída de dado nó N_i;
- w_i — Tempo de espera que dado nó N_i se submeteu aguardando dados no *buffer* de entrada;
- w_i — Tempo de espera que dado nó N_i se submeteu aguardando liberação de espaço no *buffer* de saída;

Métricas de Balanceamento Global

A estratégia global deve levar em conta os estados de todos os agentes para identificar se é necessário rebalancear o *wavefront*. Para quantificar essa necessidade o seguinte grupo de fórmulas é utilizado:

- $f_i(x)$ — Função que prevê o desempenho de dado nó N_i ante à designação de x colunas para o processamento: $f_i(x) = k \cdot (s_i/x)$ onde k é uma constante da simulação;
- T_i — Fórmula para definir o tempo restante até o fim da execução de dado nó N_i caso o balanceamento continue o mesmo: $T_i = (H - r_i)/s_i$;
- $\hat{T}$ — Fórmula que define o tempo restante até o fim da execução de todos os nós e é dada pelo valor máximo de T_i: $\hat{T} = \max(T_1...T_k)$;
- $T'_i(x)$ — Fórmula que define o tempo restante até o fim da execução de dado nó N_i caso ocorra um rebalanceamento e ele seja designado com x colunas: $T'_i(x) = (H - r_i)/f_i(x) + \alpha_i(x)$ onde $\alpha_i(x)$ é o custo adicional para reiniciar a computação pós-rebalanceamento;
- $\hat{T}'(c_1, c_2,\ldots,c_k)$ — Fórmula que define o tempo restante até o fim da execução de todos os nós caso ocorra um rebalanceamento. É dada pelo valor máximo de $T'_i(x)$: $\hat{T}'(c1, c2,\ldots,ck) = \max(T'_1(c_1), ..., T'_k(c_k))$;
- $B_i(x)$ — Fórmula que define o benefício de tempo na execução de dado nó N_i ante a um rebalanceamento e consequente redistribuição de colunas para o mesmo: $B_i(x) = T_i - T'_i(x)$. Se seu valor for maior que zero é dito que a nova distribuição de colunas é localmente aceitável, caso contrário, localmente inaceitável;
- $\hat{B}'(c_1, c_2,\ldots,c_k)$ — Fórmula que define o benefício de tempo na execução de todos os nós ante a um rebalanceamento e consequente redistribuição de colunas: $\hat{B}'(c_1, c_2,\ldots,c_k) = \hat{T} - \hat{T}'(c_1, c_2,\ldots,c_k)$. Se seu valor for maior que zero é dito que a nova distribuição de colunas é globalmente aceitável, caso contrário, globalmente inaceitável;

Métricas de Balanceamento Local

Na estratégia global existe a constante necessidade da troca de dados entre os nós para que todos possam compartilhar as informações de suas. Esse alto grau comunicação pode gerar um *overhead* e consequentemente denegrir a performance. Diferentemente da global, a estratégia local identifica a necessidade de balanceamento utilizando apenas das métricas individuais ao nó, evitando assim a onerosa comunicação entre eles. Para quantificar o estado do balanceamento as seguintes métricas são utilizadas:

- $\delta_i^{<-}$ e $\delta_i^{->}$ — Fórmulas das métricas de estabilidade dos *buffers*, correspondem ao incremento da velocidade de processamento em linhas por segundo que o nó N_i deve apresentar para pausar o aumento do número de células no *buffer* de entrada ($bi^{<-}$) e no de saída ($b_i^{->}$), respectivamente: $\delta_i^{<-} = \Delta bi^{<-}/\Delta t$, $\delta_i^{->} = \Delta b_i^{->}/\Delta t$;

- $\psi_i^{<-}$ e $\psi_i^{->}$ — Fórmulas das métricas de estabilidade de bloqueio, correspondem ao o incremento da velocidade de processamento em linhas por segundo que o nó N_i deve apresentar para pausar o bloqueio do *buffer* de entrada ($w_i^{<-}$) e do de saída ($w_i^{->}$), respectivamente: $\psi_i^{<-} = -\ \Delta ri/\Delta t * \Delta wi^{<-}/(\Delta t - w_i^{<-})$, $\psi_i^{->} = -\ \Delta ri/\Delta t * \Delta w_i^{->}/(\Delta t - w_i^{->})$;

- $\pi_i^{<-}$ e $\pi_i^{->}$ — Fórmulas que representam a estabilidade de entrada e saída do nó, respectivamente. É calculada pela soma de δ_i e ψ_i: $\pi_i^{<-} = \delta_i^{<-} + \psi_i^{<-}$, $\pi_i^{->} = \delta_i^{->} + \psi_i^{->}$;

- Π_i — Fórmula que representa a estabilidade local de dado nó N_i, calculada pela soma de $\pi_i^{<-}$ e $\pi_i^{->}$: $\Pi_i = \pi_i^{<-} + \pi_i^{->}$. O nó é considerado localmente estável se o seu valor de Π_i for próximo de zero e localmente instável caso contrário;

- $\hat{\Pi}$ — Fórmula que representa a estabilidade global do *wavefront*. É encontrada através do valor máximo de Π_i: $\hat{\Pi} = \max(\Pi_1, \ldots, \Pi_k)$. O *wavefront* é considerado globalmente estável se o seu valor de $\hat{\Pi}$ for próximo de zero e globalmente instável caso contrário;

4.6.3 Pesos e Negociação do Balanceamento

Para que a divisão de cargas possa ocorrer, é necessária a definição de um peso para cada nó. Através desses pesos $(\psi_1, \psi_2, \ldots, \psi_k)$ utiliza-se da fórmula 4.4, para encontrar a divisão de colunas $(c_1, c_2, \ldots, c_k)$ em uma redistribuição, onde γ' é o somatório de todos os pesos δ_i. Os pesos iniciais são normalmente atribuídos de acordo com a capacidade de cada nó, por meio do número de GFLOPS ou pelo resultado de algum *benchmark*. Já os pesos atualizados são calculados utilizando, na estratégia global, a fórmula 4.5 e, na local, a fórmula 4.6.

