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

Comparação Paralela de Sequências Biológicas em Múltiplas GPUs com Descarte de Blocos e Estratégias de Distribuição de Carga

Marco Antônio Caldas de Figueirêdo Júnior

Tese apresentada como requisito parcial para
conclusão do Doutorado em Informática

Orientadora
Profa. Dra. Alba Cristina M. A. de Melo

Brasília
2021

Universidade de Brasília — UnB
Instituto de Ciências Exatas
Departamento de Ciência da Computação
Doutorado

Coordenador: Profa. Dra. Genaína Nunes Rodrigues

Banca examinadora composta por:

Profa. Dra. Alba Cristina M. A. de Melo (Orientadora) — CIC/UnB
Prof. Dr. Philippe Olivier Alexandre Navaux — INF/UFRGS
Profa. Dra. Cristiana Barbosa Bentes — FEN/UERJ
Prof. Dr. Ricardo Pezzuol Jacobi — CIC/UnB

CIP — Catalogação Internacional na Publicação

Figueirêdo Júnior, Marco Antônio Caldas de.

Comparação Paralela de Sequências Biológicas em Múltiplas GPUs com Descarte de Blocos e Estratégias de Distribuição de Carga / Marco Antônio Caldas de Figueirêdo Júnior. Brasília : UnB, 2021.
172 p. : il. ; 29,5 cm.

Tese de Doutorado — Universidade de Brasília, Brasília, 2021.

1. Bioinformática, 2. GPU, 3. Descarte de Células, 4. Distribuição de Carga

CDU 004 Cutter F475c

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

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

Comparação Paralela de Sequências Biológicas em Múltiplas GPUs com Descarte de Blocos e Estratégias de Distribuição de Carga

Marco Antônio Caldas de Figueirêdo Júnior

Tese apresentada como requisito parcial para
conclusão do Doutorado em Informática

Profa. Dra. Alba Cristina M. A. de Melo (Orientadora)
CIC/UnB

Prof. Dr. Philippe Olivier Alexandre Navaux
INF/UFRGS

Profa. Dra. Cristiana Barbosa Bentes
FEN/UERJ

Prof. Dr. Ricardo Pezzuol Jacobi
CIC/UnB

Profa. Dra. Genaína Nunes Rodrigues
Coordenador do Programa de Pós-graduação em Informática

Brasília, 05 de março de 2021

Agradecimentos

Agradeço a Deus, inicialmente, por me conceder saúde e perseverança para concluir mais esta etapa de minha vida acadêmica.

Obrigado a minha esposa, Scheila Figueirêdo, pelo apoio incessante e amor incondicional que tanto me fortaleceram ao longo desta jornada: você me faz querer crescer a cada dia e é um privilégio tê-la sempre ao meu lado. Também a meus filhos, Gabriela, Lucca e Liz, que tiveram paciência para entender que o papai nem sempre estava disponível, mas que meu amor sempre estará.

A meus pais, irmãs e demais familiares, por servir de exemplo de caráter e retidão, incentivar e dar o alicerce que permite meu crescimento. Aos meus amigos, pelas palavras de apoio e incentivo, que certamente não me permitiram pensar em outro desfecho que não fosse o do trabalho finalizado. Um agradecimento especial à Profa. Dra. Alba Melo, pela orientação irreparável e oportunidades propiciadas ao longo deste caminho. Este trabalho só foi possível graças à sua dedicação e competência - me orgulho de ser seu aluno.

Algumas pessoas foram fundamentais para o sucesso desta Tese, merecendo um especial agradecimento: (a) Prof. Dr. Denis Trystram, por me receber no IMAG (Université Grenoble Alpes - França), em 2018, e pelas discussões sobre os grafos de tarefas da comparação de sequências e o balanceamento destes grafos; (b) Prof. Dr. George Teodoro (UFMG/Brasil), por prover o acesso às máquinas Comet e Bridges, via projeto na plataforma XSEDE; (c) Prof. Dr. Ganesan Narayanasamy (IBM/USA) e Prof. Dr. Samir Shende (University of Oregon/USA), por proverem acesso aos *clusters* de GPUs da University of Oregon; e (d) João Paulo Navarro (NVidia/Brasil), por prover acesso ao *cluster* de 512 V100 da NVidia.

Agradeço a todos os professores do Programa de Pós-Graduação em Informática da Universidade de Brasília pelos ensinamentos e trabalhos colaborativos, bem como aos colegas de pesquisa, em especial ao Dr. Edans Sandes por compartilhar seus conhecimentos e ajudar a todo o momento. Enfim, agradeço a todos que contribuíram de alguma forma para a conclusão deste trabalho.

Resumo

A comparação de sequências biológicas é um problema bastante importante em Bioinformática. A utilização de métodos exatos para a solução deste problema requer um alto poder computacional, sobretudo quando duas sequências biológicas longas são comparadas. Para acelerar a obtenção de resultados, diversas estratégias têm sido propostas na literatura, envolvendo o processamento paralelo usando diversos dispositivos ou modificações nos algoritmos. Uma técnica de modificação do algoritmo aplicada com certa frequência em comparações que usam um único dispositivo é a poda da matriz de programação dinâmica, que reduz bastante o espaço de computação quando as sequências comparadas são similares. Ao nosso conhecimento, até o desenvolvimento da presente Tese, só havia uma solução que aplicava técnica de poda para múltiplos dispositivos. No entanto, essa solução usava somente duas GPUs (*Graphics Processing Units*), com distribuição estática básica de carga. O principal objetivo da presente Tese de Doutorado é investigar o uso de processamento paralelo em conjunto com estratégia de poda, visando acelerar a execução de soluções de comparação de sequências biológicas longas. Para atingir esse objetivo, inicialmente fizemos o estudo estatístico de duas soluções em uma única GPU com descarte de blocos, para estimar o tempo de execução das comparações. A seguir, avaliamos soluções em ambientes heterogêneos e híbridos com descarte de blocos para determinar o impacto da poda no balanceamento da execução. Com base nesses estudos, propusemos uma estratégia leve de distribuição estática de carga (Static-MultiBP), que aplica o descarte de blocos em conjunto com a comunicação periódica do melhor escore entre as GPUs. Para ambientes heterogêneos de GPUs e para comparações entre sequências bastante similares, propusemos uma estratégia de distribuição dinâmica de carga (Dynamic-MultiBP), que divide a computação da matriz em diversas partes (ciclos) e, ao final da computação de cada ciclo, reavalia a distribuição de carga e a modifica, caso não esteja apropriada. Por fim, propusemos um módulo flexível de decisão que pode ser usado para decidir entre as duas estratégias. O Static-MultiBP, o Dynamic-MultiBP e o módulo de decisão foram integrados em um único *framework*, chamado MultiBP. O MultiBP foi integrado à ferramenta MASA-CUDAlign, que representa o estado da arte em ferramentas de comparação de sequências em GPU. As estratégias MultiBP foram executadas em diversos ambientes, sendo obtidos ganhos expressivos em comparação ao MASA-CUDAlign original. Finalmente, executamos o Static-MultiBP em um grande *cluster* de GPUs NVidia Volta (512 V100), obtendo o impressionante desempenho de 82,82 TCUPS (*Trillions of matrix Cells Updated Per Second*). A nosso conhecimento, este é o melhor desempenho na literatura de ferramentas de comparação de sequências em GPU que usam o algoritmo Smith-Waterman e suas variantes, e o melhor desempenho entre ferramentas que comparam sequências longas (em qualquer dispositivo).

Palavras-chave: Bioinformática, GPU, Descarte de Células, Distribuição de Carga

Abstract

Biological sequence alignment is a very important problem in Bioinformatics. The use of exact methods to solve this problem requires high computational power, especially if two long biological sequences are compared. To speed up results, several strategies have been proposed in the literature, involving the parallel processing using several devices or algorithm modifications. A technique frequently applied in the algorithms used to compare sequences in one device is the pruning of the dynamic programming matrix, which reduces the computation space when the compared sequences are similar. To our knowledge, until the development of this Thesis, there was only solution that used the pruning technique in multiple devices. However, this solution used only two GPUs (Graphics Processing Units), with basic static load distribution. The main objective of this PhD Thesis is to investigate the use of parallel processing in conjunction with pruning strategies, aiming to accelerate the execution of long biological sequences comparison. To achieve this objective, initially we did a statistical study of two solutions in a single GPU with block pruning, to estimate the execution time of the comparisons. Next, we evaluated solutions on heterogeneous and hybrid environments with block pruning to investigate the impact of pruning on the execution balance. Based on these studies, we proposed a lightweight static load distribution strategy (Static-MultiBP), which applies block pruning together with the periodic communication of the best score among the GPUs. For heterogeneous GPU environments and for comparisons between very similar sequences, we proposed a dynamic load distribution strategy (Dynamic-MultiBP), which divides the matrix computation into several parts (cycles) and, at the end of the computation of each cycle, reevaluates the load distribution and modifies it, if not appropriate. Finally, we proposed a flexible decision module that can be used to decide which strategy to use. Static-MultiBP, Dynamic-MultiBP and the decision module were integrated into a single framework, called MultiBP. MultiBP was integrated with the MASA-CUDAlign tool, which represents the state-of-the-art in GPU sequence comparison tools. MultiBP strategies were executed in several environments, and significant gains were obtained in comparison to the original MASA-CUDAlign. Finally, we executed Static-MultiBP on a large cluster of NVidia Volta GPUs (512 V100), achieving an impressive performance of 82.82 TCUPS (Trillions of matrix Cells Updated Per Second). To our knowledge, this is the best performance in the literature for sequence comparison tools that use Smith-Waterman algorithm and its variants in GPU, and the best performance obtained by tools that compare long sequences (on any device).

Keywords: Bioinformatics, GPU, Pruning, Workload Distribution

Sumário

Lista de Figuras

Lista de Tabelas

Lista de Abreviaturas e Siglas

ANOVA Analisys of Variance. 78

API Application Programming Interface. 34

AVX Advanced Vector Extensions. 67

BP Block Pruning. 4, 20, 54, 82, 95, 112, 138

BW Bandwidth. 81

CGBS Current Global Best Score. 20, 96, 125

CI Confidence Interval. 77, 106, 126

CLBS Current Local Best Score. 96

CP Computing Power. 81

CPU Central Processing Unit. 3, 29, 48, 86, 96, 124, 139

CUDA Compute Unified Device Architecture. 33, 49, 74, 95, 123

CUPS Cells Updated per Second. 28

DP Dynamic Programming. 3, 12, 81, 87, 96, 113, 139

EFT Earliest-Finish-Time. 44

FCFS First Come First Served. 44

FPGA Field Programmable Gate Array. 3, 29, 45, 48

GBP Giga Base Pairs. 12, 61

GCUPS GigaCells Updated per Second. 3, 28, 48, 79, 89, 103, 123, 139

GFLOPS Giga Floating-point Operations Per Second. 72, 121

GPC Graphics Processing Cluster. 31

GPGPU General Purpose Graphics Processing Unit. 29

GPU Graphics Processing Unit. 3, 20, 29, 45, 48, 76, 86, 95, 112, 138

HEFT Heterogeneous Earliest-Finish-Time. 44

IA Inteligência Artificial. 30

IST Incremental Speculative Traceback. 63

KBP Kilo Base Pairs. 12, 71, 79

Laico Laboratório de Sistemas Integrados e Concorrentes. 88, 102

LCS Longest Common Subsequence. 17

LJF Largest-Job-First. 44

MASA Multi-Platform Architecture for Sequence Aligners. 7, 21, 55, 76, 88, 139

MBP Mega Base Pairs. 12, 51, 79, 88, 103, 116, 139

MCUPS MegaCells Updated per Second. 28

MM Myers-Miller. 17, 53

MPI Message Passing Interface. 33, 61

NCBI National Center for Biotechnology Information. 88, 103, 123

NW Needleman-Wunsch. 13

OpenCL Open Computing Language. 33, 50, 95

OpenMP Open Multi-Processing. 33, 73

PTX Parallel Thread Execution. 34, 58

PU Processing Unit. 87

ReCAM Resistive Content Addressable Memories. 64

SFU Special Function Units. 30

SGE Sun Grid Engine. 70

SIMD Single Instruction Multiple Data. 66

SIMT Single Instruction Multiple Threads. 34

SJF Shortest-Job-First. 44

SLURM Simple Linux Utility for Resource Management. 8, 133

SM Streaming Multiprocessor. 30

SP Streaming Processor. 30, 66

SSE Streaming SIMD Extensions. 60

SW Smith-Waterman. 1, 14, 48, 135, 138

TCUPS TeraCells Updated per Second. 28, 68

TFLOPS Tera Floating-point Operations Per Second. 30

TPC Texture Processing Cluster. 30

UnB Universidade de Brasília. 88

VPU Vector Processing Unit. 73

Capítulo 1

Introdução

A área da Bioinformática contempla o conjunto de técnicas computacionais utilizadas para avaliar dados biológicos [65]. A gama de tarefas realizadas para inferir conhecimento a partir de dados biológicos inclui, usualmente, a organização, o tratamento e o processamento de dados genômicos [73], que se traduzem em um volume de dados extremamente grande e ainda crescente. A aplicação destas técnicas é encontrada em áreas tão diversas quanto patologia genética, farmacologia ou agricultura [30].

Dentre as operações amplamente utilizadas no âmbito da Bioinformática, a comparação de sequências biológicas permite identificar o grau de similaridade entre sequências de proteínas, DNA ou RNA, sendo de muita relevância na análise de organismos por biólogos [65]. Soluções deste tipo podem retornar apenas um escore ótimo, que representa a similaridade, ou também o alinhamento gerado. O alinhamento de pares de sequências de DNA permite, por exemplo, identificar áreas de cromossomos onde houve deleções ou mutações, e correlacionar estas alterações com doenças. É usual, também, comparar sequências de dois organismos com algum tipo de correlação genética para identificar similaridades, e assim desenvolver tratamentos mais eficientes ou vacinas.

A comparação de cromossomos inteiros pode auxiliar a esclarecer aspectos relacionados à evolução das espécies, incluindo a humana. A comparação de sequências muito longas é particularmente desafiadora, e o uso de ferramentas que usam algoritmos exatos para este tipo de trabalho foi considerado por muito tempo inviável. Em [68], um estudo comparativo detalhado foi realizado entre cromossomos homólogos do homem e de outros mamíferos. Comparações aos pares foram feitas usando BLASTZ [110], que é um método heurístico que usa o algoritmo exato Smith-Waterman (SW) em algumas de suas etapas, uma vez que usar o algoritmo em toda a comparação foi considerado intratável pelos autores [68]. Em [55], uma abordagem heurística independente de alinhamento (*alignment-free*) é usada para comparar cromossomos de várias espécies (homem, chimpanzé, vaca, cachorro, etc.). Os autores citam que uma abordagem usando este tipo de ferramenta foi

usada porque não era viável obter alinhamentos utilizando o algoritmo exato SW, embora eles reconhecessem que teriam alinhamentos mais precisos se utilizassem o algoritmo SW em todas as etapas.

Diante deste cenário, esta Tese propõe otimizações para a comparação de sequências longas de DNA em plataformas com alto poder computacional, visando diminuir o tempo de execução deste tipo de solução em ambientes com múltiplos dispositivos. As seguintes seções estão contempladas neste capítulo introdutório. Na Seção 1.1, apresenta-se o problema abordado na Tese: comparação de sequências biológicas. Já na Seção 1.2, a motivação deste trabalho é descrita. A seguir, as Seções 1.3 e 1.4 listam os objetivos e contribuições, respectivamente. Por fim, a Seção 1.5 descreve os demais capítulos desta Tese.

1.1 Problema: Comparação de Sequências Biológicas

A representação das sequências biológicas (DNA, RNA ou proteínas) é feita como uma cadeia de caracteres. No caso das sequências de DNA, o alfabeto é composto por $\Sigma = \{A, T, C, G, N\}$, onde A, T, C e G são Adenina, Timina, Citosina e Guanina, respectivamente, e N (aNy) é usado para representar um nucleotídeo não resolvido, que pode ser qualquer um dos outros quatro [73]. A comparação de sequências biológicas produz um escore, que representa a similaridade entre as sequências, e um alinhamento, que mostra claramente as regiões com caracteres coincidentes ou distintos. Geralmente é utilizado um caractere adicional (como "$-$", por exemplo) para representar situações de espaços (*gaps*) em uma das sequências. Na comparação de sequências, valores são atribuídos para ponderar as situações em que ocorrem coincidência (*match*) ou divergência (*mismatch*) de caracteres, além do cenário de *gap*. A soma dos valores referentes a cada posição respectiva nas duas sequências produz um escore final, de tal forma que, quanto maior o escore, maior a similaridade entre as sequências.

A Figura 1.1 representa um alinhamento entre duas sequências pequenas de DNA (S_0 e S_1, contendo apenas oito nucleotídeos cada), com pontuação $+1$ para *match*, -1 para *mismatch* e -2 para *gaps*. Nesse alinhamento, há ocorrências de *gaps* nas duas sequências, e o escore final (E) calculado é igual a $+1$.

Os métodos utilizados nas soluções de comparação de sequências biológicas podem ser classificados quanto à qualidade da solução como: heurísticos, que utilizam uma abordagem aproximada para produzir resultados mais rapidamente, mas sem a garantia do resultado ótimo; ou exatos, que produzem a solução ótima [73]. Alguns algoritmos exatos propostos na literatura são Needleman-Wunsch [78], Smith-Waterman [113], Gotoh [34] e Myers-Miller [77]. Apesar de produzirem o resultado ótimo, esses algoritmos possuem

```
S0:  T  A  C  G  C  -  A  C  A
     |  |     |  |     |     |
S1:  T  A  -  G  C  T  A  T  A
-------------------------------
E:  +1 +1 -2 +1 +1 -2 +1 -1 +1 = +1
```

Figura 1.1: Exemplo de alinhamento onde o escore final obtido é +1.

complexidade quadrática de tempo, o que representa um desafio se as sequências comparadas são longas, pois o tempo necessário para se obter o alinhamento pode se tornar proibitivo.

1.2 Motivação

Para realizar a busca de uma solução ótima, os algoritmos exatos citados na Seção 1.1 processam uma matriz de programação dinâmica (DP - *Dynamic Programming*), onde o cálculo da pontuação de similaridade em uma determinada célula depende de outros três valores de células vizinhas, resultando na resolução de uma equação de recorrência. O processamento desta matriz pode possuir alto custo computacional, principalmente se duas sequências longas forem comparadas. Nesse contexto, as estratégias paralelas propostas para calcular a matriz DP devem respeitar a dependência de dados entre as células. Além das tradicionais CPUs (*Central Processing Units*), outras plataformas de *hardware* vêm sendo utilizadas, tais como os *Field Programmable Gate Arrays* (FPGAs), placas aceleradoras como o Intel Xeon Phi e as unidades de processamento gráfico (GPUs - *Graphics Processing Units*). O desempenho destas soluções é usualmente medido em bilhões de células da matriz DP que são atualizadas por segundo (GCUPS - *GigaCells Updated per Second*).

As primeiras abordagens de implementação paralela dos algoritmos exatos de comparação de sequências possuíam como plataformas dispositivos únicos, sejam em CPU [31] [89] [5] [94] [20], em FPGA [40] [86] [128] [10] [97], Intel Xeon Phi [120] ou GPU [57] [58] [103] [51] [47] [91]. Visando propor uma solução que realizasse a comparação de sequências biológicas longas utilizando uma linguagem para ambientes heterogêneos, o autor desta Tese propôs e avaliou durante a Dissertação de Mestrado o MASA-OpenCL [25], cujos resultados em uma única GPU foram comparáveis e mesmo superiores a outras soluções baseadas em linguagem voltada apenas para GPUs fabricadas pela NVidia.

Mais recentemente, os trabalhos que tratam a comparação de sequências têm sido propostos considerando ambientes com múltiplos dispositivos, sejam eles do mesmo tipo

(ambiente uniforme) [93] [66] [123] [124] [59] [61] [67] [107] [104] [41] [75] ou de tipos diferentes (ambiente híbrido) [71] [112] [70] [62] [95] [96] [53] [98] [52]. Os melhores desempenhos da literatura em execuções reais em ambientes uniformes são os seguintes: (a) MASA-CUDAlign [104], que obteve 10.370 GCUPS com 384 GPUs; (b) SWRyviera [123], que obteve 6.020 GCUPS com 128 FPGAs; e (c) SW-MVM [66], com 900 GCUPS em uma plataforma com 128 CPUs. Como pode ser visto, o melhor desempenho foi obtido pelo MASA-CUDAlign [104] na execução em múltiplas GPUs: 10.370 GCUPS. Apesar de incorporar diversas otimizações, a solução em múltiplas GPUs requer que todas as células da matriz DP sejam calculadas.

O processamento em múltiplos dispositivos necessita adotar estratégias de distribuição de carga de trabalho entre os dispositivos [35] [12]. Os trabalhos encontrados na literatura vêm adotando principalmente a distribuição estática de carga de trabalho, baseando-se no poder computacional de cada dispositivo. As poucas iniciativas com distribuição dinâmica de carga se baseiam na comparação de sequências pequenas em um *grid* de CPUs [13] ou em ambiente híbrido contendo CPU e Intel Xeon Phi [96]. A nosso conhecimento, a única solução que prevê distribuição dinâmica de carga na comparação de sequências longas [108] utiliza um ambiente multi-GPUs, e foi avaliada apenas em simulação. Desta forma, a distribuição apropriada de carga na comparação de sequências biológicas longas em ambiente uniforme ou híbrido é, no momento do desenvolvimento desta Tese, um problema em aberto.

Alternativamente, técnicas de descarte de dados podem ser utilizadas, visando evitar o processamento de células da matriz DP que não podem produzir um resultado melhor que o atual, e assim reduzir o tempo de processamento. A abordagem proposta por Fickett [24] utiliza como princípio o fato de que o alinhamento de sequências muito similares se concentra nas vizinhanças da diagonal principal da matriz, e assim uma faixa de células localizadas acima e abaixo da diagonal é definida para buscar o alinhamento. O tamanho da faixa pode ser ampliado caso um ponto pertencente ao alinhamento esteja fora do limite definido. As células fora da faixa são então descartadas (*pruned*). A complexidade de tempo e espaço do algoritmo de Fickett é da ordem de $O(kn)$, onde k é o tamanho da faixa de cálculo e n é o tamanho da menor sequência. Já o LBD-Align [17] processa as células da matriz DP em diagonais, e em cada uma das diagonais um teste é realizado comparando os escores em cada célula com limites definidos para avaliar a oportunidade de descarte. Finalmente, o *Block Pruning* (BP) [102] expande este conceito para diagonais de blocos, e um cálculo do valor máximo a ser obtido por um bloco (com base nas suas coordenadas e no escore da primeira célula de cada bloco) é comparado com o melhor escore atual para decidir se o bloco pode ser totalmente descartado. A técnica foi inicialmente proposta processando a matriz de forma diagonal, e depois estendida para uma

abordagem genérica [105], mas em ambos os casos é aplicável para apenas um dispositivo. Em [51], a mesma técnica chegou a ser aplicada para dois dispositivos (no caso, GPUs). Em todas as abordagens citadas, a quantidade de células descartadas será maior caso as sequências sejam mais similares. Portanto, da mesma maneira que a distribuição apropriada da carga de trabalho, a aplicação de técnicas de descarte de blocos para comparações de sequências longas que envolvem mais de dois dispositivos também é um problema em aberto no momento do desenvolvimento desta Tese.

A principal motivação desta Tese é, portanto, avançar o estado da arte relativo aos problemas de distribuição de carga e descarte de blocos em comparações de sequências longas em ambientes multi-GPU, com algoritmos exatos tais como o algoritmo Smith-Waterman e suas variantes, obtendo o resultado ótimo. Como resultado de nossa investigação, visamos obter desempenho superior ao melhor desempenho atual da literatura. O foco principal é a distribuição apropriada de carga de trabalho em ambiente multi-GPU considerando o descarte de blocos, visto que não foram identificados na literatura trabalhos que utilizem estas abordagens concomitantemente. Consideramos a incorporação de técnicas que tratam esses problemas fundamental para a redução significativa do tempo de execução das aplicações. O ganho de desempenho esperado com as otimizações propostas na presente Tese será útil para: (a) comparar sequências de tamanho muito grande, tais como cromossomos completos, em tempo reduzido, caso uma plataforma de GPU em larga escala esteja disponível; e (b) comparar sequências de tamanho razoável em ambientes que possuam um conjunto de GPUs heterogêneas. Como consequência da distribuição adequada de trabalho e ganhos de desempenho gerados pelo *pruning*, o uso de GPUs pode ser otimizado em vários tipos de ambientes, o que pode ser importante em situações em que há limitação de tempo no uso dos recursos.

1.3 Objetivos

A presente Tese tem por objetivo principal propor e avaliar estratégias que permitam realizar a comparação de sequências longas de DNA com algoritmos exatos em múltiplas GPUs em tempos reduzidos através do uso de técnicas de descarte de blocos e de distribuição apropriada de carga. As estratégias e técnicas propostas devem ser leves, de forma que as modificações propostas não gerem impacto significativo na comparação de sequências pequenas ou pouco similares (onde há baixa taxa de descarte de blocos), mantendo um desempenho comparável ao de ferramentas do estado da arte (por exemplo, o MASA-CUDAlign 4.0), ou seja, as soluções devem apresentar baixo *overhead*. Nos casos em que o descarte de blocos seja mais relevante, visamos ultrapassar o desempenho da ferramenta mais rápida com GPUs no estado da arte (MASA-CUDAlign 4.0, com 10.370

GCUPS). Além disso, visamos propor as estratégias adequadas a ambiente homogêneo ou heterogêneo de GPUs.

Abaixo, seguem listados os objetivos específicos da Tese:

- Avaliar métodos estatísticos que permitam verificar ganhos de desempenho e estimar tempos de execução de ferramentas de comparação de sequências em GPU, considerando um determinado nível de confiança;
- Avaliar o comportamento de soluções de comparação de sequências em ambientes onde os dispositivos não possuem capacidade de processamento semelhante;
- Propor, projetar e avaliar uma estratégia eficiente de distribuição estática de carga em ambientes multi-GPU com descarte de blocos;
- Propor, projetar e avaliar uma arquitetura de *software* (*framework*) que permita a comparação de sequências longas de DNA em múltiplas GPUs do mesmo modelo ou de modelos diferentes com técnicas de descarte de blocos, e que ofereça estratégias estática e dinâmica de distribuição de carga de trabalho, com um módulo de decisão que permita a escolha entre os dois tipos de estratégia.

1.4 Contribuições

Como principal contribuição desta Tese, a arquitetura de *software* MultiBP para comparação de sequências foi projetada e integrada à ferramenta MASA-CUDAlign 4.0, permitindo a seleção de duas estratégias de execução (estática ou dinâmica) em múltiplas GPUs, com descarte de blocos. O desenvolvimento da solução passou por duas fases, a saber:

- Static-MultiBP: solução capaz de realizar a comparação em múltiplas GPUs com descarte de blocos, distribuição estática de carga e compartilhamento de melhor escore entre as GPUs. Com o Static-MultiBP, atingiu-se 694,8 GCUPS, utilizando-se 4 GPUs. O projeto e avaliação do Static-MultiBP foram publicados em [28];
- *Framework* MultiBP: arquitetura de *software* capaz de realizar a comparação em múltiplas GPUs com descarte de blocos e distribuição de carga de trabalho em ambientes homogêneos e heterogêneos. Além da estratégia Static-MultiBP [28], o *framework* MultiBP oferece também uma estratégia dinâmica de carga de trabalho (Dynamic-MultiBP). Nesta estratégia, o processamento da matriz de programação dinâmica é dividido em partes (ciclos), e o desempenho da execução é avaliado após

cada ciclo para sugerir uma nova redistribuição de carga entre as GPUs. Adicionalmente, um módulo de decisão que sugere uma das estratégias (estática ou dinâmica) foi incorporado para criar um *framework* flexível e aderente a várias arquiteturas de GPUs da NVidia. Dependendo da quantidade/modelo das GPUs e do tamanho/similaridade das sequências, os melhores resultados podem ser obtidos com a estratégia estática ou dinâmica. Nos testes com o MultiBP, a estratégia Dynamic-MultiBP foi testada em ambientes de até 8 GPUs, obtendo-se 2.410,6 GCUPS, enquanto a estratégia Static-MultiBP foi executada em ambientes de até 512 GPUs, com um máximo de 82.822,8 GCUPS, sendo agora, até onde temos conhecimento, o melhor resultado de uma solução de comparação de sequências em GPU. O projeto e avaliação do *framework* MultiBP e da estratégia dinâmica de distribuição de carga estão descritos no artigo [26], submetido a um prestigioso periódico e atualmente em processo de avaliação.

Outrossim, contribuições adicionais são apresentadas nesta Tese:

- Avaliação quantitativa: técnicas estatísticas foram avaliadas para validar os ganhos de desempenho da solução MASA-OpenCL [25], ferramenta desenvolvida pelo autor desta Tese durante o Mestrado, com objetivo final de indicar uma equação de regressão linear que permite predizer com boa aproximação o tempo de execução de uma dada comparação em uma determinada GPU, com descarte de blocos. A descrição da avaliação estatística proposta neste Tese foi publicada em [27];
- Análise empírica de extensões *Multi-Platform Architecture for Sequence Aligners* (MASA): testes com a ferramenta MASA [105] para execução em ambiente com dispositivos de capacidade de processamento díspares (2 CPUs e 2 GPUs) foram realizados para identificar o impacto de cargas de trabalho desbalanceadas no desempenho.

1.5 Sumário da Tese

A presente Tese está organizada em três partes. A primeira parte (*Background*/Contextualização) apresenta o embasamento conceitual que serviu como referência para o desenvolvimento da Tese, e está dividida em quatro capítulos. O Capítulo 2 detalha alguns conceitos inerentes ao problema da comparação de sequências biológicas, além dos algoritmos exatos utilizados para realizar esta tarefa. Já o Capítulo 3 mostra aspectos relevantes da plataforma GPU, além de ambientes de programação para desenvolvimento de soluções paralelas. Em seguida, definições e conceitos relacionados ao escalonamento

de tarefas, tipo de aplicações paralelas e distribuição de carga são objetos de estudo do Capítulo 4. Finalmente, o Capítulo 5 discute e compara soluções do estado da arte que tratam o problema em máquina única com o uso de GPUs, bem como soluções para múltiplos dispositivos em ambientes uniforme e híbrido, além de apresentar abordagens para distribuição de carga de trabalho em soluções de comparação de sequências biológicas.

A segunda parte (Contribuições) discute as contribuições da Tese. Esta parte encontra-se dividida em quatro capítulos. O Capítulo 6 aborda a utilização de técnicas estatísticas para validação e predição de desempenho de soluções de comparação de sequências em GPU, com descarte de blocos. O Capítulo 7 apresenta análises teóricas e experimentais relacionadas ao desempenho das extensões MASA em ambiente híbrido (CPU e GPU). A estratégia estática de distribuição de carga de trabalho e o *framework* MutiBP (que inclui a estratégia dinâmica de distribuição e um módulo de decisão) proposto para a comparação de sequências longas de DNA em múltiplas GPUs com descarte de blocos são apresentados nos Capítulos 8 e 9, respectivamente.

Finalmente, o Capítulo 10 concentra a conclusão e sugestões de trabalhos futuros, compondo a terceira e última parte desta Tese (Conclusão). Após este capítulo, são apresentados as referências bibliográficas e dois anexos. O Anexo I mostra o repositório criado para o projeto, que possui o código-fonte das soluções e os *scripts* desenvolvidos para execução da solução MultiBP em um ambiente com escalonador *Simple Linux Utility for Resource Management* (SLURM) [126]. Já o Anexo II descreve a produção científica advinda desta Tese, sendo: um artigo completo publicado em periódico internacional [27], um artigo completo publicado em conferência internacional [28] e um artigo completo submetido a periódico internacional [26] e que se encontra em processo de revisão.

Parte I

Background/Contextualização

Capítulo 2

Comparação de Sequências Biológicas

A comparação de sequências biológicas é uma das atividades mais relevantes e corriqueiras no âmbito da Biologia Molecular, que consiste em avaliar duas sequências de DNA, RNA ou de proteínas buscando a similaridade entre elas [7]. O problema da comparação de sequências biológicas pertence ao escopo da Bioinformática, que prevê o uso de técnicas computacionais para entender e organizar a informação associada a dados biológicos [65]. A busca de formas de otimizar a execução desta tarefa através de recursos computacionais e/ou algoritmos paralelos é fundamental para obter-se resultados em tempo adequado, sobretudo quando sequências longas são analisadas.

No presente capítulo, inicialmente são apresentados na Seção 2.1 os conceitos básicos de comparação de pares de sequências. A seguir, alguns dos principais algoritmos exatos para o problema da comparação de sequências são mostrados (Seção 2.2), e discutidas técnicas de poda que podem ser utilizadas neste problema (Seção 2.3). Finalmente, alguns aspectos relativos a estratégias de paralelização dos algoritmos são mencionados na Seção 2.4.

2.1 Conceitos Básicos

A comparação de sequências biológicas é uma das operações mais básicas e importantes em Biologia Molecular. Inicialmente, cabe destacar que essa operação consiste em comparar duas sequências ordenadas representando nucleotídeos (no caso do DNA e RNA) ou aminoácidos (para proteínas), buscando um alinhamento ótimo [73]. Em cada uma das posições analisadas, pode ocorrer um *match* (quando os caracteres das duas sequências nas mesmas posições são iguais), um *mismatch* (quando os caracteres são diferentes) ou

ser introduzido um *gap* [33] (um espaço, tratado geralmente como sendo a ocorrência de uma deleção) em uma das sequências.

Para avaliar de forma quantitativa a comparação, valores (ou pontuações) são atribuídos para as situações de *match*, *mismatch* e *gap*. Ademais, visando refletir um cenário que ocorre com muita frequência na natureza, usualmente a abertura de novos *gaps* possui penalização maior que a extensão de um *gap* já aberto, modelo denominado *affine gap* [7]. Na comparação de duas sequências, os valores obtidos em cada posição equivalente entre as duas sequências são somados para obter um escore final.

Existem três tipos de alinhamento: global, local e semiglobal. No alinhamento global, as duas sequências são analisadas em sua totalidade, buscando-se o maior número de resíduos idênticos. Ou seja, todos os caracteres das sequências avaliadas participam do alinhamento. No alinhamento local, apenas uma parte de cada uma das sequências é alinhada, objetivando ressaltar a área de maior similaridade entre elas [6]. Há ainda o alinhamento semiglobal, que permite remover somente o prefixo ou o sufixo das sequências a serem alinhadas. A Figura 2.1 ilustra casos de alinhamentos global, local e semiglobal entre as sequências $S_0 = AATTAAGCTCAG$ e $S_1 = AGCACTT$, com pontuações $+1$, -1 e -2 para *matches*, *mismatches* e *gaps*, respectivamente. Os escores finais obtidos nestes casos são, respectivamente, $G = -9$, $L = +3$ e $SG = +1$.

```
S0:  A  A  T  T  A  A  G  C  T  C  A  G
                   |  |  |     |
S1:  -  -  -  -  -  A  G  C  A  C  T  T
--------------------------------------
G:  -2 -2 -2 -2 -2 +1 +1 +1 -1 +1 -1 -1 = -9


S0:  A  A  T  T  A  A  G  C  T  C  A  G
                   |  |  |
S1:                A  G  C
----------------------------------------
L:                +1 +1 +1            = +3


S0:  A  A  T  T  A  A  G  C  T  C  A  G
                   |  |  |     |
S1:                A  G  C  A  C  T  T
--------------------------------------
SG:               +1 +1 +1 -1 +1 -1 -1 = +1
```

Figura 2.1: Exemplos de alinhamentos global (escore G), local (escore L) e semi-global (escore SG).

A comparação de sequências biológicas é geralmente feita de duas maneiras: o alinhamento de sequências longas, onde duas cadeias de tamanho considerável são comparadas; e a busca de uma sequência de referência em uma base de dados (também denominada *query* x *database*), onde geralmente a sequência que serve como *query* é uma sequência que possui tamanho menor [19].

2.2 Algoritmos Exatos para Comparação de Sequências

Os algoritmos exatos de comparação de sequências recebem como entrada as sequências S_0 e S_1 e fornecem como saída o alinhamento e o escore ótimos, ou seja, o alinhamento que possui o escore máximo. Os seguintes alfabetos são utilizados: $\Sigma = \{A, T, G, C\}$ para sequências de DNA, $\Sigma = \{A, T, G, U\}$ para sequências de RNA e $\Sigma = \{A, C, D, E, F, G, H, I, K, L, M, N, P, Q, R, S, T, V, W, Y\}$ para sequências de proteínas, além de um caractere adicional para representar os *gaps*. Sequências de DNA podem possuir de poucos pares de bases (caracteres) até milhões de pares de bases, e seu tamanho é expresso em *base pairs* ou suas variações, tais como KBP (*Kilo Base Pairs* - milhares de pares de bases), MBP (*Mega Base Pairs* - milhões de pares de bases) ou GBP (*Giga Base Pairs* - bilhões de pares de bases). Quando uma determinada base é desconhecida em uma sequência de DNA ou RNA, um caractere adicional $\{N\}$ é utilizado na representação. Nos algoritmos exatos, que obtêm a solução ótima, é calculada uma matriz de programação dinâmica (matriz DP ou matriz de similaridade), atribuindo-se valores a cada um dos cenários possíveis de alinhamento.

Nas Seções 2.2.1 a 2.2.4 serão detalhados os principais algoritmos exatos, utilizando as definições a seguir como referência:

- S_0 e S_1 são as sequências de entrada a serem comparadas, com comprimentos m e n, respectivamente;
- G é a penalidade constante atribuída a um *gap*, tendo portanto valor negativo;
- Em algoritmos que implementam o modelo *affine gap*, G_{first} representa a penalidade atribuída à abertura de uma nova sequência de *gaps* e G_{ext} representa a penalidade da extensão de um *gap* já aberto, também com valores negativos;
- Os valores dos escores atribuídos a *match* ou *mismatch* são representados pela função $sbt(x, y)$. Na comparação de proteínas, a pontuação é obtida através de uma matriz 20×20, chamada matriz de substituição [73].

2.2.1 Algoritmo Needleman-Wunsch (NW)

O algoritmo exato Needleman-Wunsch (NW) [78] retorna o alinhamento global ótimo entre duas sequências de entrada S_0 e S_1, possibilitando a introdução de *gaps* que são penalizados de forma constante. O algoritmo é executado em duas etapas: (1) cálculo da matriz de similaridade e (2) recuperação do alinhamento gerado (*traceback*).

Na etapa 1, uma matriz de similaridade A é criada, calculando-se para cada célula o escore do melhor alinhamento dos prefixos das sequências até aquela posição, retornando, ao final, o escore ótimo obtido. Assumindo os tamanhos das sequências de entrada m e n, a matriz de similaridade A construída através de programação dinâmica terá dimensões $(m+1) \times (n+1)$, onde cada célula $A_{i,j}$ conterá o escore de melhor alinhamento até aquela posição. Na inicialização, o valor zero é introduzido na posição $A_{0,0}$, e as demais posições da primeira linha e coluna são assim fixados: $A_{i,0} = -i * G$ e $A_{0,j} = -j * G$, onde G é o valor estipulado da penalização constante por *gap*.

Para o cálculo de uma posição $A_{i,j}$ $(i, j \neq 0)$, três cenários de alinhamento são avaliados, buscando-se o que retornará o maior escore: (a) partir do alinhamento $S_0[1..i-1]$ e $S_1[1..j-1]$ e adicionar os resíduos $S_0[i]$ e $S_1[j]$, que podem resultar em um *match* ou *mismatch* nesta posição; (b) partir do alinhamento $S_0[1..i]$ e $S_1[1..j-1]$ e inserir um *gap* em S_1, com a devida penalização (G); ou (c) partir do alinhamento $S_0[1..i-1]$ e $S_1[1..j]$ e inserir um *gap* em S_0, também com a devida penalização. A Equação 2.1 descreve a recorrência derivada destes cenários.

$$A_{i,j} = \max \begin{cases} A_{i-1,j-1} + sbt(S_0[i], S_1[j]) \\ A_{i-1,j} + G \\ A_{i,j-1} + G \end{cases} \tag{2.1}$$

Além do valor $A_{i,j}$, também é armazenada uma referência à posição anterior ($A_{i-1,j-1}$, $A_{i-1,j}$ ou $A_{i,j-1}$) que gerou o escore na célula $A_{i,j}$. Com isso, a matriz pode ser percorrida partindo-se da célula no canto inferior direito e em sentido inverso para retornar o alinhamento obtido. Esse processo é executado na etapa 2 e é denominado *traceback*.

O *traceback* não apresenta complexidade de tempo alta. O preenchimento da matriz de similaridade com NW, por outro lado, requer que todas as $(m+1)*(n+1)$ posições sejam calculadas, resultando em uma complexidade de tempo e espaço na ordem de $O(nm)$. Na comparação entre duas sequências longas de tamanhos similares, esta complexidade quadrática requer alto poder computacional e grande espaço de memória para que se produza o escore ótimo.

A Figura 2.2 apresenta um exemplo da matriz de similaridade e alinhamento para uma comparação utilizando NW. Os seguintes parâmetros foram adotados: $S_0 = TAGCTACT$,

$S_1 = TAGCTATA$, $match = +2$, $mismatch = -1$, $G = -5$. Os valores dos escores encontram-se no centro da célula. As células com padrão de sombreamento são as que levam ao alinhamento global ótimo, que serão percorridas no *traceback*.

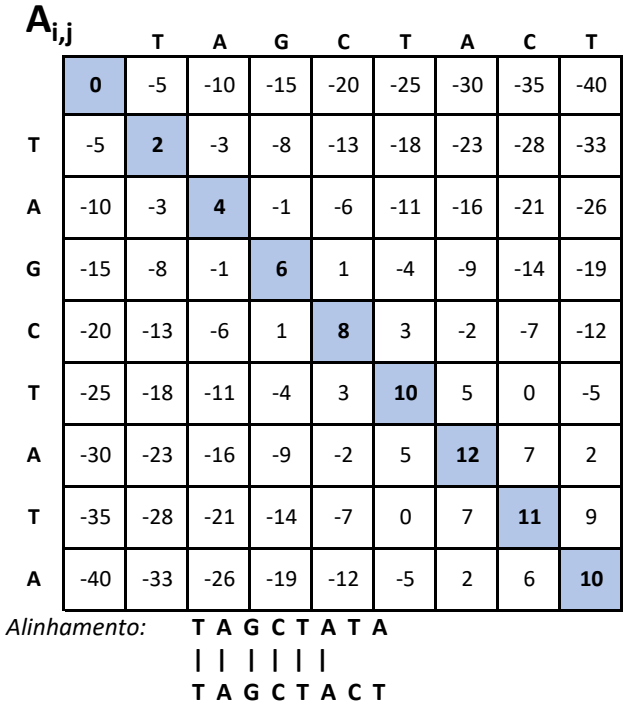


Figura 2.2: Matriz de similaridade e alinhamento global ótimo utilizando algoritmo NW.

2.2.2 Algoritmo Smith-Waterman (SW)

Também baseado em programação dinâmica, o algoritmo Smith-Waterman (SW) [113] permite a identificação de alinhamentos locais ótimos, a partir de três modificações em relação ao algoritmo NW (Seção 2.2.1). A primeira modificação ocorre na inicialização da matriz de programação dinâmica, onde a primeira linha $(0, j)$ e coluna $(i, 0)$ são inicializadas com o valor zero. A segunda modificação ocorre na definição da equação de recorrência, como pode ser observado na Equação 2.2: um termo com valor zero é adicionado à operação de máximo, impedindo que um valor negativo possa ser obtido na determinação do escore máximo em uma posição. Esta modificação propicia que um novo alinhamento local seja iniciado a partir do ponto em que ocorre um escore parcial negativo, reiniciando este escore com zero. A terceira modificação é a execução do *traceback* a partir da célula que possui o maior escore. Isso ocorre porque, a partir deste ponto, o alinhamento local obtido passa a não ser mais ótimo.

$$A_{i,j} = \max \begin{cases} A_{i-1,j-1} + sbt(S_0[i], S_1[j]) \\ A_{i-1,j} + G \\ A_{i,j-1} + G \\ 0 \end{cases} \quad (2.2)$$

A Figura 2.3 apresenta um exemplo da matriz de similaridade e alinhamento local para uma comparação utilizando SW. As sequências de entrada, os valores das pontuações e os padrões de preenchimento são os mesmos utilizados na Seção 2.2.1. A complexidade de tempo e espaço desta solução é idêntica ao algoritmo NW - $O(nm)$, o que não é ideal para a comparação de sequências longas.

$A_{i,j}$		T	A	G	C	T	A	C	T
	0	0	0	0	0	0	0	0	0
T	0	**2**	0	0	0	2	0	0	2
A	0	0	**4**	0	0	0	4	0	0
G	0	0	0	**6**	1	0	0	3	0
C	0	0	0	1	**8**	3	0	2	2
T	0	2	0	0	3	**10**	5	0	4
A	0	0	4	0	0	5	**12**	7	2
T	0	2	0	3	0	2	7	11	9
A	0	0	4	0	2	0	4	6	10

Alinhamento:
T A G C T A
| | | | | |
T A G C T A

Figura 2.3: Matriz de similaridade e alinhamento local ótimo utilizando algoritmo SW.

2.2.3 Algoritmo Gotoh

Os algoritmos citados nas Seções 2.2.1 e 2.2.2 não implementam o modelo *affine gap* (Seção 2.1), e esta é a principal contribuição do algoritmo Gotoh [34]. Para isso, a Equação 2.1 foi ajustada, permitindo que uma função de cálculo da penalização por *gap* possa ser introduzida em lugar de um valor constante G, como, por exemplo: $\lambda(k) = G_{first} + (k-1) * G_{ext}$, onde k é o número de *gaps* consecutivos.

Para que o modelo *affine gap* seja implementado, duas matrizes adicionais (E e F) são propostas, para avaliar os casos de abertura de um novo *gap* ou extensão de um *gap*

existente em cada uma das sequências. Desta forma, duas novas equações de recorrência são introduzidas, ajustando o algoritmo NW ao modelo de *affine gap*, conforme Equações 2.3, 2.4 e 2.5. Na Equação 2.3, uma condição com valor zero pode ser adicionado para evitar valores negativos, e desta forma obter o alinhamento local.

Na etapa 1, as matrizes são inicializadas, e são computadas considerando os valores de *match* e *mismatch* (H), uma sequência de *gaps* em S_0 (E) e uma sequência de *gaps* S_1 (F). As matrizes H, E e F são usadas para realizar o *traceback* dependendo da situação ocorrida. A Figura 2.4 ilustra as matrizes Gotoh.

$$H_{i,j} = \max \begin{cases} H_{i-1,j-1} + sbt(S_0[i], S_1[j]) \\ E_{i,j} \\ F_{i,j} \\ 0 \end{cases} \tag{2.3}$$

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

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

	*	C	C	C	T	T	T	T	G
*	**0**	0	0	0	0	0	0	0	0
C	0	**2**	2	2	0	0	0	0	0
C	0	2	**4**	4	1	0	0	0	0
C	0	2	4	**6**	3	1	0	0	0
A	0	0	1	2	5	2	0	0	0
A	0	0	0	**1**	2	4	1	0	0
T	0	0	0	0	**3**	4	6	3	1
T	0	0	0	0	2	**5**	6	8	4
T	0	0	0	0	2	4	**7**	8	7
T	0	0	0	0	2	4	6	**9**	7
A	0	0	0	0	0	1	3	5	8

(a) Matriz H.