$$(c_1, c_2, \ldots, c_k) = (W\gamma_1/\gamma', W\gamma_2/\gamma', \ldots, W\gamma_k/\gamma') \tag{4.4}$$

$$\gamma_i = s_i \tag{4.5}$$

$$\gamma_i = c_i * (\Delta r_i - \psi_i) \tag{4.6}$$

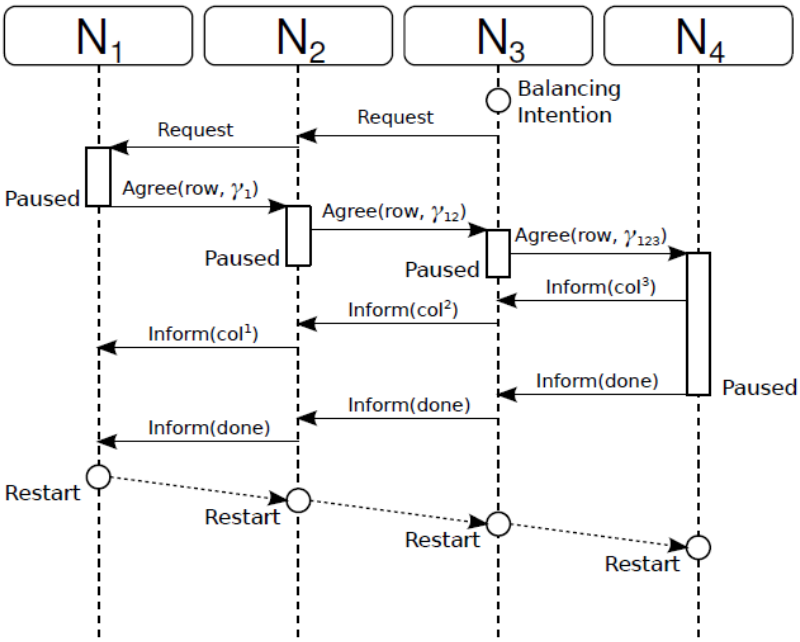

Figura 4.3: Ilustração da negociação do rebalanceamento de cargas [3]

Quando um nó percebe a necessidade da realização de um balanceamento é necessária uma negociação com todos os outros para que ela ocorra. Existem quatro fases no processo da negociação entre os nós (figura 4.3), a Requisição, Concordância, Distribuição e Reinício.

No momento em que dado nó N_i percebe a necessidade do rebalanceamento a fase de requisição se inicia. Ele envia uma mensagem de requisição para o nó à esquerda e a mesma se propaga até chegar no N_1, este altera seu estado para *Balancing* e interrompe a execução enquanto os nós de N_i até N_2 aderem ao estado *Requesting*.

A fase de concordância se inicia já quando o nó N_1 pausa a sua execução. Nesse momento ele envia uma mensagem de concordância para o nó seguinte, passando a última linha calculada e o peso y_1. O nó seguinte então, ao receber a mensagem, passa ao estado de *Balancing* e inicia o processo de pausa de execução realizando o processamento até chegar na última linha calculada por N_1. Antes de acabar tal processamento ele passa à frente a mensagem de concordância, porém passando a soma de seu peso com o do nó anterior, no caso $\gamma_1 + \gamma_2$. Essa operação progride de forma que, supondo a existência de 4 nós, em N_4 se receberia a soma dos pesos $\gamma_1 + \gamma_2 + \gamma_3$ e a última linha processada pelos nós anteriores.

Quando o último nó recebe a mensagem de concordância e chega no estado *Balancing* tem-se o início da fase de Distribuição. Nesse momento ele passa a possuir os dados necessários para calcular a sua própria parte na divisão de colunas, e assim o faz, através da fórmula 4.4. Posterior a esse cálculo, ele passa para o nó à esquerda o número de

colunas ainda não balanceadas. Esse nó, utilizando desse dado e do somatório dos pesos até o mesmo, também encontra a sua própria distribuição de colunas através da fórmula 4.4 (onde W passa a ser o número de colunas não balanceadas). Esse processo progride até que todos os nós possuam sua distribuição de colunas.

O reinício ocorre assim que o último nó completa o processamento da matriz até a linha acordada, propagando uma mensagem de finalização para os outros nós. Dessa forma todos os nós passam ao estado *Satisfied* e retomam o processamento a partir do nó N_1, assim como visto na Figura 4.3.

4.7 *Multi-platform Architecture for Sequence Aligners* (MASA)

Em [3] foi apresentado o *Multi-Platform Architecture for Sequence Aligners* (MASA), uma arquitetura composta por um conjunto de módulos de código independente de plataforma e de módulos de código específico da plataforma, que facilitam o desenvolvimento de ferramentas similares ao CUDAlign (extensões MASA) em diferentes arquiteturas de hardware e software. O MASA implementa as funcionalidades e otimizações de todas as versões do CUDAlign vistas nas Seções 4.1, 4.2, 4.3, 4.4 e 4.5.

Algumas extensões já foram implementadas em [3], são elas a MASA-CUDAlign, MASA-OpenMP/CPU, MASA-OpenMP/Phi e MASA-OmpSs/CPU. A quantidade de linhas do código específico da plataforma das três últimas extensões contém menos de 200 linhas de código [3], o que mostra a facilidade de portar o MASA para uma nova arquitetura.

4.7.1 Arquitetura MASA

A arquitetura MASA é composta por 5 módulos, o módulo de Gerência de Dados, Comunicação, Estatística, Gerenciamento dos estágios e o *Aligner*. O módulo *Aligner*, por sua vez, é composto por outros 3 módulos, o de Tipo de Processamento, *Block Pruning* e Cálculo da Matriz.

Os módulos que constituem o *Aligner* são de código específico de cada plataforma, e tem como responsabilidade o cálculo da matriz de programação dinâmica nos estágios 1 a 3, portanto, são os módulos responsáveis pela fases mais computacionalmente custosas do algoritmo. O *Aligner* é o único módulo que precisa ser implementado para portar o MASA para um nova plataforma.

Os outros módulos são de código independente de plataforma, e por conta disso, podem ser utilizados em qualquer extensão MASA. Esses módulos constituem o MASA-

Core [27] e são compostos por 90% do código original do CUDAlign, sendo responsáveis pelas operações de entrada/saída, estatísticas, comunicação, validação e gerenciamento dos estágios.

4.7.2 MASA-API

O MASA, baseado no paradigma de programação orientada a objetos, expõem uma *Application programming interface* (API) que define as interfaces de comunicação entre o módulo Gerenciamento de Estágios, que é implementado no MASA-Core, e o *Aligner*, que é implementado pelo programador.

Primeiramente, para implementar uma nova extensão o programador deve criar uma classe do módulo *Aligner* que implementa os métodos da interface *IAligner*, esta classe se comunica com o módulo de Gerenciamento de Estágios através da interface *IManager*.

A interface *IAligner* define os métodos abstratos de inicialização e finalização de execução do módulo *Aligner*, de inicialização e finalização de uma iteração de um estágio, e de alinhamento de uma ou mais partições.

A interface *IManager* define os métodos abstratos que retornam parâmetros de alinhamento, que transferem ao *Aligner* as linhas e colunas iniciais da partição, e métodos por meio do qual o *Aligner* envia valores de células para o módulo de Gerenciamento dos Estágios.

O MASA implementa uma hierarquia de classes abstratas e concretas que podem ser utilizadas para facilitar ainda mais a implementação do módulo *Aligner*. A classe abstrata *AbstractBlockPruning* e suas subclasses concretas *DiagonalBP* e *GenericBP* gerenciam a operação de *Block Pruning*. A classe abstrata *AbstractAligner* encapsula os métodos da interface *IAligner* e realiza a comunicação com o módulo de *Block Pruning*, suas subclasses abstratas *AbstractBlockAligner* e *AbstractDiagonalAligner* calculam a matriz de programação dinâmica em blocos retangulares, utilizando o *GenericBP*, ou em *wavefront*, utilizando o *DiagonalBP*, respectivamente.

Capítulo 5

Projeto do Mecanismo de Parada Sincronizada

Conforme mencionado na parte de trabalhos futuros da Tese [3], um mecanismo de *checkpoint* deve ser projetado para que o balanceamento dinâmico de carga seja incorporado. Sendo assim, o presente Projeto de Graduação projetou esse mecanismo e o incorporou à arquitetura MASA (Seção 4.7). Portanto foi proposto e implementado uma funcionalidade de parada sincronizada (*Checkpoint*) dos nós de processamento das sequencias biológicas, que tem como objetivo servir como base para o desenvolvimento da funcionalidade de balanceamento em múltiplos nós mencionados no Seção 4.6.3.

5.1 Visão geral do projeto

A partir da versão do CUDAlign 3.0 (Seção 4.4), o alinhamento das sequências biológicas passou a poder ser realizado em múltiplas GPUs, e posteriormente, na arquitetura MASA, passou-se a utilizar múltiplos nós heterogêneos (Seção 4.7). Por conta da heterogeneidade da capacidade de processamento dos nós, pode-se gerar um significativo desbalanceamento ao longo da operação de alinhamento, fato mencionado na Seção 4.6.3.

Desta forma, foi proposto em [3] e mencionado na Seção 4.6.3 um modelo para o balanceamento dinâmico de carga, no qual os nós, quando identificam um desbalanceamento na operação de alinhamento decidem se é vantajoso ou não realizar um rebalanceamento. Na decisão de rebalanceamento, os nós param de maneira sincronizada o alinhamento e realizam a operação de redistribuição de carga.

Nesse projeto, implementamos um novo módulo à arquitetura MASA, o de *checkpoint*, que realiza a operação de parada sincronizada dos nós, tendo como base o proposto em [3] e com algumas adaptações na arquitetura e na sequência da troca de mensagens entre os nós.

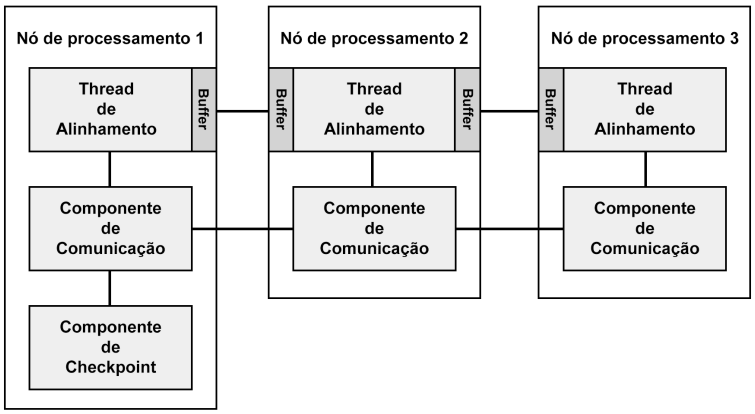

Figura 5.1: Arquitetura do módulo de *checkpoint* e suas conexões

5.1.1 Arquitetura do módulo de Checkpoint

O módulo de *checkpoint* é composto por 2 componentes principais, o componente de comunicação e o componente de *checkpoint*. O componente de *checkpoint* é responsável por decidir se o balanceamento deve ser realizado ou não. No presente este mecanismo de decisão não foi implementado, portanto foi utilizado um temporizador que lança o mecanismo após determinado número segundos. Este componente só está presente no nó responsável pelo primeiro intervalo de colunas do alinhamento (primeiro nó).

O componente de comunicação se encontra presente em todos os nós e é responsável pela troca de mensagens. Por meio dele, são compartilhadas entre todos os nós as informações referentes às decisões do módulo de *checkpoint* e outros dados, como por exemplo o número da linha a partir da qual deve-se voltar na reinicialização.

A figura 5.1 apresenta a arquitetura do módulo de *checkpoint* e a conexão entre os componentes. Os nós são conectados sequencialmente pelos componentes de comunicação na mesma ordem da execução do alinhamento. Os *buffers* que podem ser vistos na imagem, são os *buffers* utilizados pela arquitetura MASA no alinhamento das sequências.