	*	C	C	C	T	T	T	T	G
*	–	–	–	–	–	–	–	–	–
C	$-\infty$	−4	−2	−2	−2	−3	−4	−4	−4
C	$-\infty$	−4	−2	0	0	−1	−2	−3	−4
C	$-\infty$	−4	−2	0	2	1	0	−1	−2
A	$-\infty$	−4	−4	−3	−1	1	0	−1	−2
A	$-\infty$	−4	−4	−4	−3	−2	0	−1	−2
T	$-\infty$	−4	−4	−4	−4	−1	0	2	1
T	$-\infty$	−4	−4	−4	−4	−2	1	2	4
T	$-\infty$	−4	−4	−4	−4	−2	0	3	4
T	$-\infty$	−4	−4	−4	−4	−2	0	2	5
A	$-\infty$	−4	−4	−4	−4	−4	−3	−1	1

(b) Matriz E.

	*	C	C	C	T	T	T	T	G
*	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$
C	–	−4	−4	−4	−4	−4	−4	−4	−4
C	–	−2	−2	−2	−4	−4	−4	−4	−4
C	–	−2	0	0	−3	−4	−4	−4	−4
A	–	−2	0	**2**	−1	−3	−4	−4	−4
A	–	−3	−1	**1**	1	−2	−4	−4	−4
T	–	−4	−2	0	0	0	−3	−4	−4
T	–	−4	−3	−1	−1	0	2	−1	−3
T	–	−4	−4	−2	−2	1	2	4	0
T	–	−4	−4	−3	−2	0	3	4	3
A	–	−4	−4	−4	−2	0	2	5	3

(c) Matriz F.

C	C	C	A	A	T	T	T	T
C	C	C	_	_	T	T	T	T
2	2	2	−4	−1	2	2	2	2

(d) Alinhamento Gotoh (escore=9).

Figura 2.4: Matrizes Gotoh H, F e E para sequências $S_0 = CCCAATTTTA$ e $S_1 = CCCTTTTG$. Valores para *matches*, *mismatches*, abertura de *gap* e extensão de *gap* são +2, -1, -4 e -1, respectivamente. Células em negrito com setas representam o caminho de *traceback*.

Em relação à complexidade de tempo, o algoritmo de Gotoh é semelhante aos dois anteriores: $O(nm)$. A complexidade de espaço também se mantém, muito embora requeira o triplo de espaço de armazenamento dos algoritmos NW e SW, devido à criação das duas matrizes auxiliares.

2.2.4 Algoritmo Myers-Miller (MM)

O algoritmo Myers-Miller (MM) [77] se baseia na combinação do algoritmo de Gotoh (Seção 2.2.3) com a estratégia de comparação com complexidade quadrática de tempo e complexidade linear de espaço proposta por Hirschberg [39] para o problema da subsequência em comum mais longa (LCS - *Longest Common Subsequence*). Executa-se com complexidade de espaço linear para alinhamentos globais computados com o modelo *affine gap*.

A abordagem MM baseia-se na aplicação de sucessivas divisões na matriz de similaridade, que é processada em espaço linear, determinando-se em cada divisão um ponto que pertence ao alinhamento ótimo (*crosspoint*), contido na linha considerada. A operação é realizada através do cálculo do custo mínimo de conversão de vetores que são obtidos percorrendo a matriz nos dois sentidos: do início até a linha média, e do final até este mesmo ponto, em sentido reverso. Neste estágio, o procedimento é chamado recursivamente a partir do *crosspoint* para duas submatrizes resultantes, até a obtenção de problemas triviais. A Figura 2.5 representa um esquema de obtenção do ponto médio ótimo (ou *crosspoint*, representado pelas coordenadas $i*$ e $j*$) na matriz H. Cabe ressaltar que a abordagem de sucessivas divisões pode ser aplicada com sucesso aos algoritmos NW (Seção 2.2.1) e SW (Seção 2.2.2).

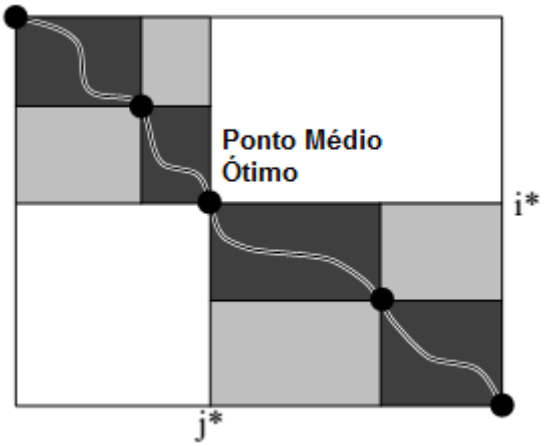


Figura 2.5: Ponto médio ótimo em matriz de similaridade do algoritmo MM - adaptado de [77].

A solução proposta por Myers-Miller permitiu a comparação de duas sequências de 62.500 bases utilizando apenas 1 MB de memória [77]. Nos algoritmos anteriores (Seções 2.2.1 a 2.2.3), se considerarmos duas sequências com o mesmo tamanho (62.500 bases cada), onde cada célula da matriz requer 4 bytes de memória, o total de bytes necessários para armazenar toda a matriz seria 15,6 GB.

2.3 Técnicas de Descarte de Dados (*Pruning*)

Técnicas de descarte de dados fazem a identificação de cálculos envolvendo dados que podem ser eliminados sem comprometer a correção dos resultados, reduzindo a utilização de recursos computacionais e, por consequência, melhorando o desempenho.

A comparação de sequências com algoritmos descritos na Seção 2.2 pode levar um longo tempo, especialmente se sequências longas são comparadas. Por exemplo, a obtenção do escore ótimo entre duas sequências com 10 MBP usando um algoritmo exato requer que 10^{14} células da matriz DP sejam processadas. Em vários casos, no entanto, muitas destas células podem ser descartadas antes de serem calculadas, porque não contribuem para o escore ótimo, o que pode reduzir significativamente o tempo de execução. Cabe ressaltar que as técnicas de descarte de dados discutidas nessa seção não possuem impacto no resultado final, ou seja, o algoritmo continua retornando o escore ótimo.

2.3.1 Algoritmo Fickett

O algoritmo de Fickett [24] faz o descarte de dados ao definir que somente uma faixa da matriz DP será calculada, ao invés de toda a matriz. Fickett usa como base o algoritmo NW (Seção 2.2.1), e define a faixa de tamanho k (k-*band*) baseando-se na observação de que sequências muito similares possuem poucos *gaps*, e, portanto, o alinhamento está localizado nas vizinhanças da diagonal principal da matriz. Como consequência, algumas áreas da matriz não são calculadas, sendo portanto descartadas, melhorando o desempenho.

A Figura 2.6 ilustra o algoritmo de Fickett. Inicialmente, é definida uma faixa de tamanho D_0 e somente a área em cinza é calculada. Ao se fazer o *traceback*, percebe-se que partes do alinhamento (linha vermelha) estão fora da faixa. Sendo assim, o tamanho da faixa é aumentado para D_1, e esta faixa é então calculada, recuperando-se o alinhamento. Caso o alinhamento não estiver totalmente contido em D_1, repete-se o processo, aumentando-se novamente a faixa, até que o alinhamento possa ser recuperado.

A complexidade de tempo e espaço do algoritmo de Fickett é da ordem de $O(kn)$, onde k é o tamanho da faixa de cálculo e n é o tamanho da menor sequência - quanto maior a similaridade, melhor o desempenho do algoritmo. Espaços de memória adicionais

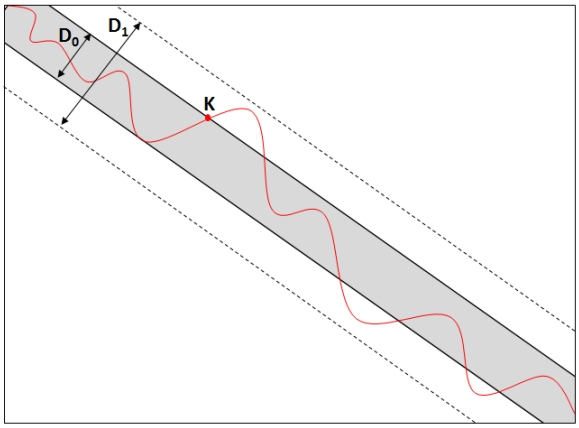


Figura 2.6: Esquema de funcionamento do algoritmo Fickett [100].

são requeridos para armazenar os valores de índices em cada linha, mas isso não impacta na complexidade de espaço.

2.3.2 LBD-Align

O LBD-Align (*Linear Bounded Diagonal Alignment*) [17] visa obter o alinhamento global ótimo, e combina o processamento da matriz de similaridade obtida com programação dinâmica com ideias do algoritmo A-Star [38], que faz uma busca *best-first* e possui capacidade de descarte de dados. Assume-se que a matriz é processada antidiagonal por antidiagonal e que somente as células que estão dentro da fronteira de descarte (*pruning frontier*) são processadas. Para cada antidiagonal, a fronteira de descarte é definida por dois limites: inferior e superior, que são estimados. O autor sugere que o limite inferior seja estimado com um algoritmo heurístico como o BLAST [1], já que o valor do escore ótimo do alinhamento BLAST é sempre igual ou inferior ao valor do escore ótimo. Isso faz com que tal limite seja admissível pelo A-Star. O limite superior é definido com a distância de Manhattan [50] do ponto em consideração até o final da matriz. Os limites podem ser estimados novamente ao longo da execução, porém o autor alerta que tal estimativa pode causar um *overhead* considerável.

A Figura 2.7 ilustra o processamento do LBD-Align. Note-se que as diagonais D_0 a D_3 foram processadas na sua totalidade, enquanto somente as células das diagonais D_4 e D_5 dentro da fronteira de descarte foram calculadas. Nos resultados experimentais, foram usadas sequências de DNA de até 230.000 caracteres e, estimando-se o limite inferior com uma estratégia gulosa, 61,48% das células da matriz foram descartadas. Com uma má estimativa, essa percentagem caiu para 48,65%.

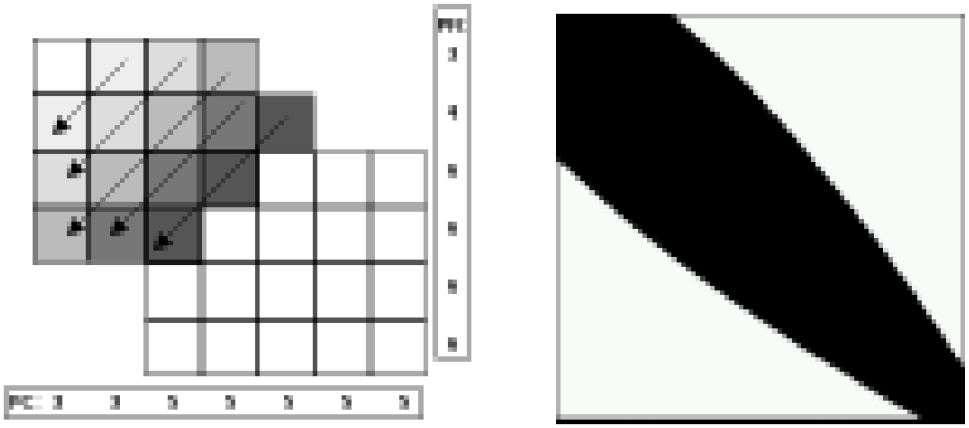


Figura 2.7: Esquema de funcionamento do algoritmo LDB-Align [17]. Apenas as células na região em preto foram processadas.

2.3.3 Descarte de Blocos

O descarte de blocos (BP - *Block Pruning*) foi proposto no CUDAlign 2.1 [102], que provê o alinhamento local de sequências em uma GPU utilizando 6 estágios. A estratégia de BP foi adicionada ao estágio 1, que corresponde à etapa 1 do algoritmo Gotoh (Seção 2.2.3), ou seja, à de computação da matriz de similaridade. As células são agrupadas em diagonais externas, compostas por blocos de células, e caso o escore calculado na célula no canto superior esquerdo de cada bloco adicionado ao máximo escore que poderia ser obtido deste ponto até o final da matriz não seja superior ao melhor escore global corrente (CGBS - *Current Global Best Score*), todo o bloco é descartado.

Na proposta inicial [102], o BP assume processamento antidiagonal de blocos por antidiagonal de blocos, e somente as duas extremidades dos blocos de diagonais são avaliadas: os primeiros blocos são analisados até encontrar um que não pode ser descartado, e da mesma forma são analisados os últimos blocos da diagonal, em sentido inverso. A determinação dos blocos que não serão calculados é feita antes do cálculo de cada antidiagonal de blocos, sempre em CPU, mesmo que a solução use GPU para o cálculo das antidiagonais. Desta forma, de maneira similar ao LDB-Align (Seção 2.3.2), apenas os blocos entre estes limites serão processados, e os demais serão descartados. Por exemplo: supondo uma diagonal com n blocos, o BP avaliará os blocos a partir do bloco 1 de forma crescente até encontrar um bloco x que não pode ser descartado, e a partir do bloco n de forma decrescente até encontrar um bloco y que também deve ser processado. Neste cenário, $(x-1)+(n-y)$ blocos serão descartados, e apenas os demais serão efetivamente processados.

A Figura 2.8 ilustra como se dá a avaliação do descarte de uma célula representativa

de um bloco em uma matriz DP de dimensões $(m+1) \times (n+1)$ no CUDAlign 2.1. A linha em diagonal dentro da matriz representa o alinhamento ótimo, e o melhor escore em um determinado momento é representado por $best(i)$. Dada uma determinada célula $H(i,j)$, devem ser calculadas então as distâncias até a última linha (Δi) e coluna (Δj) da matriz, tomando-se o valor mínimo entre elas. Este valor é então multiplicado pelo valor de *match* (ma) e acrescido ao escore desta célula para se obter o valor máximo que pode ser obtido a partir deste ponto ($H_{max}(i,j)$). Caso este valor seja inferior a $best(i)$, significa que a célula (e, portanto, todo o bloco) não pode gerar um melhor alinhamento, o que permite o descarte.

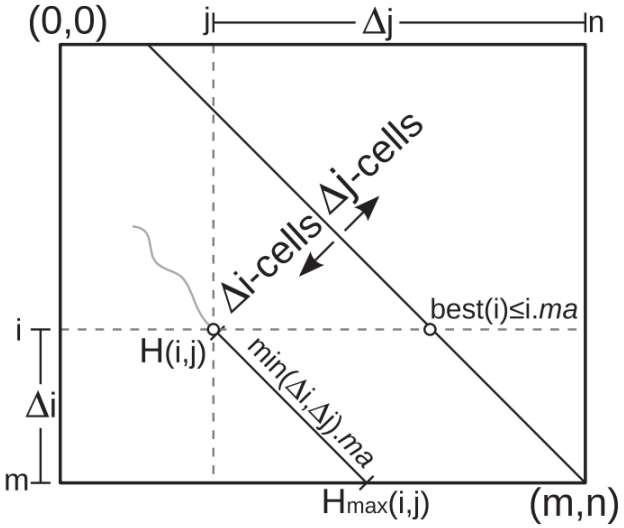
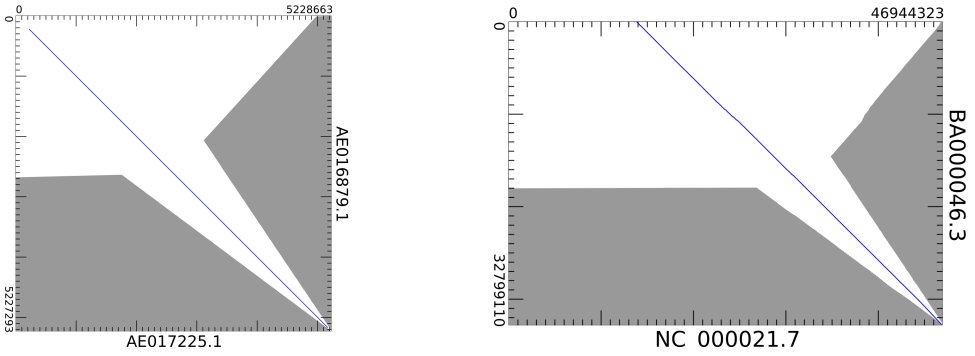


Figura 2.8: Esquema de funcionamento do BP no CUDAlign 2.1 [102].

Esta técnica melhora significativamente o desempenho da solução, principalmente se duas sequências muito similares forem comparadas. Nos testes com duas sequências de (a) 5 MBP $\times$ 5 MBP e (b) 33 MBP $\times$ 47 MBP (Figura 2.9), pode-se verificar o *pruning* real produzido pelo BP, o que permitiu um ganho de desempenho de mais de 50% em relação à mesma execução sem o descarte de blocos.

A estratégia de BP diagonal também foi utilizada na solução SW# [51], e neste caso o alinhamento pôde ser realizado em duas GPUs. Contudo, a solução não possibilita a execução do BP em mais que duas GPUs.

Na proposta do *framework Multi-Platform Architecture for Sequence Aligners* (MASA) [105], que permite o alinhamento de sequências em outros dispositivos e suporta outras formas de alinhamento (incluindo uma paralelização genérica baseada em fluxo de dados), uma nova forma de processamento do BP foi adicionada ao processamento diagonal: uma estratégica genérica. Neste caso, em vez de avaliar as oportunidades de descarte de blocos em cada diagonal, uma matriz armazena o estado de *pruning* de cada bloco, marcando

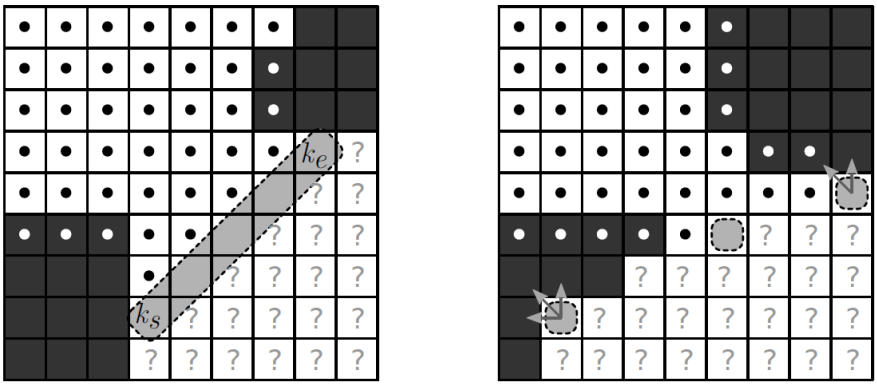


(a) (b)

Figura 2.9: Matriz de programação dinâmica em duas comparações de DNA com CUDAlign 2.1 [102]: (a) 5 MBP × 5 MBP e (b) 33 MBP × 47 MBP. A área cinza (53,7% e 48,1%, respectivamente) representa os blocos descartados. A linha diagonal indica o alinhamento ótimo obtido.

um bloco como "descartável" caso os outros da sua vizinhança também o sejam, o que requer que uma área de memória equivalente ao tamanho total de blocos da matriz seja alocada.

A Figura 2.10 ilustra as duas estratégias de execução do algoritmo BP apresentadas em [105]. Blocos com um círculo preto são totalmente processados e blocos em cinza escuro com um círculo branco são descartados. As células com contorno tracejado são as que estão sendo processadas em paralelo, enquanto as interrogações indicam os blocos que ainda não foram processados. Na abordagem diagonal (Figura 2.10a), blocos na mesma diagonal dentro da janela $[ks :: ke]$ podem ser processados em paralelo. Na abordagem genérica (Figura 2.10b), blocos de diferentes diagonais são processados em paralelo.



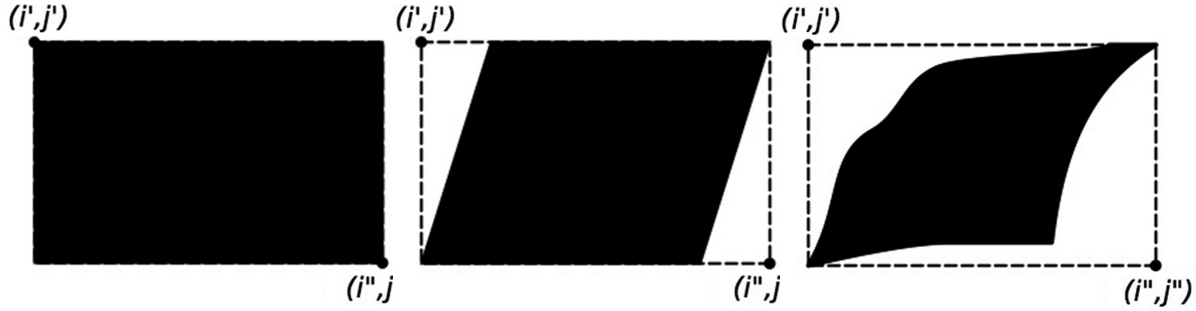
(a) BP diagonal. (b) BP genérico.

Figura 2.10: Algoritmos BP da ferramenta MASA [105].

Posteriormente, outro trabalho realizou uma formalização matemática do BP [109]. A estratégia de BP, que era aplicável apenas para alinhamentos locais, foi ampliada para

ser usada também em alinhamentos globais (Seção 2.2.1). Para tanto uma estratégia de descarte baseada na determinação de limites superiores e inferiores para uma dada célula $H_{i,j}$ da matriz DP é proposta, e provas matemáticas são utilizadas para mostrar que a utilização do *pruning* garante que o escore ótimo será obtido. Um novo algoritmo genérico linear é também proposto, que permite que as células da matriz DP possam ser processadas em qualquer ordem.

Para garantir a correção dos algoritmos e eliminar dependências cíclicas entre blocos quando o descarte é avaliado em um bloco de células contíguas (em vez de célula a célula), é necessário garantir que, caso as células $H_{i,j}$ e $H_{i-1,j-1}$ pertençam a um bloco, $H_{i,j-1}$ e $H_{i-1,j}$ também devem pertencer. Três formatos de blocos são apresentados no trabalho que atendem a este requisito, como pode ser visto na Figura 2.11: com padrão retangular, em paralelogramo ou irregular, onde (i', j') é a mínima coordenada de uma célula no bloco e (i'', j'') a máxima.

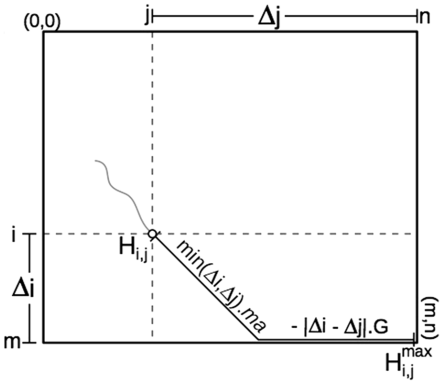


(a) Retangular. (b) Paralelogramo. (c) Irregular.

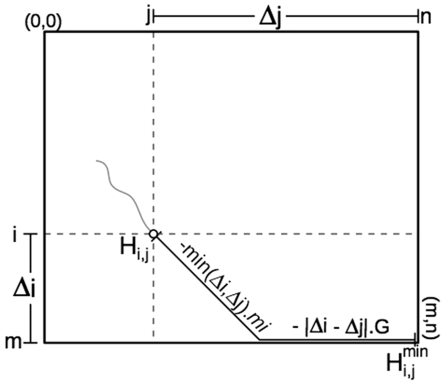
Figura 2.11: Formatos de blocos [109].

A Figura 2.12 mostra uma representação geométrica do BP em cenários com alinhamento global e alinhamento local em uma matriz com m linhas e n colunas. Considerando uma célula $H_{i,j}$, as distâncias até a última linha e última coluna são, respectivamente, Δi e Δj. O máximo valor que pode ser obtido por um escore a partir deste ponto ($H_{i,j}^{max}$) pode ser calculado como o mínimo entre estas duas distâncias multiplicado pelo valor de *match* (*ma* na Figura 2.12), enquanto o mínimo valor ($H_{i,j}^{min}$) é igual ao escore em $H_{i,j}$, visto que simula a situação em que um melhor alinhamento não pode ser obtido a partir deste ponto. Caso o valor máximo seja inferior ao melhor escore atual, esta célula pode ser então descartada. Este formalismo também pode ser utilizado para descartar um bloco de células, como foi também provado no trabalho.

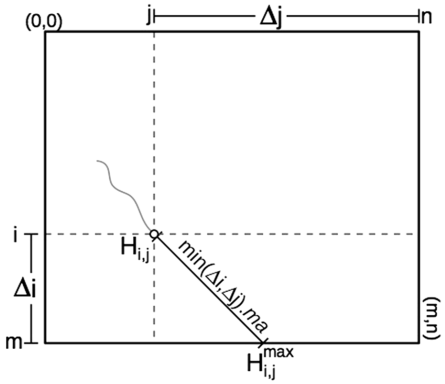
Para avaliar os diversos cenários de uso do BP, foram realizadas diversas simulações de uso no BP com quatro pares de sequências de tamanhos variando entre 50 KBP e 1 MBP. As seguintes simulações foram realizadas: variando as formas de processamento da matriz DP, variando os ângulos de processamento, testando várias situações de simila-



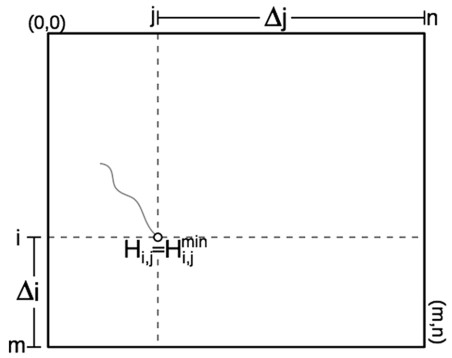
(a) $H_{i,j}^{max}$ para alinhamento global.



(b) $H_{i,j}^{min}$ para alinhamento global.



(c) $H_{i,j}^{max}$ para alinhamento local.



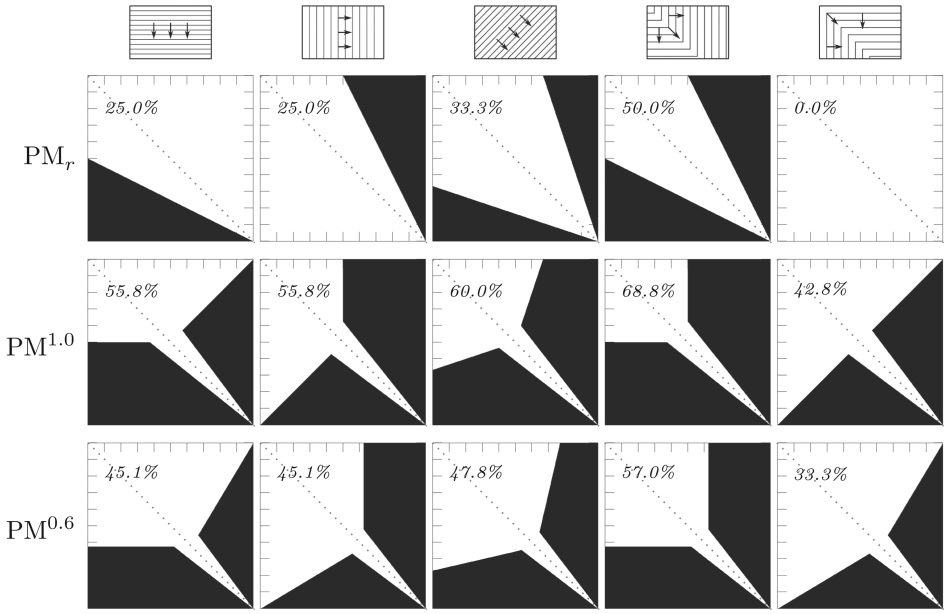
(d) $H_{i,j}^{min}$ para alinhamento local.

Figura 2.12: Representação geométrica no descarte de célula [109].

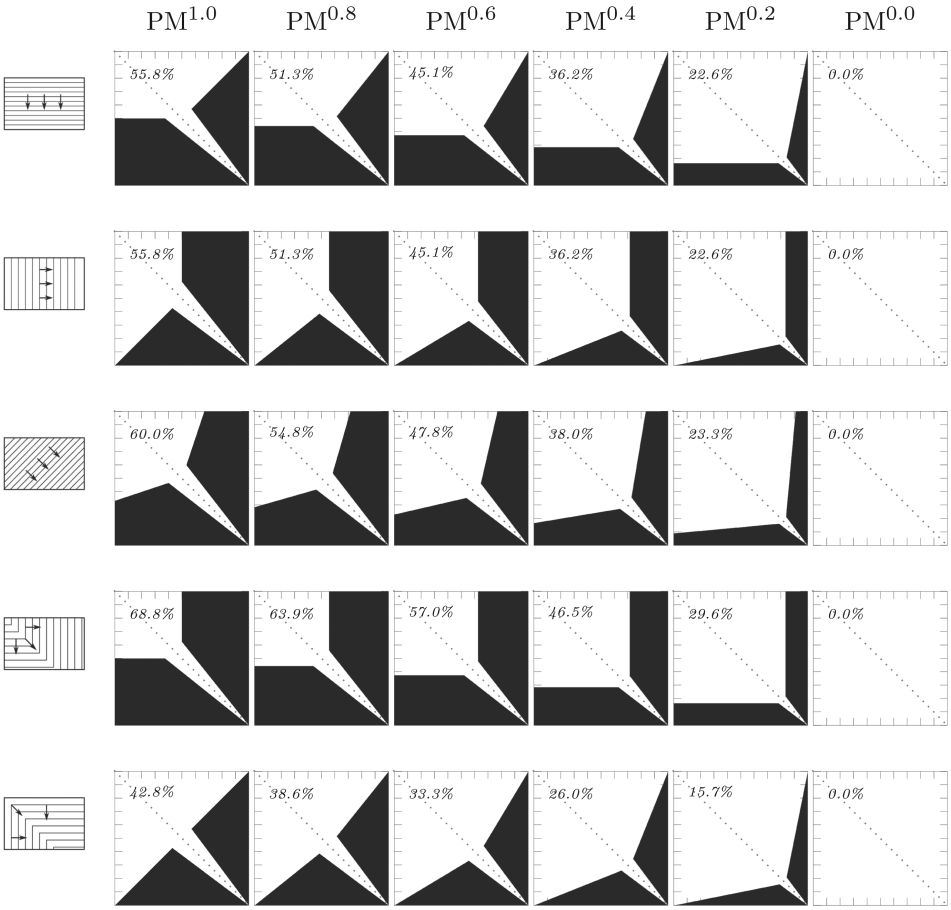
ridade entre sequências, variando o valor utilizado para a penalidade em caso de *gap* e variando o tamanho das sequências. A Figura 2.13 mostra duas destas simulações e seus efeitos na taxa de descarte de blocos. Na Figura 2.13a, são avaliadas as formas de processamento da matriz DP (por linhas, por colunas, diagonal, quadrada e antiquadrada) em três cenários de situações de *match*: perfeito com repetição de caracteres (PM_r), perfeito com caracteres distintos ($PM^{1.0}$) e parcial ($PM^{0.6}$). Como se vê, as execuções com $PM^{1.0}$ obtiveram maiores taxas de descarte de blocos (entre 42,8% e 68,8%), e as formas de processamento quadrada e diagonal foram as que se mostraram mais efetivas. Já na Figura 2.13b, observa-se que novamente as formas quadrada e diagonal geram mais blocos descartados, com um máximo de 68,8% e 60,0% de *pruning*, respectivamente. Já na variação da similaridade entre as sequências (valor x em cada PM^x nas colunas), verifica-se que quanto maior a similaridade maior a efetividade da técnica, o que era um resultado esperado.

De acordo com este trabalho, a eficiência do BP, em termos de porcentagem de blocos descartados, pode ser impactada por três aspectos:

- **a ordem usada para calcular as células da matriz:** processamento com padrão



(a) Efeito no descarte de blocos de diferentes formas de processamento.



(b) Efeito no descarte de blocos de diferentes similaridades de sequências.

Figura 2.13: Simulações de descarte de blocos [109].

quadrado tende a ser mais eficiente em relação ao *pruning*, mas o processamento em diagonal também atinge bons resultados;

- **valores dos parâmetros da equação de recorrência:** os melhores resultados são obtidos quando as penalidades de *gap* são altas, contudo penalidades mais altas para *gaps* podem reduzir a qualidade biológica do alinhamento;
- **o conteúdo das sequências:** comparações com sequências muito similares produzem maiores percentuais de blocos descartados.

2.4 Abordagens Paralelas

A exigência de alto poder computacional para execução dos algoritmos exatos de comparação de sequências demanda a utilização de técnicas que possibilitem a obtenção de resultados em tempos adequados. O processamento paralelo [14], por exemplo, que permite que várias operações sejam realizadas simultaneamente, é altamente utilizado. Para determinar que operações podem ser feitas em paralelo, deve ser feita uma análise da dependência de dados.

O cálculo das células da matriz de similaridade através das equações de recorrência discutidas nas Seções 2.2.1 a 2.2.4 possui alto custo computacional, e, portanto, é uma opção de paralelização com grande potencial de melhorar o desempenho. Um ponto em comum entre os algoritmos apresentados na Seção 2.2 é o tipo de dependência de dados, onde cada posição (i, j) da matriz de similaridade depende de três posições previamente calculadas: $(i-1, j-1)$, $(i-1, j)$ e $(i, j-1)$. Esta dependência de dados adéqua-se à técnica de processamento em *wavefront*, que possui este nome pois a computação se assemelha a uma frente de onda em propagação. No processamento *wavefront* diagonal, a matriz DP é percorrida antidiagonal a antidiagonal, a partir da antidiagonal 0 no canto superior esquerdo. Desta forma, os escores das células que compõem uma mesma antidiagonal podem ser calculados de forma paralela. A Figura 2.14 ilustra este comportamento em uma matriz DP 4×4. Após a célula $(1, 1)$, que corresponde à antidiagonal $d1$, ser processada, as células $(2, 1)$ e $(1, 2)$ podem ser processadas em paralelo (antidiagonal $d2$), e assim por diante. Vale ressaltar que no início do processamento os recursos computacionais são geralmente subutilizados, visto que não existe paralelismo suficiente no *wavefront* para que todos os elementos de processamento sejam alocados. Considera-se que o *wavefront* está cheio quando possui dados/instruções suficientes para alocar todos os elementos computacionais disponíveis, maximizando o paralelismo.

Outro aspecto a ser definido na estratégia de paralelização é a granularidade do processamento dos dados. Nas aplicações de grão grosso (*coarsed-grained*), porções maiores

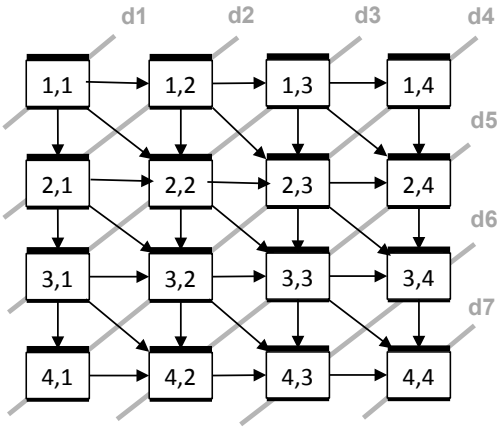


Figura 2.14: Processamento em *wavefront* diagonal de matriz 4×4.

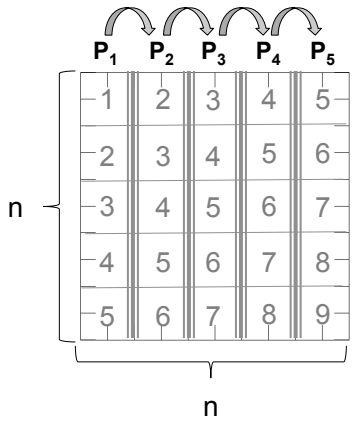
de dados são tratados pelas tarefas, enquanto nas abordagens de grão fino (*fined-grained*), os dados são processados em blocos de menor dimensão.

No problema da comparação de sequências biológicas, as aplicações de grão grosso usualmente comparam uma sequência de consulta (*query*) com um conjunto do banco de sequências (*database*). Portanto, se tivermos w entradas no banco de sequências, w comparações serão feitas, uma em cada tarefa, sem comunicação entre elas. No paralelismo de grão grosso, as sequências de banco de dados são geralmente ordenadas previamente pelo tamanho para balancear a quantidade de computação entre as unidades de computação. Por outro lado, soluções de grão fino utilizam várias unidades de computação para comparar uma sequência *query* com outra sequência, dividindo o processamento de uma única matriz entre as diversas unidades de processamento, como na abordagem *wavefront*.

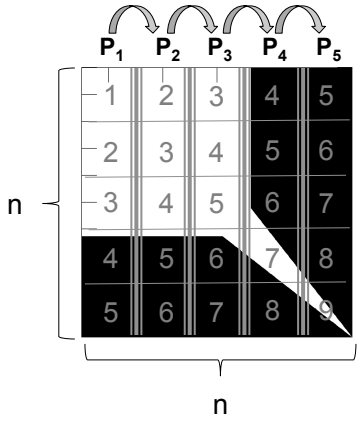
A Figura 2.15 ilustra o processamento de uma matriz $n \times n$ utilizando cinco elementos de processamento em dois cenários: sem e com *block pruning*. Neste caso, blocos da matriz DP são divididos entre os elementos de processamento por colunas. Pode-se notar que, no processamento da matriz com descarte de dados, o padrão de processamento dos dados se torna irregular. Por exemplo, P_1 e P_4 processarão muito poucas células na antidiagonal 4, enquanto P_2 e P_3 processarão todas as suas células. Neste caso, apesar de conter um considerável número de células descartadas, a velocidade da computação é determinada pela velocidade de P_2 e P_3, devido à dependência de dados.

2.5 Medida de Desempenho - CUPS

O desempenho de aplicações é geralmente expresso em unidades de tempo de execução (segundos ou milissegundos). Entretanto, devido às características do processamento da matriz de programação dinâmica e ao grande volume de dados, a métrica mais usada



(a) Matriz DP $n \times n$, sem *pruning*.



(b) Matriz DP $n \times n$, com *pruning*.

Figura 2.15: Dois cenários de *pruning* para 5 elementos de processamento.

para medição de desempenho para aplicações de comparação de sequências biológicas é expressa em número de células atualizadas por segundo (CUPS - *Cells Updated per Second*), podendo atingir ordens de grandeza de milhões (MCUPS), bilhões (GCUPS) ou trilhões (TCUPS) de células atualizadas por segundo.

A métrica CUPS é obtida dividindo-se a quantidade de células da matriz pelo tempo de execução total. A quantidade de células é obtida através do produto entre os tamanhos das duas sequências comparadas, ou pelo produto entre o tamanho da sequência *query* e o banco de sequências (mais usual na comparação de proteínas). Tomando-se por base duas sequências de tamanhos m e n, e sendo t o tempo medido em segundos, o desempenho em CUPS é obtido pela Equação 2.6. Para o cálculo do GCUPS, por sua vez, o tempo t deve ser multiplicado por 10^9.

$$CUPS = \frac{m * n}{t} \tag{2.6}$$

Em que pese esta medida não ser perfeita, visto que diferentes configurações, ambientes ou sequências podem influenciar os resultados do cálculo dos CUPS, ela provê uma boa aproximação do comportamento de uma determinada abordagem. Desta forma, esta medida será adotada como referência de desempenho nesta Tese.

Capítulo 3

Execução Paralela em GPU

A paralelização de execução dos algoritmos exatos para comparação de sequências biológicas vistos no Capítulo 2 utiliza plataformas de computação de alto desempenho. O grande número de elementos de processamento permite que várias tarefas possam ser executadas simultaneamente, desde que a dependência de dados seja respeitada, o que diminui o tempo de execução.

Plataformas que permitem a execução paralela das comparações de sequências vêm sendo muito utilizadas desde a década de 1990, com ênfase em *multicores* (CPU) e aceleradores, tais como *hardware* reconfigurável (FPGA) [29], Intel Xeon Phi [44] (já descontinuado) e placas gráficas (GPUs). Muito embora trabalhos citados nesta Tese utilizem estas plataformas e alguns testes tenham sido realizados em CPU, as soluções propostas são baseados em GPU, então esta plataforma será detalhada neste capítulo, que apresenta aspectos da plataforma, além de ambientes de programação usualmente utilizados por soluções paralelas para a comparação de sequências biológicas em GPU.

3.1 *Graphics Processing Unit* (GPU)

A placa gráfica (ou GPU) é uma arquitetura amplamente utilizada na implementação da comparação de sequências biológicas. Utilizadas inicialmente apenas com a função de processamento gráfico, as GPUs também se destacam como uma plataforma de uso genérico que permite o paralelismo em grande escala. Desta forma, as GPUs podem ser utilizadas para resolução de problemas genéricos, funcionando como unidades de propósito geral, ou *General Purpose Graphics Processing Unit* (GPGPU) [64].

Entre os fabricantes de GPUs, destacam-se atualmente a NVidia e a AMD. Apesar de compartilharem conceitos similares, as arquiteturas das GPUs variam entre esses fabricantes. Características peculiares podem favorecer a utilização de determinada placa

gráfica na execução de determinada aplicação, tomando por base aspectos como consumo de energia ou desempenho, por exemplo.

As GPUs da AMD se baseiam numa arquitetura de processadores utilizando vetores. A empresa tem focado no projeto de placas para computadores pessoais, notadamente para usuários de jogos. Tomando por base a arquitetura da placa Radeon RX 6900 XT, lançada em dezembro de 2020, as unidades de processamento são chamadas de *Computing Units*. Neste modelo específico, existem 80 *Computing Units*, além de 320 unidades de textura e 128 unidades de renderização, o que faz com que esta GPU possa atingir até 46,08 trilhões de operações de ponto flutuante por segundo (TFLOPS - *Tera Floating-point Operations Per Second*) [3].

Já a NVidia Corporation lançou a sua primeira GPU para computadores pessoais em 1999, com o nome GeForce 286 e arquitetura NV10, voltada principalmente para o mercado de jogos. Nos anos seguintes (entre 2000 e 2008), a NVidia continuou criando modelos de placa GeForce, identificadas pelos números 2 a 9. Em 2008, a empresa lançou a placa GeForce GTX 200, e passa a usar esta nova nomenclatura. As séries seguintes (GTX 300, em 2009, a GTX 900, em 2014) evoluindo em aspectos como quantidade de núcleos, *clock*, velocidade e largura de banda da memória.

As séries para computadores pessoais estão relacionadas a nomes de arquiteturas projetadas pela NVidia. Elas são descritas a seguir incluindo seus modelos de placas mais representativos: Tesla / 8800 GTX, Fermi / GTX 480, Kepler / GTX 680, Mawell / GTX 980 Ti, Pascal / GTX 1080 Ti, Turing / RTX 2080 Ti, Volta / Quadro GV100 e Ampere / RTX 3090. O modelo para GeForce RTX 3090, lançado em 2020, possui 10.496 núcleos, *clock* de 1,40 GHz e memória padrão de 24 GB com largura de banda de 384 bits, chegando a um máximo de 36 TFLOPS.

Além de componentes que possibilitam a comunicação placa gráfica/*host* e de distribuição de trabalho entre os componentes, as GPUs NVidia possuem um *array* de *Texture Processor Clusters* (TPCs), que acessa áreas de memória DRAM. A Figura 3.1 ilustra uma visão simplificada de uma TPC. Tipicamente, cada um dos TPCs possui um ou mais *Streaming Multiprocessors* (SMs), contendo área de *cache* de instruções (I-Cache - *Instruction Cache*), área de *cache* de constantes (C-Cache - *Constant Cache*), área de gerenciamento de *Multithreads* (MT *Issue*), unidade de textura (*Texture Unit*) e área de memória compartilhada (*Shared Memory*), além dos núcleos de processadores chamados de *Streaming Processors* (SP) e *Special Function Units* (SFU) [88].

A partir do ano 2016, com a arquitetura Pascal, a NVidia intensificou a produção de placas gráficas para ambiente de *datacenters*, com unidades específicas adaptadas para aplicações na área de Inteligência Artificial (IA), como os *Tensor Cores*. Além disso, o número de núcleos das GPUs e sua velocidade de *clock* continuaram aumentando, e por

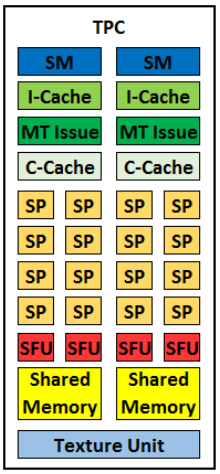


Figura 3.1: Arquitetura de TPC contendo 2 SMs, 16 SPs e 4 SFUs [88].

isso as placas NVidia continuam sendo bastante adequadas a outras aplicações GPGPU.

Como exemplo, temos a arquitetura da placa NVidia Full GV100 na Figura 3.2, onde se pode ver a arquitetura do SM da GPU. A placa NVidia Full GV100 é composta por 6 *Graphics Processing Clusters* (GPCs), cada qual com 7 TPCs, sendo que cada deles possui 2 SMs, totalizando 84 SMs. Cada SM possui 64 *cores* de ponto flutuante de 32 bits (`FP32`), 64 *cores* inteiros de 32 bits (`INT32`), 32 *cores* de ponto flutuante de 64 bits (`FP64`), 8 *Tensor Cores* e 4 unidades de textura. Os *Tensor Cores* são núcleos de computação de precisão mista, que permitem uma adaptação dinâmica dos cálculos para aumentar o desempenho mantendo a precisão, o que pode ser explorado por aplicações de IA ou de computação de alto desempenho. Outro aspecto interessante é a apresentação de uma nova geração da *High Bandwidth Memory*, a HBM2. Nesta arquitetura, módulos de memória podem ser empilhados (até 8 módulos por pilha), criando um modelo tridimensional. A GPU GV100 possui configuração básica 32 GB de memória e largura de banda de 900 GB/s. Outro destaque na arquitetura é a tecnologia de interconexão chamada NVLink, que permite um tráfego de dados de até 300 GB/s.

A Tabela 3.1 lista algumas características de GPUs de diferentes famílias da NVidia que foram utilizadas no desenvolvimento desta Tese. Como se pode notar, houve uma grande evolução da quantidade de núcleos e capacidade de processamento (TFLOPS) a cada nova arquitetura lançada. Os modelos P100, V100 e A100 [84] são voltados ao ambiente de *datacenters*.

Figura 3.2: Arquitetura da SM da GPU NVidia GV100 [83].

Tabela 3.1: Comparação entre GPUs produzidas pela NVidia.

Característica	**GeForce GTX 680**	**GeForce GTX 980 Ti**	**Tesla P100**	**Tesla V100**	**Tesla A100**
Lançamento	2012	2015	2016	2017	2020
Arquitetura	Kepler	Maxwell	Pascal	Volta	Ampere
Núcleos	1.536	2.816	3.840	5.120	6.912
Clock (MHz)	1.006	1.000	1.190	1.530	1.410
TFLOPS	3,2	5,6	21,2	31,4	78,0
Memória (GB)	2	6	12	32	40

3.2 Ambientes para Programação Paralela

Aliado às plataformas de *hardware*, o desenvolvimento de soluções paralelas demanda o uso de ambientes de programação que permitam que os dados possam ser divididos e processados por tarefas distintas. Esta seção abordará alguns destes ambientes, detalhando de forma particular as opções mais utilizadas em soluções de comparação de sequências biológicas.

Uma das estratégias utilizadas para o desenvolvimento em GPUs é a utilização de linguagens voltadas especificamente para a arquitetura desejada, como por exemplo a plataforma *Compute Unified Device Architecture* (CUDA) [81], desenvolvida pela NVidia. Uma alternativa a esta decisão de projeto é a adoção de *frameworks* de programação paralela para ambientes heterogêneos, como por exemplo o *Open Computing Language* (OpenCL). Já para o desenvolvimento em CPUs, é frequente o uso de *threads* e do *Open Multi-Processing* (OpenMP) [15], além do padrão de passagem de mensagens em ambiente distribuído *Message Passing Interface* (MPI) [36]. As Seções 3.2.1 a 3.2.3 discutem aspectos de alguns destes ambientes/linguagens.

3.2.1 Programação em GPUs NVidia: Arquitetura CUDA

A arquitetura de *hardware* e *software* CUDA [81] foi desenvolvida pela NVidia em 2006 para possibilitar a implementação de aplicações GPGPU em suas placas gráficas [99]. Os componentes da arquitetura podem ser observados na Figura 3.3.