5.1.2 Diagrama de Comunicação

A troca de mensagens entre os nós e a sequência de parada dos nós implementadas neste projeto diferem da negociação de balanceamento proposta em [3] e estão ilustradas na Figura 5.2. A operação de parada se inicia quando o componente de *checkpoint* presente no primeiro nó decide realizar o balanceamento. Em seguida, este componente notifica o componente de comunicação presente no mesmo nó sobre a operação de parada. A partir

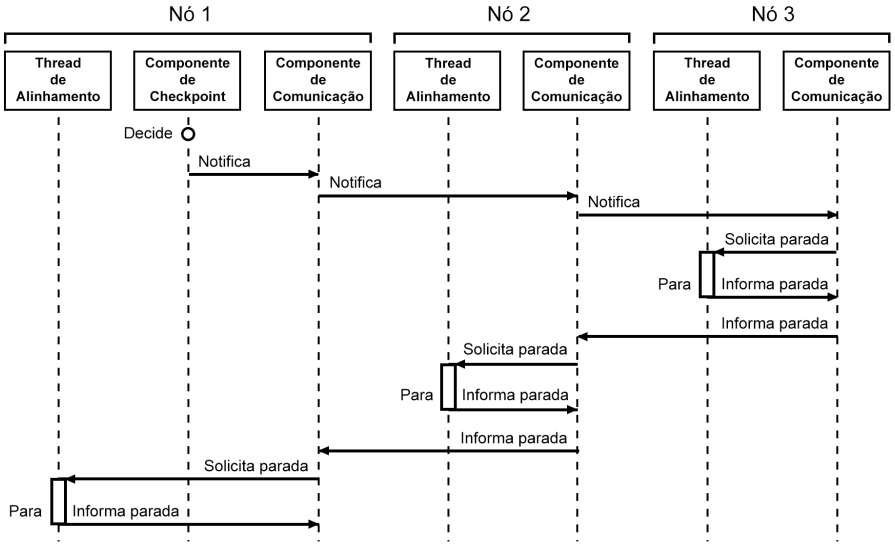

Figura 5.2: Diagrama de comunicação dos componentes, para a parada sincronizada.

daí os componentes de comunicações informam uns aos outros sequencialmente até que o componente no último nó seja informado.

Quando o componente de comunicação do último nó é informado sobre a parada, ele solicita à *thread* de alinhamento que está realizando o alinhamento das sequências que pare a execução. A *thread* de alinhamento, entre as iterações de alinhamento, verifica se existe alguma solicitação de parada, se existe, para e avisa o componente de comunicação. O componente de comunicação por sua vez informa o nó anterior que a execução no seu nó parou. E desta forma os nós param a execução do alinhamento de trás para frente e parando na última linha completa salva pelo nó final, diferentemente da proposta em [3] na qual os nodos param de frente para trás a partir da linha do primeiro nó.

A escolha de parar a execução na última linha inteira calculada pelo nó final se deu pelo fato de que, considerou-se que todos os nós utilizam o *AbstractDiagonalAligner* mencionado na seção 4.7, ou seja, o processamento ocorre em formato de *wavefront*. Por conta disso, não é trivial fazer todos os nós executarem até a última linha do primeiro nó, como mencionado na negociação de balanceamento na Seção 4.6.3, pois as células de uma determinada linha de um nó que não seja o primeiro dependem dos dados de uma diagonal externa mais adiante no nó anterior, como na ilustrado na Figura 5.3.

Portanto, ao invés de calcular até uma determinada linha e parar a execução, escolhemos por voltar a uma linha já calculada por todos os nós. Em ambos os casos ocorre o descarte de diversas células que foram calculadas.

Além disso, a escolha de realizar a parada dos nodos de trás para frente, também se deu devido a dependência de dados blocante que existe entre os nós. No momento em

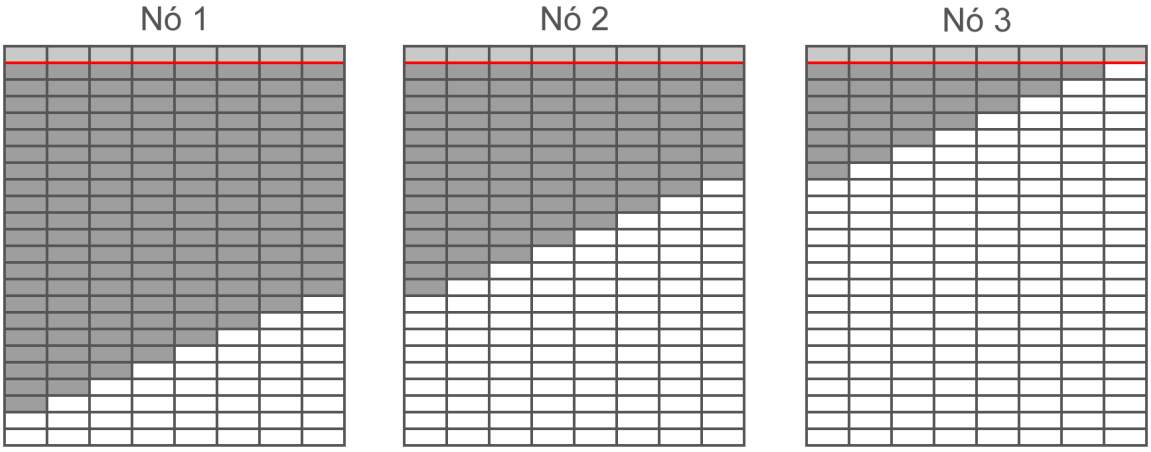

Figura 5.3: Exemplo de um alinhamento em três nós.
Blocos que devem ser recalculados e que o cálculo será descartados (cinza escuro), os blocos não calculados (branco), os blocos já calculados e que não serão descartados (cinza claro) e a linha em vermelho para qual a execução retornará na reinicialização.

que determinado nó, que não o primeiro, encontra seu *buffer* de entrada vazio, a *thread* de alinhamento é bloqueada até que novos dados cheguem ao *buffer*.

Tal comportamento pode levar a uma condição de *deadlock* caso se tente realizar a parada no sentido da execução do alinhamento, pois caso o primeiro nó pare a execução e o segundo esteja bloqueado aguardando o preenchimento do *buffer*, esse último jamais sairá da condição de bloqueio para executar a rotina de parada. A execução da parada de trás para frente evita esse problema, pois não permite a existência de tais dependências entre um nó que esteja parando e os que ainda executam. O único bloqueio que pode ocorrer utilizando o método proposto nesse Trabalho de Graduação é quando o *buffer* de saída de determinado nó se encher, porém esse pode ser resolvido de maneira mais simples, com o esvaziamento do *buffer* de entrada dos nós que se encontram parados.

A atual implementação da comunicação entre os nós, como citado, foi implementada sequencialmente. Isso se deu por que seguiu-se o padrão utilizado na estrutura de comunicação do MASA, na qual pressupõe-se que no ambiente não é possível realizar um *broadcast*.

A necessidade de escolher uma linha completa para a parada de execução vem do fato de que, na reinicialização da execução, os intervalos de colunas que cada um dos nós calcula pode ter sido modificado, devido o rebalanceamento. Nos nós em que a modificação ocorreu, o conjuntos de blocos já processados neste novo intervalo pode não seguir o formato de processamento em *wavefront*, o que impediria a realização da retomada do alinhamento neste formato. Por conta disso, é necessário ter uma linha inteira da matriz

de alinhamento salva, que servirá de ponto de reinicialização de todos os nós.

Outro possível problema existiria se o primeiro nó concluísse a execução antes de receber a mensagem de volta do último nó, gerando erro no algoritmo. Para efeitos desse projeto consideramos que isso não poderia acontecer, já que pode-se controlar quando executar o balanceamento, de forma que o mesmo não seja realizado caso o processamento da matriz já esteja no fim.

5.2 Visão detalhada do projeto

Foi implementada a classe *CheckpointManager* na linguagem de programação C++. Esta classe contém toda a lógica de execução mencionada na seção 5.1 e realiza a configuração inicial, onde os componentes comunicação estabelecem as suas conexões.

5.2.1 Inicialização do módulo

Foram adicionados dois parâmetros que são passados na inicialização do programa, que são *checkpoint-ip* e *checkpoint-port*. Para o primeiro, deve ser passado o endereço IP e a porta da máquina do nó anterior, na qual o componente de comunicação tentará se conectar. No segundo, deve ser passado o número da porta na qual o componente de comunicação neste nó esperará por uma conexão. O programa considera que deve-se realizar a inicialização do módulo de *checkpoint* se qualquer um desses dois parâmetros forem passados, e considera também que se apenas o *checkpoint-port* ou apenas o *checkpoint-ip* for passado, então o nó é o primeiro ou o último nó das conexões, respectivamente.

A inicialização do objeto *CheckpointManager* acontece junto à configuração das variáveis do Estágio 1. Nela são criados os semáforos e *mutexes*, que são utilizados na sincronização entre os componentes e a *thread* de alinhamento. Nesta inicialização, são criadas também as *threads* de comunicação e de *checkpoint*, que representam os componentes de comunicação e *checkpoint* da arquitetura mencionada na seção 5.1.

Para efeitos desse projeto, não foi feita a utilização da otimização *Block Pruning* descrita na subseção 4.3 pois o mesmo não funciona atualmente para múltiplas GPUs.

5.2.2 Thread de Checkpoint

O componente de *checkpoint* foi construído como um método da classe *CheckpointManager* e executa como uma *thread* a partir da *thread* de alinhamento. Assim, como citado na Seção 5.1.1, essa *thread* só é executada no primeiro nó.

Ao fim da etapa de inicialização, a *thread* de *checkpoint* é avisada por um semáforo para que inicie sua execução. Para efeito de testes, foi implementado um temporizador

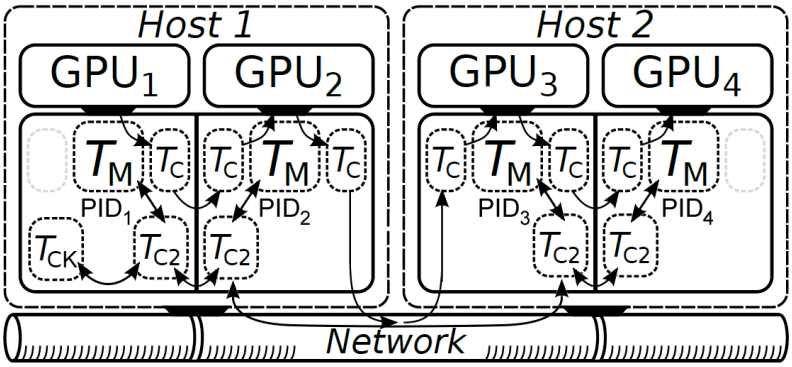

Figura 5.4: Figura da arquitetura MASA com a adição das *threads* de *checkpoint* (T_{CK}) e de comunicação (T_{C2}), Adaptado de [3].

que decorrido o tempo, avisa a *thread* de comunicação por meio de um semáforo para iniciar a execução da parada sincronizada.

Pautar a parada sincronizada em torno de um temporizador é uma escolha tomada unicamente para fins de teste. Espera-se que na implementação do balanceamento em si, os parâmetros para essa tomada de decisão advenham de informações obtidas nos diversos nós. Um exemplo de um protocolo para essa tomada de decisão se encontra descrito na subseção 4.6.3 baseada na negociação de balanceamento de [3].

5.2.3 Thread de Comunicação

Esta *thread* é responsável pela comunicação entre os nós como visto na Figura 5.4. Se inicialmente, foi passado como parâmetro o *checkpoint-ip*, então é realizada uma conexão por meio de *socket* com o *IP* e porta designados. Se foi passado como parâmetro o *checkpoint-port* então o nodo de comunicação aguarda por uma conexão na porta designada. O fluxo de conexões é realizado como pode ser visto na Figura 5.5.

A sequência de mensagens para a operação de parada de execução vistas no diagrama da figura 5.2 é feita em sua maioria por essa *thread*. Inicialmente, por meio de um semáforo, a *thread* de *checkpoint* libera a *thread* de comunicação no primeiro nó. Este, por sua vez, envia mensagens por meio do *socket* ao nó seguinte. Assim, os nós enviam sequencialmente mensagens aos próximos nós até que o último nó seja informado. No último nó a *thread* de comunicação avisa a *thread* alinhamento que ela deve parar a operação de alinhamento e isto é feito por meio de uma variável compartilhada que é protegida por um *mutex*. A *thread* de alinhamento confere o valor desta variável na execução de cada iteração. Quando a *thread* de alinhamento para, ela informa a *thread* de comunicação sobre a parada por meio de um semáforo.

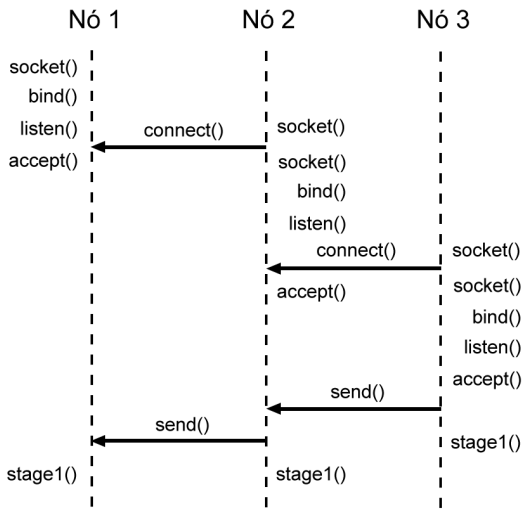

Figura 5.5: Configuração da conexão entre os nós feita pela thread de comunicação.