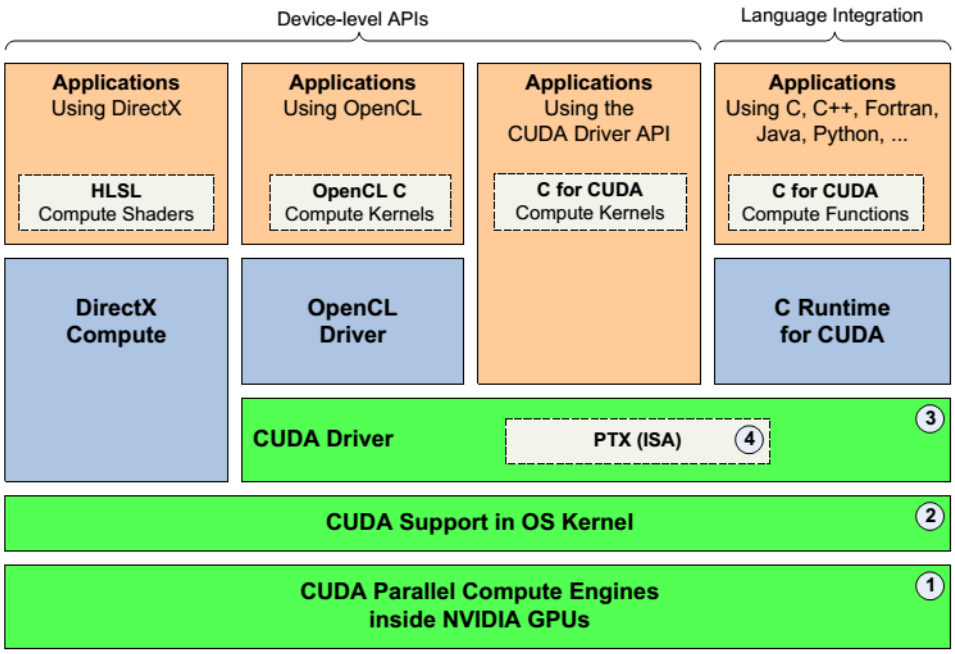


Figura 3.3: Modelagem dos componentes da arquitetura CUDA [79].

A camada inferior da arquitetura CUDA representa os elementos de computação existentes na GPU, que vão variar de acordo com o modelo da placa. Acima dela, existe o nível 2, que permite interação com o *kernel* do sistema operacional, permitindo funções como inicialização e configuração do *hardware*.

A camada subsequente (nível 3) já pode ser manipulada pelo programador, comportando um *driver* em modo usuário que possui uma *Application Programming Interface*

(API) para programação em baixo nível. Na API, está implementado um conjunto de instruções denominadas *Parallel Thread Execution* (PTX) [82], que por sua vez possuem um componente adicional que funciona como uma máquina virtual, portando um código gerado para diferentes modelos de GPUs. Se desejar, o programador pode desenvolver aplicações diretamente sobre o PTX, utilizando uma linguagem de baixo nível similar ao Assembly. Acima da camada 3, pode-se ver as interfaces de programação de alto nível: HLSL (para computação DirectX), OpenCL C (quer requer o *driver* OpenCL do dispositivo), CUDA C (usando diretamente a API do dispositivo) ou outras linguagens como C++, Fortran ou Java (o que requer o C *Runtime* para CUDA).

Quanto ao modelo de programação, o CUDA C estende a linguagem C, possibilitando ao programador definir funções a serem executadas em GPUs (denominadas *kernels*) para execução em uma quantidade definida de *threads* em paralelo, criando várias instâncias. Logicamente, as *threads* são agrupadas em blocos (*blocks*), e os blocos são então agrupados em grades (*grids*), e tanto os *blocks* quanto os *grids* podem ter até 3 dimensões.

As *threads* possuem um nível de isolamento em relação ao conjunto de registradores e memória local privada (*local memory*), e são identificadas unicamente dentro de cada bloco pelo seu `threadIdx`. Durante a execução, as *threads* são organizadas em conjuntos em um *warp*, que executam a mesma instrução paralelamente (modelo *Single Instruction Multiple Threads* - SIMT). Já os blocos também são identificados unicamente dentro de seu *grid*, e possuem uma memória compartilhada (*shared memory*) acessível apenas por suas *threads*. Já o *grid* é o conjunto de blocos executando um mesmo *kernel*, e possui uma memória que pode ser acessada por todas as *threads* de todos os blocos (*global memory*), o que requer cuidados especiais para evitar condições de corrida. Uma ilustração da hierarquia de memórias CUDA pode ser vista na Figura 3.4.

Além das memórias citadas anteriormente (*local*, *shared* e *global*), outros dois tipos de memória de apenas leitura completam a hierarquia: constante (*constant*) e textura (*texture*). Esta última é muito utilizada em aplicações que requerem escrita única e leitura múltipla, pois possui forma de acesso rápido e estrutura de *cache*. Caberá ao desenvolvedor identificar as áreas de memória adequadas para o armazenamento dos dados relativos à sua aplicação. Na comparação de sequências, por exemplo, é usual que os caracteres das sequências sejam armazenados na área de textura.

Um programa em CUDA C, desta forma, deve ser projetado de forma que as *threads* de um bloco possam ser executadas em paralelo sem que haja dependência de dados, acessando as áreas de memória adequadas. O trecho de programa 3.1 [80] representa de forma resumida o código CUDA C para soma de duas matrizes (A e B) de dimensões $N \times N$, com resultado armazenado na matriz C. A função paralelizada é definida nas linhas 2 a 7, e a chamada do *kernel* em GPU está na linha 17.

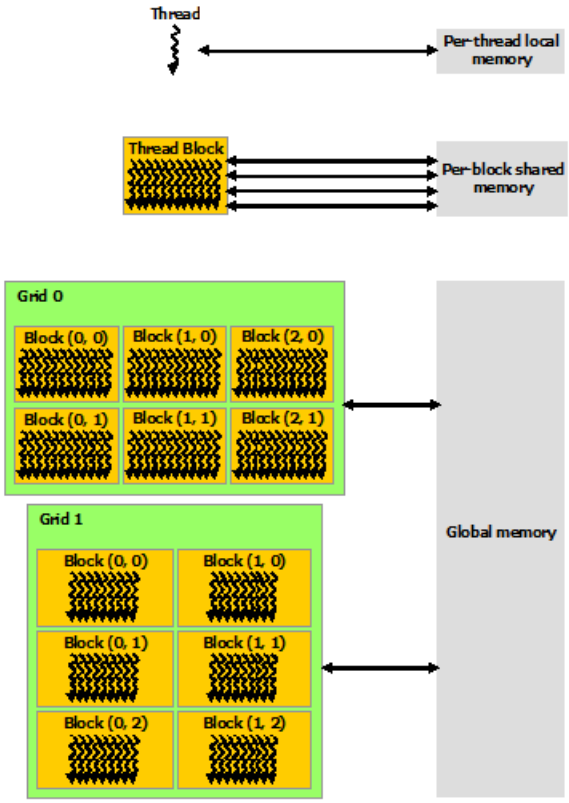


Figura 3.4: Hierarquia de memória CUDA [85].

Programa 3.1 Exemplo de código em CUDA C [80].

```
 1: // Definição do Kernel que executa a soma, utilizando modificador __global__
 2: __global__ void MatAdd(floatA[N][N], floatB[N][N],floatC[N][N])
 3: {
 4:    inti = threadIdx.x;
 5:    intj = threadIdx.y;
 6:    C[i][j] = A[i][j] + B[i][j];
 7: }
 8:
 9: // Definição da função principal
10: int main()
11: {
12: // ...
13: // Invocação da função Kernel com um bloco contendo N * N * 1 threads
14: // Todas as threads do bloco serão executadas em paralelo
15: int numBlocks = 1;
16: dim3 threadsPerBlock(N, N);
17: MatAdd<<<numBlocks, threadsPerBlock>>>(A, B, C);
18: // ...
19: }
```

3.2.2 Ambiente OpenCL

O OpenCL é um *framework* para programação paralela em ambiente heterogêneo desenvolvido pelo Khronos Group [37]. A solução permite que um mesmo código-fonte possa ser executado em diferentes plataformas de *hardware*, possibilitando que o desenvolvedor

defina explicitamente a plataforma, o contexto e a distribuição do trabalho no *hardware* disponível, explorando modelos de paralelismo de dados (quando o mesmo conjunto de instruções é aplicado a um conjunto de dados concorrentemente), paralelismo de tarefas (quando o programa é dividido em um conjunto de tarefas que podem ser executadas concorrentemente) ou híbrido. O funcionamento do OpenCL pode ser mais bem entendido avaliando-se os modelos propostos pelo *framework*: plataforma, execução e memória.

O OpenCL representa o *hardware* onde a aplicação é executada por um modelo de plataforma [76], onde o *host* é conectado a um ou mais dispositivos OpenCL (como uma CPU ou GPU, por exemplo), e estes são divididos em unidades computacionais, que são conjuntos de elementos de processamento onde a execução efetivamente ocorre.

Quanto ao modelo de execução, a arquitetura OpenCL subdivide uma aplicação em duas partes: um programa *host*, executado no *host*; e uma coleção de *kernels*, funções escritas na linguagem OpenCL C ou em linguagem nativa do *host*, que são executadas nos dispositivos OpenCL [76]. Em relação à arquitetura CUDA (Seção 3.2.1), observa-se a ausência da área de memória de textura, o que se justifica, uma vez que esta área é restrita a GPUs, e o OpenCL possui como requisito a portabilidade para vários tipos de dispositivos.

A Figura 3.5 ilustra as etapas requeridas para a execução de um programa OpenCL em um dado dispositivo de uma plataforma. Em síntese, as seguintes etapas devem ser seguidas na criação de um programa em OpenCL [76]:

- ◇ Inicialmente, realiza-se uma consulta ao *host* sobre as plataformas disponíveis;
- ◇ Dentre as plataformas retornadas, o identificador (*id*) da plataforma desejada (CPU, GPU ou outra) é selecionado;
- ◇ Para a plataforma selecionada, deve-se consultar os dispositivos existentes;
- ◇ O *id* do dispositivo desejado é então armazenado;
- ◇ A seguir, é solicitada a criação do contexto, informando o *id* do dispositivo;
- ◇ Ponteiros com a informação do contexto criado e o descritor do arquivo contendo o código a ser executado são passados como parâmetros na instrução de criação de programa, retornando um ponteiro que servirá como entrada para a instrução de compilação do programa;
- ◇ São criadas, então, referências para as funções *kernel* do programa compilado;
- ◇ Após a execução de instruções de criação de *buffer* e fila de comandos, os dados de execução são finalmente colocados na fila para a execução paralela.

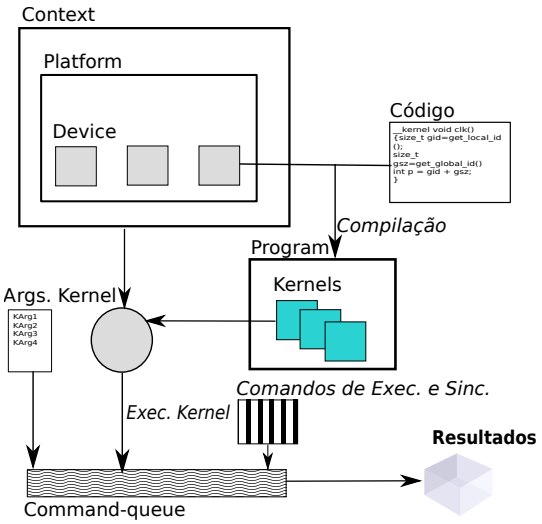


Figura 3.5: Esquema da execução de programa OpenCL em um dispositivo.

3.2.3 Paralelização em CPUs: *pthreads*

Uma das formas mais usuais de divisão da aplicação em unidades independentes é através do uso de *threads* [114]. Sob o ponto de vista do programador, a *thread* é criada em um procedimento, que é executado de forma concorrente ao programa principal. Desde que respeitadas as dependências, dados podem ser distribuídos em diferentes *threads*, que podem ser alocadas a núcleos distintos de uma CPU para permitir o paralelismo.

Uma das interfaces de programação que exploram este recurso são as *Posix Threads* (*pthreads*) [9], que surgiram a partir de uma iniciativa para padronização na forma de implementar as *threads* em sistema Unix na década de 1990. As *pthreads* são criadas através de uma chamada ao pacote, tendo como parâmetro o procedimento a ser executado por ela. Uma vez criada, a *pthread* tem acesso a todas as variáveis globais do processo, bem como acesso privado a suas variáveis locais. Possui também uma pilha de execução, que permite a chamada a outros procedimentos. Para a sincronização entre as *threads*, o pacote *pthreads* oferece *locks* e variáveis de condição [9]. Como o programador controla o momento de criação e término de cada *pthread* e é também responsável pela sincronização, é dito que o pacote *pthread* é uma API que usa paralelismo de baixo nível (*low level* API).

Ressalte-se que as chamadas de funções *kernels* CUDA ou OpenCL são invocadas a partir de uma *thread* em CPU e, por isso, o uso de múltiplas *threads* em CPU é muito frequente no desenvolvimento de soluções em GPU.

Capítulo 4

Escalonamento de Tarefas

Dentre os aspectos envolvidos na execução em múltiplos dispositivos, a distribuição da carga de trabalho entre as unidades de processamento existentes possui grande importância. Basicamente, o desafio consiste em criar uma política para alocar as tarefas de maneira a aproveitar a capacidade de processamento, buscando um melhor desempenho, um melhor aproveitamento dos dispositivos ou um menor consumo de energia, por exemplo.

O presente capítulo aborda alguns conceitos inerentes ao escalonamento de tarefas, discutindo características que devem ser avaliadas no projeto de uma solução que funcione em ambiente uniforme ou híbrido. A Seção 4.1 apresenta o padrão de nomenclatura adotado no capítulo. A Seção 4.2 apresenta a definição de escalonamento e conceitos relacionados. A Seção 4.3 apresenta a classificação de Lawer et. al para escalonamento de tarefas. Os tipos de aplicações paralelas são apresentados na Seção 4.4, e a Seção 4.5 apresenta alguns algoritmos de escalonamento de tarefas. Finalmente, a Seção 4.6 discorre sobre distribuição de carga.

4.1 Padrão de Nomenclatura Adotada

Os termos envolvidos em um sistema de escalonamento podem ter diferentes nomenclaturas dependendo do trabalho avaliado. A fim de uniformizar a abordagem neste Tese, as seguintes definições serão utilizadas neste capítulo:

- **Processador (*processor*):** qualquer elemento de processamento ou recurso computacional que possa ser utilizado na execução de uma tarefa;
- **Tarefa (*task*):** atividade única que pode ser alocada a único processador em um dado instante;

- **Trabalho (*job*):** conjunto de uma ou mais tarefas dependentes ou independentes que compõem a aplicação paralela;
- **Escalonamento (*scheduling*):** algoritmo que distribui o conjunto de tarefas entre os processadores de acordo com uma política definida.

Todos os trabalhos discutidos no presente capítulo serão descritos em termos de tarefas, trabalhos e processadores, segundo essa nomenclatura, mesmo que outros termos sejam utilizados nos artigos. Caso algum trabalho abordado neste capítulo adote uma nomenclatura diferente da citada acima, os termos utilizados originalmente serão citados no detalhamento, mas os termos definidos na uniformização serão usados preferencialmente.

4.2 Definições e Conceitos

O problema de escalonamento de processadores foi inicialmente definido na década de 1960, da seguinte maneira [35]. Existem n unidades de processamento (ou processadores) P_i $(1 \leq i \leq n)$ e um conjunto $T = T_1, ..., T_m$ de m tarefas a serem processadas pelas unidades de processamento P_i. Existe também uma ordem parcial (*partial order*) $\prec$, que determina as precedências temporais entre as tarefas. A ordem parcial $\prec$ deve ser respeitada, isto é, se $T_i \prec T_j$ então T_j só pode iniciar sua execução quando T_i terminar de executar.

Apesar de os primeiros artigos terem definido a precedência de tarefas em termos de ordens parciais, logo se convencionou representar o conjunto de tarefas como um grafo $G = (E, V)$, onde as arestas E representam a ordem de precedência e os vértices V representam as tarefas. Opcionalmente, o tempo de execução t_i de cada tarefa T_i, quando for conhecido, pode ser representado entre parênteses ao lado da tarefa. A Figura 4.1 apresenta um grafo de cinco tarefas (T_1, T_2, T_3, T_4 e T_5), com tempos de execução 2, 2, 1, 3 e 1, respectivamente, onde as seguintes ordens parciais devem ser respeitadas: $T_1 \prec T_2$, $T_1 \prec T_4$, $T_2 \prec T_3$, $T_4 \prec T_5$, $T_3 \prec T_5$.

O objetivo do escalonamento é otimizar uma função de custo (ou função objetivo), que normalmente leva em consideração o tempo de execução do conjunto de tarefas. Uma função de custo muito usada é o *makespan* (ω), que mede o tempo desde o início da primeira tarefa até o término da última tarefa. No exemplo da Figura 4.1, podem ser observadas duas linhas de execução, com tempos t=2+2+1+1=6 e t'=2+3+1=6; logo, o valor do *makespan* mínimo é ω=6.

A atividade de cada processador P_i no tempo é geralmente representada por um diagrama de Gantt, onde cada processador é representado no eixo y e o tempo é representado no eixo x [35]. Existe uma linha para cada processador, onde são colocadas as tarefas

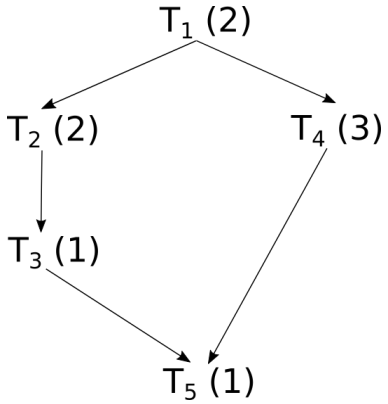


Figura 4.1: Exemplo de grafo de precedência entre tarefas.

executadas por cada P_i e seus respectivos tempos de execução. A Figura 4.2 ilustra um diagrama de Gantt onde dois processadores P_1 e P_2 executam o grafo ilustrado na Figura 4.1. Nos casos em que um processador não possui tarefa a executar, é adicionada a palavra *idle* no diagrama, que equivale a dizer que o processador está disponível ou executando uma tarefa vazia.

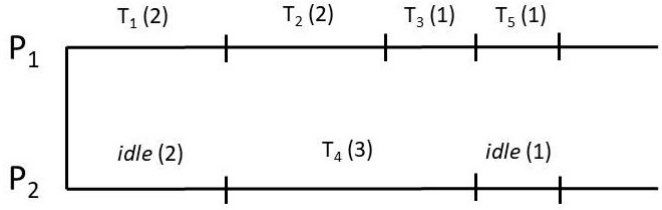


Figura 4.2: Exemplo de diagrama de Gantt para escalonamento de tarefas.

Um dos primeiros algoritmos de escalonamento de tarefas proposto se baseia no conceito de listas de execução [35]. Para tanto, inicialmente constrói-se uma lista L de tarefas $T = T_1, ..., T_m$, chamada lista de prioridades. No momento t quando um processador P_i termina de executar uma tarefa, ele consulta a lista L, do início ao final, e executa a primeira tarefa T_j que não começou a ser executada e cujos predecessores (i.e., todo $T_i \prec T_j$) já terminaram a execução antes de t. Se o processador P_i termina de consultar a lista e não acha nenhuma tarefa, P_i fica ocioso (ou, no presente algoritmo, P_i executa uma tarefa vazia – *empty task*). Neste caso, P_i fica ocioso até que um processador P_j complete a execução de alguma tarefa T_i. Nesse momento, P_i e P_j consultam a lista à busca de tarefas a serem executadas. Se P_i e P_j tentam executar a mesma tarefa, convencionou-se que o processador com menor índice irá executá-la. O algoritmo continua até que todas as tarefas sejam completadas (executadas até o final).

Com esse algoritmo, foram obtidos alguns resultados importantes em termos de limites superiores. O primeiro resultado diz que, se existem dois processadores P_1 e P_2, e duas listas de tarefas L e L', a razão entre o *makespan* mínimo de L (ω) e de L' (ω') respeita as seguintes inequações: $\frac{2}{3} \leq \frac{\omega'}{\omega} \leq \frac{3}{2}$ [35]. Adicionalmente, se forem tomados dois grafos de execução G e G', tais que as precedências de G' estão contidas nas precedências de G, e os tempos de execução das tarefas em G' são menores ou iguais que os respectivos tempos de execução em G, tem-se que $\frac{\omega'}{\omega} \leq 1 + \frac{n-1}{n'}$, onde n e n' são os números de processadores em G e G', respectivamente.

4.3 Classificação de Lawer et al.

Em um trabalho relevante na área de escalonamento, Lawer et al. [54] propõem em 1993 uma notação de três campos $(\alpha|\beta|\gamma)$ para classificar os problemas de escalonamento, onde α representa as características do ambiente computacional, β representa as características das tarefas e γ representa características da função objetivo.

Quanto ao ambiente (α), existem as seguintes possibilidades:

- $\alpha = 1$: processador único;
- $\alpha = P$: processadores paralelos (p_i) idênticos, ou seja, existem m processadores idênticos e o tempo de execução da tarefa j não depende do processador na qual ela está alocada ($p_{ij} = p_j$);
- $\alpha = Q$: processadores paralelos uniformes ou relacionados; m processadores com velocidades s_1, ..., s_m; o tempo de processamento da tarefa T_j no processador P_i é $p_{ij} = p_j/s_i$;
- $\alpha = R$: processadores paralelos não relacionados; m processadores com velocidades s_1, ..., s_m; tempos arbitrários de processamento $p_{ij} = p_j/s_{ij}$;
- Alternativamente, pode-se ter $\alpha = P_m$ ou Q_m ou R_m, quando o número de processadores é fixado em m.

Quanto às tarefas (β), temos as seguintes opções:

- r_j = *release date*: a tarefa não pode iniciar sua execução antes do tempo r_j;
- *pmtn* = preempção: a tarefa pode ser interrompida e sua execução pode ser retomada em outro processador;

- $prec$ = existem relações de precedência. Nesta categoria, existem alguns tipos particulares, tais como *chain* (encadeamento linear), *intree* (quando o número de arestas que saem - *outdegree* - é no máximo 1) e *outtree* (quando o número de arestas que chegam – *indegree* - é no máximo 1). O grafo de bifurcação (*fork graph*) é um exemplo de *outtree*, enquanto o grafo de junção (*join graph*) é um exemplo de *intree*.
- d_j: as tarefas possuem *deadlines* e devem terminar sua execução em tempo menor ou igual a este tempo máximo.

Finalmente, quanto à função objetivo (γ), são feitas inicialmente algumas definições:

- C_j: tempo de término da tarefa;
- $F_j = C_j - r_j$: tempo de fluxo (*flow time*) da tarefa;
- $L_j = C_j - d_j$: atraso (*lateness*) da tarefa;

Com estas definições, algumas funções objetivo são introduzidas:

- C_{max}: função que contabiliza o máximo tempo de execução das tarefas ou *makespan*;
- F_{max}: função que maximiza o *flow time*, ou seja, $F_{max} = max_j(F_j)$;
- L_{max}: função que maximiza o *lateness*, ou seja, $L_{max} = max_j(L_j)$;
- são definidas também variações das funções anteriores considerando um peso (funções *weighted*).

Para exemplificar, o algoritmo proposto em [35] é classificado, segundo Lawer et al., como $P|prec|C_{max}$. Essa classificação continua sendo utilizada nos dias de hoje em diversos trabalhos teóricos de escalonamento, tais como [2] e [49].

Outra taxonomia que foi muito utilizada até o início da década de 2000 é a de Casavant e Kuhl [12]. Nessa taxonomia, o escalonamento de tarefas é definido como um sistema de gerenciamento de recursos (processadores), que consiste basicamente em um mecanismo ou política usada para gerenciar de maneira eficiente e efetiva o acesso e a utilização de um recurso pelos seus vários consumidores (tarefas). Em um ambiente com múltiplos dispositivos, a capacidade de processamento pode ser obtida basicamente por duas estratégias: 1) características do *hardware*, como quantidade e velocidade dos núcleos, velocidade/capacidade do barramento ou desempenho teórico dos núcleos, podem ser utilizadas como referência para estimar o tempo de execução; ou 2) uma execução prévia com parte dos dados pode ser realizada para se obter um padrão de processamento (*profiling*) dos dispositivos.

4.4 Tipos de Aplicações

As tarefas que compõem a aplicação paralela (ou *job*) podem ser organizadas de diversas maneiras. A seguir, serão apresentadas as organizações mais comuns [72] [118] [16]:

- **Tarefas independentes (*Bag of tasks*):** ocorre quando o trabalho pode ser dividido em tarefas independentes, que podem ser processadas individualmente, requerendo apenas a disponibilidade do recurso. Quando todas as tarefas executam o mesmo programa, porém com diferentes parâmetros de entrada, o termo "*parameter sweep*" é algumas vezes utilizado [11];

- **Tarefas com dependência de entrada/saída (*workflow*):** neste cenário, ocorre dependência temporal entre as tarefas, e o início de uma execução pode precisar ser retardada até uma outra tarefa da qual ela depende seja finalizada. Neste caso, usualmente um grafo é construído para representar as dependências do tipo início-fim entre as tarefas, conforme Figura 4.1. O padrão de dependência pode variar de situações mais simples (como em uma relação linear ou circular) para interações mais complexas;

- **Tarefas comunicantes:** algumas aplicações podem requerer que as tarefas troquem com certa frequência informações entre si. Padrões de comunicação neste caso usualmente são representados de maneira linear, onde uma tarefa se comunica com outra à direita ou em uma malha, onde cada tarefa pode se comunicar com 4 ou 8 vizinhos, por exemplo.

Nos dois últimos casos citados acima, a dependência entre tarefas pode seguir um padrão regular ou irregular. O padrão irregular é geralmente produzido de maneira dinâmica, ao longo da execução, o que complica significativamente o escalonamento.

4.5 Algoritmos de Escalonamento

Computacionalmente, o problema de escalonamento de tarefas é um problema NP-Completo [87]. Sendo assim, a busca de heurísticas que permitam uma boa alocação de tarefas a processadores e que consigam atingir os objetivos do escalonamento no caso médio tem sido priorizada.

Muitos algoritmos vêm sendo propostos para realizar o escalonamento. Um dos algoritmos mais simples é o *Round-Robin* [90], onde as tarefas cujas dependências já foram recebidas são armazenadas em uma fila circular, e a primeira delas é alocada a um processador durante uma fatia de tempo (*quantum*). Caso a tarefa não seja finalizada neste

intervalo, ocorre uma preempção, e esta tarefa é colocada no final da fila. A seguir, a primeira tarefa da fila é selecionada para ocupar o processador.

Já na estratégia *Largest-Job-First* (LJF) [56], as tarefas aguardando execução são ordenadas pelo tamanho, sendo que as maiores possuem maior prioridade de execução. Uma abordagem oposta - mas semelhante - é o *Shortest-Job-First* (SJF), onde as menores tarefas são executadas primeiro. Em ambos os casos, contudo, pode ocorrer subutilização dos processadores.

O algoritmo *Earliest-Finish-Time* (EFT) [48], por outro lado, se baseia na alocação das tarefas aos processadores respeitando-se as precedências e priorizando as tarefas que possuem menor tempo de finalização esperado. A Figura 4.3 mostra um grafo de tarefas (Figura 4.3a) e seu escalonamento para dois processadores seguindo o algoritmo EFT (Figura 4.3b). A tarefa T_1 possui três tarefas dependentes: T_2, T_3 e T_4. Dentre estas, as tarefas T_2 e T_4 são selecionadas primeiro para processamento, pois possuem menor duração.

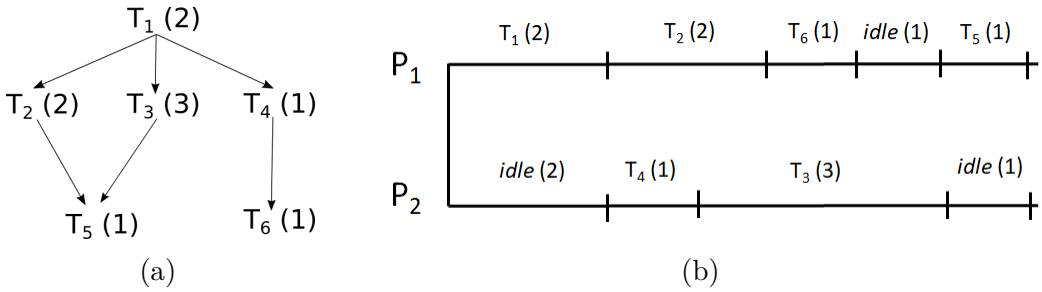


Figura 4.3: Exemplo de escalonamento com algoritmo EFT: (a) grafo de tarefas e (b) distribuição entre dois processadores.

Na variação do EFT para ambiente heterogêneo (HEFT - *Heterogeneous Earliest-Finish-Time*) [116], as tarefas são previamente ordenadas a partir de um *rank* calculado com base no custo médio de computação e de comunicação, e as tarefas são alocadas de acordo com este critério para um processador que minimiza o tempo de finalização da tarefa.

O algoritmo *Backfilling* [23] [74] provê uma modificação em relação ao algoritmo que processa as tarefas na mesma ordem em que chegam (*First Come First Served* - FCFS): o escalonador estima o tempo de execução quando uma tarefa que está na lista em espera é alocada, e busca localizar outras tarefas que possam ser executadas em paralelo com a selecionada, desde que haja processadores disponíveis. As tarefas com menor tempo de execução são então passadas à frente na lista de execução, visando preencher os "buracos" que podem ser criados durante o processo de alocação. Esta abordagem geralmente possui uma alocação mais eficiente que as anteriores, mas pode ser impactada se existe uma latência alta entre os processadores disponíveis. Aprimoramentos nesta estratégia

foram propostos por [69], aplicando restrições de contiguidade e localidade para reduzir a fragmentação das tarefas.

Já o algoritmo *Gang Scheduling* [22] é aplicável em *jobs* paralelizáveis que possuem diversas tarefas, escalonadas em processadores distintos de forma simultânea. Se o número de tarefas em todos os *jobs* excedem a quantidade total de processadores, um esquema de fatia de tempo é adotado. O processamento é coordenado, de forma que se houver uma troca de contexto, todas as tarefas de um mesmo *job* são retiradas dos processadores ao mesmo tempo, independentemente da quantidade de processadores que o novo *job* precise.

Por fim, cabe destacar que o procedimento de alocação de tarefas pode ser dificultado em ambientes híbridos, devido às características diferentes dos processadores. Algumas tarefas, por exemplo, podem ser incompatíveis com algum tipo de processador (como GPU ou FPGA); logo, esta afinidade deve ser considerada pelo algoritmo. Além disso, algoritmos que requerem informação prévia do tempo de execução vão exigir estimativas diferentes para cada tipo de processador.

4.6 Distribuição de Carga

O balanceamento de carga (*load balancing*) consiste em se distribuir as tarefas de maneira uniforme entre os processadores, de forma que todos os processadores tenham a mesma carga [115]. O termo distribuição de carga (*load sharing*) oferece uma definição mais relaxada, pois não exige que os processadores tenham a mesma carga, mas permite que possuam cargas diferentes, desde que nenhum processador esteja saturado (*overloaded*) [121]. No presente capítulo e ao longo da Tese, o termo distribuição de carga será utilizado, pois este conceito engloba o balanceamento de carga.

Em linhas gerais, a distribuição de carga pode ser estática ou dinâmica [12]. Na distribuição estática, as tarefas que compõem o grafo da aplicação são alocadas aos processadores antes do início da execução da aplicação, levando em conta uma distribuição que não gere processadores saturados. A distribuição dinâmica de carga pode ocorrer de duas maneiras: sem ou com reatribuição. Na distribuição dinâmica sem reatribuição, as tarefas são alocadas aos processadores quando suas dependências são resolvidas, considerando o grafo da aplicação e levando em conta as cargas dos processadores neste momento. Uma vez iniciada a execução, a tarefa executa-se no mesmo processador até o seu término. Na distribuição dinâmica com reatribuição, as tarefas também são criadas quando suas dependências são resolvidas, levando em conta a carga atual dos processadores. No entanto, caso o processador origem se torne saturado, pode ocorrer preempção nas tarefas, e elas podem ser migradas para um outro processador [12].

Em uma das formulações mais simples para a distribuição de carga, o sistema possui um conjunto de P processadores conectados por uma rede de *broadcast*. Todas as tarefas são independentes (Seção 4.4) e do mesmo tipo, e chegam aos nodos com a mesma taxa média. As tarefas são criadas localmente se o processador não tiver atingido o limite (*threshold*) de saturação. Caso o processador esteja saturado, a tarefa é transferida e criada em outro processador. A escolha do processador destino é feita segundo uma política de distribuição de carga. Utilizando-se políticas simples de distribuição de carga, foi mostrado que tanto o tempo de espera da tarefa na fila de entrada como o *makespan* (Seção 4.2) são reduzidos [63].

O trabalho [111] definiu o problema da distribuição de carga considerando-se tarefas e servidores (processadores). O problema abordado consiste em distribuir n tarefas entre m processadores segundo um critério, onde cada processador só pode processar uma tarefa por vez, e cada tarefa executa de forma ininterrupta em um dos processadores. Quando os tempos de chegada e saída das tarefas são conhecidos previamente, este procedimento pode ser realizado estaticamente, sendo definida uma alocação estática de tarefas para os processadores. Contudo, usualmente apenas é conhecido um peso para uma determinada tarefa, mas não sua duração. Este cenário requer uma distribuição dinâmica ("*on-line*"), em que o problema consiste em atribuir cada tarefa a um processador apropriado de forma que todos os processadores executem cargas similares [4].

No modelo de distribuição dinâmica de carga proposto por Azar [4], uma tarefa é alocada a um dos processadores capazes de executá-la no momento que chega, e não é possível transferi-la para outro servidor até o seu encerramento, ou seja, sem preempção segundo o modelo de Lawer et al. (Seção 4.3), e sem reatribuição segundo Casavant e Kuhl. Um algoritmo guloso atribui uma tarefa recém chegada ao processador com menor carga no momento para obter um limite superior. Na definição do problema dinâmico ótimo, a ordem de chegada e a duração das tarefas são inteiramente conhecidas, e elas são alocadas de forma a minimizar o custo. Ainda nesse caso, trata-se de um problema NP-Completo [4].

Para avaliar a distribuição dinâmica de carga, o autor classifica as tarefas quanto à sua perenidade entre temporárias (que possuem tempos de início e fim definidos) e permanentes (que iniciam em tempos arbitrários e continuam indefinidamente), e quanto à sua restrição ou não de execução em determinados processadores entre restritas e irrestritas. Os processadores verificam seu desempenho através da relação entre as cargas máximas das execuções estática e dinâmica para o pior caso de ordem de tarefas. Os autores apresentam alguns limites de execução de algoritmos gulosos para tarefas irrestritas $(2 - \frac{1}{n})$ e para tarefas restritas e permanentes $(log(n))$, onde n é a quantidade de tarefas. Para tarefas restritas e temporárias, contudo, o limite obtido é bem elevado: $\frac{(3n)^{2/3}}{2}(1 + o(1))$.

Em um trabalho mais recente, Buchbinder e Naor [8] reavaliam a distribuição dinâmica de carga para tarefas restritas em um modelo específico denominado $1-\infty$, onde todas as tarefas possuem o mesmo peso, que é uma unidade. Os autores avaliaram a justiça (*fairness*) da distribuição centrada na máquina (*machine-centric*) e na tarefa (*task-centric*). Na abordagem *fair*, busca-se maximizar a banda de tráfego dada à tarefa alocada ao processador com maior carga de trabalho. O algoritmo guloso utilizado para distribuição das tarefas é o mesmo utilizado por [4]. As medidas de desempenho utilizadas foram minimizar a carga máxima na avaliação *machine-centric* e maximizar a utilização de banda do *job* mais lento na abordagem *task-centric*. Os autores informam que, dentre as estratégias analisadas, a estratégia gulosa é a melhor abordagem *on-line* com respeito à justiça global. O algoritmo adotado possui complexidade de $O(log(p))$ baseando-se no critério de *fairness*, onde p indica o número de processadores.

Capítulo 5

Trabalhos Relacionados

Soluções paralelas para o problema de comparação de sequências biológicas que utilizam o algoritmo SW e suas variantes (Seção 2.1) vêm sendo propostas na literatura há décadas. Em um primeiro momento, foram propostas soluções que exploravam o paralelismo em uma única máquina, com diferentes tipos de plataformas de *hardware*, tais como as CPUs [31] [89] [5] [94] [20], FPGAs [40] [86] [128] [10] [97], GPUs [57] [58] [47] [91] [103] [51] [25] e o coprocessador Intel Xeon Phi [120]. Dentre essas, as GPUs são uma plataforma muito utilizada na implementação das soluções paralelas, aproveitando a grande quantidade de núcleos presentes nestes dispositivos. Em um segundo momento, e de maneira a aumentar ainda mais o desempenho, as soluções para o problema da comparação de sequências biológicas adotaram múltiplas máquinas, em plataformas uniformes homogêneas, uniformes heterogêneas ou híbridas.

O presente capítulo aborda inicialmente algumas das soluções mais relevantes que utilizam uma ou mais GPUs (Seção 5.1) em máquina única, uma vez que esta é a plataforma utilizada nas soluções propostas nesta Tese. Adicionalmente, são apresentadas e comparadas soluções que foram projetadas para múltiplos dispositivos (Seção 5.2), com foco nas que apresentaram os melhores desempenhos em cada plataforma de *hardware*. Finalmente, trabalhos que tratam de distribuição de carga em soluções de comparação de sequências são apresentados na Seção 5.3. Para comparação dos desempenhos das soluções apresentadas neste capítulo é usada a métrica GCUPS (Seção 2.5).

5.1 Comparação de Sequências com GPU em uma Única Máquina

Nas últimas décadas, as GPUs vêm sendo muito utilizadas para a execução de algoritmos paralelos de comparação de sequências biológicas. Devido à grande concentração

de núcleos, o processamento em GPUs permite muito paralelismo, com alto desempenho. Nas Seções 5.1.1 a 5.1.8 serão apresentadas soluções de comparação/alinhamento de sequências em GPUs instaladas em máquina única. A Seção 5.1.9 apresenta uma tabela comparativa e a discute.

5.1.1 Ligowski e Rudnicki (2009)

A solução de Ligowski e Rudnicki [57] implementa o algoritmo de Gotoh (Seção 2.2.3) em plataforma CUDA (Seção 3.2.1) para comparação de uma sequência *query* de proteína com um banco de dados (Seção 2.1).

Inicialmente, as sequências do banco de dados são ordenadas pelo tamanho (maiores sequências primeiro) e organizadas em blocos, de maneira a melhorar a distribuição da carga (Seção 4.6). A sequência *query* é comparada através de *kernels* CUDA com diferentes entradas do banco de sequências, retornando o maior escore produzido por cada comparação. O paralelismo é de grão grosso (Seção 2.4), onde cada *thread* da GPU compara a sequência *query* com uma sequência diferente do *database*, de forma independente. A solução retorna somente o escore ótimo de cada comparação.

Os testes foram realizados na placa gráfica NVidia 9800 GX2, que possui duas GPUs integradas e é capaz de executar 768 *threads* concorrentemente. As sequências testadas são um subconjunto da base de sequências Swiss-Prot 56.5, limitando-se o tamanho da sequência a cerca de 1.000 aminoácidos, gerando um subconjunto de 388.517 proteínas ordenadas de forma decrescente em relação ao tamanho. O melhor desempenho foi obtido na comparação de sequências mais longas e utilizando as duas GPUs integradas, atingindo 14,5 GCUPS. Os autores mostraram que esse resultado está próximo ao limite teórico e consideram que o bom desempenho da sua abordagem se deve ao acesso à memória global apenas na inicialização do laço, uso intensivo de registradores e da utilização de 4.096 *threads* CUDA.

5.1.2 CUDASW++ (2009, 2010)

O CUDASW++ implementa o algoritmo Gotoh (Seção 2.2.3) em GPUs para comparação de proteínas. Na versão 1.0 [58], a paralelização é obtida através da criação de várias tarefas, e em cada uma delas a sequência de referência é comparada com uma sequência do banco de sequências, também ordenadas pelo tamanho. A paralelização é otimizada através de dois métodos: intertarefas e intratarefas, com paralelismo de grão grosso e fino, respectivamente.

Na abordagem intertarefas, cada comparação é realizada por apenas uma *thread*. Um conjunto de 256 *threads* é agrupado em um bloco, sendo possível processar tantos blocos

em paralelo quanto o número de multiprocessadores existentes na GPU utilizada. Esta abordagem é voltada a sequências menores.

Na metodologia intratarefas, o bloco de 256 *threads* é responsável pela execução da tarefa, e as *threads* cooperam durante a execução. A matriz de programação dinâmica é processada de forma paralela em sua diagonal - técnica de *wavefront* (Seção 2.4). Otimizações também foram realizadas na forma de acesso à memória global, de forma que sucessivas *threads* acessem áreas contíguas da memória, padrão de acesso denominado *coalesced*.

Foram analisadas 25 sequências de busca, com tamanhos entre 144 e 5.478 aminoácidos, e o banco de sequências Swiss-Prot 56.6. Nas sequências com até 3.072 aminoácidos, o método intertarefas foi utilizado, enquanto o método intratarefas foi implementado nas comparações de sequências maiores. Duas GPUs foram utilizadas nos testes: GTX 280 e GTX 295 (dual). Os máximos desempenhos obtidos foram 9,66 GCUPS, na placa GTX 280, e 16,09 GCUPS, na placa GTX 295.

A versão 2.0 do CUDASW++ [60] apresenta duas implementações do algoritmo Gotoh (Seção 2.2.3): uma otimização do modelo intertarefas e uma variação do padrão de processamento em tiras (*striped* [21]). Na primeira abordagem, duas modificações foram realizadas no algoritmo inicial: uma abordagem *sequential query profile*, onde os valores da matriz de substituição (Seção 2.2) referentes à sequência de referência são pré-calculados antes da busca na base de sequências; e *packed data format*, onde cada sequência representada numericamente é armazenada em um tipo vetor `uchar4`. Na segunda técnica, foi proposta uma variante do modelo *striped* sugerido por Farrar [21], além de particionar as sequências e armazená-las em área de memória de textura da GPU (Seção 3.2.1). Desta forma, são armazenados nesta área de memória os escores obtidos na comparação da sequência de busca (*query*) contra todos os possíveis símbolos do alfabeto (para proteínas, 20 caracteres - Seção 2.1), e eles são lidos em conjuntos de quatro caracteres para se ajustar ao tamanho do barramento da memória (32 bits).

As mesmas GPUs utilizadas nos testes da versão 1.0 foram escolhidas para os testes da nova solução, atingindo-se como desempenhos máximos: 16,9 GCUPS (GTX 280) e 28,8 GCUPS (GTX 295), para a implementação da intertarefas otimizada; e 17,8 GCUPS (GTX 280) e 29,9 GCUPS (GTX 295), para a variante do modelo *striped*.

5.1.3 SW 2.0 (2010)

O algoritmo Gotoh (Seção 2.2.3) também é utilizado como base na solução SW 2.0 [47], desenvolvida em OpenCL, que realiza a comparação de uma sequência de referência de proteína com sequências existentes no banco. A paralelização é obtida fazendo com que cada *thread* (implementadas em OpenCL como *work-itens* - Seção 3.2.2) calcule um

alinhamento ótimo entre a sequência *query* e uma das sequências da base de sequências. A paralelização ocorre, portanto, com estratégia de grão grosso.

Duas otimizações utilizadas na solução CUDASW++ 2.0 (Seção 5.1.2) também são usadas no SW 2.0: o padrão de acesso à memória global *coalesced*, que neste trabalho é realizado por todas as *threads* de um mesmo *work-group* OpenCL; e o uso da técnica de *sequential query profile*.

Os testes foram realizados nas GPUs NVidia 9800GT e AMD HD 5850, obtendo-se melhor desempenho de aproximadamente 66,0 GCUPS no segundo modelo.

5.1.4 Razmyslovich et al. (2010)

A solução proposta por Razmyslovich et al. [91] é um outro exemplo de comparação de sequências utilizando OpenCL. Apesar de trabalhar com sequências de DNA, a ferramenta foi testada apenas com uma sequência longa (cromossomo 21 humano, mapeado na época com cerca de 28 milhões de pares de bases), que serve como base para a busca de um conjunto de sequências com apenas 36 nucleotídeos, realizada de forma paralela. Logo, o problema tratado é a comparação de um conjunto de sequências muito pequenas (*short reads*) com uma sequência de referência muito grande. O número máximo de sequências do conjunto utilizado no trabalho é 600.

Para obter uma melhor eficiência, os dados e *threads* foram organizados para comportar o modelo de *work-group* do OpenCL, adotando o processamento em *wavefront*, com os dados organizados em um paralelogramo. Neste caso, o uso de uma sequência muito pequena acaba simplificando a técnica.

A aplicação foi modularizada em oito subprocessos: inicialização do *host*, transferência da entrada de dados, agendamento de execução, *kernel* de pré-cálculo, *kernel* de cálculo, transferência da matriz para a memória do *host*, cálculo de caminhos e impressão de resultados. Um estudo em relação ao tempo de execução de cada subprocesso foi realizado, tendo sido identificados os estágios com maior impacto no desempenho e que podiam ser paralelizados. Para tanto, a solução prevê uma forma de executar simultaneamente as tarefas de transferência de dados e execução do *kernel* através de um *buffer* em anel alocado na memória do dispositivo, composto por três janelas: duas usadas para cálculo da matriz e uma contendo os dados que podem ser transferidos.

Duas versões da solução foram propostas e testadas em uma GPU NVidia GTX 260: uma que informa apenas o escore ótimo e outra que retorna também o alinhamento. A versão que provê apenas o escore calculado obteve desempenho compatível ou superior a outras abordagens, possibilitando a pesquisa concorrente de até 600 sequências contendo 36 nucleotídeos na sequência de referência do cromossomo 21 humano (28 MBP). A versão que provê um dos alinhamentos ótimos calculados foi testada com 40 comparações simul-

tâneas. O trabalho não informa explicitamente os resultados de desempenho obtidos, mas a partir dos gráficos com o tempo de execução, pode-se calcular um GCUPS aproximado de 0,112 (28.000.000 de células / 250 ms).

5.1.5 CUDAlign 1.0, 2.0 e 2.1 (2010, 2011, 2013)

A comparação de duas sequências longas de DNA é feita no CUDAlign através do algoritmo Gotoh, executando em GPU. A versão 1.0 do CUDAlign [103] produz o escore ótimo e as suas coordenadas na matriz de programação dinâmica. Para tanto, as células da matriz são agrupadas em blocos, que são processados diagonalmente em um padrão de paralelogramo adaptado do modelo *wavefront* (Seção 2.4).

O CUDAlign propôs o processamento em paralelogramo, que maximiza o paralelismo. Duas técnicas foram propostas para permitir a execução em paralelogramos, pois existirão células pendentes nas bordas esquerda e direita da matriz. Na delegação de células (Figura 5.1), as células pendentes de processamento em um bloco são processadas por outro bloco na próxima diagonal, o que permite o máximo paralelismo entre as *threads*, em um aprimoramento do processamento em *wavefront*. Além disso, uma subdivisão da fase de cálculo das matrizes de programação dinâmica Gotoh é realizada para forçar uma sincronização no processamento dos blocos, evitando assim que uma dependência de dados gere uma condição de corrida [103]. Finalmente, a utilização dos registradores e áreas de memória global, compartilhada e textura da GPU é otimizada na solução, permitindo que longas sequências possam ser comparadas mesmo com placas gráficas com menos recursos.

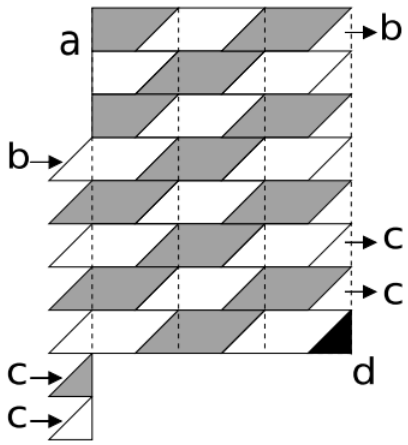


Figura 5.1: Delegação de células no CUDAlign 1.0 [103].