Assim, a *thread* de comunicação deste último nó escolhe qual linha será utilizada como ponto de retorno na inicialização. A linha escolhida é a mais recente salva no conjunto de linhas especiais descritas na Seção 4.2.1. Essas *linhas especias* são salvas de tempos em tempos em *RAM* e/ou disco, seguindo um intervalo que é calculado de acordo com o tamanho da *RAM* e/ou do disco (parâmetros *–ram* e *–disk* respectivamente), e podem, portanto, variar entre os nós. Para efeitos desse projeto, foi considerada a utilização apenas do salvamento de linhas em disco. Foi considerado também uma mesma quantidade de espaço em disco para todas as máquinas. Dessa forma, o intervalo de armazenamento das *linhas especiais* é o mesmo em todos os nós, podendo assim, serem utilizadas como linhas de *checkpoint* na reinicialização. O número da linha é enviada para o nodo anterior, que por sua vez realiza a processo para parada da *thread* de alinhamento. Isso se repete até todos os nós pararem, de trás para frente.

Capítulo 6

Resultados Experimentais

Foram realizados diversos testes com o objetivo de verificar se o mecanismo implementado neste projeto introduz um *overhead* significativo à execução da arquitetura MASA. Executamos o estágio 1 (Subseção 4.2.1) do algoritmo de alinhamento sob três pares de sequências (Tabela 6.3) utilizando os computadores do LAICO (LAboratório de sistemas Integrados e COncorrentes da Universidade de Brasília). Todas as sequências utilizadas para os testes foram obtidas do *National Center for Biotechnology Information* (NCBI). Na Seção 6.1 é descrito o ambiente de testes e na seção 6.2 os testes e seus resultados são apresentados.

6.1 Ambiente de testes

O ambiente utilizado para os testes foi composto por três máquinas do LAICO conectadas por uma rede Ethernet de 1 GBit/s, cada uma com uma GPU. O detalhamento da especificação de cada uma dessas máquinas se encontra na Tabela 6.1. Como pode ser observado, foi utilizado um ambiente heterogêneo.

	Nó 1 - 680	Nó 2 - 580	Nó 3 - 980
GPU	GeForce GTX 680 (Kepler)	GeForce GTX 580 (Fermi)	GeForce GTX 980 Ti (Maxwell)
CPU	Intel Core i7-3770 @ 3.40GHz	Intel Core i7-2600 @ 3.40GHz	Intel Core i7-3770 @ 3.40GHz
Sistema operacional	Ubuntu 12.04.2 LTS	Ubuntu 12.04.1 LTS	CentOS Linux 7.2.1511
Disco	909GB	923GB	923GB
Memória	8GB	8GB	8GB

Tabela 6.1: Especificação das máquinas do LAICO utilizadas nos testes.

Dada a heterogeneidade do ambiente, foi necessário que realizássemos um *profiling* para determinar a capacidade de processamento de cada máquina e, consequentemente, a distribuição de carga do alinhamento para cada uma.

Para tal, executamos o estágio 1 do algoritmo de alinhamento do MASA, sem a funcionalidade de parada, para as sequências biológicas de 5227K e 5229K descritas na Tabela 6.3.

O tempo que cada máquina levou para realizar o alinhamento dessas sequências nos levou à construção da Tabela 6.2, que possui o tempo de execução e a proporção devida a cada máquina.

	Tempo do Estágio 1	Proporção da matriz
Nó 1 - 680	646s	24.93%
Nó 2 - 580	760s	21.20%
Nó 3 - 980	299s	53.87%

Tabela 6.2: *Profiling* das máquinas para decisão da distribuição de cargas entre os nós.

6.2 Testes

Os testes foram realizados em dois conjuntos de máquinas, um utilizando os três nós da tabela 6.1 e outro apenas os nós 1 - 680 e 3 - 980. Foram utilizadas sequências de diferentes tamanhos, que variam de 5M até 24M (tabela 6.3).

Nome	Sequência 1	Organismo	Sequência 2	Organismo
5227K x 5229K	AE016879.1	*Bacillus anthracis str. Ames*	AE017225.1	*Bacillus anthracis str. Sterne*
10236K x 10236K	NC_014318.1	*Amycolatopsis mediterranei U32 chromosome*	NC_017186.1	*Amycolatopsis mediterranei S699*
23012K x 24544K	NT_033779.4	*Drosophila melanogaster chromosome 2L sequence*	NT_037436.3	*Drosophila melanogaster chromosome 3L sequence*

Tabela 6.3: Pares de sequências biológicas utilizadas nos testes.

Até o momento, o ambiente MASA não possui uma rotina para decidir pela realização ou não de um balanceamento, portanto, como critério para a realização da parada, utilizamos um temporizador. Dessa forma o mecanismo desenvolvido nesse projeto decide realizar a parada em 120 segundos após o início da execução para as sequências 5227K x 5229K e 10236K x 10236K, sendo definido para 180 segundos na execução do alinhamento das sequências 23012K x 24544K. A diferença do tempo de inicio da operação de parada se deu pelo fato de que 120 segundos é insuficiente para que pelo menos uma *linha especial* seja salva no alinhamento entre as sequências de 23012K x 24544K, e em 180 segundo o alinhamento das sequências de 5227K x 5229K já teriam finalizado.

Foram medidos os tempos de inicialização do módulo de *checkpoint* e de execução do algoritmo de parada. O primeiro consiste do tempo de criação dos componentes somado ao tempo que os componentes de comunicação levam para realizarem a conexão inicial através de *sockets*. O segundo consiste do tempo entre o início do mecanismo de parada até a parada de todos os nós.

Observando os valores da linha especial escolhida como ponto de retorno e o número da última diagonal externa calculada pelo MASA antes da parada, estimamos o percentual de blocos que precisariam ser recalculados em relação ao total de blocos da matriz de processamento, devido ao retorno do processamento à linha de *checkpoint*.

Os resultados se encontram presentes em duas tabelas (6.4 e 6.5) para a execução em 2 e 3 nós respectivamente. Cada um desses valores foi obtido a partir da média entre o encontrado em três execuções. Mais execuções não foram necessárias por se tratar de um ambiente controlado e dedicado, além de o desvio padrão (σ) encontrado para a maioria das entradas ter sido desprezível.

O único que variou significativamente entre os 3 experimentos foi o tempo de execução do algoritmo de parada. Isso provavelmente se deve ao fato de que, em dado nó, o inicio de execução da parada é condicionado à finalização da iteração corrente de alinhamento e essa iteração pode estar ou não próxima do seu fim quando a parada é solicitada ao nó. Além disso, os tempos estão na ordem de dezenas de milissegundos e muito provavelmente sofrem interferência de processos *daemons* do sistema operacional.

Foi feita também uma execução completa, sem realização de paradas, do alinhamento de cada um dos pares de sequências presentes na tabela 6.3. Essas execuções (tabela 6.6) foram feitas com o objetivo de mostrar o percentual de tempo gasto com o módulo de *Checkpoint* comparada com o tempo total de execução.

	5227K x 5229K (Média)	5227K x 5229K (σ)	10236K x 10236K (Média)	10236K x 10236K (σ)	23012K x 24544K (Média)	23012K x 24544K (σ)
Inicialização do Módulo	0,996s	0,00023	0,996s	0,00010	0,996s	0,00034
Execução do algoritmo	0,012s	0,003	0,039s	0,030	0,084s	0,038
Overhead total	1,008s	-	1,035s	-	1,08s	-
Blocos desperdiçados	4,54%	-	2,85%	-	1,94%	-

Tabela 6.4: Resultados da execução do alinhamento em 2 nós

	5227K x 5229K (Média)	5227K x 5229K (σ)	10236K x 10236K (Média)	10236K x 10236K (σ)	23012K x 24544K (Média)	23012K x 24544K (σ)
Inicialização do Módulo	1,995s	0,00079	1,995s	0,00096	1,994s	0,00080
Execução do algoritmo	0,027s	0,005	0,051s	0,009	0,164s	0,005
Overhead total	2,022s	-	2,046s	-	2,159s	-
Blocos desperdiçados	7,23%	-	3,87%	-	2,38%	-

Tabela 6.5: Resultados da execução do alinhamento em 3 nós

	5227K x 5229K	10236K x 10236K	23012K x 24544K
2 Nós (680 - 980)	204s	708s	4196s
3 Nós (580 - 680 - 980)	181s	634s	3360s

Tabela 6.6: Tempo de execução do estágio 1 para sequências de diferentes tamanhos, utilizando 2 e 3 nós.

6.3 Análise dos resultados

Como pode ser visto nas tabelas 6.4 e 6.5 o tempo de execução do algoritmo de parada cresce com o aumento do tamanho sequências a serem alinhadas. Isso ocorre por que a *thread* de alinhamento verifica entre cada iteração a necessidade de parar a execução ou não. O tempo de cada iteração aumenta com o tamanho das sequências, levando a um aumento do tempo entre as verificações de parada. Esse tempo também aumenta com o número de nós utilizados, o que se deve à necessidade da realização de um mecanismo de comunicação de parada com mais nós.

Observa-se também que o tempo de inicialização do módulo de *checkpoint* aumenta com o aumento do número de nós, pois durante a inicialização é realizada uma verificação síncrona que envolve uma troca de mensagens par a par entre todos os nós. Entre as diferentes sequências de entrada, porém, ele se mantém constante, pois o tamanho da sequência não interfere com a inicialização.

Com um número expressivo de nós talvez esse tempo possa vir a se tornar significativo. Para esse caso talvez seja interessante a realização do *setup* do módulo de *checkpoint* de forma assíncrona em relação à realização do alinhamento, já que espera-se que no início do mesmo a parada não seja solicitada.

Como visto nas tabelas 6.4 e 6.5, o percentual de blocos desperdiçados diminui com o aumento do tamanho das sequências de entrada. Esse comportamento é esperado e se deve ao fato de que, com o aumento do tamanho das sequências, o número de blocos que são desperdiçados devido ao formato de processamento em *wavefront* (Figura 5.3), representa um percentual menor do total de blocos. Com o aumento do número de nós, esse percentual tende a crescer, pois com o aumento do número de divisões em um *wavefront* desbalanceado, a linha escolhida (pelo último nó) se encontrará cada vez mais no começo do *wavefront*, gerando maior desperdício.

Comparando os resultados dos tempos apresentados nas tabelas 6.4 e 6.5 com a tabela 6.6 vemos que o módulo de *checkpoint* não introduz um *overhead* significativo na execução do MASA para as sequências comparadas. Nos nossos testes esse overhead representa no máximo 1.1% do tempo de execução total, sendo menos significativos no alinhamento de sequências maiores. Também em termos absolutos, o overhead adicionado é muito pequeno, ficando entre 1s e 2s (Tabelas 6.4 e 6.5), em execuções que demoram de 2min a 1h e 9 min.

Capítulo 7

Conclusão

Neste trabalho de graduação foi proposto, implementado e avaliado um mecanismo de parada sincronizada dos nós de processamento para a arquitetura MASA. Foi implementado o módulo de *checkpoint*, que é capaz de estabelecer a comunicação inicial entre os nós, realizar a operação de parada sincronizada e informar uma linha de *checkpoint* em comum a todos os nós.

Os resultados experimentais com até 3 GPUs heterogêneas mostraram que o *overhead* causado pela introdução do módulo de *checkpoint* não é significativo pois, para as sequências comparadas, o tempo de inicialização do módulo e a operação de parada sincronizada consistiu de no máximo 1.1% do tempo de execução total do estágio 1 do MASA-CUDAlign. Verificamos também que com o aumento do tamanho das sequências o *overhead*, apesar de aumentar linearmente, passa a representar um fração menor do tempo total de execução do estágio 1. O percentual de blocos que precisavam ser recalculados como consequência do retorno do processamento à linha de *checkpoint* também diminuía com o aumento das sequências.

A introdução de mais nós no alinhamento também aumentou linearmente o *overhead*, resultado esperado já que a operação de parada ocorre de maneira sincronizada e sequencial.