Na execução do CUDAlign 1.0, inicialmente a matriz DP é dividida em blocos de células com mesmas dimensões, com base no tamanho de das sequências e na quantidade de *blocks* (B) e *threads* (T) CUDA que serão utilizados. Os blocos são então organizados em

diagonais, chamadas diagonais externas, contendo entre 1 e B blocos. Para cada diagonal, ocorre uma chamada a uma função CUDA, que processa todos os blocos, repetindo-se o procedimento até que todas as diagonais sejam processadas. Dentro de bloco também ocorre paralelismo: as células são agrupadas em diagonais internas, e T (quantidade de *threads* CUDA) células de cada diagonal podem ser processadas em paralelo.

Os testes do CUDAlign 1.0 foram realizados utilizando dois modelos de placas gráficas da NVidia, tendo melhores resultados na GPU GTX 280, na comparação dos cromossomos 21 *Homo sapiens* (ser humano) e 22 *Pan troglodytes* (chimpanzé), contendo aproximadamente 33 MBP e 47 MBP, respectivamente. A solução obteve um desempenho máximo de cerca de 20,4 GCUPS na comparação destas sequências longas.

O CUDAlign 2.0 [101] obtém o escore ótimo e recupera o alinhamento produzido entre duas sequências longas de DNA com um algoritmo que combina Gotoh (Seção 2.2.3) e MM (Seção 2.2.4). Executa-se em seis estágios, sendo que o primeiro estágio corresponde ao cálculo da matriz de programação dinâmica, enquanto os estágios 2 a 5 correspondem ao *traceback*, que é executado em uma abordagem dividir-para-conquistar. O estágio 6 permite a visualização do alinhamento. O primeiro estágio equivale ao CUDAlign 1.0 modificado para salvar algumas linhas da matriz de similaridade em disco, que serão usadas no *traceback*.

As tarefas executadas em cada estágio do CUDAlign 2.0 (Figura 5.2) são detalhadas a seguir:

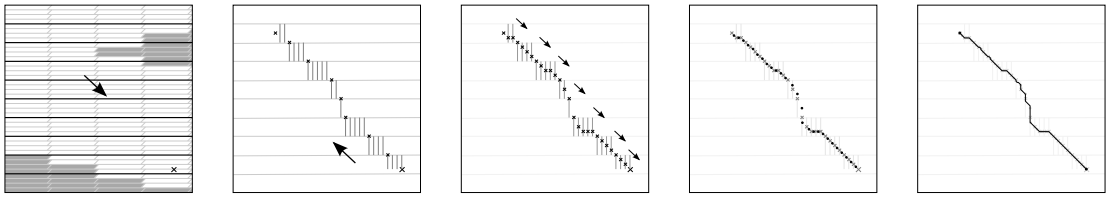
- No estágio 1, as matrizes de programação dinâmica Gotoh são processadas usando a abordagem *wavefront* em forma de paralelogramo e dados de linhas especiais são armazenados em disco, sendo gerados como saída o escore ótimo e as suas coordenadas;

- No estágio 2, um alinhamento semiglobal é realizado no sentido reverso de forma otimizada a partir do ponto onde o escore máximo ocorre, usando as linhas especiais salvas em disco. É aplicada a otimização chamada execução ortogonal, que é uma variante do algoritmo MM que permite não só o processamento linha a linha, mas também coluna a coluna, reduzindo consideravelmente a área processada [101]. Ao final desse estágio, é obtida uma lista de coordenadas dos pontos finais do alinhamento ótimo, denominados *crosspoints*;

- No estágio 3, mais *crosspoints* são obtidos, com a diferença que partições são definidas com pontos de início e fim;

- No estágio 4, o algoritmo MM é executado em CPU entre cada par sucessivo de *crosspoints*, buscando que a distância entre *crosspoints* consecutivos seja menor ou igual que um determinado limite;

- Os *crosspoints* restantes são obtidos dentro de cada partição e todos são concatenados, gerando assim o alinhamento final, que é o objetivo do estágio 5, sendo os resultados gerados em formato binário;
- Finalmente, uma visualização da representação binária do alinhamento é obtida (opcionalmente) no estágio 6.



(a) Estágio 1 (b) Estágio 2 (c) Estágio 3 (d) Estágio 4 (e) Estágios 5/6

Figura 5.2: Estágios de execução do CUDAlign 2.0 [102].

O algoritmo foi testado na placa NVidia GTX 285, recuperando o alinhamento dos cromossomos 21 humano (47 MBP) e 22 do chimpanzé (33 MBP), além de outros pares de sequências com menos bases, obtendo em desempenho máximo de 23,8 GCUPS.

Na versão CUDAlign 2.1 [102], os autores utilizaram como base a versão 2.0 [101], e adicionalmente propuseram a técnica de descarte de blocos (BP) para o estágio 1, conforme apresentado na Seção 2.3.3. As mesmas funções *kernel* e forma de processamento utilizadas no CUDAlign 1.0 são utilizadas no CUDAlign 2.1. A diferença é que, caso ocorra descarte de blocos em alguma diagonal externa, apenas os blocos não descartados são processados pelos *kernels* CUDA, reduzindo o tempo de execução.

Os testes do CUDAlign 2.1 foram realizados em um ambiente com GPU NVidia GTX 560 Ti, obtendo melhor resultado na comparação de duas sequências contendo 33 MBP e 47 MBP, atingindo 52,8 GCUPS. Nessa comparação, mais de 50% da matriz de programação dinâmica não foi calculada, devido à otimização de descarte de blocos. Mesmo assim, o alinhamento ótimo foi obtido.

5.1.6 SW# (2013)

A solução SW# [51] foi projetada em três fases: resolução, localização e reconstrução. A fase de resolução utiliza técnicas propostas no CUDAlign 2.1 (Seção 5.1.5) para execução do algoritmo MM na comparação de sequências longas de DNA com o descarte de blocos e execução ortogonal propostos no CUDAlign 2.1 em plataforma CUDA. Os autores dividem a computação da matriz em dois subproblemas independentes após a execução da primeira recursão do algoritmo, sendo obtidos o primeiro *crosspoint* e seu escore. Isto permite que o problema possa ser dividido em dois, e então distribuído para duas placas gráficas neste

ponto, para execução em paralelo. Na fase de localização, o algoritmo Gotoh é executado de maneira reversa para localizar o ponto de partida dos alinhamentos.

O método de *wavefront* é utilizado também na fase de reconstrução, combinado com a execução recursiva do algoritmo MM, modificado para que a execução seja interrompida quando o tamanho de uma submatriz for inferior ao limite definido. Neste ponto, a execução é passada para a CPU, responsável por executar a tarefa de *traceback* para gerar o alinhamento obtido nesta iteração, que será combinado aos demais para gerar o alinhamento completo. Uma representação básica do funcionamento das três etapas pode ser vista na Figura 5.3, sendo: (a) resolução, (b) localização, e (c) reconstrução.

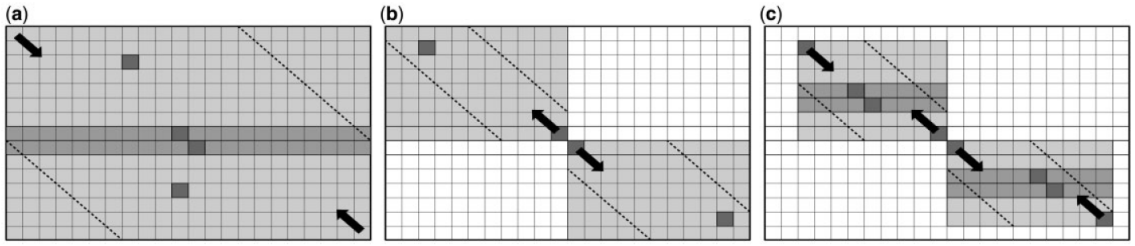


Figura 5.3: Fases da solução SW# [51].

O desempenho obtido foi comparável ao CUDAlign 2.1 (Seção 5.1.5) em uma única placa gráfica, sendo até mesmo mais lento que o CUDAlign 2.1 nos testes com sequências mais longas. Um melhor desempenho (que foi medido no trabalho considerando o tempo de execução) foi observado com a utilização da placa NVidia GTX 690 dual (duas GPUs integradas). Baseando-se no tamanho da sequência testada e no tempo de execução informado, calcula-se que a solução obteve 65,2 GCUPS. Cabe ressaltar que esta abordagem não necessita de espaço em disco adicional para armazenamento de linhas/colunas especiais para produzir o alinhamento ótimo, como era feito no CUDAlign 2.1. Nas versões subsequentes do CUDAlign, as linhas/colunas especiais são salvas em memória, e o disco somente é utilizado quando a memória se esgota.

5.1.7 MASA (2016)

O principal objetivo do *framework* MASA [105] é prover uma infraestrutura flexível para o desenvolvimento de soluções de alinhamento de sequências em múltiplas plataformas de *hardware* e *software*. O código reutilizável do MASA (desenvolvido em C/C++) pode ser agregado a um código desenvolvido para uma plataforma de *hardware* e *software* específica, permitindo a implementação de soluções em tempo reduzido.

O CUDAlign (Seções 5.1.5 e 5.2.1.6) foi utilizado como base para o desenvolvimento do MASA. Para tanto, seu código foi reorganizado e dividido em três módulos, de acordo

com as funcionalidades existentes: as funções independentes de plataforma, contemplando o gerenciamento de dados (armazenamento e manipulação das sequências, divisão da matriz em blocos, entre outros), o gerenciamento de estágios e as funções estatísticas; as otimizações/funções específicas do MASA, contendo a estratégia de processamento paralelo da matriz de programação dinâmica e a otimização de descarte de blocos (Seção 2.3.3); e, finalmente, a implementação da equação de recorrência, que é dependente de *hardware*. A integração entre estes módulos pode ser observada na Figura 5.4.

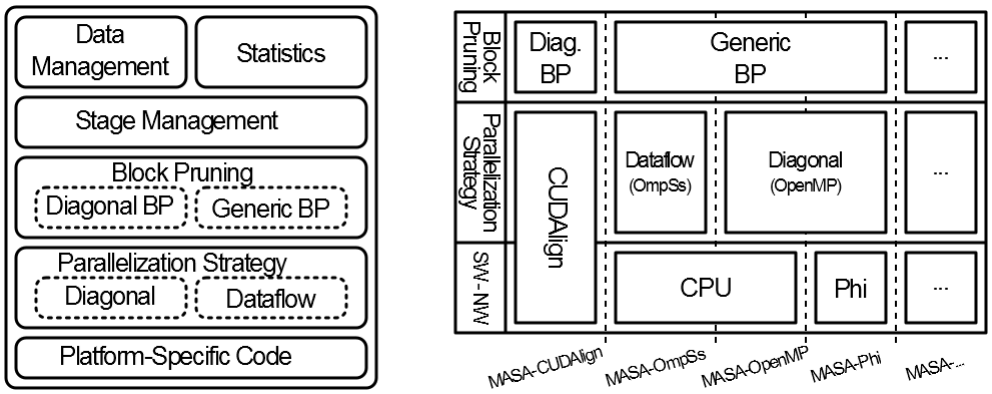


Figura 5.4: Arquitetura MASA [105].

Como estratégia de paralelização, duas abordagens são oferecidas: o método diagonal, onde o processamento inicia-se no canto superior esquerdo da matriz e propaga-se diagonalmente, permitindo o paralelismo *wavefront* (Seção 2.4); e o método de fluxo de dados (*dataflow*), propagando-se de forma genérica entre nós que representam blocos de células, respeitando as dependências da equação de recorrência. De forma similar, a técnica de descarte de blocos é também oferecida para execução de forma diagonal ou genérica (Seção 2.3.3), evitando o cálculo desnecessário de blocos que não têm possibilidade de contribuir para o escore ótimo.

Para a criação de uma implementação específica (chamada de extensão), o código comum do MASA deve ser compilado em conjunto com o código dependente da plataforma, gerando um programa executável único capaz de executar o algoritmo de comparação na plataforma de *hardware* e *software* escolhida. No trabalho, foram propostas e avaliadas as seguintes extensões (Figura 5.4): baseada em CUDA (derivada do CUDAlign), para GPUs NVidia; utilizando OpenMP (Seção 3.2.3), para *multicore* CPU e Intel Xeon Phi; e integrado ao modelo de programação OmpSs [18], para *multicore* CPU. Vale notar que as extensões para CPU não utilizaram o conjunto de instruções vetoriais, o que poderia gerar ganhos adicionais de desempenho, porém tornaria as soluções dependentes de um fabricante de CPU (Intel, ARM, IBM, etc.).

Comparações envolvendo sequências de tamanhos variando entre 10 KBP e 47 MBP foram utilizadas nos testes das extensões propostas. A soluções MASA-OpenMP e MASA-OmpSs foram testadas em um ambiente de CPU com 12 núcleos, obtendo um desempenho de 3,35 GCUPS e 4,18 GCUPS, respectivamente. Já a solução MASA-Phi foi testada no Intel Xeon Phi SE10P, que possui 60 núcleos, atingindo 2,64 GCUPS. Exceto pelas menores sequências testadas, o MASA-CUDAlign apresentou o melhor desempenho entre todas as extensões, chegando a 56,16 GCUPS no processamento dos seis estágios do alinhamento envolvendo duas sequências de 10 MBP, executando em uma GPU Nvidia Tesla M2090.

5.1.8 MASA-OpenCL (2015)

O MASA-OpenCL [25] foi realizado durante o Mestrado do autor desta Tese e utilizou como base a primeira fase do CUDAlign (Seção 5.1.5), criando uma nova extensão MASA (Seção 5.1.7) em OpenCL, que permitiu a execução do mesmo código-fonte em CPUs e GPUs de diferentes fabricantes.

Para o MASA-OpenCL, foram identificados 15 métodos ou funções no primeiro estágio que requeriam tradução de instruções CUDA para OpenCL (Seção 3.2.2). Entre as mudanças requeridas, destaca-se a mudança do local de alocação das sequências na memória do dispositivo: em vez da área de textura (que é restrita a GPUs), foi utilizada a memória global. A versão 1.2 do OpenCL foi utilizada na implementação para que o código fosse compatível com todos os dispositivos testados. O código inicial, compatível com CPUs Intel e AMD e com GPUs NVidia, sofreu pequenas alterações para que pudesse ser compilado também em GPU AMD, o que confirmou a portabilidade do OpenCL.

O ambiente de testes possuía cinco plataformas: (a) 1 CPU Intel i7-4500, com 2 processadores de 2 núcleos com 1,80 GHz e 8 GB RAM; (b) 1 CPU AMD FX-8350, com 1 processador de 8 núcleos, com 4,00 GHz e 8 GB RAM; (c) 1 GPU NVidia GTX 580, com 512 núcleos CUDA e 1,5 GB de memória; (d) 1 GPU NVidia GTX 680, com 1.536 núcleos CUDA e 2 GB de memória; (e) 1 GPU AMD R9 280X, com 2.048 núcleos de GPU e 3 GB de memória. Foram utilizadas 11 sequências de DNA cujos tamanhos variavam entre 10 KBP e 47 MBP.

Nos testes com CPUs, identificou-se resultados superiores na CPU Intel (2,57 GCUPS), embora o processador possuísse quantidade de núcleos e *clock* inferiores aos da CPU AMD. O melhor desempenho foi fruto da vetorização implícita no compilador OpenCL para o processador Intel, que obteve resultados até 2,5x mais rápidos que a CPU AMD.

Nos testes em GPUs, a execução do MASA-OpenCL foi comparada com outras duas ferramentas: SW# (Seção 5.1.6) e CUDAlign 2.1 (Seção 5.1.5). O MASA-OpenCL apresentou desempenho superior às demais em quase todas as comparações testadas. Após

análise do código intermediário PTX gerado (Seção 3.2.1), foram identificados dois fatores que contribuíram para este resultado: a compilação em 32 bits do OpenCL e o armazenamento das sequências na memória global da GPU. Após isso, tanto o SW# como o CUDAlign 2.1 foram compilados em 32 bits com as sequências armazenadas em memória global, e os resultados obtidos pelas três ferramentas em duas GPUs NVidia (GTX 580 e GTX 680) foram muito próximos. O melhor resultado pelo MASA-OpenCL, no entanto, foi obtido na GPU AMD R9 280X: 179,2 GCUPS na comparação de duas sequências de 10 MBP.

5.1.9 Tabela Comparativa: Soluções em GPU em uma Máquina Única

A Tabela 5.1 apresenta uma comparação entre alguns aspectos das soluções apresentadas na Seção 5.1. Os artigos são referenciados preferencialmente pelo nome dado pelo(s) autor(es). As seções relativas aos trabalhos também são referenciadas entre parênteses.

A coluna "Sequências" indica se a solução realiza a comparação de proteínas, geralmente de tamanhos reduzidos e comparados contra uma base de sequências, ou de sequências de DNA, que em geral possuem tamanho mais significativo. A coluna "Saída" informa se a solução produz como saída de dados apenas o escore ou também o alinhamento.

Já a coluna "Algoritmo" destaca o algoritmo utilizado como base para a tarefa de comparação de sequências (Seção 2.2), enquanto a coluna "Plataforma" informa se a solução se baseia na arquitetura CUDA (Seção 3.2.1) ou no *framework* OpenCL (Seção 3.2.2). Finalmente, a coluna "GCUPS" representa o desempenho no melhor caso obtido nos testes, medido em bilhões de células atualizadas por segundo (GCUPS).

Avaliando-se os trabalhos mais relevantes que abordam a comparação de sequências biológicas em GPU em máquina única, observa-se que diversas soluções tratam a comparação de sequências longas (como cadeias de DNA), tarefa que demanda maior poder computacional e mais espaço de armazenamento. Da mesma forma, a maioria dos programas utilizam a linguagem CUDA, restringindo o uso da aplicação para placas de vídeo produzidas pela NVidia - apenas três soluções foram testadas em GPUs produzidas pela AMD. A maioria das soluções existentes que utilizam a linguagem OpenCL possuem foco na comparação de sequências pequenas, com exceção do MASA-OpenCL. Adicionalmente, o máximo desempenho reportado por qualquer das aplicações para processamento em GPUs em máquina única foi de 179,2 GCUPS.

Tabela 5.1: Soluções de comparação de sequências em máquinas únicas com GPUs.

Artigo	Sequências	Saída	Algoritmo	Plataforma	GCUPS
Ligowsky e Rudinick (5.1.1)	Proteínas	Escore	Gotoh	CUDA	14,5
CUDASW++ 1.0 (5.1.2)	Proteínas	Escore	Gotoh	CUDA	16,1
CUDASW++ 2.0 (5.1.2)	Proteínas	Escore	Gotoh	CUDA	29,9
SW 2.0 (5.1.3)	Proteínas	Escore	Gotoh	OpenCL	66,0
Razmyslovich et al. (5.1.4)	DNA	Escore	SW	OpenCL	0,1
CUDAlign 1.0 (5.1.5)	DNA	Escore	Gotoh	CUDA	20,4
CUDAlign 2.1 (5.1.5)	DNA	Escore + Alinhamento	Gotoh + MM	CUDA	52,8
SW# (5.1.6)	DNA	Escore + Alinhamento	Gotoh + MM	CUDA	65,2
MASA-OpenCL (5.1.8)	DNA	Escore	Gotoh	OpenCL	179,2

5.2 Comparação de Sequências Biológicas com Múltiplos Dispositivos

A demanda por menores tempos de execução vem conduzindo as pesquisas à utilização de múltiplos dispositivos para a execução de algoritmos de comparação de sequências. As unidades de processamento podem ser agrupadas em ambiente uniforme (onde os dispositivos são do mesmo tipo) ou em ambiente híbrido (onde dispositivos de tipos diferentes são utilizados na computação).

Nesta seção, serão apresentadas, em ordem cronológica, algumas soluções de comparação de sequências biológicas com execução em múltiplos dispositivos, seja em ambiente uniforme (Seção 5.2.1) ou híbrido (Seção 5.2.2). Será apresentada também uma tabela comparativa entre estas soluções (Seção 5.2.3), destacando as plataformas utilizadas e o desempenho obtido.

5.2.1 Soluções em Ambiente Uniforme

5.2.1.1 GPU - Ino et al. (2009, 2012)

Em Ino et al. [41], múltiplas GPUs não dedicadas são usadas para executar o algoritmo de Gotoh (Seção 2.2.3) com sequências de proteínas, em uma comparação *query* x *database* com abordagem de grão grosso. A disponibilidade das GPUs para executarem o processamento é verificada pela inicialização do processo de proteção de tela (*screensaver*). As GPUs funcionam como escravos em uma arquitetura mestre/escravo, onde a CPU realiza o gerenciamento e a distribuição das tarefas. Uma tarefa é definida como a comparação de uma sequência *query* contra todo o banco de sequências genômicas, e as tarefas são consumidas cada vez que a GPU se torna disponível, até que todo o trabalho seja processado.

Os testes envolveram 64 sequências de referência com 367 aminoácidos e um banco de sequências. Foi utilizado um ambiente com 8 GPUs (NVidia séries 8800 e 7900), obtendo um máximo de 3,1 GCUPS.

A solução foi aprimorada em 2012 [42], eliminando a necessidade de aguardar pela ativação da proteção de tela. Nesta versão, a solução aguarda que as GPUs fiquem no estado disponível (*idle*) para iniciar as comparações. As tarefas são distribuídas de maneira a priorizar a execução em GPUs que estão há mais tempo disponíveis. O algoritmo permite também o cancelamento de uma tarefa em execução e sua realocação para outra GPU.

O ambiente de testes possuía 9 GPUs NVidia de diferentes modelos: 5 placas GTX 285, 1 placa GTX 295, 1 placa FX 5800 e 1 placa 8800 GTX, cada uma conectada a um *host* diferente. A comparação de sequências contendo entre 63 e 511 aminoácidos atingiu 64,0 GCUPS.

5.2.1.2 CPU - SWIPE (2011)

O algoritmo Gotoh foi utilizado por Rognes para comparação de sequências de proteínas utilizando o conjunto de instruções vetoriais SSE. Na solução SWIPE [93], cada tarefa compara uma sequência de referência com um subconjunto do banco de sequências (grão grosso). Nesta abordagem, 16 sequências do banco podem ser comparadas em paralelo, sem requerer que as sequências sejam ordenadas por tamanho. Visando maximizar o desempenho, as instruções do *loop* mais interno foram codificadas em linguagem de baixo nível.

Além das instruções vetoriais, o SWIPE utiliza também *pthreads* (Seção 3.2.3) para paralelização, em uma estratégia de grão grosso. Nesta solução, as *threads* podem não processar a mesma quantidade de trabalho, já que os pedaços de sequências são alocados às *threads* assim que elas ficam disponíveis.

Nos testes, foram comparadas 32 sequências com tamanhos variando entre 24 e 5.478 aminoácidos com uma base de sequências, e foram utilizados 2 processadores Intel Xeon de 6 núcleos cada. Obteve-se um máximo de 106,2 GCUPS na comparação de uma sequência de 375 aminoácidos contra o banco de sequências UniProt Knowledgebase, utilizando 19 *threads*. Contudo, há redução do desempenho na comparação de sequências menores que 100 aminoácidos, bem como nas sequências maiores que 1.000 aminoácidos, caso muitas *threads* sejam usadas.

5.2.1.3 FPGA - SW-Rivyera (2013, 2014)

A plataforma Rivyera S3-5000 é utilizada por Wienbrandt [123] para executar o algoritmo SW (Seção 2.2.2) na comparação *query* x *database* de proteínas ou de duas sequên-

cias de DNA, provendo o escore ótimo como saída. A implementação em FPGA utiliza uma paralelização de grão fino, onde existe um elemento de processamento na FPGA para cada nucleotídeo da sequência de referência. O processamento da matriz de programação dinâmica é realizado de forma diagonal, seguindo o padrão *wavefront* para FPGA, ou seja, todos os elementos de processamento são conectados em *array* sistólico, de forma que cada elemento tem acesso ao valor da célula do seu predecessor.

O sistema utilizado nos testes possuía 16 placas, cada uma com 8 FPGAs XLINX Spartan3-5000, com 32 MB de memória, totalizando 128 FPGAs conectadas por um barramento de alto desempenho. Esta arquitetura permite que até 512 comparações de sequências de DNA possam ser realizadas simultaneamente no sistema. Nos testes, um milhão de sequências contendo 100 *base pairs* foram comparadas contra o genoma humano (3,2 GBP), obtendo 3.040 GCUPS.

Em um trabalho posterior [124], o autor testou a solução em outro *cluster* de FPGAs: RIVYERA S6-LX150, com 128 placas do tipo Xilinx Spartan6-LX150. Essa solução permitiu a execução de 1.024 comparações concorrentes, e nos testes obteve 6.020 GCUPS, na comparação de sequências de DNA com 100 *base pairs* cada.

5.2.1.4 Xeon Phi - Swaphi-LS (2014)

O Swaphi-LS foi proposto por Liu et al. [61] para comparação de sequências longas de DNA no acelerador Intel Xeon Phi, sendo implementado a partir de modificações em uma solução criada anteriormente para comparação de sequências pequenas [59], gerando como saída apenas o escore ótimo.

Três abordagens de paralelização foram propostas: (a) uma abordagem ingênua (*naive*), em que o alinhamento é realizado diagonal a diagonal, a partir do canto superior esquerdo, com técnicas de vetorização em um time de *threads* usando OpenMP; (b) uma abordagem em tiras, que particiona a matriz de programação dinâmica em tiras retangulares e executa o processamento através de múltiplas *threads* nas tiras da mesma diagonal (paralelização de grão grosso), além de vetorização dentro de cada tira (paralelização de grão fino), executado em um Intel Xeon Phi; e (c) uma abordagem distribuída, que particiona a matriz em blocos processados em uma rede de Intel Xeon Phis, utilizando MPI para comunicação entre os processos.

Sequências com tamanhos entre 4 MBP e 50 MBP foram comparadas utilizando um *host* ligado a quatro Intel Xeon Phis. Testes foram realizados variando a quantidade de *threads*, e com as abordagens em tiras e distribuída. O melhor desempenho foi obtido na versão distribuída executando nas quatro placas aceleradoras: 111,4 GCUPS. A solução também foi comparada com o SW# (Seção 5.1.6), com resultados superiores aos obtidos em duas GPUs NVidia.

5.2.1.5 CPU - SW-MVM (2014)

Maleki et al. [66] utilizaram álgebra linear para transformar problemas de programação dinâmica em um conjunto de multiplicações de vetor-matriz independentes. Entre estes problemas estão incluídos a comparação local e global de sequências com modelo *affine gap* (Gotoh - Seção 2.2.3). Cada estágio do processamento iterativo do algoritmo pode ser executado em paralelo, porém requer sincronização ao final do estágio. Segundo [66], alinhamentos globais requerem mais estágios para convergir que alinhamentos locais. O algoritmo proposto por Farrar [20] foi utilizado em cada estágio do alinhamento local.

O ambiente de testes da solução SW-MVM incluiu 8 nós homogêneos, cada um com 2 processadores Intel Xeon 2,7 GHz com 8 núcleos cada, totalizando 128 núcleos. A comunicação entre os nós do *cluster* utilizou MPI. Quatro sequencias de referência com até 800 nucleotídeos foram pesquisadas e comparadas com os cromossomos humanos 1 a 4, com até 249 MBP. A solução obteve 900 GCUPS no alinhamento local de sequências.

5.2.1.6 GPU - CUDAlign 3.0 e 4.0 (2014, 2016)

O CUDAlign 3.0 [107] permite a execução do algoritmo de Gotoh em múltiplas GPUs, com paralelismo de grão fino, obtendo o escore ótimo e incorporando as otimizações do CUDAlign 1.0 (Seção 5.1.5). Para realizar a comparação de sequências longas em múltiplas GPUs, componentes de comunicação foram introduzidos para transportar dados entre GPUs vizinhas enquanto a computação é realizada, através de três *threads* assíncronas na CPU: uma de gerenciamento (T_M) e duas de comunicação (T_C) para cada GPU existente, como pode der visto na Figura 5.5. Cada GPU possui um identificador de processo (PID_i), que equivale a uma execução do CUDAlign específica para cada GPU_i. Caso as GPUs estejam em *hosts* diferentes, a troca de dados se dá pela rede de comunicação (*Network*).

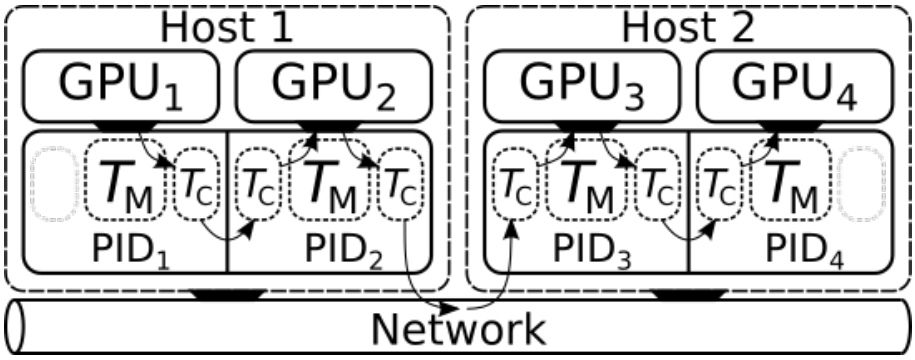


Figura 5.5: *Threads* de comunicação no CUDAlign 3.0 [107].

Para troca das informações das últimas colunas, são usados *buffers* de entrada (I - *input*) e saída (O - *output*) em cada GPU, como pode ser visto na Figura 5.6. Os *buffers*

possuem tamanho fixo de 128 KB, e podem gerar retenção de comunicação caso fiquem cheios em algum momento do processamento.

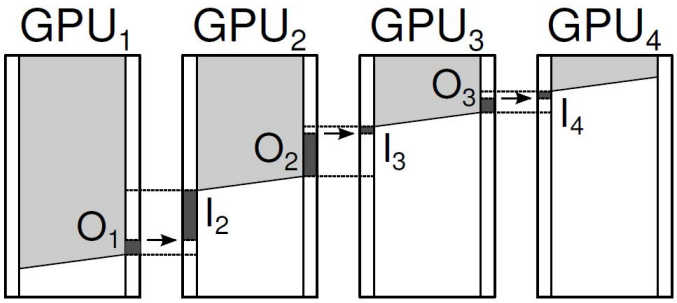


Figura 5.6: Modelo de comunicação do CUDAlign 3.0. O_i e I_i são, respectivamente *buffers* de saída (*output*) e entrada (*input*) [107].

No CUDAlign 3.0, as GPUs são conectadas através de *sockets* TCP, e organizadas logicamente de forma linear, havendo comunicação de cada GPU apenas com as vizinhas da esquerda e da direita. Cada GPU computa um conjunto de colunas da matriz de programação dinâmica, transferindo as células de sua última coluna para a próxima GPU.

O CUDAlign 3.0 foi testado em ambiente homogêneo [107] e heterogêneo [106]. No ambiente heterogêneo com 3 GPU, atingiu-se 140,3 GCUPS. No ambiente homogêneo, foram utilizadas 64 GPUs NVidia Tesla M2090, permitindo a comparação dos cromossomos 1 do homem e do chimpanzé (249 MBP e 228 MBP, respectivamente) em pouco mais de 9 horas, atingindo 1.726 GCUPS.

A versão 4.0 do CUDAlign [104] executa também o *traceback* na matriz para recuperar o alinhamento local ótimo com várias GPUs. A técnica especulativa *Incremental Speculative Traceback* (IST) foi proposta no CUDAlign 4.0 e funciona da seguinte maneira. Assim que a GPU_i termina de calcular seu subconjunto de colunas no estágio 1, ela usa o tempo em que ficaria ociosa para fazer o IST de sua parte da matriz, assumindo que o ponto de maior escore na coluna de borda é um *crosspoint* (Seção 5.1.5). Como várias GPUs podem estar executando de maneira especulativa, elas se comunicam, e várias especulações podem ser feitas por cada GPU, de maneira incremental [104]. Ao receber o *crosspoint* correto, a GPU_i o compara com os *crosspoints* utilizados no início de cada especulação. Se forem iguais, a especulação está correta, e os *crosspoints* obtidos a partir dela são imediatamente passados para a GPU_{i-1}. Caso contrário, a especulação está incorreta, e os *crosspoints* corretos são calculados.

Algumas métricas de execução para cada GPU são periodicamente escritas em arquivos de *log* pelo CUDAlign 4.0. Estas métricas podem ser usadas para avaliar a distribuição da carga de trabalho (Seção 5.3.2).

Nos testes, foram utilizados todos os cromossomos homólogos entre o homem e o chimpanzé, com tamanhos variando entre 26 MBP e 249 MBP. Atingiu-se um desempenho de 10.370 GCUPS executando em 384 GPUs NVidia M2090 divididas em 128 nós. A adoção do *traceback* incremental especulativo gerou um ganho de desempenho (*speedup*) de até 21,03x na fase de recuperação do alinhamento em GPU. Vale ressaltar que, para a execução do alinhamento das sequências em múltiplas GPUs no CUDAlign 4.0, o descarte de blocos existente na versão 2.1 foi desabilitado. Cabe ressaltar que, no final de 2016, o código do CUDAlign foi adaptado à arquitetura MASA (Seção 5.1.7) e, a partir daí, passa a se chamar MASA-CUDAlign.

5.2.1.7 CPUs em Nuvem - CloudSW (2017)

A solução CloudSW [125] permite a comparação de proteínas do tipo *query* x *database* em um conjunto de CPUs distribuídos em nuvem computacional (*cloud computing*) [117]. Uma técnica de *query profile* é utilizada para reduzir repetições de cálculo e operações de cópia entre áreas de memória. A solução retorna o escore e o alinhamento gerado.

Inicialmente a solução realiza um pré-processamento das sequências *query*, das bases de sequências e da matriz de substituição, para que as comparações possam ser mapeadas em tarefas. Estas tarefas são gerenciadas através do Apache Spark [127], que é uma *engine* de computação que permite o agendamento, distribuição e monitoramento de tarefas de uma aplicação através de diversos servidores em *cluster*.

Os experimentos foram realizados utilizando 8 nós físicos armazenando 50 máquinas virtuais (200 *cores*) na nuvem Alibaba Alyium. Sequências de proteínas de referência contendo até 4.096 aminoácidos foram comparadas contra 10 diferentes bases de sequências. A avaliação de desempenho analisou diferentes modos de operação (retornando apenas o escore ou também o alinhamento) e diferentes valores para situações de *match*/*mismatch*/*gaps*. O máximo desempenho obtido foi de 529,9 GCUPS.

5.2.1.8 ReCAM - BioSEAL (2020)

BioSEAL [46] é uma solução de comparação de sequências com o algoritmo Gotoh que usa uma tecnologia denominada *Resistive Content Addressable Memories* (ReCAM), que é usada tanto para armazenamento quanto para processamento de dados. Conceitualmente, sua arquitetura é composta de centenas de milhões de linhas, cada uma servindo como uma unidade computacional [45], permitindo um alto paralelismo de grão fino para operações lógicas e aritméticas simples.

A solução permite a comparação tanto de sequências de DNA quando de proteína, possuindo duas aplicações: (a) alinhamento de duas sequências longas de DNA; e (b) alinhamento de uma sequência *query* contra uma base de sequências, seja de DNA ou

proteínas. Para permitir a implementação do algoritmo, a ReCAM foi conceitualmente modificada para explorar a repetição de valores de escrita e execução da escrita em um único ciclo, permitindo a gravação em múltiplos pontos.

Na avaliação da comparação de sequências DNA, o trabalho utilizou oito sequências, com tamanhos variando entre 1 MBP e 249 MBP, e comparou com outras soluções que executam em CPU, GPU, FPGA e Intel Xeon Phi. Cabe ressaltar que os testes com BioSEAL foram realizados em simulador. O melhor desempenho informado para a solução na comparação de duas sequências longas de DNA foi de 32.800 GCUPS. Para a aplicação que realiza a comparação de uma sequência pequena de referência contra uma base, chegou a atingir 90.700 GCUPS, comparando sequências de 100 *base pairs* (*short read*) com o genoma humano. Como a execução somente foi feita em um simulador, admite-se que esses GCUPS são os limites superiores de desempenho para essa solução e acreditamos que um desempenho bem inferior é esperado para execuções reais em ReCAM.

5.2.2 Soluções em Ambiente Híbrido

5.2.2.1 FPGA+CPU - Meng e Chaudhary (2010)

No trabalho de Meng e Chaudhary [71], uma solução que utiliza CPU e FPGA é proposta para executar comparações do tipo *query* x *database* em uma arquitetura mestre/escravo. O mestre executa em CPU e aloca tarefas de comparação grão grosso para os escravos e combina os resultados recebidos. A distribuição das tarefas entre os dispositivos é feita de forma proporcional à capacidade de processamento dos dispositivos.

A solução proposta combina paralelização de grão grosso e fino em uma mesma arquitetura usando diferentes tipos de componentes de processamento. Em um primeiro nível, o banco de sequências é dividido em múltiplos fragmentos de tamanhos iguais e distribuídos para cada nó de computação dependendo de sua capacidade. Em um nível mais baixo relacionado ao *hardware*, técnicas adicionais são implementadas de acordo com as funcionalidades específicas da arquitetura do dispositivo (CPU e FPGA). Para as CPU, foram utilizadas instruções SSE2; não foi informado no artigo a linguagem utilizada para programação da FPGA.

A solução utiliza um arquivo de configuração lido pelo nó mestre com as informações de *hardware* referentes aos nós de processamento. A comunicação entre o nó mestre e os escravos utiliza MPI. A distribuição dos fragmentos é feita de forma proporcional às características do *hardware* nos nós escravos, visando aproveitar o tempo ocioso de cada dispositivo. Devido às limitações de armazenamento na FPGA, sequências *query* com mais de 1.420 aminoácidos são segmentadas para subsequências de no máximo este tama-

nho. Caso a comparação seja com sequências que possuem mais de 8.188 aminoácidos, a pesquisa é realizada pela CPU.

Sequências de referência foram comparadas contra três bancos de sequências de proteínas. A plataforma era composta por 10 CPUs Dual Core AMD Opteron 2,2 GHz (com instruções SSE), 1 FPGA (Xlinx Virtex II) e 1 CPU Pentium IV 1,9 GHz (sem instruções SSE). O desempenho obtido foi de 11,13 GCUPS, comparando uma sequência de tamanho 1.223. Devido ao tempo requerido para divisão da sequência para execução em FPGA, o desempenho para sequências *query* de mais de 1.420 bases diminui sensivelmente.

5.2.2.2 GPU+CPU - Mendonça e Melo (2013)

O ambiente híbrido de CPUs e GPUs também foi escolhido por Mendonça e Melo [70] em sua solução, que implementa o algoritmo Gotoh com um ajuste de distribuição de carga de trabalho dinâmica através de replicação. Nesta abordagem, as GPUs executam o CUDASW++ 2.0 (Seção 5.1.2), enquanto CPUs rodam uma versão modificada do algoritmo de Farrar [20]. A solução executa comparações *query* x *database* para proteínas. Inicialmente, o trabalho é distribuído estaticamente de acordo com a capacidade dos dispositivos; contudo, se um dispositivo termina seu trabalho e ainda existem comparações sendo executadas em outros dispositivos, ele também realiza esta computação, replicando a execução e visando terminar o trabalho o mais cedo possível.

Os testes foram conduzidos em dois *hosts* com mesma configuração: 2 GPUs NVidia GTX 580 e 1 CPU Intel i7 com 4 núcleos. Um conjunto de sequências de até 5.000 aminoácidos foram comparados com cinco bancos de sequências genômicas, resultando em um desempenho de 172,82 GCUPS quando os dois *hosts* foram utilizados (totalizando 2 GPUs e 8 núcleos de CPU).

5.2.2.3 GPU+CPU - CUDASW++ 3.0 (2013)

Liu, Wirawan e Schmidt [62] realizam a comparação de sequências de proteínas em CPU e GPU. A carga de trabalho é distribuída de forma estática baseando-se nas frequências de *clock*, quantidade de núcleos para as CPUs e número de SPs (Seção 3.1) para as GPUs. O código executado na CPU é baseado no SWIPE (Seção 5.2.1.2), enquanto o código executado na GPU utiliza instruções SIMD CUDA PTX (Seção 3.2.1) aliadas à abordagem SWIPE.

A paralelização ocorre em níveis *inter-task* e *intra-task*, equivalendo a estratégias de grão grosso e fino, respectivamente. Também foi mantida nesta versão a estratégia *sequential query profile* apresentada no CUDASW++ 2.0 (Seção 5.1.2).

O algoritmo funciona em quatro estágios [62]: (a) distribuição da carga de trabalho entre CPUs e GPUs de acordo com sua capacidade de processamento; (b) computação

concorrente em CPU e GPU; (c) recomputação de todos os alinhamentos que excederam a acurácia de 8 bits usando instruções SIMD de 16 bits em CPU; e (d) ordenação de todos os escores em ordem decrescente e saída dos resultados.

O ambiente de testes possuía 1 CPU (Intel i7 3,5 GHz, com 4 núcleos) combinada com 3 GPUs (1 NVidia GTX 680 e 2 NVidia GTX 690). Foram pesquisadas sequências contendo até 5.478 aminoácidos contra a base de sequências Swiss-Prot, atingindo o máximo de 185,6 GCUPS quando a CPU foi utilizada em conjunto com a GPU GTX 690.

5.2.2.4 FPGA+CPU - Oswald (2016)

Rucci et al. [98] propuseram uma solução do tipo *query* x *database* para comparação de proteínas implementada utilizando OpenCL para execução em ambiente híbrido de FPGA e CPU. A aplicação utiliza instruções vetoriais SSE e diretivas OpenMP na porção executada em CPU e explora o paralelismo dos FPGAs para retornar o escore ótimo das comparações, estendendo o processamento para múltiplas FPGAs.

O banco de sequências é ordenado pelo tamanho das sequências em ordem crescente. Três implementações foram propostas: com apenas uma FPGA, com múltiplas FPGAs e com ambiente híbrido CPU + FPGA. A distribuição de carga entre os dispositivos é feita de forma estática com *profiling*: um conjunto de dados é processado previamente para coletar a capacidade de processamento de cada unidade computacional, e então o trabalho é distribuído de maneira proporcional à capacidade de processamento, em uma paralelização *inter-task* (grão grosso).

Nos testes, a solução Oswald comparou um conjunto de 20 sequências com tamanhos entre 144 e 5.478 aminoácidos com os bancos de sequências Swiss-Prot e Environmental NR. Foi avaliada em relação a outras soluções em ambiente GPU + CPU (CUDASW++ 3.0, Seção 5.2.2.3) e Intel Xeon Phi + CPU (SWIMM - Seção 5.3.3), apresentando desempenho superior na maioria dos casos. O melhor desempenho, 441,6 GCUPS, foi obtido utilizando a segunda versão do conjunto de instruções vetoriais AVX (AVX2) em um ambiente com 2 CPUs Intel E5-2695 e 2 placas Altera Stratix V GSD5. Este resultado, contudo, foi inferior à execução das comparações utilizando a ferramenta SWIMM em ambiente CPU + Intel Xeon Phi: 450,5 GCUPS.

5.2.2.5 GPU+Phi+CPU - SWHybrid (2017)

A solução de Lan et al. [52] se baseia na construção de um modelo de máquina através da hierarquia de classes C++ compostas de uma parte geral e outa específica. Isto permite a criação de módulos *kernel* especializados para comparação de proteínas em diversas arquiteturas diferentes, integrando funções escritas em CUDA, *pthreads* ou OpenMP para

permitir a execução em CPUs, GPUs e em Intel Xeon Phi. Otimizações adicionais foram realizadas em cada código plataforma-específico para melhorar o desempenho.

A base de sequências é pré-processada para criar lotes globais de sequências. Para a distribuição da carga, as comparações são colocadas em lotes de tarefas consolidadas em um *pool*, que é consultado pelos processos de cada dispositivo até que todo o banco de sequências seja processado. Em cada processo *worker*, existe um *buffer* para o dispositivo, e eles são preenchidos pelos lotes globais.

O trabalho compara 8 sequências de tamanhos entre 464 e 5.478 aminoácidos a duas bases de sequências de proteínas. No ambiente de teste, 5 ambientes foram utilizados com diferentes dispositivos. O desempenho mais significativo (em torno de 1.000 GCUPS) foi obtido em um servidor com 2 CPUs Xeon E5-2683v4, 1 GPU Titan X e 1 GPU GTX 1080.

5.2.3 Tabela Comparativa: Soluções com Múltiplos Dispositivos

A Tabela 5.2 sumariza as soluções de comparação de sequências biológicas apresentadas nas Seções 5.2.1 e 5.2.2. Na coluna "Seção" consta o número da seção que apresenta a solução. Os tipos de dispositivos utilizados nos resultados experimentais estão na segunda coluna. O tamanho máximo (ordem de grandeza) da maior sequência *query* (para proteínas ou sequências *short read* de DNA) ou cadeia de DNA comparada é apresentado na quarta coluna. Já a coluna "Paralelismo" indica os modelos de programação paralela e comunicação entre dispositivos utilizados pela solução. A quinta coluna apresenta a quantidade de cada tipo de dispositivo utilizada, sendo que, para as CPUs, está sendo considerado o total de núcleos dos processadores envolvidos. Finalmente, a coluna "GCUPS" apresenta desempenho máximo obtido pela solução.

Como pode ser observado, os ambientes de *hardware* utilizados são bem variados. Nas soluções híbridas, a CPU é sempre associada a algum outro dispositivo devido às características da plataforma de *hardware*. As alternativas de paralelismo também são diversas (instruções vetoriais em CPU, CUDA, OpenCL e uso de *hardware* específico), mostrando que a busca por melhores soluções de comparação de sequências biológicas ainda é intensa, independente da(s) arquitetura(s) de *hardware* escolhida(s).

Algumas soluções obtiveram desempenho superiores a 1,0 TCUPS, o que é um resultado bem expressivo. O SW-Ryviera (Seção 5.2.1.3) utiliza 128 FPGAs para obter 6,0 TCUPS com paralelização de grão fino implementado no *array* sistólico das placas. Em ambiente híbrido, 1,0 TCUPS foi obtido pela solução SWHybrid (Seção 5.2.2.5), que além de utilizar os recursos de CPUs e GPUs ainda provê uma solução básica de distribuição da carga. O BioSEAL (Seção 5.2.1.8) apresentou resultados impressionantes em ReCAM, tanto na comparação de duas sequencias longas (32,8 TCUPS) como na comparação de

Tabela 5.2: Soluções de comparação de sequências em múltiplos dispositivos.

Seção	Tipos de Dispositivos	Tamanho Máximo	Paralelismo	Quantidade de Dispositivos	GCUPS
Ambiente Uniforme					
5.2.1.1	GPU	10^3	CUDA	8 GPUs	64,0
5.2.1.2	CPU	10^3	SSE	12 CPUs	106,2
5.2.1.3	FPGA	10^3	*Hardware* customizado	128 FPGAs	6.020,0
5.2.1.4	Intel Xeon Phi	10^7	MPI e IMICS	4 Intel Xeon Phis	111,4
5.2.1.5	CPU	10^2	SSE e MPI	128 CPUs	900,0
5.2.1.6	GPU	10^8	CUDA	384 GPUs	10.370,0
5.2.1.7	CPU em nuvem	10^3	Apache Spark	50 CPUs	529,9
5.2.1.8	ReCAM	10^8	Simulação	Simulação	32.800,0
		10^2			90.700,0
Ambiente Híbrido					
5.2.2.1	FPGA + CPU	10^3	n.i., MPI e SSE2	1 FPGA + 11 CPUs	11,3
5.2.2.2	GPU + CPU	10^3	CUDA e SSE	2 GPUs + 8 CPUs	172,8
5.2.2.3	GPU + CPU	10^3	CUDA e SSE	2 GPUs + 4 CPUs	185,6
5.2.2.4	FPGA + CPU	10^3	OpenCL e AVX2	2 FPGAs + 28 CPUs	441,6
5.2.2.5	GPU + CPU	10^3	CUDA e SSE	2 GPUs + 32 CPUs	1.000,0

uma sequência *query* (90,7 TCUPS), mas a solução foi testada apenas em simulação. Desta forma, o melhor desempenho reportado em execuções reais foi de 10,37 TCUPS, obtido pela solução CUDAlign 4.0 (Seção 5.2.1.6) em um ambiente homogêneo de 384 GPUs.