Como trabalho futuros sugerimos o projeto e implementação do balanceamento de carga, a ser realizado após a parada, e a adição de um algoritmo que avalie se deve ser executado ou não o balanceamento em determinado momento. O algoritmo de balanceamento e avaliação do momento de parada pode ser implementado de acordo com o sugerido em [3] e descrito na Seção 4.6.

Sugerimos também a implementação da reinicialização do processamento após a parada, avaliando a possibilidade do aproveitamento do valor de linhas especiais incompletas à frente da linha de *checkpoint* com o intuito de diminuir o *overhead* causado pelo descartes de linha após a parada.

Outra sugestão é avaliar a utilização de outras linhas como *checkpoint*, ou até mesmo utilizar linhas incompletas em conjunto com colunas como *checkpoint*, e comparar o *overhead* do tempo da decisão de balanceamento até a parada de todos os nós e o percentual de desperdício de linhas entre esses diferentes métodos.

Referências

[1] NVIDIA. Cuda C Programming Guide. *Programming Guides*, (September):1–261, 2015. ix, 15, 16, 17, 19

[2] NVIDIA. NVIDIA Tesla P100: The Most Advanced Datacenter Accelerator Ever Built, 2016. ix, 21, 23

[3] Edans Flavius De O Sandes. *Algoritmos Paralelos Exatos e Otimizações para Alinhamento de Sequências Biológicas Longas em Plataformas de Alto Desempenho*. PhD thesis, 2015. ix, 2, 24, 25, 28, 29, 30, 31, 32, 34, 35, 38, 39, 41, 42, 43, 46, 54

[4] Refseq growth statistics. `https://www.ncbi.nlm.nih.gov/refseq/statistics/`. 1

[5] David W. Mount. *Bioinformatics : sequence and genome analysis*. Cold Spring Harbor Laboratory Press, 2004. 1, 3, 4, 5, 11

[6] Saul B. Needleman and Christian D. Wunsch. A general method applicable to the search for similiarities in the amino acid sequence of two proteins. *Journal of molecular biology*, 48(3):443–453, 1970. 1, 6

[7] T. F. Smith and M. S. Waterman. Identification of common molecular subsequences. *Journal of Molecular Biology*, 147(1):195–197, 1981. 1, 2, 7

[8] W. R. Pearson and D. J. Lipman. Improved tools for biological sequence comparison. *Proceedings of the National Academy of Sciences*, 85(8):2444–2448, 1988. 1, 11

[9] S. F. Altschul, W. Gish, W. Miller, E. W. Myers, and D. J. Lipman. Altschul et al.. 1990. Basic Local Alignment Search Tool.pdf, 1990. 1, 12

[10] E.F.O. Sandes and A.C. de Melo. CUDAlign: using GPU to accelerate the comparison of megabase genomic sequences. *ACM SIGPLAN Notices*, 45(5):137–146, 2010. 2, 24, 27

[11] Osamu Gotoh. An improved algorithm for matching biological sequences. *Journal of Molecular Biology*, 162(3):705–708, 1982. 2, 8

[12] Bruce Alberts, Alexander Johnson, Julian Lewis, David Morgan, Martin Raff, Keith Roberts, and Peter Walter. *Molecular Biology of the Cell 6e*, volume 6. 2014. 3, 4

[13] Ce Charles E Leiserson, Rl Ronald L Rivest, Clifford Stein, and Thomas H Cormen. *Introduction to Algorithms, Third Edition*, volume 7. 2009. 5

[14] D. S. Hirschberg. A linear space algorithm for computing maximal common subsequences. *Communications of the ACM*, 18(6):341–343, 1975. 9

[15] Eugene W. Myers and Webb Miller. Optimal alignments in linear space. *Bioinformatics*, 4(1):11–17, 1988. 10

[16] Dennis A. Benson, Mark Cavanaugh, Karen Clark, Ilene Karsch-Mizrachi, David J. Lipman, James Ostell, and Eric W. Sayers. GenBank. *Nucleic Acids Research*, 45(D1):D37–D42, 2017. 11

[17] I O Bucak and V Uslan. An analysis of sequence alignment: heuristic algorithms. *Conf Proc IEEE Eng Med Biol Soc*, 2010:1824–1827, 2010. 11

[18] D. Lipman and W. Pearson. Rapid and sensitive protein similarity searches. *Science*, 227(4693):1435–1441, 1985. 11

[19] Stephen F. Altschul, Thomas L. Madden, Alejandro A. Schäffer, Jinghui Zhang, Zheng Zhang, Webb Miller, and David J. Lipman. Gapped BLAST and PSI-BLAST: A new generation of protein database search programs, 1997. 12

[20] U. Röhm and T.M. Diep. How to BLAST your Database—A Study of Stored Procedures for BLAST Searches. *Database Systems for Advanced Applications*, pages 807–816, 2006. 12

[21] Peter Pacheco. *An Introduction to Parallel Programming*. 2011. 14

[22] Nicholas Wilt. *The CUDA Handbook*. Number 1. 2013. 14, 15, 17, 18, 19, 21

[23] A. Driga, P. Lu, J. Schaeffer, D. Szafron, K. Charter, and I. Parsons. FastLSA: A fast, linear-space, parallel and sequential algorithm for sequence alignment. In *Proceedings of the International Conference on Parallel Processing*, volume 2003-January, pages 48–57, 2003. 28

[24] Edans Flavius and Alba Cristina M.A. De Melo. Retrieving smith-waterman alignments with optimizations for megabase biological sequences using GPU. *IEEE Transactions on Parallel and Distributed Systems*, 24(5):1009–1021, 2013. 30, 31

[25] Edans F.De O. Sandes, Guillermo Miranda, Alba C.M.A.De Melo, Xavier Martorell, and Eduard Ayguadé. CUDAlign 3.0: Parallel biological sequence comparison in large GPU clusters. In *Proceedings - 14th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing, CCGrid 2014*, pages 160–169, 2014. 31

[26] Edans Flavius De Oliveira Sandes, Guillermo Miranda, Xavier Martorell, Eduard Ayguade, George Teodoro, and Alba Cristina Magalhaes Melo. CUDAlign 4.0: Incremental Speculative Traceback for Exact Chromosome-Wide Alignment in GPU Clusters. *IEEE Transactions on Parallel and Distributed Systems*, 27(10):2838–2850, 2016. 33, 34

[27] Edans Flavius De O Sandes. Masa-core. `https://github.com/edanssandes/MASA-Core`. 40