5.3 Distribuição de Carga em Comparação de Sequências Biológicas

A adoção de uma política adequada de distribuição de carga pode causar reduções significativas no tempo de execução de soluções de comparação de sequências biológicas em múltiplos dispositivos. Em ambientes uniformes homogêneos dedicados, a abordagem mais simples é a divisão das tarefas equitativamente entre os dispositivos.

Em ambientes uniformes heterogêneos ou híbridos, a abordagem mais usual é a distribuição de carga estática (Seção 4.6), onde as tarefas são alocadas de acordo com a capacidade de processamento teórica de cada dispositivo [71] [62] [95] ou a partir de parâmetros obtidos realizados após um *profiling* da aplicação [112] [98]. Em [70], como mostrado na Seção 5.2.2.2, algumas tarefas distribuídas estaticamente podem ser processadas de forma redundante por dispositivos *idle* para obter menor tempo de execução. As Seções 5.3.1 a 5.3.3 discutirão abordagens que utilizam distribuição dinâmica de carga.

5.3.1 Chen e Schmidt (2005)

Em [13], uma técnica para distribuição dinâmica de carga em um *grid* hierárquico de CPUs é proposta para a comparação de sequências com uma variante do algoritmo SW (Seção 2.2.2), que permite computar alinhamentos ótimos e próximos do ótimo (*near optimal*) sem interseção e em espaço linear, adaptando uma proposta apresentada em [122]. O paralelismo é em *wavefront* e todas as unidades de processamento calculam um subconjunto de colunas da mesma matriz de programação dinâmica. Sendo assim, segundo a classificação da Seção 4.4, a aplicação é *workflow* e, uma vez criadas as tarefas, elas se comunicam (tarefas comunicantes), trocando os valores da coluna de borda.

Na arquitetura proposta, existem vários *clusters* de computadores, que se comunicam via MPI. Uma camada superior é executada no nó controlador de cada *cluster*, e uma camada inferior reside em todos os nós do *cluster*, permitindo uma comunicação mais rápida. Também é utilizado o *software* de gerenciamento de recursos distribuídos *Sun Grid Engine* (SGE) [32] para escalonar tarefas dentro de cada *cluster*. A Figura 5.7 mostra o esquema da arquitetura proposta.

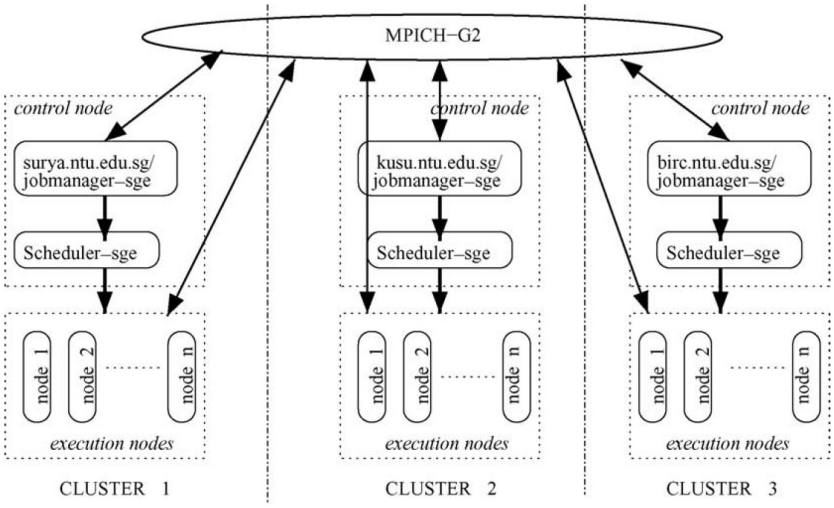


Figura 5.7: Arquitetura de solução em *grid* hierárquico [13].

A solução prevê um funcionamento em ambiente de *grid* não dedicado, e, portanto, sujeito a perturbações. Uma perturbação ocorre quando outra aplicação entra em execução no ambiente durante o processamento do trabalho.

Inicialmente, foi proposta uma distribuição de carga estática: a matriz de programação dinâmica é dividida em conjuntos de colunas adjacentes que são distribuídos entre *clusters* baseando-se na capacidade computacional de cada um deles. Em seguida, as colunas alocadas para cada *cluster* são divididas entre seus nós. Neste caso, o desempenho pode ser degradado caso ocorra uma perturbação no ambiente. Posteriormente, duas estratégias

de escalonamento foram propostas: mestre-escravo, onde cada nó escravo recebe do nó principal (mestre) os trabalhos a ele designados, e *scheduler-worker*, onde existe troca de dados entre os nós trabalhadores (*workers*) e entre estes e o nó principal (*scheduler*).

O trabalho propõe uma abordagem de distribuição dinâmica de carga em nível de aplicação utilizando MPI. Na abordagem mestre-escravo, a matriz de programação dinâmica é dividida em blocos retangulares pelo processo mestre, que são distribuídos como tarefas entre os escravos. Na solução proposta baseada em abordagem *scheduler-worker*, cada trabalhador comunica com o "agendador" único sempre que finaliza seu bloco retangular, reportando o seu desempenho. Este, por sua vez, é responsável por produzir uma nova forma de distribuição de trabalho com base no desempenho (quantidade de blocos processados por unidade de tempo) de cada nó. Para tanto, um novo tamanho de bloco é calculado baseado no desempenho de cada nó em relação ao desempenho dos demais nós. Caso seja necessária uma redistribuição dos blocos, o valor recalculado para cada "trabalhador" envolvido no processamento é enviado. O "agendador" verifica se há alguma perturbação nos nós antes de distribuir o trabalho.

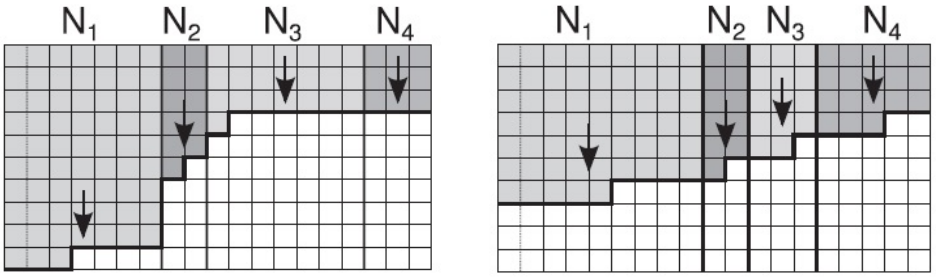
O ambiente de testes consistia em três *clusters*, sendo dois contendo 8 nós Intel Pentium III e outro contendo 8 nós Intel Itanium-I. Duas sequências de DNA contendo 144 KBP e 132 KBP (consideradas longas à época) foram usadas nos testes. O melhor desempenho na comparação isolada destas sequências em cada um dos *clusters* foi de cerca de 2,9 GCUPS.

Para avaliar o desempenho das estratégias propostas, foi utilizado como métrica o *speedup* dos tempos de execução calculado em relação ao processamento sem nenhuma política de distribuição de carga. A distribuição estática foi avaliada considerando uma perturbação causada por processo que ocupava 50% da CPU, obtendo um *speedup* máximo de 12x. Valores semelhantes foram obtidos utilizando a abordagem *scheduler-worker*. Já a abordagem mestre-escravo obteve um *speedup* máximo em torno de 10x. As duas soluções com distribuição dinâmica, contudo, foram bem menos afetadas pela perturbação causadas por tarefas locais: em torno de 10%, enquanto a distribuição estática foi degradada em até 80%.

5.3.2 Sandes et al. (2014)

A proposta de Sandes et al. [108] para distribuição dinâmica de carga utiliza o conceito de agentes e métricas locais e globais para identificar se as computações estão desbalanceadas em uma aplicação que executa o algoritmo Gotoh (Seção 2.2.3) em vários nós, onde cada nó processa um subconjunto de colunas da matriz de programação dinâmica. O processamento em *wavefront* é feito em múltiplos nós dispostos em plataformas heterogêneas e não dedicadas. Da mesma maneira que Chen e Schmidt (Seção 5.3.1), a aplicação é

um *workflow* com tarefas comunicantes. Exemplos de processamento de colunas de forma desbalanceada e balanceada em um ambiente com quatro nós podem ser vistos na Figura 5.8.



(a) (b)

Figura 5.8: Distribuição de colunas entre 4 nós [108]: (a) *wavefront* desbalanceado e (b) *wavefront* balanceado.

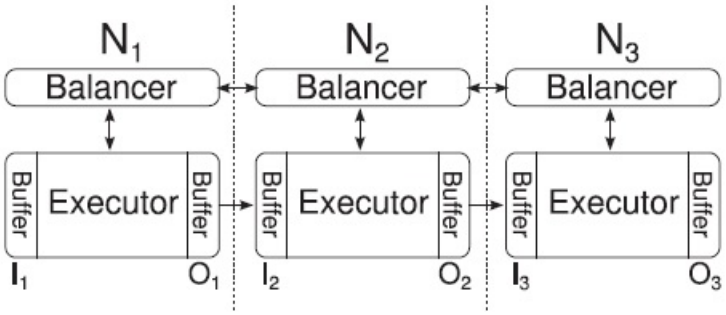


Figura 5.9: Arquitetura de solução multiagente [108].

Na arquitetura proposta (Figura 5.9), cada um dos nós possui um agente executor, responsável por executar a aplicação e manter os *buffers* de comunicação, e um agente balanceador, que tem a função de analisar o *status* do agente executor e identificar se seria melhor manter ou redistribuir as colunas. Neste último caso, o agente balanceador negocia com os demais nós considerando pesos dinâmicos. Estes pesos são definidos inicialmente com base em alguma métrica de desempenho de cada nó (como os GFLOPS, por exemplo). Durante a execução, a métrica utilizada na avaliação da distribuição é a quantidade de linhas da matriz de programação dinâmica processadas por segundo, e os pesos são atualizados assim que os nós decidem pela redistribuição, de forma a reduzir o tempo computacional. Para tanto, duas estratégias são usadas: global (onde agentes conhecem o estado dos demais agentes) e local (onde agentes só conhecem seu próprio estado).

Algumas métricas de execução para cada GPU são periodicamente escritas em arquivos de *log*. As relevantes que indicam a retenção nos *buffers* de entrada e saída são a métrica de

instabilidade causada por bloqueios na entrada ($\texttt{psi\_in}_i$) e o número de linhas processadas por segundo ($\texttt{r}_i$). A abordagem de distribuição dinâmica se baseia na análise destas métricas, e a Equação 5.1 foi adotada para calcular um novo subconjunto de colunas para cada GPU$_i$ (`Nsplit`) baseando-se na divisão de colunas atual (`Csplit`) e as métricas $\texttt{psi\_in}_i$ e r_i [108].

$$Nsplit = Csplit * (r_i - psi\_in_i) \tag{5.1}$$

Para testar os cenários de distribuição, os autores implementaram um simulador, e o testaram em um cenário com cinco nós de capacidade de processamento 160%, 160%, 160%, 100% e 300%, ou seja, as GPUs 1 a 3 possuíam 60% a mais de capacidade em relação à GPU 4, enquanto a GPU 5 possuía 200% a mais. As execuções foram medidas em unidades de tempo. Os resultados mostraram que cenários de comparação desbalanceados podem conduzir a desempenhos ruins (execução em 3,04 unidades de tempo), que pode ser atenuado se uma estratégia de distribuição estática com base no poder computacional dos nós for adotada (execução em 1,78 unidades de tempo). Quanto às estratégias de distribuição dinâmica de carga, foi simulada uma mudança no poder computacional do segundo nó para 240% e redução do terceiro nó para 120%, em momentos diferentes. Neste caso de teste, o cenário desbalanceado (onde a instabilidade é ignorada) passou a ser executado em 2,00 unidades de tempo, enquanto a estratégia local (1,76 unidades de tempo) produziu resultados comparáveis à estratégia global (1,72 unidades de tempo). As políticas permitiram um ganho de 12% se comparado ao cenário sem redistribuição, muito embora a aplicação tenha sido testada apenas em um simulador.

5.3.3 Rucci et al. (2015)

Rucci et al. [96] realizam uma análise de desempenho da solução SWIMM, além de uma análise do consumo de energia. O SWIMM executa o algoritmo SW (Seção 2.2.2) em plataformas híbridas (CPU, Intel Xeon Phi e GPU), aplicado à busca de proteínas em banco de sequências usando OpenMP. Ou seja, a aplicação é do tipo *bag of tasks* (Seção 4.4). Utiliza a abordagem de paralelização intertarefas proposta por Rognes (Seção 5.2.1.2), onde múltiplas sequências são comparadas simultaneamente. A estratégia de ordenar as sequências pelo tamanho proposta no CUDASW++ (Seção 5.1.2) também foi utilizada para distribuir as sequências de acordo com o tamanho das unidades de processamento vetoriais (VPUs - *Vector Processing Units*) de forma balanceada.

A distribuição de carga neste caso utiliza uma abordagem em dois níveis. Inicialmente, o banco de sequências de proteínas é ordenado pelo tamanho e agrupado de acordo com a largura dos VPUs na plataforma de destino. No caso do processamento em Intel Xeon Phi

e na implementação em ambiente híbrido, o banco de sequências é dividido em pedaços, que são alocados ao dispositivo assim que ele se torna disponível (*self-scheduling*). No segundo nível, a CPU ou o coprocessador Intel Xeon Phi distribui os pedaços (*chunks*, configurados na instrução OpenMP) recebidos para processamento entre seus núcleos.

A estratégia permitiu resultados significativos na implementação do SWIMM em ambiente híbrido. Dois ambientes foram testados: um com 2 CPUs cada qual com 8 núcleos e 1 Intel Xeon Phi 3120P (contendo 57 núcleos), e outro com 2 CPUs com 14 núcleos cada, 1 Intel Xeon Phi 3120P e 1 GPU NVidia Tesla K20c (contendo 2.496 núcleos CUDA). Foram comparadas 20 sequências *query* com tamanhos variando entre 144 e 5.478 aminoácidos com dois bancos de sequência - Swiss-Prot e Environmental NR. Em um dos testes, foram obtidos 117,8 GCUPS no processamento em CPU, 41,9 GCUPS em Intel Xeon Phi e 160,0 GCUPS na implementação híbrida, que realizou uma distribuição estática de carga - 75% para CPU e 25% para Intel Xeon Phi. Nos testes no segundo ambiente (incluindo uma GPU), a solução obteve 380,0 GCUPS.

5.3.4 Tabela Comparativa: Soluções com Distribuição de Carga

Os trabalhos listados nas Seções 5.3.1 a 5.3.3 adotaram estratégias diferentes de distribuição de carga. Os principais aspectos das soluções podem ser vistos na Tabela 5.3.

Tabela 5.3: Soluções com distribuição de carga na comparação de sequências biológicas.

Solução	**Tipo de aplicação**	**Tipo de distribuição de carga**	**Estratégia de distribuição de carga**
5.3.1	*workflow* e tarefas comunicantes	estático e dinâmico	mestre-escravo e *scheduler-worker*
5.3.2	*workflow*	estático e dinâmico	redistribuição
5.3.3	*bag of tasks*	estático e dinâmico	*self-scheduling*

As propostas de Chen e Schmidt (Seção 5.3.1) e Sandes et al. (Seção 5.3.2) dividem o trabalho a partir do processamento de uma única matriz de programação dinâmica em aplicação do tipo *workflow*, enquanto a abordagem de Rucci et al. (Seção 5.3.3) se baseia na distribuição de várias comparações, sem comunicação entre as tarefas (*bag of tasks*). A estratégia utilizada para distribuição da carga também diverge: enquanto Sandes et al. distribui todo o trabalho, redistribuindo dinamicamente se necessário, as demais soluções distribuem o trabalho aos poucos, realizando o balanceamento estático e dinâmico através do recálculo do tamanho do bloco (Chen e Schmidt) ou da estratégia de escalonamento estático com o uso do OpenMP (Rucci et al.), combinado ao *self-scheduling*. Devido às diferentes métricas e ambientes, não é possível realizar uma comparação direta entre os desempenhos dos trabalhos.

Parte II

Contribuições

Capítulo 6

Avaliação Estatística de Soluções MASA

A primeira contribuição desta Tese é uma análise quantitativa de soluções de comparação de sequências em GPU (Seção 3.1) que fazem parte da arquitetura MASA [27]. Duas soluções foram utilizadas na análise: o MASA-CUDAlign (Seção 5.1.5) e o MASA-OpenCL (Seção 5.1.8). Os objetivos do trabalho eram validar estatisticamente os ganhos obtidos pelo MASA-OpenCL em relação ao MASA-CUDAlign na execução em uma GPU [25], além de derivar uma equação de regressão que possibilitasse estimar o tempo de execução de uma dada comparação com base nas informações das sequências envolvidas e da GPU utilizada. Este resultado pode ser útil para, por exemplo, estimar o tempo de alocação de um recurso em um ambiente compartilhado de supercomputação com uso de escalonadores. Os resultados obtidos mostraram que os ganhos do MASA-OpenCL em relação ao MASA-CUDAlign foram validados estatisticamente, e foi possível construir uma equação de regressão multilinear que permite estimar com boa aproximação o tempo de execução de uma dada comparação.

A seguir, detalha-se a organização deste capítulo. Na Seção 6.1 são abordados alguns aspectos de análise quantitativa, técnica utilizada na avaliação dos dados experimentais. A Seção 6.2, por sua vez, descreve a avaliação do MASA-CUDAlign e do MASA-OpenCL. A avaliação da equação de regressão é discutida na Seção 6.3. Por fim, na Seção 6.4 são apresentadas as conclusões do capítulo.

6.1 Aspectos de Análise Quantitativa

O desempenho de soluções que comparam proteínas ou sequências de DNA é normalmente avaliado na literatura através do uso de métodos estatísticos básicos, tais como média aritmética e desvio padrão, o que pode ser insuficiente para garantir a confiança dos

resultados. A adoção de métricas de desempenho adequadas e técnicas estatísticas pode ser crucial para interpretar os dados, permitindo a inspeção da contribuição de fatores para os resultados, bem como uma predição de desempenho em experimentos posteriores.

Dentre as técnicas que podem ser utilizadas para avaliação de uma carga de trabalho [43], pode-se destacar a avaliação se a média de uma amostra (quantidade de experimentos realizados) é uma boa estimativa para a população. O primeiro passo para realizar esta avaliação é definir um nível de confiança. Este valor indica a probabilidade de os resultados da avaliação amostral corresponderem à realidade. Um nível de confiança de pelo menos 90%, por exemplo, pode ser usado para validar estatisticamente, com boa precisão, os resultados de desempenho de uma aplicação.

Definido o nível de confiança, é possível calcular o intervalo de confiança (CI - *Confidence Interval*) para a média. O CI é calculado utilizando a média da amostra, o desvio padrão e um fator estatístico obtido de tabelas que representam uma distribuição normal, tomando por base o número de experimentos e o nível de confiança. A fórmula utilizada quando a quantidade de experimentos é menor do que 30 é dada pela Equação 6.1, onde $\widehat{y}$ é a média, α é o coeficiente de confiança, n é o número de experimentos, s_y é o desvio padrão e $t_{[\alpha;n-1]}$ é a constante obtida da tabela estatística (*t-student*). Para um nível de confiança de 90%, o valor adotado para o coeficiente de confiança α é de 0,95, uma vez que o erro deve ser distribuído entre as duas extremidades do intervalo.

$$CI = \widehat{y} \pm \frac{t_{[\alpha;n-1]} * s_y}{\sqrt{n}} \tag{6.1}$$

Em alguns casos, um teste com zero pode ser usado para verificar se o valor medido é significativamente diferente de zero. Se o CI calculado inclui o zero, a média não pode ser considerada significativa no nível de confiança adotado [43]. Este procedimento é conhecido como "teste t". A Figura 6.1 mostra exemplo de testes t, onde cada linha vertical representa um experimento diferente: nos casos (a) e (b), os valores medidos falham no teste com zero.

Este teste pode ser útil quando o desempenho de duas soluções é comparado para identificar qual a melhor, através de uma comparação pareada. Para tanto, deve-se realizar as diferenças entre cada par de experimentos, e avaliar este novo conjunto de valores com o teste t. Se o CI desta diferença inclui o zero, as soluções não podem ser consideradas significativamente diferentes, pois os desempenhos são similares. Caso contrário, pode-se afirmar que uma solução é melhor ou pior que a outra dentro do nível de confiança adotado.

Outra ferramenta estatística frequentemente usada é o modelo de regressão linear [43], visando estimar o resultado de uma variável de resposta baseado variação de um (regressão

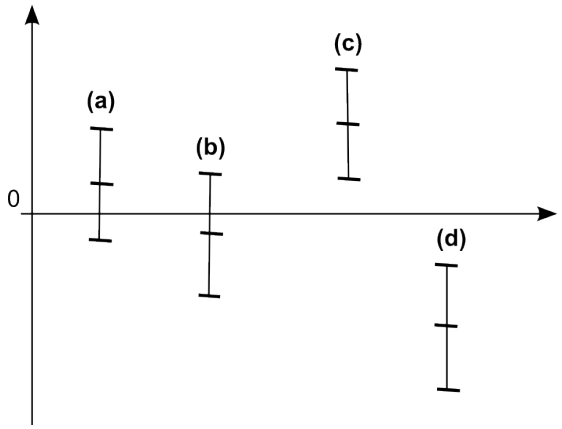


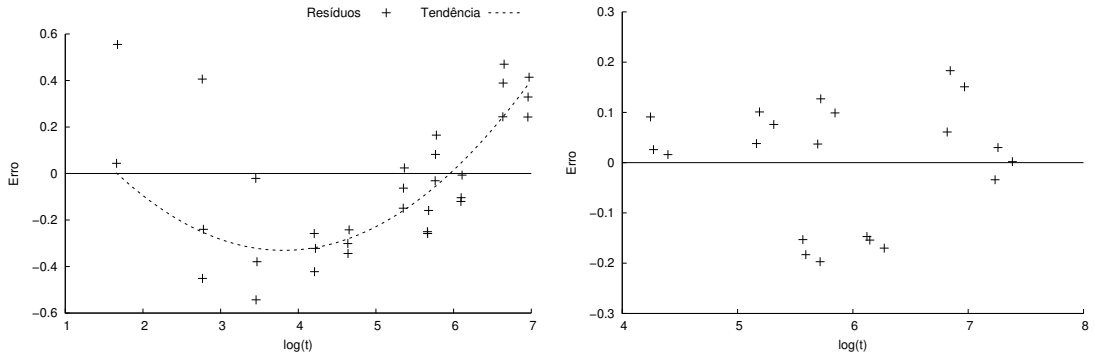
Figura 6.1: Teste t em quatro intervalos de confiança (adaptado de [43]).

linear simples) ou vários (regressão linear múltipla) fatores através de uma função que representa a relação. Os dados coletados durante os experimentos são utilizados para calcular os coeficientes da equação de regressão. Considerando b a matriz dos coeficientes da equação, X a matriz de fatores, X^T a matriz transposta e y o vetor com as variáveis de resposta, a Equação 6.2 é usada para determinar o modelo de regressão linear múltipla.

$$b = (X^T X)^{-1}(X^T y) \tag{6.2}$$

A qualidade de um modelo de regressão pode ser avaliada verificando-se o coeficiente de determinação (R^2) e fazendo-se uma inspeção visual do gráfico que avalia os resíduos (erro percentual calculado entre o valor estimado e o valor real, podendo ser positivo ou negativo) e as respostas previstas pela equação. A Figura 6.2 mostra exemplos de gráficos residual X previsto, onde os pontos relacionam um valor estimado pela equação (no eixo X) e o erro de estimativa (no eixo Y). Na Figura 6.2a existe uma tendência quadrática na distribuição do erro, como pode ser observado na linha de tendência tracejada, e, portanto, a regressão linear não é ideal. Já na Figura 6.2b, os dados estão distribuídos aleatoriamente no gráfico, sem que se possa ser identificada uma tendência, o que indica um bom resultado na regressão.

A qualidade do modelo da regressão pode também ser avaliada através da análise de variância (ANOVA - *Analysis of Variance*). As somas dos quadrados médios são testadas contra uma distribuição F. Estes valores são considerados significativamente diferentes se o valor computado para a variância é maior que o obtido de uma tabela de quantis F, o que é chamado de "teste F" [43].



(a) (b)

Figura 6.2: Gráfico residual X previsto: (a) tendência quadrática (b) relação homocedástica.

6.2 Avaliação Estatística entre MASA-CUDAlign e MASA-OpenCL

Uma análise estatística detalhada do desempenho da solução MASA-OpenCL (Seção 5.1.8) em GPU foi realizada com base em técnicas de análise quantitativa [27]. Para tanto, foram utilizadas quatro GPUs da NVidia (GTX 580, GTX 680, GTX 980 Ti e GTX 1080) e uma GPU AMD (R9 280X), avaliando-se comparações entre 12 pares de sequências com tamanhos variando entre 10 KBP e 56 MBP. Cada um dos experimentos foi repetido 3 vezes, e a média das execuções (expressas em GCUPS - Seção 2.5) foi adotada como referência de desempenho.

A primeira análise teve como objetivo validar se o número de repetições era adequado. Os resultados obtidos foram então avaliados usando um nível de confiança de 90%. Considerando que o número de repetições (n) para cada experimento foi 3, a média entre as execuções de cada par (y_1, y_2 e y_3) é designada como $\widehat{y}$ e o desvio padrão é s_y. As Equações 6.3 e 6.4 foram utilizadas para calcular o CI e o máximo erro de medição (E), respectivamente.

$$CI = \widehat{y} \pm \frac{t_{[0,95;2]} * s_y}{\sqrt{3}} \tag{6.3}$$

$$E = (s_y/\widehat{y}) * 100\% \tag{6.4}$$

A Tabela 6.1 mostra os resultados obtidos para comparação para a GPU GTX 680. Como observado, todos os experimentos têm valores próximos à sua respectiva média, com desvio padrão muito baixo. Isto ocorre porque os testes foram conduzidos em um ambiente isolado e controlado, com pouca interferência na execução. Alguns aspectos poderiam

Tabela 6.1: Intervalo de confiança - MASA-OpenCL GTX 680 (GCUPS).

Id	**Medidas**			**Média**	**Desvio**	**CI**		**Erro**
	y_1	y_2	y_1	$\widehat{y}$	s_y	**mín.**	**máx.**	**E (%)**
10Kx10K	2,000	2,100	2,100	2,067	0,058	1,969	2,164	4,710
57Kx57K	17,600	18,400	18,000	18,000	0,400	17,326	18,674	3,746
162Kx172K	34,800	35,100	34,900	34,933	0,153	34,676	35,191	0,737
543Kx536K	47,300	47,700	47,800	47,600	0,265	47,154	48,046	0,937
1Mx1M	56,400	56,500	56,500	56,467	0,058	56,369	56,564	0,172
3Mx3M	52,800	52,900	52,900	52,867	0,058	52,769	52,964	0,184
5Mx5M	106,400	106,600	106,500	106,500	0,100	106,331	106,669	0,158
7Mx5M	52,700	52,800	52,900	52,800	0,100	52,631	52,969	0,319
10Mx10M	106,000	106,200	106,300	106,167	0,153	105,909	106,424	0,243
23Mx25M	53,000	53,100	53,000	53,033	0,058	52,936	53,131	0,184
47Mx32M	78,900	79,200	79,000	79,033	0,153	78,776	79,291	0,326

interferir no resultado, como a taxa de sucesso no acesso aos dados em memória *cache*, mas, nos testes realizados, o desempenho final não é significativamente afetado. Além disso, o máximo erro calculado foi de 4,7%, sendo maior nas pequenas sequências, que estão mais susceptíveis a estas variações, uma vez que nestes casos o *wavefront* não explora os núcleos de GPU por muito tempo (Seção 2.4). Desta forma, podemos afirmar que a realização de 3 repetições dos experimentos é suficiente para gerar resultados significativos em um nível de confiança de 90%. Este resultado foi semelhante quando os desempenhos obtidos nas demais GPUs foram avaliados.

Os desempenhos do MASA-OpenCL e MASA-CUDAlign foram então avaliados através de uma comparação pareada (Seção 6.1) para confirmar estatisticamente os ganhos obtidos pelo MASA-OpenCL [25]. Para este fim, os testes com zero foram realizados considerando todos os pares de sequências comparados para cada GPU. O CI foi calculado de acordo com a Equação 6.5, onde $\widehat{y}$ é a média da diferença de desempenho entre MASA-OpenCL (y_O) e MASA-CUDAlign (y_C), t é o valor na tabela *t-student* para o nível de confiança desejado (90%) e s é o desvio padrão.

$$CI = \widehat{y} \pm \frac{t_{[0,95;11]} * s_{y_O - y_C}}{\sqrt{10}} \tag{6.5}$$

O CI resultante para a GPU NVidia GTX 680 foi (11,938 ; 19,771) e para a GTX 580 foi (1,517 ; 6,139). De acordo com o teste t, estes valores não incluem o zero, então pode-se afirmar que os ganhos de desempenho do MASA-OpenCL em relação ao MASA-CUDAlign, apesar de pequenos, são significativos no nível de confiança de 90% nas duas GPUs. Contudo, cabe ressaltar que, conforme descrito no trabalho que apresentou o MASA-OpenCL [25], os resultados obtidos pela solução foram comparáveis ao MASA-CUDAlign em uma mesma GPU NVidia quando duas modificações foram realizadas no

MASA-CUDAlign: mudança da área de armazenamento para memória global (em vez de textura) e compilação do programa em 32 bits. Neste novo cenário, os ganhos não foram verificados pela comparação pareada. O melhor desempenho do MASA-OpenCL foi atingido em uma GPU AMD (R9 280X).

6.3 Construção de um Modelo de Regressão Linear

A estimativa do tempo de execução de uma determinada comparação em uma GPU pode ser bastante útil em ambientes em que o acesso aos recursos não é ilimitado, seja pela alocação de um recurso em nuvem computacional ou em um ambiente no qual o acesso aos recursos é controlado por um escalonador de tarefas. Desta forma, optou-se por construir um modelo de regressão linear (Seção 6.1) que pudesse determinar o tempo na execução do MASA-OpenCL de um par de sequências em uma dada GPU. O MASA-OpenCL foi escolhido pois obteve melhor desempenho que o MASA-CUDAlign em uma GPU, ainda que pequenos.

O modelo inicial avaliado usava apenas o tamanho da matriz de programação dinâmica (matriz DP - Seção 2.2) como único fator da equação, e o tempo de execução como variável resposta. A métrica GCUPS (Seção 2.5) não foi considerada como saída da regressão porque ela é calculada baseando-se no tamanho da matriz (produto entre os tamanhos das sequências), então haveria uma correlação entre a variável de resposta e o fator escolhido. Além disso, uma vez que os tamanhos das sequências possuíam diferentes ordens de grandeza, uma transformação logarítmica foi utilizada para normalizar os valores. Contudo, a avaliação do modelo mostrou que uma regressão simples não era adequada para representar a execução do MASA-OpenCL em diferentes GPUs, uma vez que outros fatores interferem no desempenho.

Devido a este cenário, optou-se então por uma regressão linear múltipla, introduzindo dois novos fatores: o poder computacional da GPU (CP - *Computing Power*), calculado como o produto entre o número de núcleos da GPU e a frequência de cada núcleo (em MHz), e a largura de banda da memória global da GPU (BW - *Bandwidth*), medida em Gigabytes por segundo (GB/s).

Todos os resultados das comparações nas GPUs NVidia GTX 580, NVidia GTX 680 e AMD R9 280X foram usados como parâmetros. O coeficiente de determinação obtido com este modelo foi $R^2 = 0,949$, o que pode ser considerado adequado, e os resultados foram validados pelo teste F (Seção 6.1). Contudo, a inspeção visual dos gráficos quantil-quantil e residual X previsto não confirmaram a qualidade da regressão, especialmente nas comparações envolvendo sequências menos longas, porque, nestes casos, a execução

da solução não explora completamente os recursos da GPU, uma vez que menos dados são processados em paralelo.

Para melhorar a qualidade do modelo de regressão, uma técnica de clusterização [43] foi aplicada ao tamanho da matriz DP, visando caracterizar de forma mais adequada a carga de trabalho. Dois grupos similares foram identificados: comparações com sequências com 1 MBP ou mais, e sequências menores que este limite. Considerando que o MASA-OpenCL é voltado principalmente para a comparação de sequências longas, o primeiro grupo foi usado como referência para determinar o modelo.

Adicionalmente, as mesmas sequências foram comparadas em uma GPU NVidia GTX 980 Ti, e os resultados das comparações com pelo menos 1 MBP foram adicionados aos resultados anteriores, aumentando o número de experimentos para 28. Além disso, outro fator foi incluído no modelo: a função `log` aplicada ao percentual de blocos descartados durante o processamento do algoritmo devido à técnica de *block pruning* (BP - Seção 2.3.3), uma vez que o desempenho tende a ser superior quando as sequências são mais similares.

A técnica de regressão linear múltipla foi então aplicada ao novo conjunto de dados com uma amostra de tamanho $n = 28$, resultando em um coeficiente de determinação $R^2 = 0,999$, superior ao obtido anteriormente, resultado validado pelo teste F em um nível de confiança de 90%. O modelo de regressão multilinear obtido está representado na Equação 6.6, onde m e n são os tamanhos das sequências, CP é o poder computacional da GPU, BW é a largura de banda da memória global e BP é a taxa de *block pruning*. O tempo estimado (em segundos) pode ser obtido aplicando-se a função exponencial à variável de resposta $log(t)$. Como pode ser observado na Equação 6.6, a taxa de descarte de blocos (BP) é um fator relevante na regressão e, portanto, essa taxa deve ser levada em consideração na estimativa do tempo de execução.

$$log(t) = -3,036 + 0,979 * log(m*n) - 0,344 * log(CP) - 1,001 * log(BW) + 0,777 * log(1-BP) \tag{6.6}$$

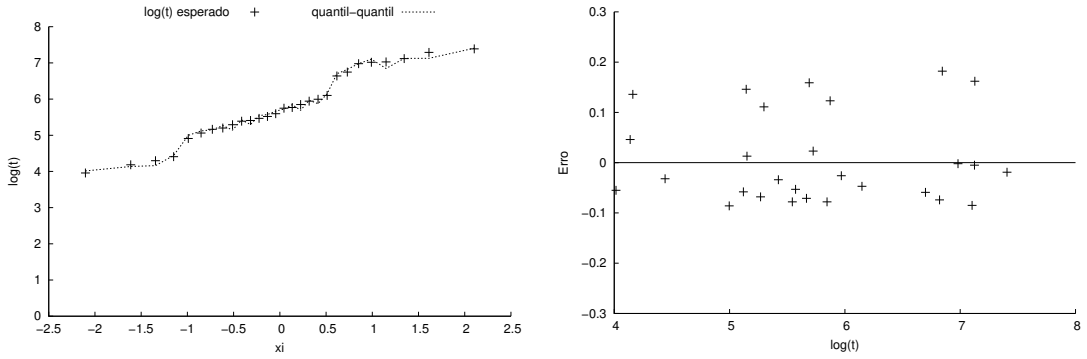
Os resultados obtidos na regressão linear múltipla podem ser vistos na Tabela 6.2, onde a coluna "GPU" representa o modelo da GPU utilizada, a coluna "$log(t)$ obs." contém o tempo de execução real observado aplicando-se a função `log`, a coluna "$log(t)$ est." contém o tempo de estimado pela equação (também aplicando-se a função `log`) e a coluna "Dif." é a diferença entre as duas medidas anteriores. As colunas "t obs." e "t est." são os valores absolutos dos tempos reais e estimados e, finalmente, a coluna "Erro (%)" é o erro relativo entre o tempo estimado e o observado.

Tabela 6.2: Experimentos em GPU e resultados da regressão linear.

Comp.	**GPU**	$log(t)$ **obs.**	$log(t)$ **est.**	**Dif.**	t **obs.**	t **est.**	**Erro (%)**
1Mx1M	R9 290X	4,184	4,090	0,094	15,272	12,296	19,49
	GTX 680	4,297	4,307	-0,009	19,821	20,257	-2,20
	GTX 580	4,408	4,407	0,001	25,607	25,553	0,21
	GTX 980	3,959	3,974	-0,014	9,100	9,408	-3,39
3Mx3M	R9 290X	5,062	5,072	-0,010	115,284	118,031	-2,38
	GTX 680	5,291	5,290	0,001	195,266	194,946	0,16
	GTX 580	5,389	5,390	-0,001	244,982	245,297	-0,13
	GTX 980	4,912	4,957	-0,045	81,570	90,518	-10,97
5Mx5M	R9 290X	5,201	5,222	-0,020	158,955	166,637	-4,83
	GTX 680	5,409	5,444	-0,035	256,717	278,127	-8,34
	GTX 580	5,519	5,539	-0,021	330,164	373,311	-4,89
	GTX 980	5,165	5,111	0,054	146,145	129,170	11,62
7Mx5M	R9 290X	5,595	5,619	-0,024	393,705	415,855	-5,63
	GTX 680	5,850	5,837	0,013	708,009	686,865	2,99
	GTX 580	5,943	5,937	0,007	877,615	864,245	1,52
	GTX 980	5,467	5,504	-0,037	292,849	318,928	-8,91
10Mx10M	R9 290X	5,767	5,798	-0,031	584,629	627,482	-7,33
	GTX 680	5,995	6,018	-0,024	988,060	1.043,178	-5,58
	GTX 580	6,101	6,115	-0,014	1.262,279	1.304,057	-3,31
	GTX 980	5,748	5,685	0,063	560,398	484,485	13,55
23Mx25M	R9 290X	6,747	6,774	-0,027	5.580,333	5.941,186	-6,47
	GTX 680	7,027	6,989	0,038	10.648,382	9.752,734	8,41
	GTX 580	7,119	7,092	0,027	13.148,263	12.347,194	6,09
	GTX 980	6,640	6,659	-0,019	4.361,168	4.556,163	-4,47
47Mx32M	R9 290X	7,018	7,053	-0,035	10.421,302	11.293,462	-8,37
	GTX 680	7,290	7,271	0,018	19.480,545	18.684,713	4,09
	GTX 580	7,387	7,373	0,013	24.350,681	23.631,105	2,96
	GTX 980	6,978	6,941	0,037	9.510,969	8.729,489	8,22

O erro máximo (em módulo) do tempo de execução estimado foi de 19,5% e a média dos erros foi de 5,9%. Estes resultados mostram uma boa qualidade do modelo de regressão, obtendo-se a estimativa do tempo de execução com boa aproximação. Observando a Figura 6.3 e analisando os gráficos quantil-quantil (Figura 6.3a) e residual X resposta prevista (Figura 6.3b), observa-se que os resultados mostram a boa qualidade da regressão, indicando uma tendência homocedástica.

A Equação 6.6 foi utilizada para estimar os tempos de execução em uma nova GPU (NVidia GTX 1080), comparando as estimativas com os tempos de execução reais. Nesse caso, o erro máximo obtido entre os valores dos tempos estimado e observado foi de 8,77%, sendo mais significativo nas sequências menores. A Figura 6.4 mostra a relação entre os valores esperados e observados (em $log(t)$) para cada par de sequências.



(a) (b)

Figura 6.3: Regressão linear múltipla: (a) quantil-quantil (b) residual X previsto.

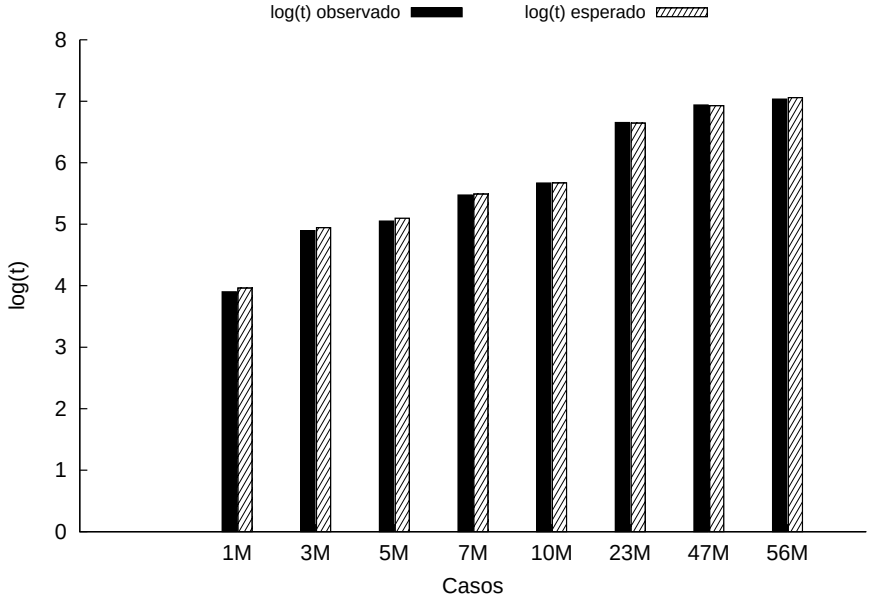


Figura 6.4: Tempo de execução esperado X observado em GPU GTX 1080.

6.4 Conclusão do Capítulo

Neste capítulo foram apresentadas algumas técnicas estatísticas que permitem avaliar resultados obtidos por soluções computacionais. Elas foram aplicadas na avaliação do desempenho das soluções MASA-CUDAlign e MASA-OpenCL, visando validar os resultados apresentados em [25] dentro de um nível de confiança de 90%. A análise dos GCUPS das duas soluções ratificou os ganhos obtidos pelo MASA-OpenCL através de uma comparação pareada, mostrando que eles foram significativos.

Os resultados da análise quantitativa mostram que a equação de regressão linear múltipla proposta para o MASA-OpenCL em uma GPU apresentou alto coeficiente de determinação e erro máximo aceitável na comparação entre os valores previsto e observado para a GPU GTX 1080. O trabalho permitiu um conhecimento mais detalhado do funcionamento da estratégia de comparação de sequências biológicas com *pruning*, fornecendo subsídios para o projeto de algoritmos de distribuição de carga estática e dinâmica que serão mostrados nos Capítulos 8 e 9, respectivamente. Além disso, a predição do tempo de execução da solução MASA-OpenCL em uma GPU através da Equação 6.6 pode ser utilizada na previsão de alocação do recurso para uma determinada comparação, o que pode ser útil em locais onde o acesso à GPU é controlado.

A proposta e os resultados apresentados neste capítulo foram publicados em [27].

Capítulo 7

Avaliação de Execuções MASA em Ambiente Híbrido (GPU e CPU)

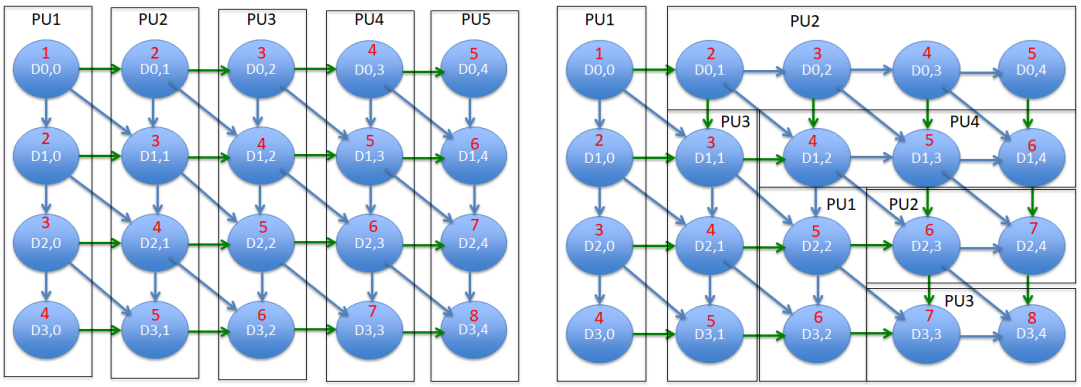
A segunda contribuição desta Tese é uma avaliação da execução das soluções MASA-OpenMP e MASA-CUDAlign (Seção 5.1.7) em ambiente híbrido composto de CPU e GPU, além de uma análise de aspectos relacionados à distribuição de carga (Seção 4.6) no problema de comparação de sequências. A comparação de sequências biológicas em múltiplos dispositivos demanda estratégias de distribuição de carga, e alguns trabalhos já abordaram este tema (Seção 5.3). Contudo, a forma de processamento em diagonal usualmente utilizada em soluções que comparam sequências longas de DNA faz com que esta tarefa não seja trivial, devido principalmente à dependência de dados. Neste capítulo, então, é apresentada uma análise do desempenho destas soluções em várias simulações de distribuição de carga em múltiplos dispositivos. Esta análise serviu como base para a construção da estratégia de distribuição dinâmica de carga de trabalho que será discutida no Capítulo 9.

Este capítulo está organizado da seguinte forma. Na Seção 7.1 são discutidos alguns aspectos inerentes à distribuição de carga de trabalho em soluções de comparação de sequências biológicas. A seguir, a Seção 7.2 descreve as variáveis avaliadas nos experimentos e o planejamento dos testes. Já a Seção 7.3 detalha e avalia os resultados obtidos. Finalmente, a Seção 7.4 conclui o capítulo.

7.1 Aspectos da Distribuição de Carga no Problema da Comparação de Sequências

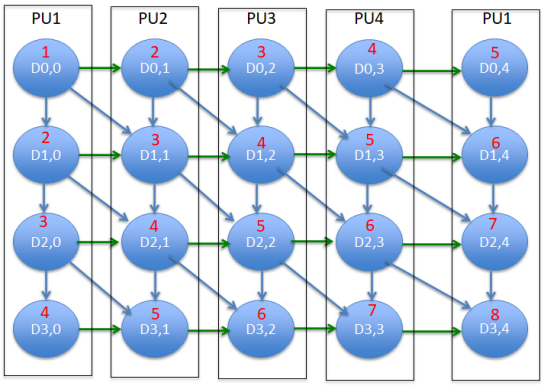
Visando discutir as estratégias a serem avaliadas na execução das soluções em ambiente híbrido, a forma de processamento utilizada nas soluções MASA foi discutida de forma

conceitual com pesquisadores do *Institut Polytechnique de Grenoble* (INP), em Grenoble, França, buscando formas de distribuição das tarefas entre unidades de processamento de forma a minimizar o tempo de execução. Alguns modelos teóricos de divisão e comunicação de tarefas foram adaptados para o caso de comparação de sequências, e um grafo de dependências de tarefas (Seção 4.2) da aplicação foi derivado. As Figuras 7.1a e 7.1b apresentam o processamento em colunas e o processamento em blocos *square*, respectivamente, para o grafo em questão, enquanto a Figura 7.1c representa um processamento em colunas usando ciclos.



(a) Modelo em colunas padrão.

(b) Modelo em blocos *square*.



(c) Modelo em colunas cíclico.

Figura 7.1: Modelos de divisão e comunicação de tarefas.

Na Figura 7.1, cada círculo representa uma tarefa, que pode ser uma célula ou um bloco de células da matriz de programação dinâmica (DP). As setas representam dependências de dados entre as tarefas. Os números em vermelho nos círculos indicam a ordem na qual as tarefas são executadas, enquanto os retângulos agrupam as tarefas que são executadas por uma determinada unidade processamento (PU - *Processing Unit*).

Observando a Figura 7.1, pode-se ver que, apesar de exigirem a mesma quantidade de comunicações entre PUs (16), o modelo em blocos *square* (b) requer menos PUs do que o

modelo em colunas padrão (a) para calcular a mesma quantidade de elementos da matriz (4 PUs, em vez de 5), como pode ser visto nas Figuras 7.1a e 7.1b.

As soluções MASA (Seção 5.1.7) já possuem uma forma otimizada de comunicação entre dispositivos utilizando *buffers* de entrada e saída de dados através de *threads* de comunicação (Seção 5.2.1.6), baseando o processamento em um *wavefront* (Seção 2.4) diagonal ou genérico. Admitindo-se que o processamento é em *wavefront* diagonal (com sincronização no final do cálculo de cada diagonal) e que uma mesma PU pode enviar e receber mensagens ao mesmo tempo (por exemplo, usando *threads* de comunicação), o modelo de colunas cíclico (Figura 7.1c) também permite que 4 PUs sejam utilizadas, já que no começo do segundo ciclo (diagonal 5) a $PU1$ já pode ser reutilizada.

7.2 Planejamento de Testes

Em paralelo com a análise conceitual, foram feitos vários testes de execução de extensões do MASA para CPU (MASA-OpenMP) e GPU (MASA-CUDAlign) para avaliar empiricamente o comportamento das soluções em função da distribuição dos dados entre os dispositivos. O ambiente de testes foi composto por duas máquinas do Laboratório de Sistemas Integrados e Concorrentes (Laico) da Universidade de Brasília (UnB), cada qual com um processador i7-3770 com 4 núcleos e uma placa gráfica NVidia, dos modelos GTX 680 e GTX 980 Ti. Os dois *hosts* são conectados por uma rede Gigabit. Diversos parâmetros foram avaliados nos experimentos, a saber:

- **Quantidade de dispositivos:** um único (CPU ou GPU), para servir como referência de comparação para os resultados em múltiplos dispositivos, dois (2 GPUs ou 1 CPU com 1 GPU) e quatro (2 CPUs com 2 GPUs);
- **Sequências:** foram utilizados dois pares de sequências com aproximadamente 1 MBP (referências CP000051.1 e AE002160.2 no *National Center for Biotechnology Information* - NCBI) e 5 MBP (referências AE016879.1 e AE017225.1). Estas sequências foram selecionadas pois apresentam similaridade média ou alta (cenário desejado para os testes), além de não levarem um tempo elevado para obter o escore ótimo, o que permite a realização de mais testes em menor espaço de tempo;
- **Divisão de colunas entre dispositivos:** 1:50, 1:100, 1:150 e 1:200 entre CPU:GPU com dois dispositivos, e 1:200:1:460 e suas variações em relação à ordem de execução entre CPU1:GPU1:CPU2:GPU2 em quatro dispositivos;
- **Ordem de execução:** iniciando pelas CPUs ou iniciando pelas GPUs;
- **Tamanho dos blocos em CPU:** 512, 1.024 ou 2.048;

- **Tamanho dos blocos em GPU:** 128, 256 ou 512.

O objetivo dos testes era avaliar o impacto de cada um dos fatores no desempenho de uma execução em ambiente heterogêneo ou híbrido, visando obter uma estratégia que possa ser utilizada na execução das soluções em plataformas que apresentam diferença de desempenho significativo entre os dispositivos. Além do desempenho em si, foi avaliada também a taxa de preenchimento dos *buffers* de comunicação (Seção 5.2.1.6), que foram propostos no CUDAlign 3.0 e incluídos no projeto do *framework* MASA (Seção 5.1.7). As execuções foram realizadas sem o descarte de blocos, uma vez que as extensões MASA não permitem esta funcionalidade na execução em múltiplos dispositivos. Os desempenhos obtidos estão expressos em GCUPS (Seção 2.5).

7.3 Resultados Experimentais

Os testes iniciais foram realizados combinando a CPU i7 e a GPU GTX 980 Ti, instaladas no mesmo *host*, com diferentes divisões de colunas entre os dispositivos (campo "Div." nas tabelas a seguir) e configurações de tamanho de bloco, considerando a execução iniciando pela CPU e depois iniciando pela GPU. Os resultados da comparação das sequências de 1 MBP foram compilados na Tabela 7.1.

Como pode ser observado na Tabela 7.1, a escolha de uma boa distribuição de colunas é relevante para a obtenção de melhores resultados, visto que os desempenhos variaram de 34,60 GCUPS a 86,69 GCUPS para a sequência 1M em 1 GPU e 1 CPU. Cabe ressaltar que a determinação de uma boa distribuição não é uma tarefa trivial: embora a diferença de desempenho na execução apenas com 1 GPU (93,88 GCUPS) e 1 CPU (1,00 GCUPS) seja na ordem de 94:1, os melhores resultados foram obtidos com uma relação 200:1. A ordem de execução da matriz interfere também nos resultados, principalmente quando mais colunas são destinadas à GPU. Quando a execução se inicia pela CPU, ocorre uma retenção de dados que afeta o desempenho. A configuração de tamanho de blocos que obteve melhor resultado (86,69 GCUPS) foi de 128 para GPU e 2.048 para CPU, muito embora estas variáveis não sejam as que mais afetam o resultado. Entretanto, podemos observar que o melhor resultado obtido no ambiente híbrido (86,69 GCUPS) foi inferior ao obtido à execução apenas com a GPU GTX 980 Ti (93,88 GCUPS).

Em um outro experimento com dois dispositivos, o MASA-CUDAlign foi executado para comparar os pares de sequência de 1M e 5M, dividindo-se o processamento da matriz entre as 2 GPUs, utilizando-se 128 e 256 *threads*, e comparando-se o desempenho obtido na execução da solução em cada uma das GPUs individualmente. Os resultados podem ser vistos na Tabela 7.2.

Tabela 7.1: Comparação 1M em ambiente híbrido com 2 dispositivos (CPU + GPU).

Disposit.	**Div.**	**Blocos (CPU)**	**Blocos (GPU)**	**GCUPS**	**Disposit.**	**Div.**	**Blocos (CPU)**	**Blocos (GPU)**	**GCUPS**
CPU	-	512	-	0,91	GPU	-	-	128	92,93
		1.024		1,00				256	83,43
		2.048		1,00				512	**93,88**
CPU+GPU	1:50	512	128	39,33	GPU+CPU	50:1	512	128	35,37
		512	256	37,79			512	256	34,99
		512	512	38,73			512	512	34,60
		1.024	128	39,32			1.024	128	38,43
		1.024	256	37,85			1.024	256	37,34
		1.024	512	38,76			1.024	512	37,06
		2.048	128	39,34			2.048	128	37,29
		2.048	256	37,79			2.048	256	36,96
		2.048	512	38,67			2.048	512	35,92
	1:100	512	128	61,97		100:1	512	128	58,61
		512	256	59,45			512	256	57,55
		512	512	60,21			512	512	54,55
		1.024	128	61,56			1.024	128	62,18
		1.024	256	59,00			1.024	256	61,91
		1.024	512	60,05			1.024	512	58,35
		2.048	128	61,97			2.048	128	67,07
		2.048	256	59,33			2.048	256	63,15
		2.048	512	60,55			2.048	512	61,23
	1:150	512	128	68,56		150:1	512	128	67,46
		512	256	64,15			512	256	66,94
		512	512	66,39			512	512	65,01
		1.024	128	67,32			1.024	128	85,76
		1.024	256	63,60			1.024	256	73,46
		1.024	512	65,63			1.024	512	78,98
		2.048	128	67,63			2.048	128	84,94
		2.048	256	63,33			2.048	256	78,06
		2.048	512	65,33			2.048	512	79,18
	1:200	512	128	66,43		200:1	512	128	77,15
		512	256	63,58			512	256	75,88
		512	512	68,93			512	512	72,99
		1.024	128	66,30			1.024	128	85,50
		1.024	256	65,06			1.024	256	76,23
		1.024	512	69,02			1.024	512	80,43
		2.048	128	66,39			2.048	128	**86,69**
		2.048	256	63,54			2.048	256	77,04
		2.048	512	69,50			2.048	512	82,90

Como pode ser observado, a execução em 2 GPUs otimiza o desempenho da solução se comparado a uma GPU única, como era de se esperar. A utilização de blocos com tamanho 512 apresentou melhor desempenho para a comparação 1M se comparado com a execução com blocos de tamanho 128, mas este resultado não pode ser tomado como padrão, pois a melhor configuração na comparação 5M foi com blocos de tamanho 128. Com efeito, a configuração de tamanho de blocos mais adequada é muito afetada pela arquitetura da GPU e pelo tempo de execução da comparação, e portanto assumimos que

Tabela 7.2: Comparações 1M e 5M em ambiente heterogêneo com 2 dispositivos (GPU + GPU)

Id	Div.	Blocos	Tempo de Execução (ms)		GCUPS
			GTX 680	GTX 980	
1M	-	128	28.119,2	-	39,9
		256	26.020,5	-	43,1
		512	24.409,9	-	45,9
	-	128	-	11.412,1	98,2
		256	-	13.431,9	83,4
		512	-	11.325,6	98,9
	10:15	128	13.376,6	11.891,7	83,8
		512	12.582,6	11.826,8	89,1
	10:20	128	11.285,9	9.730,1	99,3
		512	9.987,6	10.346,7	108,3
	10:25	128	10.968,9	8.863,1	102,2
		512	8.877,6	9.805,8	114,3
5M	-	128	679.890,1	-	40,2
		256	654.485,5	-	41,8
		512	600.353,6	-	45,5
	-	128	-	268.426,2	101,8
		256	-	388.077,8	70,4
		512	-	292.712,4	93,4
	10:15	128	270.523,8	270.190,1	101,0
		512	238.783,7	245.410,9	111,4
	10:20	128	229.357,5	229.902,1	118,9
		512	202.902,8	210.223,8	130,0
	10:25	128	195.449,0	196.665,1	**139,0**
		512	172.197,6	218.696,2	125,0

uma configuração adequada de tamanho de bloco é feita. Tal configuração geralmente é obtida de maneira empírica, variando-se o tamanho do bloco na execução de comparações de sequências menores (e.g. 1M e 5M). Pode-se notar também que, quando mais colunas são distribuídas proporcionalmente ao segundo dispositivo (campo "Div." na Tabela 7.2), os GCUPS obtidos são maiores, visto que há menos retenção de trabalho no primeiro dispositivo (GTX 680), que possui desempenho inferior. Os melhores desempenhos nas comparações das sequências 1M e 5M foram obtidos com a divisão 10:25 - 114,3 GCUPS e 139,0 GCUPS, respectivamente.

No próximo experimento, os quatro dispositivos foram utilizados para execução das comparações 1M e 5M. A Tabela 7.3 mostra os resultados da comparação 5M nas duas máquinas (1 CPU e 1 GPU em cada), com diferentes configurações de execução e dois tamanhos de blocos nas GPUs (128 e 512). A coluna "Div." mostra a relação de distribuição de colunas entre os dispositivos e a ordem em que os programas são executados: o valor

200:1:460:1, por exemplo, reflete que para cada coluna em cada CPU são atribuídas 200 colunas à GPU GTX 680 e 460 colunas para a GPU GTX 980 Ti, sendo que, em cada *host*, as primeiras colunas são processadas pelo MASA-CUDAlign (GPU) e as seguintes pelo MASA-OpenMP (CPU), ou seja, na ordem GPU680-CPU1-GPU980-CPU2. As colunas seguintes na Tabela 7.3 indicam o tamanho dos blocos e os tempos de execução obtidos em cada caso. A última coluna mostra o desempenho da execução total.

Tabela 7.3: Execução em ambiente híbrido com 4 dispositivos na comparação 5M.

Div.	**Blocos**	**Tempo de Execução (ms)**				**GCUPS**
		GTX 680	**CPU1 i7**	**GTX 980**	**CPU2 i7**	
200:1:460:1	128	207.110,8	207.096,2	207.926,0	208.446,4	131,1
	512	180.852,3	181.897,1	210.385,2	210.651,2	129,7
200:1:1:460	128	207.400,1	207.323,9	207.957,5	206.230,0	**131,4**
	512	180.553,7	181.595,0	209.690,7	180.608,7	130,3
1:200:460:1	128	207.828,6	163.118,5	209.085,7	209.260,4	130,6
	512	181.349,9	139.741,0	210.174,3	211.273,6	129,4
1:200:1:460	128	208.015,5	163.145,0	206.452,1	208.540,7	131,1
	512	182.273,0	140.201,3	209.954,7	181.619,7	130,2

Como pode-se verificar, a utilização da configuração de blocos de tamanho 128 causa maior esforço de processamento no primeiro computador (com GPU GTX 680), enquanto com blocos de tamanho 512 a execução leva mais tempo no segundo computador. A ordem de execução (primeiro CPU e depois GPU, ou vice-versa) não influencia no desempenho geral, visto que os GCUPS são bem semelhantes. Entretanto, da mesma forma que no experimento com dois dispositivos, o melhor resultado obtido nesta configuração (131,4 GCUPS) foi inferior ao obtido quando apenas as 2 GPUs são utilizadas (139,0 GCUPS - Tabela 7.2), reforçando a ideia de que a adoção de um ambiente híbrido com CPUs não gera ganhos de desempenho (MASA-CUDAlign e MASA-OpenMP) em relação ao ambiente heterogêneo com GPUs (somente MASA-CUDAlign).

Em outro teste, diferentes valores de divisão de trabalho (colunas da matriz de programação dinâmica) foram testados na execução das comparações entre sequências de 1 MBP e 5 MBP utilizando 1 CPU e 1 GPU, com duas diferentes ordens de execução: CPU processando as primeiras colunas e GPU as últimas, e vice-versa. Os testes foram realizados acrescentando duas colunas à quantidade de colunas distribuídas para a GPU a cada caso de teste. Na comparação com sequências de 1 MBP, foram utilizados os limites 1:50 a 1:200 (CPU:GPU), enquanto na comparação com sequências 5 MBP os testes variaram entre 1:100 e 1:220. Pode-se verificar na Figura 7.2 que os tempos de execução decresceram até uma certa distribuição de colunas, que em ambos os casos ocorreu por volta da relação 1:134. A partir deste ponto, o tempo de execução permanece praticamente está-

vel. Verifica-se também que a ordem dos dispositivos não gera impacto no desempenho: as duas curvas (GPU:CPU e CPU:GPU) em cada uma das comparações avaliadas (1M e 5M) são bem semelhantes.

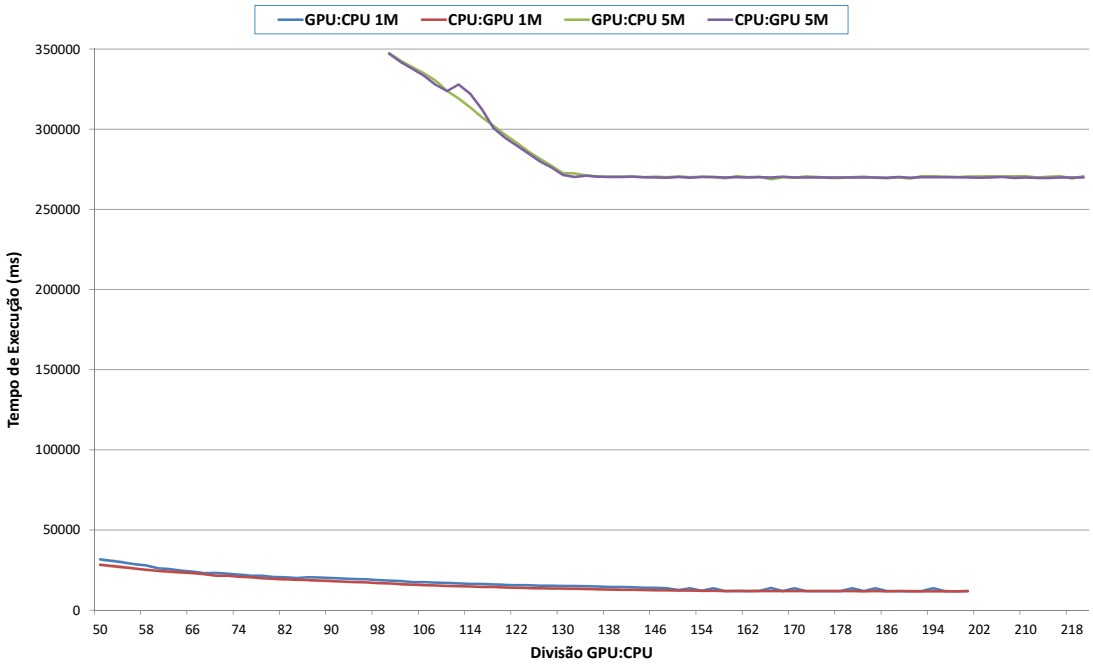


Figura 7.2: Tempo de execução em ambiente GPU:CPU variando as divisões de trabalho.

Os resultados experimentais indicam que uma escolha inadequada da divisão de colunas em uma execução em múltiplos dispositivos pode afetar o desempenho, visto que o primeiro dispositivo pode gerar retenções nos *buffers* de comunicação que afetam o fluxo de processamento nos demais dispositivos. Torna-se importante, então, escolher uma distribuição estática de trabalho adequada para que o desempenho não seja afetado.

7.4 Conclusão do Capítulo

Os resultados obtidos nos experimentos realizados neste Capítulo confirmam a alta capacidade de processamento das GPUs em relação às CPUs, o que requer que uma distribuição adequada seja escolhida para melhorar o desempenho no caso de execuções com extensões MASA em ambiente híbrido ou em ambientes heterogêneos onde a diferença de desempenho entre os dispositivos seja muito discrepante. Por outro lado, conforme observado nos testes em ambiente híbrido, as execuções envolvendo CPU e GPU utilizando o MASA-OpenMP e MASA-CUDAlign, respectivamente, não geram ganho de desempenho se comparado à execução apenas em GPUs. Já em relação à execução em 2 GPUs, o ganho obtido na execução no ambiente uniforme heterogêneo é bem relevante se comparado

à execução em apenas 1 GPU, aproximando-se o tempo de execução, em alguns casos, do limite teórico (soma do desempenho individual de cada uma das GPUs). Ademais, os resultados são mais significativos nos testes com a sequência de 5 MBP, pois com sequências menores o paralelismo máximo da GPU não é totalmente explorado.

Vale ressaltar que os experimentos foram conduzidos sem o uso da técnica de descarte de blocos e com distribuição estática de carga de trabalho, pois estas são as condições disponíveis nas extensões MASA na execução em múltiplos dispositivos. A implementação destas funcionalidades na execução em múltiplas GPUs pode melhorar o desempenho, o que motivou o desenvolvimento das soluções que serão apresentadas nos Capítulos 8 e 9.

Capítulo 8

Static-MultiBP: Comparação de Sequências Longas em Múltiplas GPUs com *Pruning*

O Static-MultiBP [28] é a terceira contribuição desta Tese. O objetivo do Static-MultiBP é permitir que o descarte de blocos seja utilizado em execuções com múltiplas GPUs, com baixo *overhead*. Para se aproximar de um cenário de execução do descarte de blocos em GPU única, optou-se por projetar uma estratégia de envio periódico do melhor escore obtido em cada GPU para a GPU vizinha à direita, organizando as GPUs em um anel. Isto permitiu se aproximar de uma visão global (considerando o BP em todas as GPUs) em relação ao melhor escore corrente, permitindo que a técnica de *pruning* pudesse ser aplicada com ganhos significativos para múltiplas GPUs. Adicionalmente, o Static-MultiBP foi projetado para execução não apenas em ambientes uniformes homogêneos, onde as GPUs possuem a mesma capacidade computacional (Seção 4.3), mas também em ambientes uniformes heterogêneos, permitindo uma distribuição não equitativa de colunas entre os dispositivos.

O Static-MultiBP foi integrado à solução MASA-CUDAlign, pois ela oferece o melhor desempenho da literatura para múltiplas GPUs (Tabela 5.2.3), e utiliza o descarte de blocos (Seção 2.3.3) em 1 GPU. Além disso, seu código está disponível publicamente para *download* [1]. Como base para o projeto da solução, foi utilizado o conceito de BP diagonal (Seção 2.3.3), apresentado inicialmente no CUDAlign 2.1 (Seção 5.1.5), e os conceitos de *buffers* de comunicação e execução em múltiplas GPUs presentes no CUDAlign 4.0 (Seção 5.2.1.6), com desenvolvimento em C++ utilizando *pthreads* (Seção 3.2.3) e CUDA (Seção 3.2.1). A decisão pela utilização da linguagem CUDA (em vez de OpenCL) e GPUs NVidia (em vez de AMD) como plataformas para as soluções propostas deve-se à

[1] Acessível em `https://github.com/edanssandes/MASA-CUDAlign` (fevereiro, 2021).

evolução das versões CUDA nos últimos anos, além da possibilidade de obter melhores desempenhos utilizando os modelos de GPUs pra *datacenter* recentemente lançados pela NVidia (Seção 3.1), que apresentam capacidade de processamento bem superior às GPUs AMD mais atuais.

O restante deste capítulo está organizado da seguinte forma. A Seção 8.1 avalia aspectos do descarte de blocos em um único dispositivo. A seguir, a Seção 8.2 apresenta o projeto do Static-MultiBP. Na Seção 8.5 são mostrados os resultados experimentais obtidos em ambientes homogêneos e heterogêneos. Finalizando, a Seção 8.6 conclui o capítulo.

8.1 Avaliação do *Block Pruning* em 1 GPU

No CUDAlign 2.1 (Seção 5.1.5) e no MASA-CUDAlign (Seção 5.1.7), o descarte de blocos é proposto com um processamento em *wavefront* (Seção 2.4) diagonal. Nele, os blocos em cada extremidade de cada diagonal externa são avaliados pela CPU para verificar a oportunidade de *pruning*, comparando-se o máximo valor que pode ser obtido a partir de um determinado ponto com o melhor escore global corrente (CGBS). Caso o CGBS seja maior, o bloco é descartado, acelerando a computação. Isto pode reduzir a quantidade de blocos processados pela GPU, principalmente se sequências similares são comparadas. A Figura 8.1 mostra a computação da matriz de programação dinâmica (DP) com a técnica de BP em uma GPU. Os blocos dentro da diagonal corrente ($d+1$, com bordas em negrito e fundo branco) serão verificados para determinar se serão descartados ou não, tomando como base os limites definidos na diagonal anterior (d, com bordas tracejadas). Na Figura 8.1, os blocos com fundo pontilhado foram processados, os blocos em cinza foram descartados e os blocos com fundo em branco ainda não foram avaliados.

No CUDAlign 3.0 (Seção 5.2.1.6) e no CUDAlign 4.0, a comparação de sequências em múltiplas GPUs é proposta, mas sem utilizar o BP. Neste caso, cada GPU_i recebe informações relativas à última coluna processada pela GPU_{i-1}, mas possui apenas sua visão local. Para que o BP possa funcionar em múltiplos dispositivos, cada dispositivo então deve considerar sua ordem em relação às demais GPUs. Tomando por base a Figura 2.12 (Seção 2.3.3), a coordenada horizontal da célula/bloco deve então ser recalculada, assim como a distância para a última coluna da matriz (Δj), que será processada pela última GPU. Desta forma, o valor do escore máximo que pode ser obtido a partir de uma célula em qualquer GPU pode ser calculado corretamente.

Esta modificação, contudo, é insuficiente para que se obtenha uma boa taxa de *pruning*. Isto porque cada GPU conheceria apenas seu melhor escore local corrente (CLBS - *Current Local Best Score*), sem considerar eventuais melhores escores obtidos por outras GPUs.

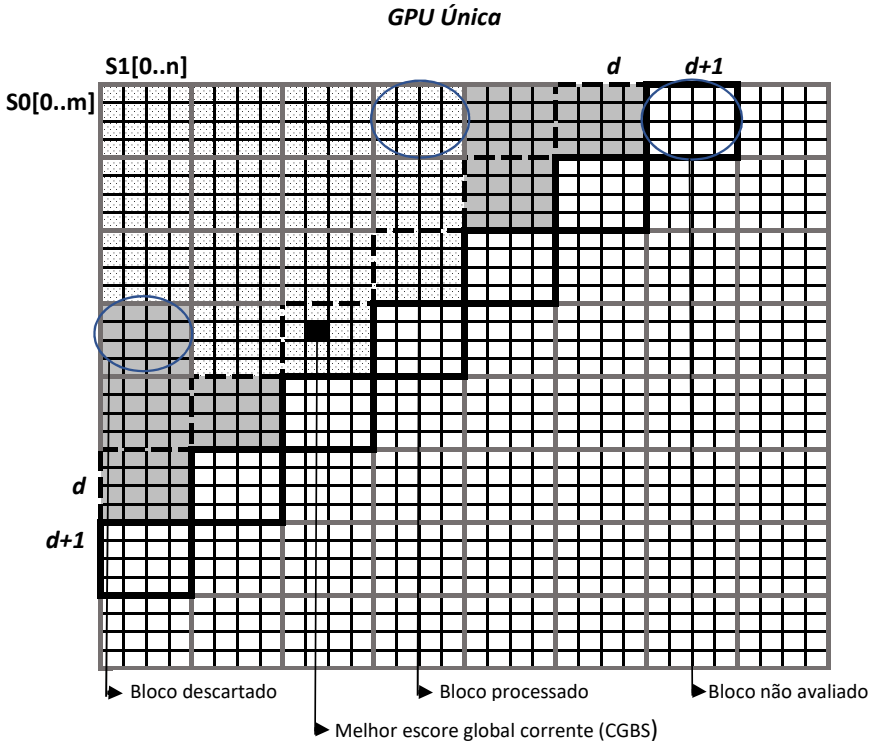


Figura 8.1: Método de *pruning* diagonal usado no CUDAlign 2.1 em uma GPU.

Torna-se importante, portanto, projetar uma forma de simular (mesmo que de forma aproximada) uma visão global do melhor escore, para diminuir a quantidade de blocos processados por cada GPU. Esta funcionalidade, chamada de compartilhamento periódico do melhor escore local, foi também incorporada no projeto do Static-MultiBP, que será apresentado na seção seguinte.

8.2 Projeto do Static-MultiBP

O Static-MultiBP é uma solução para comparação de sequências longas de DNA em múltiplas GPUs com descarte de blocos, que tem como base uma distribuição estática de carga de trabalho e o compartilhamento dinâmico do melhor escore local, visando ganho de desempenho em ambientes com várias GPUs. Os desafios no projeto do Static-MultiBP foram os seguintes: (a) garantir que o projeto do Static-MultiBP em múltiplas GPUs não afete negativamente o desempenho; (b) adaptar o cálculo do BP um ambiente de múltiplas GPUs; e (c) disseminar o melhor escore local obtido pelas GPU com uma abordagem leve, com pouca interferência na computação das matrizes de programação dinâmica.

Tanto o CUDAlign 4.0 (Seção 5.2.1.6) como o CUDAlign 2.1 (Seção 5.1.5), que agora estão integrados no MASA-CUDAlign, foram utilizados como ponto de partida para o projeto do Static-MultiBP. O Static-MultiBP foi, portanto, projetado como uma camada

adicionada ao MASA-CUDAlign, com mínima modificação do seu código e em suas estruturas de dados.

8.2.1 Cálculo do BP em Múltiplas GPUs

Na execução em múltiplas GPUs do MASA-CUDAlign, as três matrizes de Gotoh são calculadas colocando uma das sequências (S_0) verticalmente e a outra sequência (S_1) horizontalmente. Cada GPU realiza a computação de um subconjunto de colunas das matrizes e, portanto, usa uma subsequência de S_1 de acordo com uma divisão (*split*) definida pelo usuário e toda a sequência S_0. Como no MASA-CUDAlign, a GPU_i somente necessita receber a coluna de borda da esquerda da GPU_{i-1} e enviar a coluna da direita para a GPU_{i+1}, sem necessidade de visão global das matrizes. A primeira coluna de cada GPU no MASA-CUDAlign possui o índice 1.

Contudo, para que se possa avaliar se um bloco pode ser descartado em um ambiente multi-GPU, a matriz inteira deve ser considerada (Seção 2.3.3). Portanto, o índice da primeira coluna de cada GPU foi ajustado considerando a matriz inteira. Desta forma, as coordenadas $(i0, j0)$ de cada primeira célula em cada bloco (bx, by) foram recalculadas, considerando o número de GPUs e a posição relativa de cada uma delas em uma organização linear (Figura 5.6), para que se obtenham as coordenadas corretas de cada célula em relação a toda a matriz.

Adicionalmente, também foi necessário modificar o procedimento que verifica se o bloco pode ser descartado, uma vez que ela agora deve considerar a distância até o final da sequência (isto é, até a última coluna da última GPU), e não apenas a subsequência computada.

8.2.2 Compartilhamento Periódico dos Melhores Escores Locais

O modelo assíncrono de comunicação do MASA-CUDAlign entre GPUs vizinhas (Figura 5.6) foi mantido no Static-MultiBP, para a troca de células das colunas de borda entre as GPUs. Neste modelo, existem três *threads* em cada máquina (CPU): duas *threads* de comunicação - *input* (I) e *output* (O) -, que recebem e enviam dados, respectivamente, e uma *thread* gerenciadora (*manager*) que controla a execução e invoca *kernels* em GPU (Figura 5.5).

O BP usa o melhor escore corrente para calcular se um bloco pode ou não ser descartado (Seção 2.3.3). Quando executado em múltiplas GPUs, cada GPU calcula o seu melhor escore corrente, que pode ser bem menor do que o melhor escore corrente de outra GPU. Sendo assim, foi proposto no Static-MultiBP que o melhor escore corrente de cada GPU seja enviado periodicamente para outras GPUs. O compartilhamento periódico dos

melhores escores locais visa, portanto, aumentar a porcentagem de BP, tentando aproximar a taxa de BP com múltiplas GPUs da taxa de BP com uma única GPU, criando uma visão próxima da global quanto ao valor do melhor escore no momento.

Para permitir a troca dos escores locais entre GPUs sem impactar a computação das três matrizes DP, optou-se por projetar um novo conjunto de *threads* assíncronas chamadas T_S (*Threads* para troca do eScore). As *threads* T_S são executadas de maneira independente das *threads* originais do MASA-CUDAlign, de maneira assíncrona, conforme ilustrado na Figura 8.2. Nessa figura, as *threads* T_M (*thread manager*) e T_C (*thread* de comunicação) são as *threads* originais do MASA-CUDAlign. As *threads* T_S são numeradas de 1 a P, onde P é o número de GPUs, e conectam-se em uma topologia de anel, onde cada *thread* i se comunica com a *thread* $((i + 1)$ `mod` $P)$. Periodicamente, cada *thread* envia o valor do melhor escore obtido até o momento para a *thread* destino. Ao receber o escore, cada *thread* o compara com o seu maior escore local. Se o valor recebido for maior do que o maior escore local, o maior escore local é atualizado, e será utilizado nas próximas operações do BP.

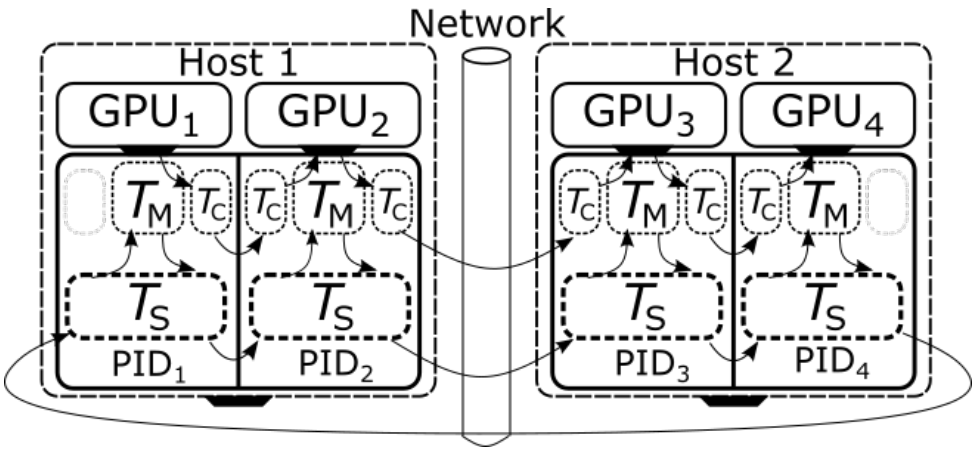


Figura 8.2: *Threads manager*, de comunicação e troca de escore do Static-MultiBP.

8.3 Ilustração do Static-MultiBP

A abordagem utilizada no Static-MultiBP com 2 GPUs é ilustrada na Figura 8.3. Cada GPU irá computar um subconjunto de colunas da matriz DP e receberá a sequência S_0 e uma parte da sequência S_1 ($x + 1$ colunas para GPU_0 e $n - x$ colunas para GPU_1). Considerando a divisão de colunas proposta na figura, as primeiras três diagonais externas são processadas inteiramente pela GPU_0, sem oportunidades de descarte de blocos. Em algum ponto (quarta diagonal da Figura 8.3), o valor do melhor escore local corrente ($CLBS_0$) é alto o suficiente para aplicar o BP ao primeiro bloco de células na GPU_0.

Quando a última coluna da quarta diagonal é processada, a GPU_0 envia células da coluna de borda para a GPU_1 usando a as *threads* T_C do MASA-CUDAlign. Então, cada GPU_i processa sua parte da próxima diagonal, atualizando seu próprio $CLBS_i$. Periodicamente, a GPU_i envia seu valor de $CLBS_i$ para a GPU_j, onde, neste exemplo, $j = (i + 1)$ `mod` 2, usando as *threads* T_S (Seção 8.2.2).

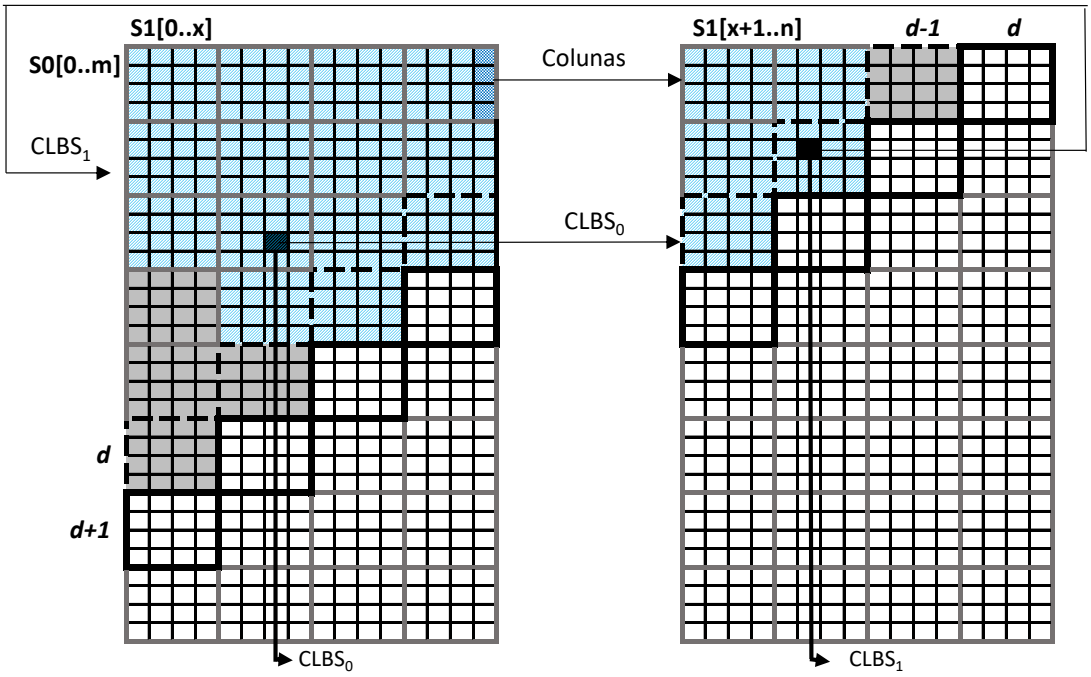


Figura 8.3: Representação do Static-MultiBP em 2 GPUs. Blocos com o padrão de fundo em azul claro foram processados, blocos em cinza foram descartados e blocos em branco ainda não foram avaliados. As setas representam a comunicação: transferência de dados de colunas entre GPU_0 para GPU_1, e a troca dos valores dos melhores escores locais correntes ($CLBS_i$) entre GPUs.

8.4 Algoritmo

O algoritmo do Static-MultiBP é mostrado no Algoritmo 1.

O procedimento (*procedure*) `shareScore` é executado na *thread* T_S, que periodicamente envia o melhor escore local corrente para a próxima GPU (linha 3) e recebe a mesma informação da GPU anterior (linha 4) usando uma topologia circular, em um *loop* infinito (linhas 2 a 9). Se o melhor escore recebido da GPU anterior ($leftBest$) é maior que o atual (linha 5), o melhor escore local ($grid.bestGlobalScore$) é atualizado. A instrução `sleep` na linha 8 força uma tempo de espera definido na variável $waitTime$.

O procedimento `updatePruningWindow` é executado pela *thread manager* (T_M) do MASA-CUDALign (Figura 8.2) e foi modificado pelo Static-MultiBP. É responsável pela

Algoritmo 1 *Block pruning* em múltiplas GPUs.

```
 1: procedure shareScore(job, grid, leftBest, waitTime)
 2:     while (TRUE) do
 3:         job.dispatchBestScore(grid.bestGlobalScore)
 4:         leftBest = job.receiveBestScore()
 5:         if (leftBest > grid.bestGlobalScore) then
 6:             grid.bestGlobalScore = leftBest
 7:         end if
 8:         sleep(waitTime)
 9:     end while
10: end procedure

11: procedure updatePruningWindow(diagonal, score, struct grid, MultipleGPU, matchValue)
12:     while (grid.windowStart < grid.gridWidth) do
13:         bx = grid.windowStart ; by = diagonal - bx
14:         if MultipleGPU == TRUE then
15:             if ! (isBlockPrunableMulti(bx, by, score, grid, matchValue)) then
16:                 break
17:             end if
18:         end if
19:         grid.windowStart = grid.windowStart + 1
20:     end while
21:     while (grid.windowEnd < grid.windowStart do
22:         bx = grid.windowEnd - 1 ; by = diagonal - bx
23:         if (MultipleGPU) == TRUE then
24:             if ! (isBlockPrunableMulti(bx, by, score, grid, matchValue)) then
25:                 break
26:             end if
27:         end if
28:         grid.windowEnd = grid.windowEnd - 1
29:     end while
30: end procedure

31: procedure isBlockPrunableMulti(bx, by, score, struct grid, matchValue)
32:     i0 = getBlockHIndexMulti(bx)
33:     j0 = getBlockVIndex(by)
34:     if (i0 == -1) then
35:         return (TRUE)
36:     end if
37:     grid.maxj = grid.seq1Size
38:     distI = grid.maxi - i0 ; distJ = grid.maxj - j0
39:     distMin = min(distI,distJ) ; inc = distMin * matchValue
40:     maxScore = score + inc
41:     if (maxScore < grid.bestGlobalScore) then
42:         maxScore = grid.bestGlobalScore
43:     else
44:         grid.bestGlobalScore = maxScore
45:     end if
46:     if (maxScore ≤ grid.bestScore) then
47:         return (TRUE)
48:     else
49:         return (FALSE)
50:     end if
51: end procedure
```

avaliação de cada diagonal externa e por ajustar a janela de blocos não descartados (os quais serão efetivamente processados). Ele recebe como parâmetros (linha 11) a diagonal externa atual (*diagonal*), o escore da primeira célula do bloco (*score*), um valor lógico que informa se a computação é dividida em múltiplas GPUs ($MultipleGPU$), uma estrutura com a informação sobre o *grid* de blocos (*grid*) e o valor usado nos casos de *match* ($matchValue$). Cada diagonal contém um conjunto de blocos, e a computação inicia no índice $grid.windowStart$ e encerra em $grid.windowEnd$, e estas varáveis são inicializadas

com 0 e o valor do tamanho do *grid* ($gridWidth$), respectivamente. O primeiro *loop* (linhas 12 a 20) verifica se os blocos iniciais da diagonal podem ser descartados, usando a função `isBlockPrunableMulti` se a execução é realizada em múltiplas GPUs (linha 14). O valor de $grid.windowStart$ é incrementado cada vez que um bloco descartável é encontrado (linha 19). De forma similar, o segundo *loop* (linhas 21 a 29) avalia a diagonal no sentido oposto a partir do final, atualizando a variável $grid.windowEnd$ à medida que os blocos são identificados como descartáveis. Desta forma, ao final, apenas a janela de blocos não descartados (entre $grid.windowStart$ e $grid.windowEnd$) será processada pela GPU_i, e todos os blocos restantes na diagonal serão descartados.

Na função `isBlockPrunableMulti` (linhas 31 a 51), cada bloco é avaliado pela *thread* principal para que seja verificado se deve ser descartado ou não. Para tanto, o índice do bloco (bx,by) deve ser analisado para a obtenção do índice da primeira célula do bloco, porque este valor será usado para comparar o escore do bloco ($score$) ao escore máximo local corrente ($grid.bestScore$). A função `getBlockHIndexMulti` foi criada para retornar o índice horizontal desta célula ($i0$), baseando-se no índice horizontal do bloco (linha 32) que foi recalculado. Esta função usa um vetor criado durante a inicialização que mantém os índices para todos os blocos de todas as GPUs. Então, a função retornará o índice considerando a sequência S_1 inteiramente, e não apenas sua subsequência. A função que retorna o índice vertical ($j0$) não foi modificada em relação à versão do CUDAlign 4.0, porque a sequência S_0 não é dividida durante o processamento (Figura 8.3). Adicionalmente, a função deve considerar a o tamanho total da sequência S_1 e não apenas o tamanho da subsequência processada pela GPU para avaliar a possibilidade de BP, então este valor é atualizado na linha 37. Nas linhas 38 a 40, o valor mínimo entre as distâncias horizontal e vertical é calculado ($distMin$), e o máximo escore possível que pode ser obtido a partir deste ponto ($maxScore$) é calculado. Na linha 41, um teste condicional compara $maxScore$ e o melhor escore global ($bestGlobalScore$), atualizando uma ou outra variável se for necessário. Finalmente, um teste é realizado na linha 46 para verificar se $maxScore$ é menor que $bestScore$, que é a condição que permite que o bloco seja descartado, retornando $TRUE$ ou $FALSE$ de acordo com o resultado.

8.5 Resultados Experimentais

8.5.1 Ambiente de Testes

Para a avaliação do desempenho do Static-MultiBP, dois ambientes uniformes foram utilizados: (a) um ambiente heterogêneo (Laico), com dois *hosts* interconectados por uma rede Gigabit Ethernet, com GPUs NVidia GTX 680 (arquitetura Kepler) e NVidia GTX

980 Ti (arquitetura Maxwell); e (b) um ambiente homogêneo (Comet), com um *host* contendo 4 GPUs NVidia Tesla P100 (arquitetura Pascal), como visto na Tabela 8.1.

Tabela 8.1: Ambientes usados nos experimentos Static-MultiBP.

Amb.	**GPUs**	**Modelo**	**Núcleos**	***Clock* (GHz)**	**Memória (GB)**
Laico	2	GTX 680	1.536	1,006	1,5
		GTX 980 Ti	2.816	1,000	6,0
Comet	4	Tesla P100	3.840	1,190	12,0

8.5.2 Sequências Comparadas

Onze casos de comparação (Tabela 8.2) foram selecionados para os testes, com sequências de DNA reais com tamanhos variando de 1 MBP e 64 MBP obtidos da base de dados NCBI, com diferentes graus de similaridade, sendo identificados nos experimentos pelo seu identificador ("Id"). Os últimos quatro casos se referem a comparações de cromossomos homólogos 19, 20, 21 e 22 entre o homem e o chimpanzé.

Tabela 8.2: Sequências usadas nos experimentos Static-MultiBP.

Id	**Sequência 1**		**Sequência 2**		**Escore Ótimo**
	Acesso	***Tam.***	***Acesso***	***Tam.***	
1M	CP000051.1	1M	AE002160.2	1M	88.353
3M	BA000035.2	3M	BX927147.1	3M	4.226
5M	AE016879.1	5M	AE017225.1	5M	5.220.960
7M	NC_005027.1	7M	NC_003997.3	5M	172
10M	NC_017186.1	10M	NC_014318.1	10M	10.235.188
23M	NT_033779.4	23M	NT_037436.3	25M	9.063
47M	NC_000021.7	47M	BA000046.3	32M	27.206.434
Ch19	NC_000019.9	59M	NC_006486.3	63M	17.297.608
Ch20	NC_000020.10	63M	NC_006487	61M	40.050.427
Ch21	NC_000021.8	48M	NC_006488.2	46M	36.006.054
Ch22	NC_000022.10	51M	NC_006489.3	49M	31.510.791

Apesar de o número de células processadas ser reduzido pelo descarte de blocos do Static-MultiBP, os resultados de desempenho nas seções a seguir são expressos em GCUPS efetivos (Seção 2.5), métrica que leva em consideração todas as células da matriz DP. Os testes buscam sempre comparar a versão original do CUDAlign 4.0 (sem BP, identificado nos gráficos e tabelas como "No-BP") e a solução Static-MultiBP (identificado como "BP"). Em todos os experimentos com Static-MultiBP em múltiplas GPUs, o tempo de espera na *thread* que realiza o recebimento/envio de melhor escore local (variável $waitTime$, utilizada na linha 8 do Algoritmo 8.4) foi definido como 5 segundos. Este valor

foi definido empiricamente após testes com algumas sequências similares, permitindo uma taxa de descarte de blocos próxima à obtida na execução em 1 GPU sem que sobrecarregar o processamento.

8.5.3 Desempenho do Static-MultiBP em 1 GPU

Inicialmente, o Static-MultiBP foi executado em uma única GPU nos ambientes Laico e Comet separadamente, com dois objetivos principais: (a) confirmar os ganhos do procedimento de BP nestas GPUs; e (b) usar os resultados como base na avaliação dos testes com múltiplas GPUs. Desta forma, sequências com tamanhos variando entre 1 MBP e 47 MBP (Tabela 8.2) foram comparadas com e sem o módulo de BP habilitado. As sequências de cromossomos (Ch19 a Ch22) não foram testadas em 1 GPU pois os experimentos levariam várias horas. Os resultados obtidos (expressos em GCUPS nas colunas "No-BP" e "BP", e de forma percentual na coluna "Ganho") podem ser vistos na Tabela 8.3.

Tabela 8.3: Execuções No-BP versus BP em 1 GPU.

Id	**GTX 680**			**GTX 980 Ti**			**P100**		
	No-BP (GCUPS)	***BP (GCUPS)***	***Ganho (%)***	***No-BP (GCUPS)***	***BP (GCUPS)***	***Ganho (%)***	***No-BP (GCUPS)***	***BP (GCUPS)***	***Ganho (%)***
1M	45,1	54,9	21,7	92,1	106,2	15,3	137,4	159,1	15,8
3M	46,6	51,8	11,1	94,8	95,7	1,0	152,5	153,5	0,7
5M	45,9	102,9	124,2	93,1	192,1	106,2	155,9	288,5	85,1
7M	46,8	52,1	11,4	93,2	95,2	2,3	155,8	156,0	0,1
10M	46,9	103,3	120,5	92,7	192,4	107,6	157,3	**291,6**	85,4
23M	46,3	52,1	12,5	92,8	95,5	2,9	159,0	159,3	0,2
47M	46,8	77,2	65,0	92,8	143,6	54,7	159,1	224,7	41,2

Como pode ser observado, a técnica de BP provê um relevante ganho de desempenho quando duas sequências similares são comparadas em 1 GPU, em particular nos casos 5M, 10M e 47M. Pode-se notar também que, para estes casos, os maiores ganhos foram obtidos pela GPU GTX 680 (124,2%, 120,5% e 65,0%, respectivamente), enquanto a GPU P100 atingiu ganhos menores (85,1%, 85,4% e 41,2%), mas ainda muito bons. Mesmo nos casos envolvendo sequências muito diferentes (3M, 7M e 23M, com valor do escore ótimo pequeno na Tabela 8.2), não houve perda de desempenho quando o módulo BP foi habilitado. O melhor desempenho neste cenário foi obtido no caso de teste 10M na GPU P100, com BP: 291,6 GCUPS.

8.5.4 Desempenho do Static-MultiBP em 2 GPUs (Heterogêneo)

O ambiente heterogêneo do Laico (Tabela 8.1) foi usado para o segundo experimento. O objetivo era avaliar o Static-MultiBP em um cenário com dois modelos diferentes de GPUs. Novamente, os testes foram realizados comparando a execução com e sem BP,

e o ganho de desempenho entre as duas execuções foi avaliado. As execuções sem BP utilizaram o código original de CUDAlign 4.0, mas restrito apenas à execução do estágio 1, que equivale à comparação realizada pelo Static-MultiBP.

Quando 2 GPUs são usadas, a distribuição de trabalho entre elas é feita estaticamente através do parâmetro `-split=`x`,`y: pesos são atribuídos às GPUs, de forma que a primeira GPU processará as primeiras $(x/(x+y))\%$ de colunas da matriz DP, enquanto a segunda GPU processará as $(y/(x+y))\%$ colunas restantes. A mesma abordagem é usada quando existem mais GPUs: uma divisão proporcional baseada nos pesos é calculada para definir que colunas serão processadas por cada GPU.

Inicialmente, uma distribuição equitativa (1:1) foi usada nos testes, mas os ganhos foram inferiores aos esperados (máximo de 45%). Isto ocorre porque as 2 GPUs usadas neste ambiente (GTX 680 e GTX 980 Ti) são consideravelmente diferentes em termos de capacidade computacional (Tabela 8.1), e a GPU GTX 980 Ti processa suas colunas muito mais rapidamente, resultando em um cenário desbalanceado, onde a segunda GPU perde tempo esperando por dados das últimas colunas processadas pela primeira GPU, como identificado nos testes realizados na Seção 7.3. Algumas diferentes configurações de *split* foram testadas empiricamente para definir uma distribuição adequada (técnica de *profiling* - Seção 4.3). Para tanto, a diferença entre os GFLOPs informados para cada GPU foi usado como referência, e o percentual de colunas processadas pela GPU GTX 980 TI foi incrementado/decrementado até obter-se o menor tempo de execução para uma comparação de referência (1M). Ao final, a divisão 10:25 proporcionou os melhores resultados, e esse *split* foi usado em todos os testes subsequentes.

A Tabela 8.4 e a Figura 8.4 mostram as comparações entre as execuções No-BP e BP no ambiente Laico (resultados em GCUPS). O ganho de desempenho foi calculado dividindo-se as execuções com BP pela execução sem BP e subtraindo 1.

Verifica-se que o Static-MultiBP teve desempenho superior à execução No-BP em todas as sequências similares (1M, 5M, 10M, 47M, Ch19, Ch20, Ch21 e Ch22). No caso de teste Ch21, por exemplo, o ganho de tempo de execução gerado pelo Static-MultiBP foi de 73,1%. Na comparação 47M, o ganho de 53,3% nos GCUPS reflete em uma tempo de execução 1 hora mais rápido (3h12min sem BP e 2h09min com BP), atingindo 224,4 GCUPS. Ganhos menos significativos (ou mesmo pequenas perdas) para os casos 3M, 7M e 23M eram esperados, uma vez que as sequências são muito dissimilares.

Cabe ressaltar que os ganhos obtidos pelo Static-MultiBP foram validados utilizando a comparação pareada (Seção 6.1). Para tanto, os diferenças entre os desempenhos das execuções com e sem BP (coluna "Dif." na Tabela 8.4) foi considerada, em um total de 11 experimentos, e um nível de confiança de 90% foi adotado. Os parâmetros da Equação 6.1, neste caso, são: $\widehat{y} = 55,927$, $n = 11$, $\alpha = 0,95$, $t_{[0,95;10]} = 2,228$ e $s_y = 44,335$. O

Tabela 8.4: Execuções No-BP versus BP em ambiente heterogêneo (GTX 680 + GTX 980 Ti).

Id	GTX 680 + GTX 980 Ti			
	No-BP (GCUPS)	***BP (GCUPS)***	***Dif. (BP - No-BP)***	***Ganho (%)***
1M	107,0	117,2	10,2	9,5
3M	122,2	121,9	-0,3	-0,2
5M	124,1	224,6	100,5	80,9
7M	125,5	125,4	-0,1	-0,1
10M	127,8	**231,2**	103,4	80,9
23M	128,9	128,9	0,0	0,0
47M	129,5	198,5	69,0	53,3
Ch19	129,9	189,4	59,5	45,8
Ch20	130,5	219,9	89,4	68,6
Ch21	129,6	224,4	94,8	73,1
Ch22	130,0	218,8	88,8	68,4

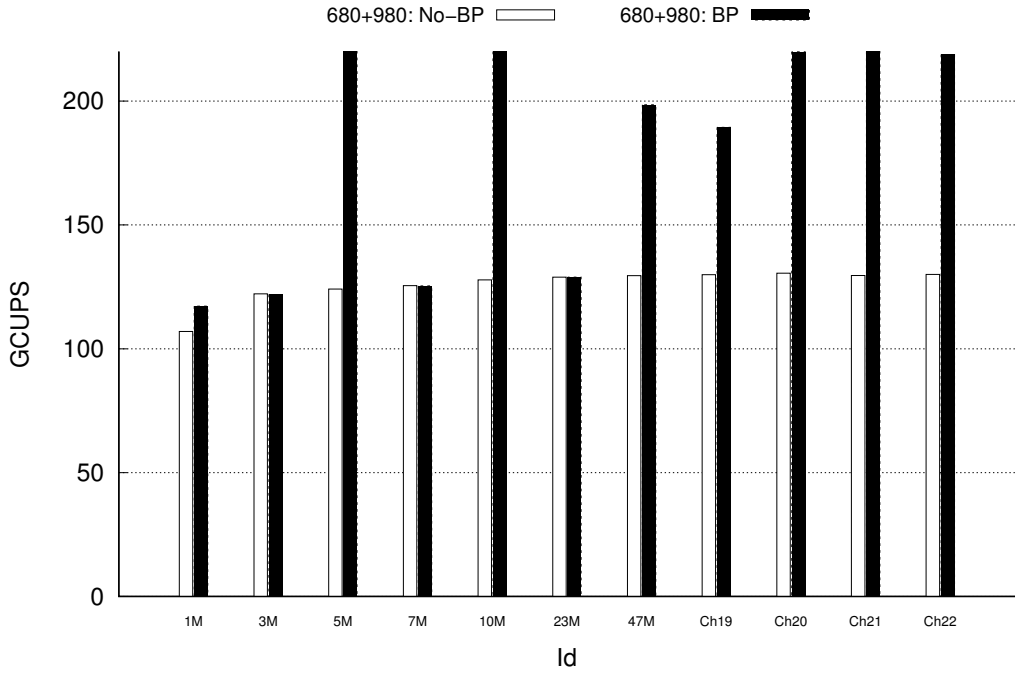


Figura 8.4: Resultados Static-MultiBP em ambiente heterogêneo Laico (GTX 680 + GTX 980 Ti).

cálculo resultou em um CI de (26,145 ; 85,710), o que, portanto, confirma estatisticamente os ganhos, já que o intervalo não inclui o zero (Seção 6.1). O melhor desempenho neste cenário foi obtido no caso de teste 10M com BP: 231,2 GCUPS.

8.5.5 Desempenho do Static-MultiBP em 4 GPUs (Homogêneo)

Neste cenário de testes, o Static-MultiBP foi executado no ambiente Comet (4 GPUs idênticas - Tabela 8.1). Todas as sequências listadas na Tabela 8.2 foram comparadas usando 2 e 4 GPUs P100, com uma distribuição igualitária entre elas (1:1 e 1:1:1:1, respectivamente). A Tabela 8.5 e a Figura 8.5 mostram os resultados obtidos.

Tabela 8.5: Execuções No-BP versus BP em ambiente heterogêneo (2 e 4 P100).

Id	**2 P100**			**4 P100**		
	No-BP (GCUPS)	***BP (GCUPS)***	***Ganho (%)***	***No-BP (GCUPS)***	***BP (GCUPS)***	***Ganho (%)***
1M	173,3	216,5	24,9	183,8	249,3	35,6
3M	257,2	271,2	5,4	383,6	441,1	15,0
5M	279,7	379,9	35,8	460,9	573,9	24,5
7M	288,2	290,2	0,7	494,3	524,9	6,2
10M	298,4	395,4	32,5	536,9	630,6	17,4
23M	309,1	305,1	-1,3	589,7	592,9	0,6
47M	314,9	370,8	17,8	612,6	656,7	7,2
Ch19	315,5	362,3	14,8	619,1	**694,8**	12,2
Ch20	315,8	401,6	27,2	620,0	683,8	10,3
Ch21	314,7	402,8	28,0	613,4	680,8	11,0
Ch22	314,2	400,3	27,4	614,4	680,3	10,7

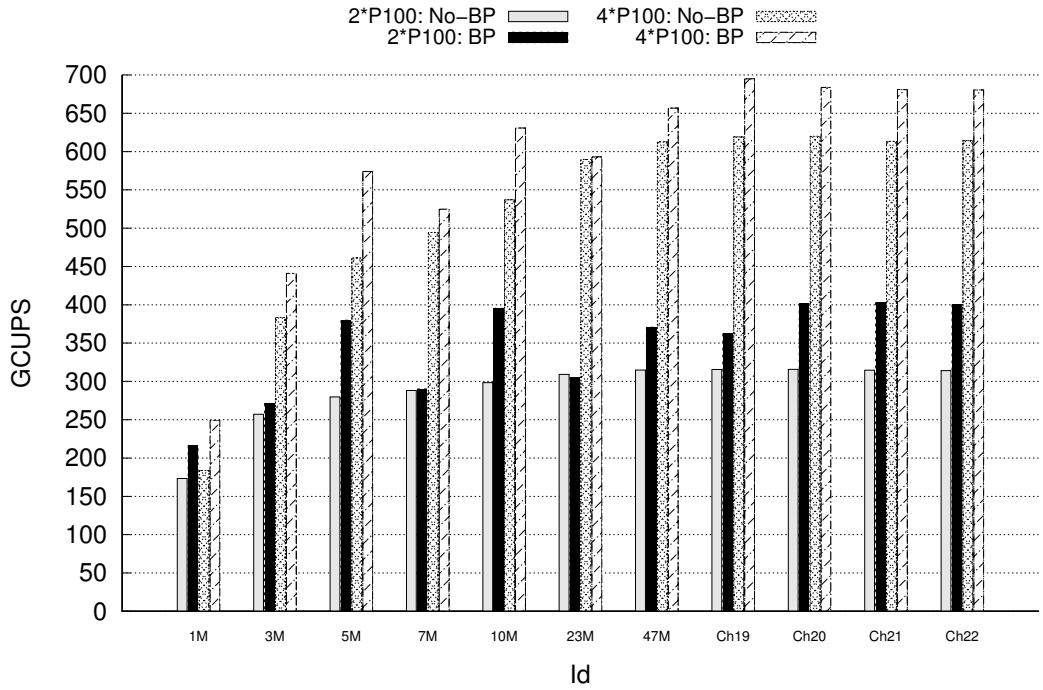


Figura 8.5: Resultados Static-MultiBP em ambiente homogêneo Laico (2 e 4 P100).

Como pode ser observado, os ganhos para sequências com maior oportunidade de descarte de blocos (5M, 10M, 47M, Ch19, Ch20, Ch21 e Ch22) é bem relevante para 2 GPUs, e ainda muito bons para 4 GPUs. Por exemplo, a comparação Ch20 (63 MBP × 61 MBP) levou 3h25min sem BP com 2 P100 e 2h41min com BP, um ganho de 27,2%. Com 4 GPUs, a mesma comparação levou 1h44min na execução do CUDAlign e 1h34min na execução do Static-MultiBP, com um ganho de 10,3%. Estes resultados confirmam os benefícios da implantação da técnica de BP em múltiplas GPUs quando as sequências são similares.

Da mesma forma que na Seção 8.5.4, a Equação 6.1 foi aplicada para realizar uma comparação pareada, visando confirmar os ganhos a um nível de confiança de 90%. No caso da execução em 4 P100, por exemplo, o CI obtido foi (42,112 ; 81,597), validando o intervalo de acordo com o teste t. O melhor desempenho obtido neste ambiente foi de 694,8 GCUPS (Ch19 em 4 GPUs P100, com BP), o que é mais de 40% do desempenho obtido pelo CUDAlign 3.0 (Seção 5.2.1.6), mas usando bem menos GPUs (apenas 4 nos testes com Static-MultiBP, contra 64 usadas pelo CUDAlign 3.0).

Considerando os testes em 2 GPUs, pode ser observado que algumas sequências dissimilares (3M e 7M) apresentaram resultados levemente superiores executando no Static-MultiBP, e em um caso (23M), uma pequena perda de desempenho (1,3%) ocorreu. Isto pode acontecer eventualmente, uma vez que a quantidade de blocos descartados é muito pequena. Basicamente, o que está sendo medido neste caso é o *overhead* adicionado pelo BP, o que pode ser considerado bem pequeno. No cenário com 4 GPUs, por outro lado, o desempenho é melhorado em todos os casos no Static-MultiBP.

Adicionalmente, pode-se ver também que, apesar de relevantes, os ganhos em 4 GPUs são relativamente menores se comparados com o ambiente com 2 GPUs. Este resultado pode ser explicado pela distribuição estática e equitativa utilizada nos testes. Devido ao comportamento do BP em múltiplas GPUs, uma GPU pode processar suas colunas mais rapidamente que outras, causando desbalanceamento.

A Tabela 8.6 mostra o *speedup* obtido pela execução em 2 e 4 GPUs com o Static-MultiBP quando comparado com a execução em 1 GPU sem o BP para as sequências mais longas (o *speedup* linear ideal foi incluído na tabela para melhor avaliação). Como pode ser observado, o *speedup* variou entre 1,60x e 1,92x com 2 P100 (máximo teórico de 2x), e 2,70x a 3,72x com 4 P100 (máximo teórico de 4x), o que pode ser considerado adequado para este tipo de solução.

Para avaliar como a distribuição estática interfere no enchimento dos *buffers* de entrada e saída utilizados para realizar a comunicação de dados entre as GPUs (Seção 5.2.1.6), os *logs* de execução foram analisados na comparação 47M, que apresenta uma alta similaridade, tanto em 2 quanto em 4 P100. Na Figura 8.6, pode-se ver a variação do uso

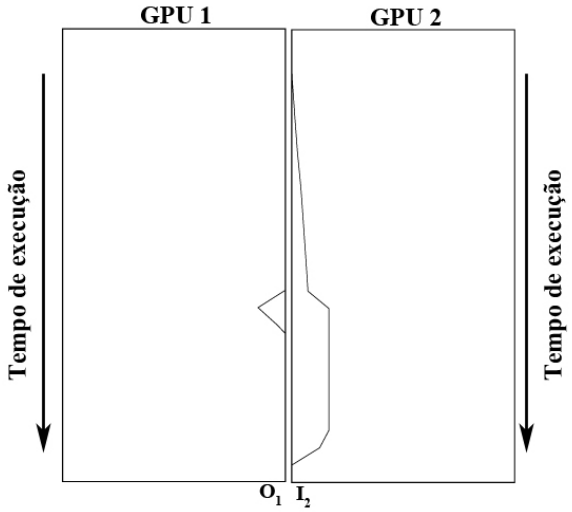
Tabela 8.6: Tempo de execução (em horas) e *speedup* do Static-MultiBP em GPUs P100.

Id	**1 P100**	**2 P100**	**4 P100**
Linear	1,00x	2,00x	4,00x
23M	1,2h	0,5h / 1,92x	0,3h / 3,72x
47M	2,2h	1,2h / 1,65x	0,7h / 2,92x
Ch19	5,5h	2,9h / 1,75x	1,5h / 3,35x
Ch20	4,9h	2,7h / 1,68x	1,6h / 2,86x
Ch21	2,8h	1,5h / 1,60x	0,9h / 2,70x
Ch22	3,2h	1,8h / 1,66x	1,0h / 2,83x

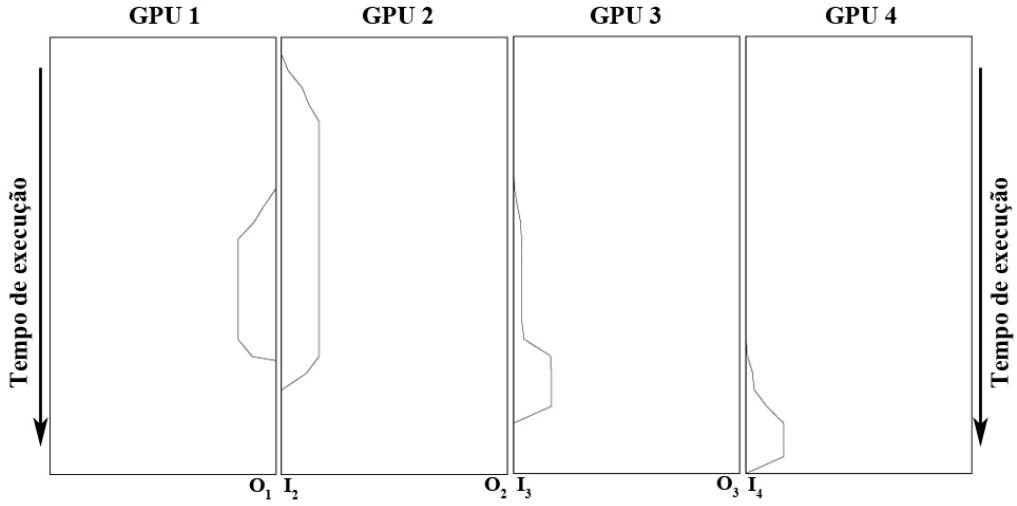
dos *buffers* de entrada (I_i) e saída (O_i) ao longo do tempo na execução. O enchimento dos *buffers* está representado pelas linhas dentro dos retângulos que representam cada GPU, e podem atingir um limite quando ele é totalmente preenchido (máximo de 131.072 bytes). Identificou-se que neste ambiente homogêneo o enchimento dos *buffers* ocorre devido à diferença entre a quantidade de blocos efetivamente processados pelas GPUs, visto que parte dos blocos estão sendo descartados pela rotina de BP. No processamento em 2 GPUs (Figura 8.6a), ocorre uma retenção pequena no *buffer* de saída da GPU 1, mas bem mais significativa no *buffer* de entrada da GPU 2. Verificando os dados de *pruning*, observamos que enquanto na GPU 1 houve uma taxa de BP de 55,8%, esta quantidade foi de apenas 33,6% na GPU 2. Por ter que processar mais blocos, acaba havendo uma retenção no *buffer* de entrada na GPU 2, o que pode prejudicar o desempenho. Já na execução em 4 GPUs (Figura 8.6b), verificou-se que as taxas de BP nas GPUs de 1 a 4 foram, respectivamente, 38,9%, 52,7%, 34,7% e 18,5%. Observando a figura, pode-se notar o efeito deste desbalanceamento nos *buffers*. A retenção no *buffer* de saída da GPU 1 é propagada para o *buffer* de entrada da GPU 2, mas não para a sua saída, visto que ela processa menos blocos devido ao BP. Já nas GPUs 3 e 4, ocorre o enchimento dos *buffers* de entrada devido ao volume maior de dados processados (menor BP), mas durante uma faixa de tempo menor. Para tratar este cenário de desbalanceamento causado pelo BP em uma distribuição de carga de trabalho estática, uma forma de distribuição dinâmica de carga de trabalho deve ser proposta, o que será o objeto do Capítulo 9.

8.6 Conclusão do Capítulo

O Static-MultiBP, solução apresentada neste capítulo, possibilitou ganhos expressivos de desempenho em ambientes de GPUs de porte médio, através de uma abordagem leve que permite o descarte de blocos em múltiplos dispositivos, sejam eles homogêneos ou heterogêneos. Diferentemente da solução SW# (Seção 5.1.6), o Static-MultiBP não está



(a) Enchimento de *buffers* em 2 P100.



(b) Enchimento de *buffers* em 4 P100.

Figura 8.6: Enchimento de *buffers* na execução do Static-MultiBP na comparação 47M.

restrito à execução em apenas 2 GPUs, sendo, até onde se tem conhecimento, a primeira solução que implementa técnicas de poda em 3 ou mais GPUs.

O Static-MultiBP foi testado em dois ambientes diferentes - um homogêneo e um heterogêneo. Ganhos de até 80,9% foram obtidos devido à implantação do algoritmo de descarte de blocos, sendo mais relevantes em sequências muito similares. O melhor desempenho obtido foi de 694,8 GCUPS quando comparando duas sequências com 59 MBP e 64 MBP em 4 GPUs P100, com BP. A avaliação do *speedup* com 4 GPUs também mostrou bons resultados, sendo viável para ser testada em ambientes com mais placas gráficas, sejam elas do mesmo modelo ou de modelos diferentes.

O Static-MultiBP, contudo, ainda se baseia em uma distribuição estática de carga de trabalho entre GPUs. Esta abordagem pode gerar contenções nos *buffers* de entrada e saída que podem afetar o desempenho final, uma vez que o volume de dados varia durante a execução, principalmente devido às mudanças nas quantidades de blocos processados em cada GPU causadas pela técnica de descarte de dados. Desta forma, o foco do Capítulo 9 será propor uma solução que provê uma distribuição dinâmica de carga de trabalho, visando melhorar o desempenho principalmente em ambientes heterogêneos.

A proposta e os resultados apresentados neste capítulo foram publicados em [28].

Capítulo 9

Framework para Comparação de Sequências em Múltiplas GPUs com Descarte de Blocos e Estratégias de Distribuição de Carga

Quando as sequências comparadas possuem alta similaridade, o descarte de blocos (BP) pode levar a um alto desbalanceamento quando múltiplas GPUs são utilizadas. Isto corre porque algumas partes da matriz não serão computadas, como visto na Figura 2.15b. Isto pode gerar desbalanceamento em toda a computação, com efeito negativo no desempenho. Para mitigar este efeito, técnicas de distribuição dinâmica de carga de trabalho (Seção 4.3) devem ser empregadas.

Para tratar o problema da distribuição irregular de trabalho em ambientes heterogêneos ou causado pela introdução de descarte de blocos em comparação de sequências com alta similaridade, uma nova solução foi proposta incorporando o Static-MultiBP (Capítulo 8) em uma arquitetura de *software* (*framework*) de comparação de sequências chamado MultiBP, adicionando uma estratégia dinâmica de distribuição de carga de trabalho que visa corrigir problemas de desbalanceamento causados pelo descarte de blocos ou por eventuais distribuições estáticas inadequadas de carga trabalho . Além disso, foi criado um módulo de decisão que avalia a melhor estratégia de distribuição de carga baseando-se em parâmetros das sequências, das GPUs envolvidas e de um algoritmo que estima a similaridade entre as sequências. Para o desenvolvimento da estratégia dinâmica, foi utilizado o conceito de redistribuição de colunas usado em Sandes et al. (Seção 5.3.2), que foi testado em ambiente de simulação, mas com a introdução do conceito de divisão da matriz em ciclos. Diferentemente do modelo de agentes sugerido por Sandes et al., o MultiBP possui uma arquitetura modular com funções de gerenciamento, decisão, monitoramento

e execução.

Este capítulo está dividido em três seções. Na Seção 9.1, será detalhado o projeto do Dynamic-MultiBP e do módulo de decisão que compõem, junto com o Static-MultiBP, o *framework* MultiBP para execução da comparação de sequências longas de DNA em múltiplas GPUs com descarte de blocos. Na Seção 9.2, serão detalhados os experimentos e resultados obtidos, enquanto na Seção 9.3 serão apresentadas as considerações finais do capítulo.

9.1 Projeto do *Framework* MultiBP

O projeto de um *framework* que utiliza o descarte de blocos em plataformas que possuem múltiplos dispositivos tem dois principais desafios. Primeiro, o padrão de descarte de blocos só é conhecido em tempo de execução, e o formato final da matriz efetivamente computada é obtido apenas no final do processamento, o que levou a considerar uma redistribuição de trabalho dinâmica. Segundo, o *overhead* gerado na redistribuição dinâmica não é desprezível, uma vez que o processamento precisa ser totalmente interrompido e adaptado ao novo cenário. Para contornar estes desafios, duas estratégias foram propostas. Na estratégia Static-MultiBP, as colunas da matriz de programação dinâmica são distribuídas estaticamente entre as GPUs e processadas com compartilhamento de escore, como descrito no Capítulo 8. Na estratégia Dynamic-MultiBP, o processamento da matriz de programação dinâmica (DP) é dividido em ciclos baseando-se na divisão de colunas entre os dispositivos, e a carga de trabalho é avaliada entre ciclos consecutivos (chamados *breakpoints*) para ajustar a distribuição de colunas, se necessário. Esta abordagem se assemelha com a discutida na Seção 7.1, mas com uma diferença em relação à Figura 7.1c: na solução proposta no Dynamic-MultiBP, as GPUs do segundo ciclo aguardam até que todas as colunas alocadas no primeiro ciclo tenham sido processadas para iniciarem a sua execução do novo ciclo, pois a quantidade de colunas processadas por cada GPU pode mudar em relação à alocação inicial.

Adicionalmente, um módulo de decisão foi projetado que indica qual estratégia deve ser usada (estática ou dinâmica), baseando-se na estimativa de similaridade entre as sequências, o que pode indicar o percentual de blocos descartados. O módulo projetado é flexível, e a rotina de estimativa da similaridade pode ser fornecida pelo usuário, ou a rotina padrão (*default*) fornecida pelo *framework* pode ser utilizada. Se o valor retornado pela rotina é menor que um limite, o Static-MultiBP é usado; caso contrário, o Dynamic-MultiBP é escolhido.

9.1.1 Visão Geral

As soluções Static-MultiBP e Dynamic-MultiBP usam uma arquitetura comum composta de quatro módulos: (I) *Executor*: módulo onde a matriz DP é processada e o descarte de blocos é avaliado; (II) *Decision-Maker*: módulo que pode ser habilitado pelo usuário para selecionar o modo de execução apropriado para o MultiBP (Static ou Dynamic); (III) *Controller*: módulo centralizado que divide a computação entre os dispositivos; e (IV) *Monitor*: módulo distribuído que faz a intermediação da comunicação entre *Controller* e *Executor*.

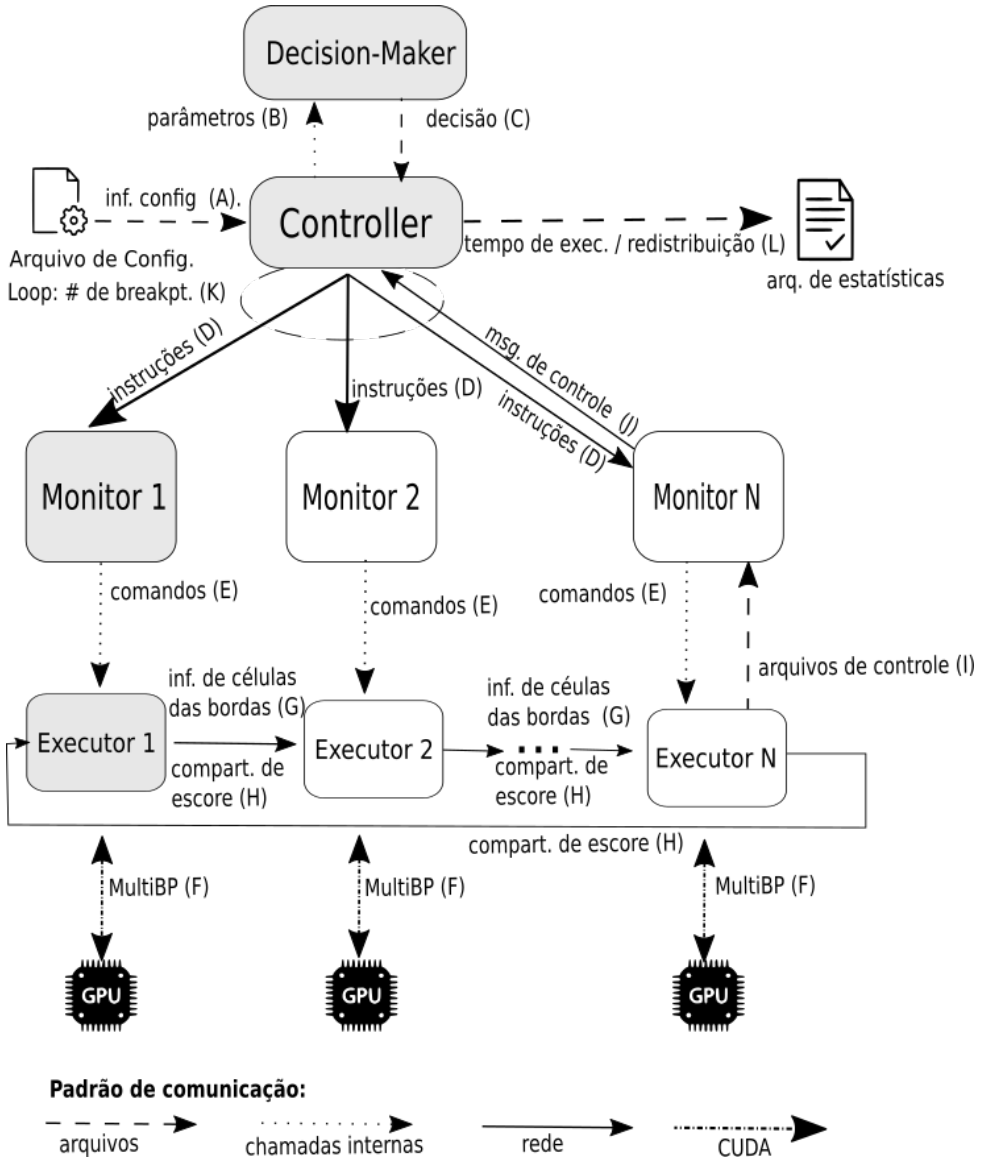


Figura 9.1: Visão geral do projeto das soluções MultiBP.

O funcionamento do *framework* MultiBP é descrito na Figura 9.1. Inicialmente, o *Controller* lê o arquivo de configuração (A). Se o módulo de decisão estiver habilitado,

a informação sobre as sequências e GPUs é enviada para o módulo *Decision-Maker* (B), que executa seu algoritmo e retorna a estratégia apropriada (C). Se o modo Static for selecionado pelo usuário ou retornado pelo *Decision-Maker*, o número de *breakpoints* é atualizado como 0, e o Static-MultiBP é então executado; caso contrário, será executado o Dynamic-MultiBP. O *Controller* constrói então os comandos de execução e envia-os para os módulos *Monitor* (um para cada GPU), indicando a execução do modo estático ou dinâmico (D). Quando um comando de execução é recebido, o *Monitor* chama o módulo *Executor* com a linha de comando recebida (E), que executa a comparação usando a técnica de BP para um subconjunto de colunas assinaladas para a respetiva GPU (F). Durante a execução, o *Executor* troca informações sobre as células da última coluna de sua submatriz (G) e o melhor escore local (H), usando uma abordagem em anel dentro de cada ciclo para a comunicação do escore, como descrito na Seção 8.2.1. O último *Monitor* em cada ciclo verifica a existência de arquivos de controle, que indicarão se os arquivos de *log* dos *buffers* de entrada e saída devem ser lidos ou se a execução foi finalizada (I). Caso o ponto de leitura dos *logs* seja atingido, o *Monitor* envia uma mensagem de controle para o *Controller* (J). O módulo *Controller* verifica o número de *brekpoints* informado pelo usuário (K) e repete as etapas (D) a (J) para cada ciclo, até que seja atingida a última iteração. Ao final do último ciclo, o *Controller* escreve os resultados em um arquivo de saída (L).

Na Figura 9.1, os módulos em caixas cinza (*Controller*, *Decision-Maker*, *Monitor* 1 e *Executor* 1) são armazenados por padrão no *host* contendo a primeira GPU do ciclo. Contudo, os módulos *Controller* e *Decision-Maker* podem ser colocados em um *host* independente, se assim desejado, requerendo apenas um sistema de arquivos compartilhado. O projeto de cada um dos módulos será detalhado nas Seções 9.1.2 a 9.1.5.

9.1.2 Projeto do Módulo *Executor*

9.1.2.1 Static-MultiBP

O projeto do Static-MultiBP já foi detalhado no Capítulo 8. Nesta estratégia, a carga de trabalho é definida estaticamente e, periodicamente, GPUs recebem o melhor escore da GPU anterior e enviam seu melhor escore local para a próxima GPU, de acordo com a organização em anel definida. A principal ideia do compartilhamento de escore é simular uma visão global do melhor escore para aumentar a quantidade de blocos descartados em cada GPU.

No Static-MultiBP, as GPUs computam todas as linhas e um subconjunto de colunas da matriz DP. O número de colunas processadas por cada GPU é calculado baseando-se em pesos informados pelo usuário em um parâmetro de execução. Se, por exemplo,

existem 2 GPUs heterogêneas, pode-se definir um *split* 2:1, alocando 2/3 de colunas da matriz DP para a GPU_1 e 1/3 para a GPU_2.

9.1.2.2 Dynamic-MultiBP

Como visto na Figura 2.15, a implantação da técnica de descarte de blocos em múltiplos dispositivos pode gerar um padrão irregular no processamento das células da matriz DP, causando desbalanceamento em toda a computação. Adicionalmente, se GPUs heterogêneas são usadas, usualmente não se sabe a priori a melhor divisão de colunas, e um *split* ruim também causa desbalanceamento. Para lidar com estes casos, o *framework* MultiBP possui a estratégia Dynamic-MultiBP. A ideia principal é evitar que os *buffers* de entrada e saída (I/O - Figura 5.6) se tornem cheios, uma vez que isto reduziria a velocidade de processamento.

O funcionamento do Dynamic-MultiBP é descrito a seguir. A atribuição de colunas às GPUs é feita inicialmente como no Static-MultiBP. Contudo, não são todas as colunas da matriz DP que são distribuídas para as GPUs. Em vez disso, um conjunto de pontos de parada (*breakpoints*) são definidos (conforme informado pelo usuário), e a computação da matriz DP é feita em partes (ou ciclos). Por exemplo, se o número de *breakpoints* é igual a 1, metade das colunas da matriz DP são processadas na primeira parte. Antes do final deste ciclo, os *buffers* de I/O de cada GPU são avaliados, e, se necessário, as colunas são redistribuídas antes do início do segundo ciclo, tentando evitar o enchimento dos *buffers*. Então, dados são trocados entre as GPUs e elas processam a segunda parte usando o novo *split* calculado.

A Figura 9.2 ilustra a divisão da matriz DP. Neste caso, a carga de trabalho inicial (20:10) foi redistribuída no *breakpoint*, e mais colunas são processadas pela GPU_2 no segundo ciclo (*split* 16:14).

É importante notar que as diferentes estratégias do MultiBP conduzem a diferentes números de blocos descartados. Para uma mesma comparação com sequências reais de 5 MBP, a Figura 9.3 mostra as áreas de *pruning* para: (a) MASA-CUDAlign 4.0 em 1 GPU - com *block pruning*; (b) atribuição estática de colunas em 2 GPUs - com *block pruning* e sem compartilhamento de escore; (c) Static-MultiBP em 2 GPUs - com *block pruning* e com compartilhamento de escore; e (d) Dynamic-MultiBP em 2 GPUs - com *block pruning*, com compartilhamento de escore e com redistribuição de carga trabalho. Como pode ser observado, o número de blocos descartados na Figura 9.3b é consideravelmente menor que a execução MASA-CUDAlign em 1 GPU (Figura 9.3a). O formato da área de *pruning* do Static-MultiBP (Figura 9.3c) é mais similar ao da Figura 9.3a. Já o formato do Dynamic-MultiBP (Figura 9.3d) é bastante diferente das outras, porque, neste caso, o valor do melhor escore já obtido é maior quando o segundo ciclo inicia (já que toda a

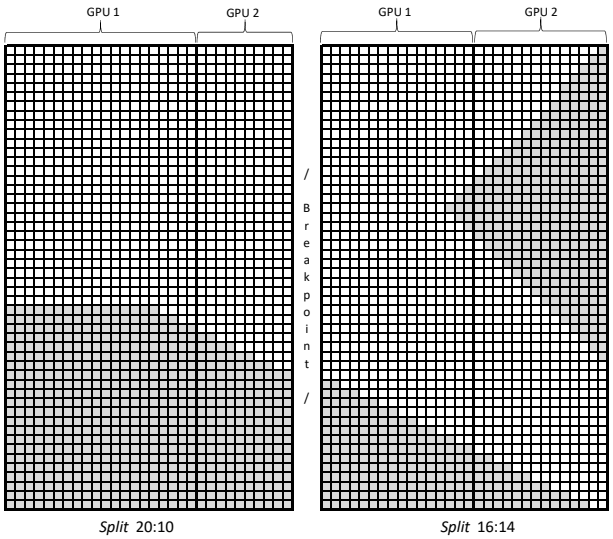
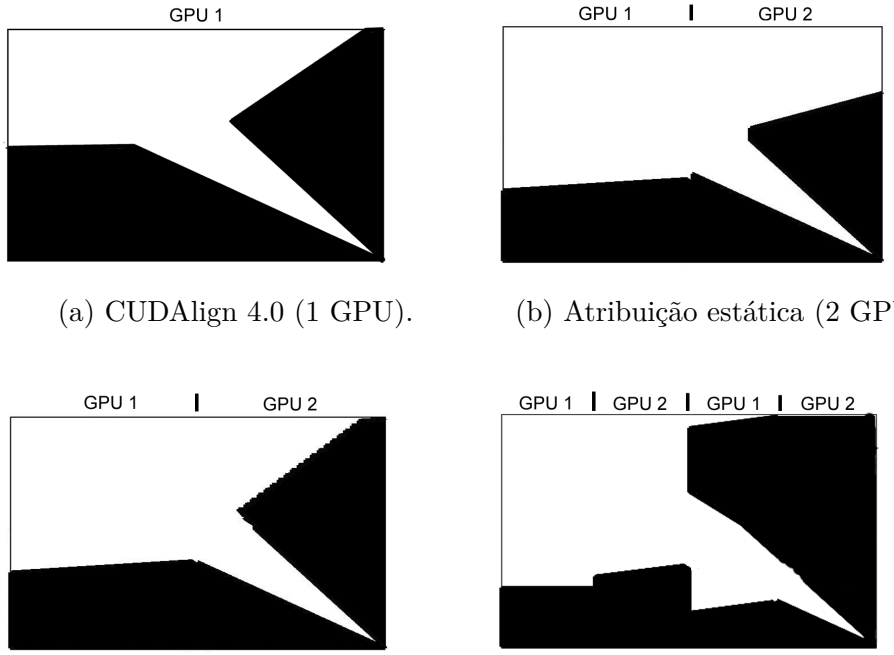


Figura 9.2: Dynamic-MultiBP: execução em 2 GPUs com redistribuição de carga de trabalho.

primeira parte da matriz foi processada), aumentando o número de blocos descartados nesta segunda parte da execução.



(a) CUDAlign 4.0 (1 GPU). (b) Atribuição estática (2 GPUs).

(c) Static-MultiBP (2 GPUs). (d) Dynamic-MultiBP (2 GPUs).

Figura 9.3: Formatos de descarte de células para diferentes estratégias de execução.

No Dynamic-MultiBP, um parâmetro de entrada informa o número de GPUs configuradas na execução. Esta informação é usada para determinar a última GPU do ciclo. O programa *Executor* que está rodando na última GPU do ciclo escreve um arquivo de con-

trole quando o número de diagonais processadas atinge um determinado limite (*threshold*), e outro arquivo é criado quando todas as diagonais são processadas. Estes arquivos são lidos pelo módulo *Controller* e usados de acordo com o estágio de execução do ciclo. Finalmente, a troca do melhor escore local proposta no Static-MultiBP (Seção 8.2.2) foi modificada para ser realizada entre GPUs dentro do mesmo ciclo, ou seja, o melhor escore local obtido na última GPU de um ciclo é enviado para a *thread* T_S do primeiro *Executor* dentro do mesmo ciclo, formando um anel interno de comunicação dentro de cada ciclo.

9.1.2.3 Algoritmo

O algoritmo do módulo *Executor* é mostrado no Algoritmo 2, que processa as colunas alocadas para cada GPU baseando-se no comando de execução enviado pelo módulo *Monitor*.

Algoritmo 2 Algoritmo MultiBP: o módulo Executor.

```
 1: procedure EXECUTOR
 2:     while (i ≤ total_diagonals) do
 3:         block_pruning_diagonal(i)
 4:         process_diagonal(i)
 5:         share_score(bestscore)
 6:         if (i == READINGPOINT*total_diagonals) AND (gpu == last_gpu) then
 7:             write_file(READ_BUFFERS)
 8:         end if
 9:         i ← i+1
10:     end while
11:     if (gpu == last_gpu) then
12:         write_file(END)
13:     end if
14: end procedure
```

O *loop* mostrado nas linhas 2 a 10 inicialmente executa o BP (linha 3), computa cada diagonal (linha 4) e compartilha o melhor escore local para a GPU vizinha (linha 5) através de *thread* de comunicação T_S. Quando o número de diagonais processadas atinge um determinado *threshold* (`READINGPOINT`), a última GPU em um ciclo escreve um arquivo que será lido pelo *Monitor* (linha 7). O valor do `READINGPOINT` foi definido quando 80% do número total de diagonais são processadas, pois este foi o ponto que melhor representou o comportamento dos *buffers* de I/O em execuções preliminares comparando sequências similares. A última GPU no ciclo também escreve o arquivo `END` quando todas as diagonais a ela designadas são processadas (linha 12).

9.1.3 Projeto do Módulo *Controller*

O algoritmo do módulo *Controller* é detalhado no Algoritmo 3.

O módulo *Controller* inicialmente processa o arquivo de configuração informado pelo usuário (linha 2). Então, o *Controller* estabelece comunicação através de *sockets* com

Algoritmo 3 Algoritmo MultiBP: o módulo Controller.

```
1:  procedure CONTROLLER(config)
2:      parser_file(config)
3:      while (i ≤ config.gpus) do
4:          connect_monitors(config.ips[i])
5:          i ← i+1
6:      end while
7:      mode ← confg.mode
8:      if (mode == Decision) then
9:          mode ← Decision-Maker(config_parameters)
10:     end if
11:     if (mode == Static) then
12:         while (i ≤ config.gpus) do
13:             send_msg(config.ips[i], EXEC, S0, S1, config.split, Static)
14:         end while
15:     else
16:         while (j ≤ config.breakpoints) do
17:             while (l ≤ config.gpus) do
18:                 send_msg(config.ips[i], EXEC, S0, S1, config.split, Dynamic)
19:                 wait_msg(READ_BUFFERS)
20:                 if (mode == Dynamic) then
21:                     parameters ← read(buffer_files)
22:                     config.split ← rebalance(parameters)
23:                 end if
24:                 wait_msg(END)
25:                 l ← l+1
26:             end while
27:             j ← j+1
28:         end while
29:         while (i ≤ config.gpus) do
30:             send_message(TERMINATE)
31:             i ← i+1
32:         end while
33:     end if
34:     write_statistics()
35: end procedure
```

os módulos *Monitor* (linhas 3 a 6). Se a estratégia Static ou a Dynamic é selecionada explicitamente pelo usuário no arquivo de configuração, a *string* de execução é montada e enviada para cada *Monitor*. Caso contrário, o modo Decision é selecionado (linha 8), e o *Controller* envia os parâmetros lidos para o módulo *Decision-Maker*. Se as sequências são pequenas (por exemplo, menores que 23 MBP) ou se a execução do procedimento de decisão levará tempo demais (definido através de uma estimativa), o módulo *Decision-Maker* retorna "Static". Caso contrário, ele informa que o procedimento de decisão será executado. Neste caso, a informação "Dynamic" é enviada pelo *Controller*, e o comando é então executado de forma assíncrona pelos módulos *Executor* (linha 18). A efetividade do procedimento de distribuição de colunas é avaliada no primeiro *breakpoint*. A opção por esta abordagem de iniciar a execução do Dynamic-MultiBP sem esperar a decisão deve-se ao fato de que o procedimento realizado pelo módulo *Decision-Maker* padrão pode levar um tempo considerável para ser executada. Por exemplo, a rotina *default* (Seção 9.1.5) demora cerca de 1 minuto. Sendo assim, adiar a execução tanto tempo poderia afetar negativamente o desempenho.

O *Controller* então espera por uma mensagem enviada do último *Monitor* no ciclo

(linha 19). Quando a mensagem chega (linha 21), ele lê os parâmetros de retenção de *buffer* para cada GPU e então calcula o *split* adequado usando a Equação 5.1, mudando a *string* de execução para o próximo ciclo de acordo com os novos valores de divisão de colunas calculados (linha 22). Logo, a heurística de redistribuição de colunas se baseia na instabilidade causada por bloqueios nos *buffers* de entrada e no número de linhas processadas por segundo, conforme proposto na Seção 5.3.2, que causam retenção nos *buffers* de comunicação.

O *Controller* espera até que o último *Monitor* do ciclo informe que o processamento foi finalizado (linha 24) para enviar o próximo comando de execução para todos os módulos *Monitor*. Ao final da última iteração do último ciclo, o *Controller* envia uma mensagem terminativa para os módulos *Monitor* (linha 30), e a execução termina.

É importante mencionar que a informação relacionada à última coluna processada por cada GPU é enviada para a próxima GPU no mesmo ciclo usando *sockets*. A troca de dados nos *breakpoints*, contudo, requer que toda a computação esteja finalizada antes que o próximo ciclo (com a nova distribuição de colunas) comece. Desta forma, neste ponto, os dados são enviados através de um arquivo binário armazenado em uma pasta compartilhada. Assume-se que em plataformas com 10 ou mais GPUs um sistema de arquivos rápido tal como o Lustre [119] existe, e que a gravação/leitura deste arquivo não gerará grande impacto no desempenho.

9.1.4 Projeto do Módulo *Monitor*

O Algoritmo 4 mostra o algoritmo do módulo *Monitor*.

Algoritmo 4 Algoritmo MultiBP: o módulo Monitor.

```
 1: procedure MONITOR
 2:     wait_connection()
 3:     exec_string ← receive_message()
 4:     while (exec_string ≠ TERMINATE) do
 5:         run_Executor(exec_string)
 6:         if (thisMonitor == lastMonitor) then
 7:             wait_file(READ_BUFFERS)
 8:             send_message(Controller, READ_BUFFERS)
 9:             wait_file(END)
10:             send_message(Controller, END)
11:             exec_string ← receive_message()
12:         end if
13:     end while
14: end procedure
```

O *Monitor* é responsável por prover a interface entre o *Controller* e os módulos *Executor*. Para tanto, primeiro ele espera pela conexão via *socket* do módulo *Controller* (linha 2) e, depois de receber os comandos do *Controller* (linha 3), executa o programa *Executor* (linha 5).

O último *Monitor* no ciclo é responsável por monitorar a informação escrita pelo módulo *Executor* rodando na última GPU. Se o módulo identifica o arquivo que sinaliza que os arquivos de *log* dos *buffers* devem ser lidos (linha 7), esta informação é enviada para o *Controller* (linha 8), e os arquivos são então lidos. De forma similar, o final da execução também é informado ao *Controller* (linha 10). Após o *Controller* calcular a nova distribuição de carga de trabalho (no caso da estratégia Dynamic-MultiBP), ele informa a nova *string* de execução para os módulos *Monitor*, repetindo estas ações em um *loop* (linhas 4 a 13), até que todos os ciclos se encerrem. Quando, a mensagem `TERMINATE` é recebida do *Controller*, o *Monitor* finaliza a execução.

9.1.5 Projeto do Módulo *Decision-Maker*

O objetivo do módulo *Decision-Maker* é decidir qual estratégia MultiBP será executada de acordo com parâmetros recebidos. Uma vez que há várias maneiras de fazer isto, o módulo proposto é flexível, e usuários podem prover seus próprios módulos para que sejam incorporados ao *framework*. Nesta seção, o algoritmo do *Decision-Maker* padrão (*default*) do *framework* é descrito, o que pode ser visto no Algoritmo 5.

Algoritmo 5 Algoritmo MultiBP: o módulo Decision-Maker.

```
 1: procedure DECISION-MAKER(seq1, seq2, gflops)
 2:     initialize_parameters()
 3:     if (seq1.size() < minsize) OR (seq2.size() < minsize) then
 4:         return Static
 5:     end if
 6:     if (gflops > 0) then
 7:         estimate ← (seq1.size() * seq2.size())/gflops
 8:         if (estimate < mintime) then
 9:             return Static
10:         end if
11:     end if
12:     ret ← alfn2 (seq1, seq2, kmers)
13:     if (ret < minalf) then
14:         return NoRebalance
15:     else
16:         return Dynamic
17:     end if
18: end procedure
```

O módulo *Decision-Maker* padrão utiliza a ferramenta *alignment-free* Alf-N2 [92], que computa a similaridade de um par de sequências com base na frequência $k - mer$. O objetivo, neste caso, é utilizar uma ferramenta que possa indicar de forma aproximada e rápida uma estimativa de similaridade entre as sequências, para sugerir qual a estratégia seria mais indicada.

Após fazer as inicializações (linha 2), se as sequências são muito pequenas, o módulo responde "Static" (linha 4). Caso contrário, o módulo estima o tempo de execução do primeiro ciclo do Dynamic-MultiBP baseando-se nos tamanhos das sequências e nos GFLOPS das GPUs informados pelo usuário (linha 7). Se o tempo de execução estimado

é menor que o tempo que o Alf-N2 levará para executar, o programa também retorna "Static" imediatamente (linha 9), para não prejudicar o desempenho. De acordo com resultados empíricos, o tempo limite no Alf-N2 é em torno de um minuto com $k-mer$=12, independentemente do tamanho das sequências. Se o tempo é mais longo, o programa vai escrever o estado "Executing" em um arquivo local, e o procedimento Alf-N2 será executado (linha 12). Em paralelo, o Dynamic-MultiBP executa.

Se o valor retornado pelo Alf-N2 é maior que um dado *threshold* (linha 13), o módulo retorna "Dynamic", porque estima-se que as sequências possuem similaridade média ou alta [109], conduzindo a uma área de *pruning* considerável; caso contrário, o módulo retorna "NoRebalance", o que informa para o *Controller* que não haverá redistribuição de carga de trabalho no *breakpoint*. Neste caso, a execução é equivalente à introdução de *breakpoints* na estratégia Static-MultiBP, com divisão da matriz DP em ciclos. De acordo com resultados experimentais, valores de Alf-N2 de 0,90 refletem, em média, os casos em que as sequências têm uma taxa de descarte de blocos de 30% ou mais, o que significa que as sequências possuem similaridade considerável [109].

Desta forma, o módulo *Decision-Maker* padrão possui três heurísticas para realizar a decisão: (a) caso o tamanho de uma das sequências seja inferior a um dado limite, o modo estático é selecionado; (b) se a estimativa do tempo de execução do primeiro ciclo, baseando-se no tamanho das sequências e nos GFLOPS das GPUs, for inferior ao tempo que levará a execução do Alf-N2, o modo estático é selecionado; e (c) caso o valor de similaridade retornado pelo Alf-N2 seja inferior a um dado limite, o modo estático escolhido, enquanto o modo dinâmico é selecionado caso o valor retornado seja superior a este limite.

O usuário pode implementar seu próprio *Decision-Maker*, bastando para isso informar caminho/nome_do_programa no arquivo de configuração. O módulo *Decision-Maker* provido pelo usuário deve conter os mesmos parâmetros de entrada que o padrão, e o resultado da decisão deve ser gerado em um arquivo com mesmo padrão de nomenclatura informada pelo usuário, no formato texto. O *Controller* processará este arquivo para decidir qual o modo de execução deverá ser usado.

9.2 Resultados Experimentais

Nesta seção, será avaliado o desempenho das estratégias MultiBP em cenários homogêneos e heterogêneos, comparando também com a execução sem BP. As seguintes opções foram testadas: (a) **C** - MASA-CUDAlign 4.0, sem BP (Seção 5.2.1.6); (b) **SM** - Static-MultiBP (Capítulo 8); (c) **DM** - Dynamic-MultiBP (Seção 9.1.2.2); e (d) **Dec** - com o uso do módulo *Decision-Maker* (Seção 9.1.5). Cada opção é um programa diferente, e

Tabela 9.1: Sequências usadas nos experimentos MultiBP.

Id	**Acesso**	**Tam.**	**Acesso**	**Tam.**	**Alf-N2**	***Prun.*** **(%)**
1M	CP000051.1	1M	AE002160.2	1M	0,12	10,96
3M	BA000035.2	3M	BX927147.1	3M	0,15	0,13
5M	AE016879.1	5M	AE017225.1	5M	0,99	53,17
7M	NC_005027.1	7M	NC_003997.3	5M	0,02	0,01
10M	NC_017186.1	10M	NC_014318.1	10M	0,99	53,34
23M	NT_033779.4	23M	NT_037436.3	25M	0,71	0,04
47M	NC_000021.7	47M	BA000046.3	32M	0,98	34,79
ChY	NC_000024.9	59M	NC_006492.3	26M	0,81	7,52
Ch21	NC_000021.8	48M	NC_006488.2	46M	0,98	47,08
Ch22	NC_000022.10	51M	NC_006489.3	49M	0,99	46,48
Ch19	NC_000019.9	59M	NC_006486.3	63M	0,99	28,83
Ch20	NC_000020.10	63M	NC_006487	61M	0,99	44,74
Ch16	NC_000016.9	90M	NC_006483.3	90M	0,98	35,89
Ch05	NC_000005.9	180M	NC_006472.3	182M	0,99	12,17
Ch01	NC_000001.9	249M	NC_006468.3	228M	0,98	N/D

eles foram compilados utilizando `gcc` (código C++) e `nvcc` (código CUDA) com a opção `-O3`. Todos os resultados de desempenho estão expressos em GCUPS (Seção 2.5).

9.2.1 Sequências e Plataformas

A Tabela 9.1 apresenta as sequências de DNA usadas nos testes. Estas sequências foram obtidas a partir do sítio do NCBI (`https://www.ncbi.nlm.nih.gov`).

Os tamanhos das sequências variam de 1 MBP a 249 MBP. A similaridade obtida pelo Alf-N2 é mostrada na coluna 6 e a taxa de descarte de blocos (%) obtida após execução real em 1 GPU é mostrada na coluna 7. A comparação envolvendo o cromossomo 1 (Ch01) não foi executada em 1 GPU para obter a taxa de *pruning* porque levaria alguns dias para executar. Como pode ser visto na Tabela 9.1, quando as sequências são muito grandes (>= 60 MBP), o Alf-N2 retorna similaridades maiores ou iguais a 0,98, mesmo nos casos nos quais as sequências não são tão similares (por exemplo, Ch05). Isso ocorre porque a ferramenta usa uma heurística muito rápida para calcular a similaridade, e sua estimativa não é muito boa em alguns casos, notadamente com sequências muito grandes. Por este motivo, apesar de este algoritmo ter sido adotado no *Decision-Maker* padrão (Seção 9.1.5), este módulo do *framework* foi projetado de forma flexível, de forma que o usuário possa desenvolver seu próprio módulo com outra heurística que julgue mais adequada.

Os testes foram executados em cinco ambientes dedicados:

- **Laico:** plataforma heterogênea com 1 NVidia GTX 680 (arquitetura Kepler - 1.536 núcleos, 1,01 GHz, 2 GB de memória global) e 1 NVidia GTX 980 Ti (arquitetura Maxwell - 2.816 núcleos, 1,00 GHz, 6 GB de memória global), instaladas em 2 *hosts*, cada um com 1 CPU Intel i7 3770 4-*core* 1,3 GHz, rede a 1 Gbps;
- **UOregonHet:** plataforma heterogênea com 1 NVidia Tesla P100 (arquitetura Pascal - 3.840 núcleos, 1,19 GHz, 12 GB de memória global) e 1 NVidia Tesla V100 SMX2 (arquitetura Volta - 5.120 núcleos, 1,53 GHz, 32 GB de memória global), instaladas em 1 *host* com 1 CPU IBM Power9 20-*core* 3,8 GHz;
- **UOregonHom:** plataforma homogênea com 2 NVidia Tesla A100 (arquitetura Ampere - 6.912 núcleos, 1,41 GHz, 40 GB de memória global), instaladas em 1 *host* com 2 CPUs Intel Xeon Gold 6148, 20-*core*, 3,7 GHz;
- **Bridges:** plataforma homogênea com 8 NVidia Tesla V100 SMX2 GPUs, instaladas em 2 *hosts* com 2 CPUs Intel Xeon Gold 6148 20-*core* 3,7 GHz (4 GPUs em cada *host*), rede a 100 Gbps;
- **NVidia:** plataforma homogênea com 512 NVidia Tesla V100 SMX2 GPUs, instaladas em 32 *hosts* com 2 CPUs Intel Xeon Gold 6148 20-*core* 3,7 GHz (8 GPUs em cada *host*), rede a 100 Gbps.

9.2.2 Resultados em Ambiente Heterogêneo

9.2.2.1 Resultados no Laico: GTX 680 + GTX 980

Neste cenário, o objetivos eram: (a) avaliar o comportamento do MultiBP em uma plataforma de GPUs com baixo poder computacional, com uma distribuição de carga de trabalho inicial apropriada; e (b) avaliar se o *Decision-Maker* seria capaz de escolher corretamente entre os modos estático e dinâmico. Todos os programas descritos no início desta seção e as 7 menores sequências na Tabela 9.1 (1M a 47M) foram utilizadas nos testes, além do caso Ch20.

Para definir a divisão de colunas inicial, três comparações similares (1M, 5M e 10M) foram feitas previamente. Para cada sequência, quatro casos de divisões de colunas foram testados (170:100, 180:100, 190:100 e 200:100), obtendo resultados similares (máxima variação de 1% no desvio padrão nos tempos de execução). Com isso, a decisão foi adotar o *split* 186:100, que é relação aproximada entre os GFLOPS das 2 GPUs (GTX 980 Ti : GTX 680).

A Figura 9.4 mostra o GCUPS (quanto maior, melhor) para 8 comparações. O Dynamic-MultiBP (DM) foi testado usando 1 e 2 *breakpoints* e o *Decision-Maker* (Dec)

foi testado com 1 *breakpoint*. Pode-se ver que, para sequências iguais ou maiores que 5 MBP, todas as estratégias MultiBP possuem desempenho superior ao MASA-CUDAlign 4.0 (C). Para as comparações 1M e 3M, pelo menos uma das estratégias MultiBP é melhor que C.

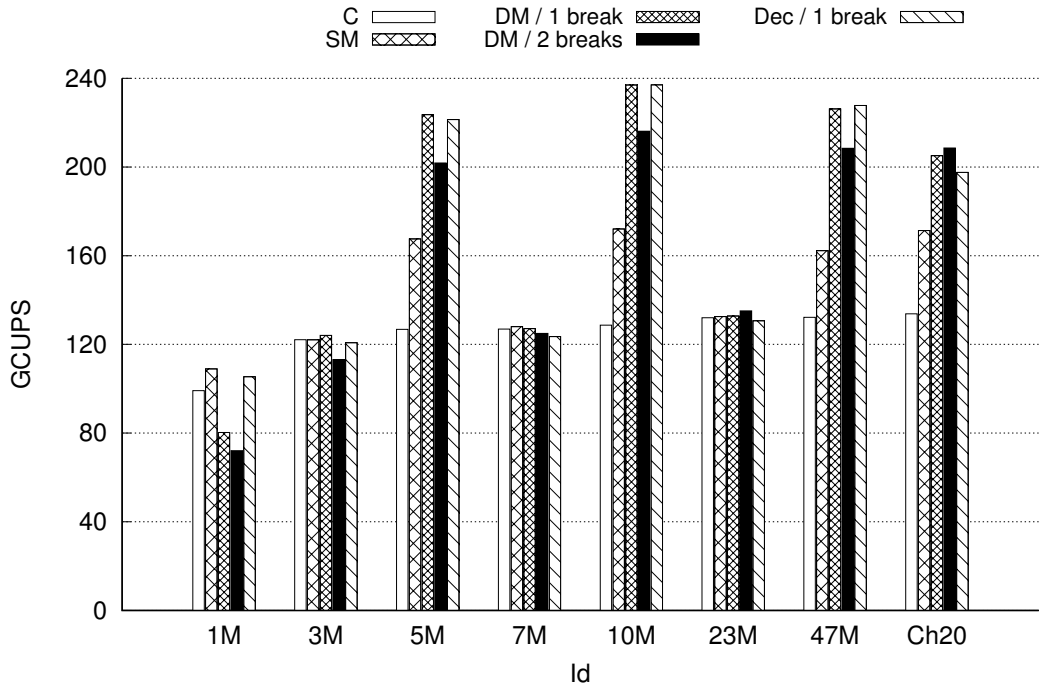


Figura 9.4: GCUPS no ambiente Laico (1 GTX 680 + 1 GTX 980 Ti).

O Static-MultiBP (SM) obteve o melhor GCUPS para as comparações 1M e 7M. O caso 1M possui taxa de BP (Tabela 9.1) considerada média e, neste caso, SM se beneficia da divisão de colunas configuradas estaticamente (186:100). Já na comparação 7M, as sequências são muito diferentes, com baixíssima atuação do descarte de blocos. Apesar disso, o SM apresenta um pequeno ganho em relação ao MASA-CUDAlign (C): 0,8%. Para todos os casos restantes, o uso da estratégia DM melhorou o desempenho, usando 1 (3M, 5M e 10M) ou 2 (23M e Ch20) *breakpoints*.

Além da redistribuição de colunas com base nos resultados parciais de uso de *buffer*, existe um efeito adicional introduzido pelo DM: como o primeiro ciclo é processado completamente antes que o segundo ciclo inicie, no momento da inicialização da segunda parte pode existir, caso as sequências sejam similares, um alto valor de melhor escore calculado pelo primeiro ciclo, e este CGBS (Seção 2.3) é compartilhado com as GPUs no início do ciclo seguinte. Desta forma, o BP no segundo (e demais) ciclos pode descartar mais blocos (como observado na Figura 9.3d), reduzindo o número de blocos processados pelas GPUs. A introdução de um segundo *breakpoint* (Figura 9.4), por outro lado, produziu na maioria dos casos resultados piores, porque os ganhos da redistribuição não superaram o *overhead* causado pela segunda divisão na matriz DP.

O uso do módulo Dec produziu os resultados esperados: para sequências menores ou pouco similares (1M, 3M, 7M e 23M), o modo de execução Static foi selecionado, e o desempenho é comparável ao SM; para as outras comparações, o modo Dynamic foi escolhido, e os resultados são comparáveis a "DM / 1 break".

O melhor desempenho neste ambiente foi obtido pelo caso 10M usando DM com 1 *breakpoint* ("DM / 1 break"): 240,7 GCUPS.

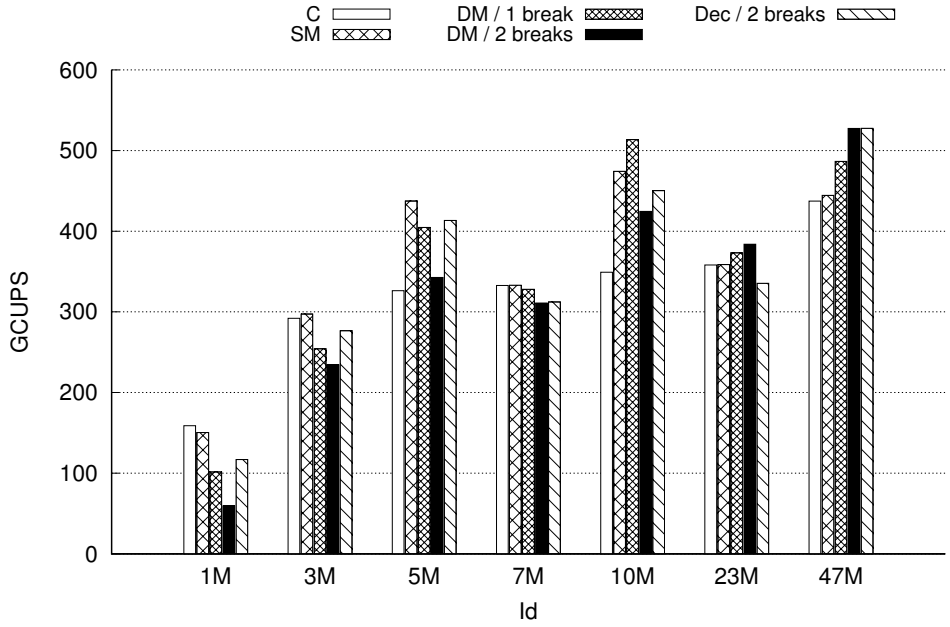
9.2.2.2 Resultados na U. Oregon com GPUs Heterogêneas: P100 + V100

Nesta plataforma, o objetivo era avaliar o comportamento do MultiBP em um ambiente heterogêneo com GPUs de alto poder computacional, onde a distribuição de trabalho inicial não fosse configurada de maneira ideal. O módulo de decisão (Dec) também foi avaliado neste cenário. Todas as estratégias foram testadas em UOregonHet (Seção 9.2.1), incluindo todas as comparações descritas na Tabela 9.1, exceto Ch01. Um número igual de colunas foi distribuído para as 2 GPUs (*split* 1:1), porque um dos objetivos era avaliar um cenário onde o usuário não sabe o poder de processamento de cada GPU.

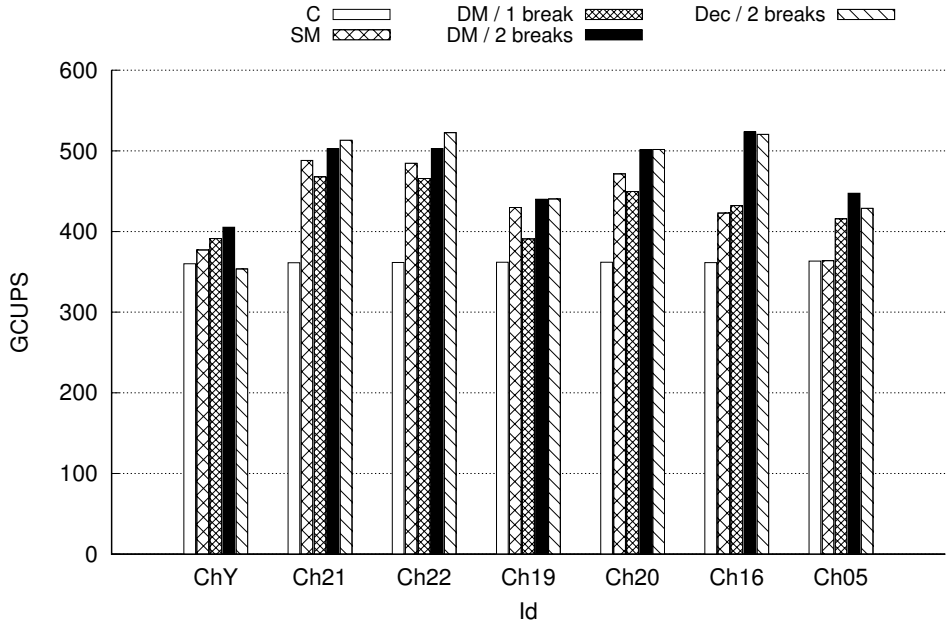
Da mesma forma que na Seção 9.2.2.1, as estratégias MultiBP superam C em todos os casos, exceto na comparação 1M, porque neste caso a comparação executa muito rapidamente em GPUs P100 e V100 sem o BP (7,5 segundos), e as sequências não são muito similares. Para os casos 3M, 5M e 7M, o SM apresentou melhores resultados (Figura 9.5a). Para todas as sequências maiores e muito similares, os melhores resultados foram obtidos por DM (10M com 1 *breakpoint*, e ChY, Ch16 e Ch05 com 2 *breakpoints*). Em alguns casos (Ch19, Ch20, Ch21 e Ch22), o Dec com 2 *breakpoints* obteve os melhores resultados (Figura 9.5b). Nestes casos, a redistribuição de colunas do DM muda a carga de trabalho nos *breakpoints*, o que melhorou o desempenho. Os ganhos causados pelo DM sobre o SM foram maiores que 23% em alguns casos, melhorando o GCUPS em 45% se comparado a C.

A fim de validar estatisticamente os ganhos obtidos pelo Dynamic-MultiBP neste cenário através de uma comparação pareada (Seção 6.1), os resultados das execuções "DM / 1 break" e C foram selecionados para cálculo do CI (Seção 6.1), de acordo com a Equação 6.1 e o nível de confiança de 90%. O CI obtido foi de (14,784 ; 83,576), o que valida os ganhos de acordo com o teste t (Seção 6.1).

O melhor desempenho neste ambiente foi obtido pelo caso 47M usando Dec com 2 *breakpoints* ("Dec / 2 breaks"): 527,6 GCUPS.



(a) Sequências menores que 48 MBP.



(b) Sequências maiores que 48 MBP.

Figura 9.5: GCUPS no ambiente UOregonHet (1 P100 + 1 V100).

9.2.3 Resultados em Ambiente Homogêneo

9.2.3.1 Resultados na U. Oregon com GPUs Homogêneas: 2 A100

Neste cenário, o objetivo era a avaliação do MultiBP em uma plataforma homogênea com um pequeno número de GPUs extremamente rápidas. As opções de programa C, SM e DM (1 e 2 *breakpoints*) foram usadas. O caso Ch01 não foi testado neste ambiente.

Visando verificar se a adição de um terceiro *breakpoint* produziria melhores resultados, foi utilizado um simulador de *block pruning* [1] com o parâmetro `row-cyclical` configurado como 3 (o que simula os 3 *breakpoints*), bem como alguns testes com o padrão "DM / 3 breaks" foram realizados com sequências reais. Em ambas as avaliações (teórica e empírica), a taxa de *pruning* foi pior que a obtida com 2 *breakpoints*, então esse cenário de testes com 3 *breakpoints* não foi utilizado. Adicionalmente, o módulo Dec também não foi habilitado nestes testes, uma vez que seu comportamento já foi avaliado nas Seções 9.2.2.1 e 9.2.2.2. Uma distribuição de carga de trabalho equitativa (1:1) foi usada, visto que o ambiente é homogêneo. Neste tipo de ambiente homogêneo, a estratégia dinâmica considera apenas o efeito do desbalanceamento causado na retenção de *buffers* pelo padrão irregular de *pruning*.

Como observado nas Figuras 9.6a e 9.6b, para todas as comparações, pelo menos uma variação do MultiBP supera C. Os melhores resultados obtidos pelo SM sobre o C têm o mesmo comportamento observado nos ambientes heterogêneos para todas as sequências similares, mas com ganhos menores. Os ganhos menos significativos se devem ao uso de 2 GPUs idênticas com grande número de núcleos e com alto poder de processamento. O uso do DM com 1 e 2 *breakpoints* produziram melhores resultados em 5 casos (5M, 7M, 10M, ChY e Ch22), enquanto SM foi superior nas outras comparações (1M, 3M, 23M, 47M, Ch21, Ch19, Ch20, Ch16 e Ch05). Observando os arquivos de *log*, notou-se que as execuções SM geraram mínima retenção de *buffer*, então a distribuição de carga inicial (1:1) mostrou-se bem balanceada neste ambiente, mesmo com o uso do BP.

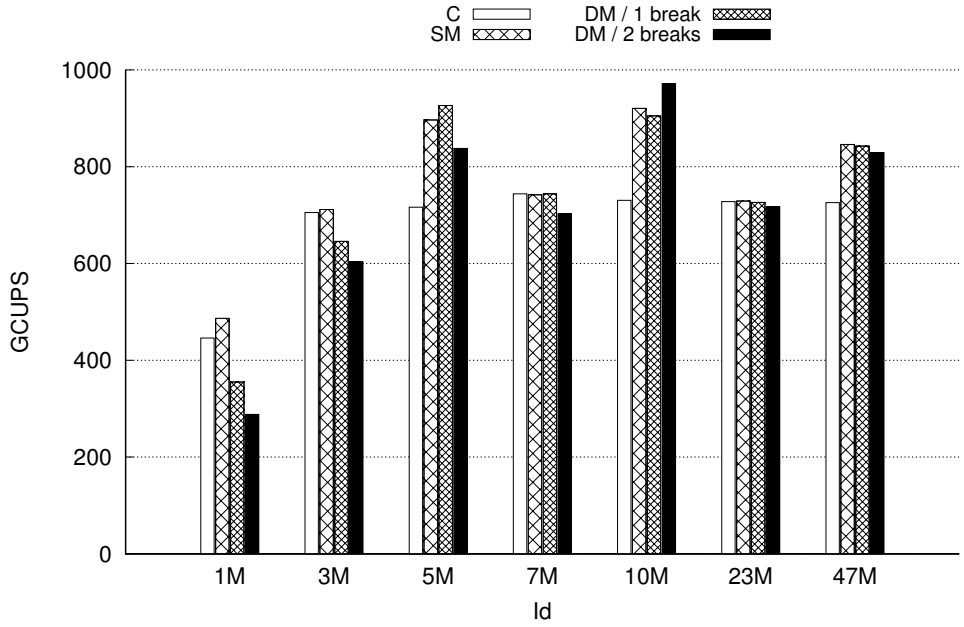
Os resultados dos ganhos do SM sobre C foram também validados utilizando a técnica de comparação pareada. Aplicando-se os dados de execução à Equação 6.1 e utilizando um nível de confiança de 90% obteve-se o CI (59,414 ; 142,954), confirmando os ganhos. Em contrapartida, a mesma técnica aplicada aos resultados do "DM / 2 breaks" resultou em um CI de (-8,447 ; 115,658). Como este intervalo inclui o zero, pode-se afirmar que a solução DM não pode ser considerada melhor que C na execução com 2 *breakpoints* neste ambiente com 2 A100 e um nível de confiança de 90%.

O melhor desempenho neste ambiente foi obtido pelo caso 10M usando DM com 2 *breakpoints* ("DM / 2 breaks"): 971,9 GCUPS.

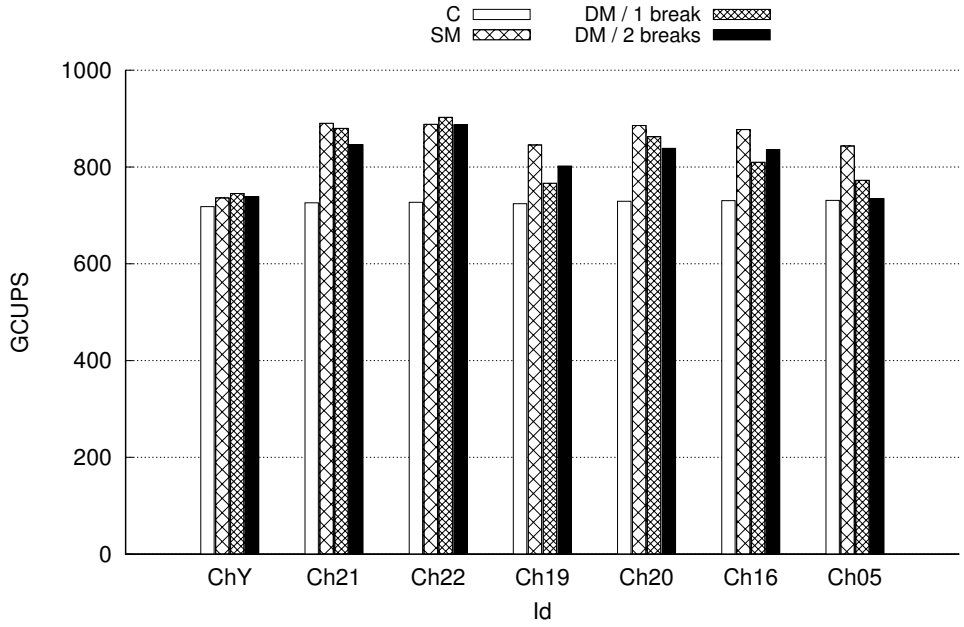
9.2.3.2 Resultados no Bridges: 8 V100

A execução no ambiente Bridges tinha dois objetivos principais: verificar se o comportamento observado na Seção 9.2.3.1 seria reproduzido em um ambiente com mais GPUs e observar o *speedup* da solução quando mais GPUs são adicionadas. Desta forma, inicialmente foram testadas algumas sequências utilizando-se apenas 2 e 4 V100, com

[1] Acessível em https://edanssandes.github.io/sim-block-pruning/ (fevereiro, 2021).



(a) Sequências menores que 48 MBP.



(b) Sequências maiores que 48 MBP.

Figura 9.6: GCUPS no ambiente UOregonHom (2 A100).

distribuição equitativa de colunas entre as GPUs (1:1 e 1:1:1:1, respectivamente). Foram executados os programas C, SM e DM, sendo este último executado usando 1 e 2 *breakpoints*, e os resultados em 2 e 4 GPUs podem ser vistos nas Tabelas 9.2 e 9.3, respectivamente.

Inicialmente, cabe destacar que, com poucas exceções nos casos com sequências menos similares (casos 1M em 2 V100, e 7M e 23M em 4 V100), pelo menos uma solução MultiBP

Tabela 9.2: GCUPS no ambiente NVidia (2 V100).

Id	**C** ***(GCUPS)***	**SM** ***(GCUPS)***	**DM / 1 break** ***(GCUPS)***	**DM / 2 breaks** ***(GCUPS)***
1M	331,2	322,5	265,3	189,3
3M	450,1	485,0	451,1	412,0
5M	498,9	671,5	725,1	533,4
7M	508,8	540,4	512,0	485,5
10M	548,4	713,3	807,4	636,2
23M	575,6	581,8	581,2	574,6
47M	581,8	691,1	720,8	649,8
Ch19	588,0	685,9	667,8	652,3
Ch20	588,2	748,5	755,6	668,0
Ch21	586,3	746,2	788,8	681,4
Ch22	589,9	745,5	**826,8**	645,4
Ch16	594,5	730,6	666,7	681,9
Ch05	600,8	699,0	640,8	617,5
ChY	593,7	602,5	612,8	607,4

Tabela 9.3: GCUPS no ambiente NVidia (4 V100).

Id	**C** ***(GCUPS)***	**SM** ***(GCUPS)***	**DM / 1 break** ***(GCUPS)***	**DM / 2 breaks** ***(GCUPS)***
1M	314,7	367,1	165,3	120,8
3M	754,9	757,1	647,9	528,9
5M	902,6	1019,3	929,0	727,5
7M	945,2	941,7	830,8	750,9
10M	1.032,3	1.157,9	1.135,3	1.081,6
23M	1.123,0	1.120,7	1.108,2	1.078,2
47M	1.150,9	1.254,5	1.215,6	1.209,2
Ch19	1.164,2	1.280,6	1.235,8	1.262,3
Ch20	1.165,9	1.307,4	1.254,7	1.267,6
Ch21	1.157,2	1.298,9	1.261,8	1.244,6
Ch22	1.160,0	1.308,0	1.264,5	1.260,0
Ch16	1.160,7	**1.316,4**	1.261,2	1.269,3
Ch05	1.192,9	1.199,6	1.226,9	1.232,8

apresentou desempenho superior a C. Na maioria dos casos (16 de 27 experimentos), a solução SM atingiu o melhor desempenho, indicando que esta estratégia é adequada para ambientes homogêneos com GPUs com alto poder computacional, como observado na Seção 9.2.3.1. Em 2 V100, o uso da estratégia DM produz ganhos adicionais em relação ao SM, como observado nos casos 5M, 10M, 47M, Ch20, Ch21, Ch22 e ChY. A adição de um segundo *breakpoint* ("DM / 2 breaks" nas Tabelas 9.2 e 9.3), contudo, não trouxe benefícios ao processamento, só obtendo melhor resultado no caso Ch05 em 4 GPUs, mas

quase insignificante em relação ao obtido com 1 *breakpoint*: ganho de 0,5%. O melhor desempenho com 4 V100 foi obtido na comparação Ch16 com o SM: 1.316,4 GCUPS.

O efeito da estratégia dinâmica nos *buffers* de entrada e saída foi avaliado na execução em 4 GPUs V100, tomando por base a comparação Ch19 e uso de 1 *breakpoint* (Figura 9.7), o que implica na execução em dois ciclos. Como pode-se verificar, no primeiro ciclo o enchimento dos *buffers* é insignificante. Na execução na primeira GPU no segundo ciclo, ocorre a saturação do *buffer* de entrada (I_1), o que era esperado, visto que a GPU está lendo os dados do arquivo gerado pela última GPU do ciclo anterior, que já estão disponíveis para leitura. Contudo, este enchimento não causa retenção na execução, como pode ser observado no pequeno uso dos *buffers* das próximas GPUs. Cabe destacar que, no ambiente homogêneo, a redistribuição altera pouco a distribuição de colunas na comparação Ch19. Neste exemplo, a distribuição uniforme inicial 250:250:250:250 foi modificada para 250:250:249:251 após o *breakpoint*.

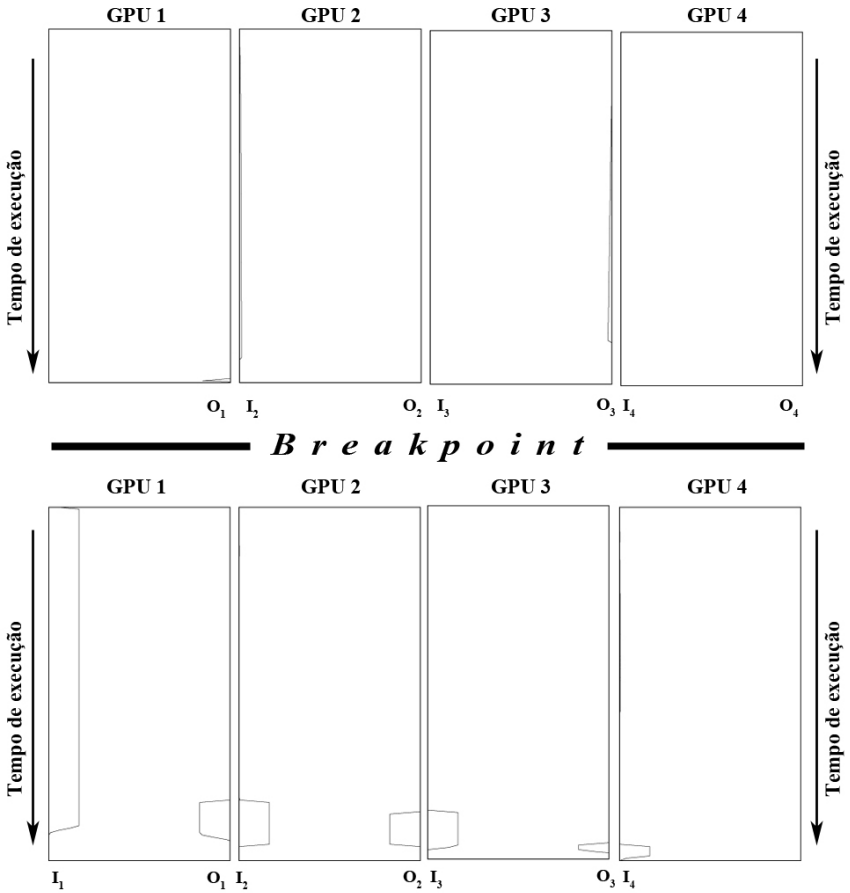
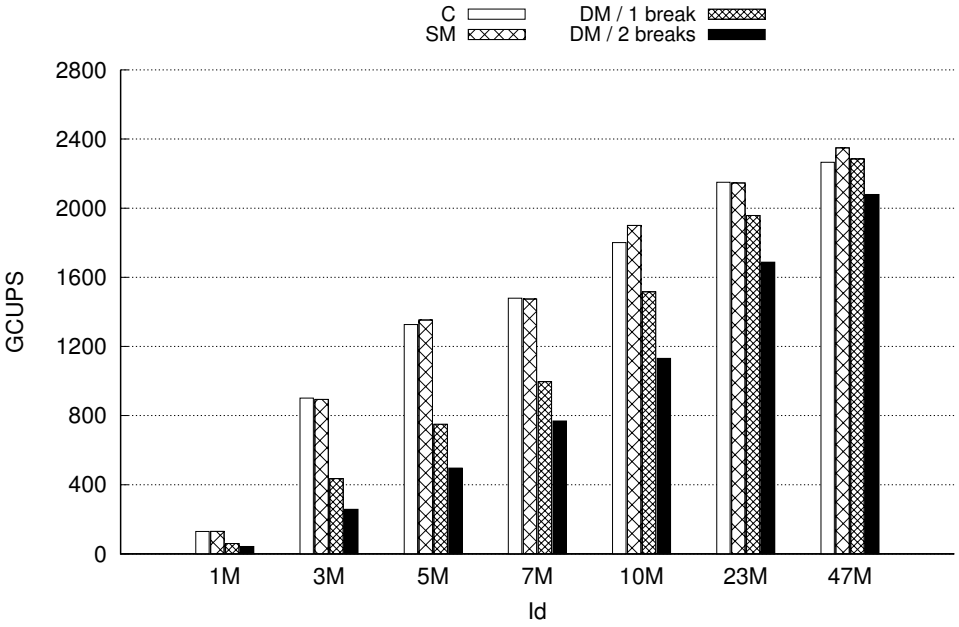
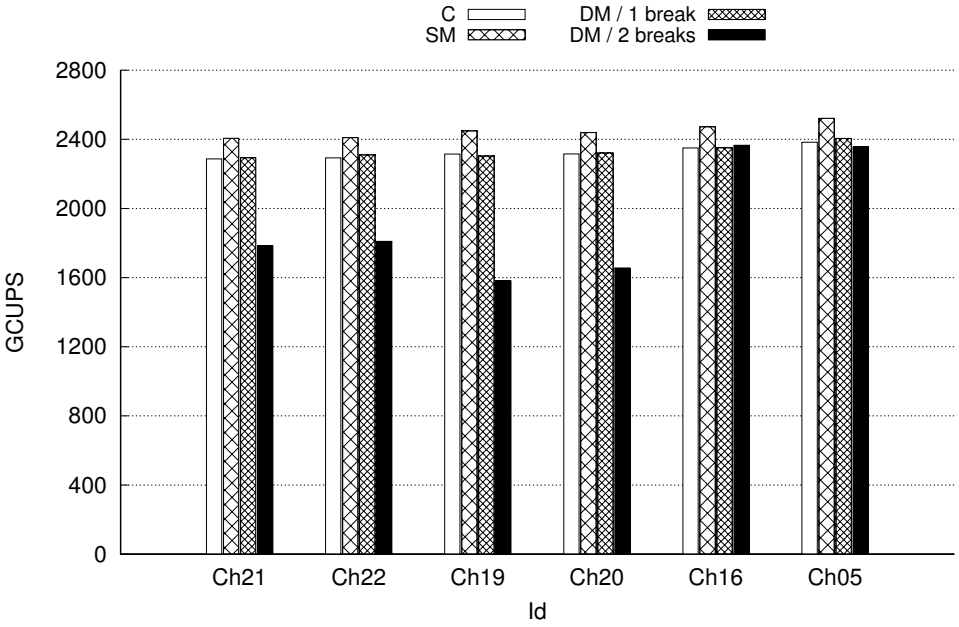


Figura 9.7: Enchimento de *buffers* na execução dinâmica na comparação Ch19.

Para os testes com 8 V100, foram usados os mesmos programas testados com 2 e 4 GPUs, e todos os casos de teste, exceto Ch01. Uma distribuição uniforme (1:1:1:1:1:1:1:1) foi usada nesta configuração. A Figura 9.8 mostra os resultados consolidados.



(a) Sequências menores que 48 MBP.



(b) Sequências maiores que 48 MBP.

Figura 9.8: GCUPS no ambiente Bridges (8 V100).

Como pode ser visto na Figura 9.8, o CUDAlign (C) obteve resultados similares ou levemente melhores para sequências pequenas ou dissimilares (1M, 3M, 7M e 23M - Figura 9.8a). Contudo, para todos os demais casos (5M, 10M, 47M, Ch21, Ch22, Ch19, Ch20, Ch16 e Ch05), o SM atingiu o melhor desempenho, como no ambiente UOregonHom (Seção 9.2.3.1). A estratégia DM não atingiu resultados melhores neste ambiente porque os *buffers* não chegaram a ser completamente preenchidos, e então o fluxo de processamento

da matriz DP não foi afetado. O pico de desempenho de 2.521,5 GCUPS foi obtido pela estratégia SM na comparação Ch05, com sequências contendo mais de 180 MBP.

Para avaliar o *speedup*, os resultados em 8 V100 foram comparados aos obtidos com 2 e 4 V100. Os *speedups* da estratégia SM em seis das maiores sequências testadas (comparações Ch19, Ch20, Ch21, Ch22, Ch16 e Ch05) podem ser vistos na Figura 9.9, comparando as execuções em 4 e 8 GPUs em relação à execução em 2 GPUs. Pode-se verificar que a solução possui boa escalabilidade, aproximando-se do limite teórico. Avaliando-se, por exemplo, os resultados obtidos pelo SM na comparação Ch05 em 8 V100 (2.521,5 GCUPS) e em 4 V100 (1.193 GCUPS) em relação à execução em 2 V100 (699 GCUPS), verifica-se um *speedup* de 3,61x e 1,72x, respectivamente, ante um *speedup* teórico linear de 4x e 2x.

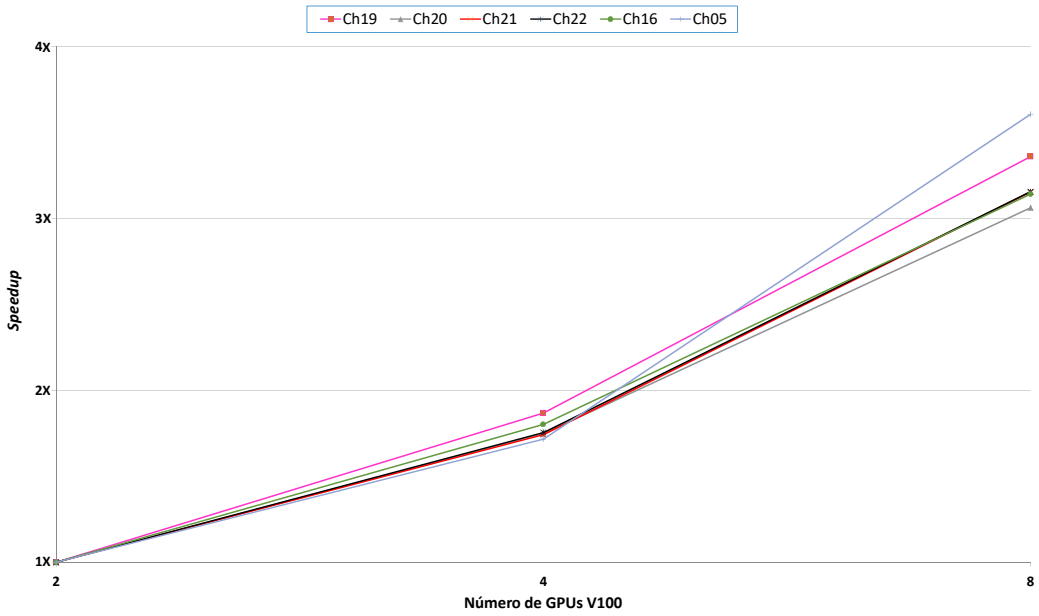


Figura 9.9: *Speedup* da estratégia SM em até 8 GPUs V100.

9.2.3.3 Resultados na NVidia: 512 V100

Para os testes no *cluster* da NVidia, foi necessário criar *scripts* de execução das estratégias em ambiente SLURM [126]. Estes *scripts*, bem como o código-fonte das soluções, foram disponibilizados na página do projeto criado no GitHub (Anexo I). No *cluster* NVidia foram testadas as soluções C e SM. O DM não foi executado neste ambiente uma vez que não gerou melhores resultados em ambientes homogêneos com 8 GPUs (Seção 9.2.3.2). Foram selecionadas sequências de cromossomos de diferentes tamanhos: pequeno (Ch21), médio (Ch16) e grande (Ch05). Adicionalmente, o caso de teste Ch01 (muito grande - mais de 228 MBP) também foi testado neste ambiente no cenário com pelo menos 128 GPUs.

Após realizar os testes com até 64 GPUs, foi identificado que os comportamentos do SM e do C eram muito próximos, e que entre 32 e 64 GPUs o CUDAlign passava a ficar levemente superior ao SM. Acreditamos que isto ocorreu pois o poder computacional das GPUs V100 é elevado, não havendo ganho significativo em executar a rotina de BP: é mais vantajoso simplesmente deixar as GPUs processarem todos os blocos da matriz DP. Para validar este cenário, foram testadas as três sequências menores (Ch21, Ch16 e Ch05) variando-se o número de GPUs de 8 em 8, obtendo-se o gráfico da Figura 9.10.

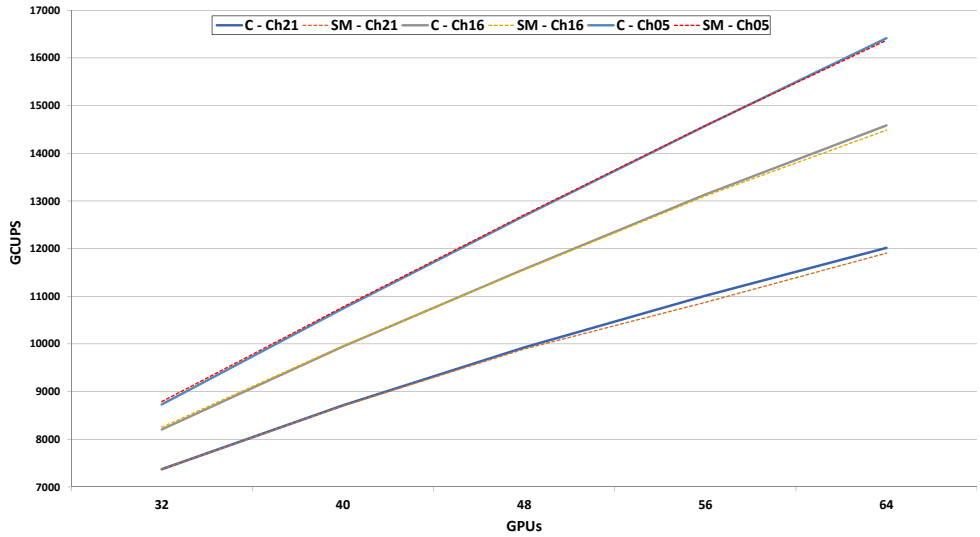


Figura 9.10: GCUPS no ambiente NVidia (32 a 64 V100).

Um aspecto relevante que se observa na Figura 7.2 é que o crescimento de desempenho quando a quantidade de GPUs aumenta é menos acentuado nas sequências Ch21 e Ch16 que na sequência Ch05. Isto ocorre pois, ao se distribuir as colunas da matriz DP entre as GPUs, as sequências menores não chegam a explorar totalmente o poder de processamento de todas as GPUs, diminuindo o *speedup*. Ademais, observa-se que os desempenhos das soluções C e SM para cada par de sequências são muito próximos. Com efeito, através da comparação pareada (Seção 6.1) não foi possível afirmar com um nível de confiança de 90% que os resultados sejam significativamente diferentes. Outrossim, o ponto onde a solução C se torna mais vantajosa é diferente em cada sequência, não sendo possível determinar a priori qual solução apresentaria o melhor desempenho em uma dada configuração.

Desta forma, optou-se por executar apenas a estratégia SM nesta plataforma homogênea NVidia, visando avaliar a escalabilidade desta estratégia MultiBP em um robusto *cluster* com grande número de GPUs, usando 16, 32, 64, 128, 256 e 512 GPUs V100. Os resultados são mostrados na Figura 9.11 e na Tabela 9.4, bem como o *speedup*.

Como pode ser visto, o SM possui escalabilidade muito boa quando o número de GPUs aumenta, mas atinge um limite se o paralelismo das GPUs não é totalmente explorado. Para o caso Ch05, por exemplo, o *speedup* obtido ("Spd." na Tabela 9.4) foi muito bom

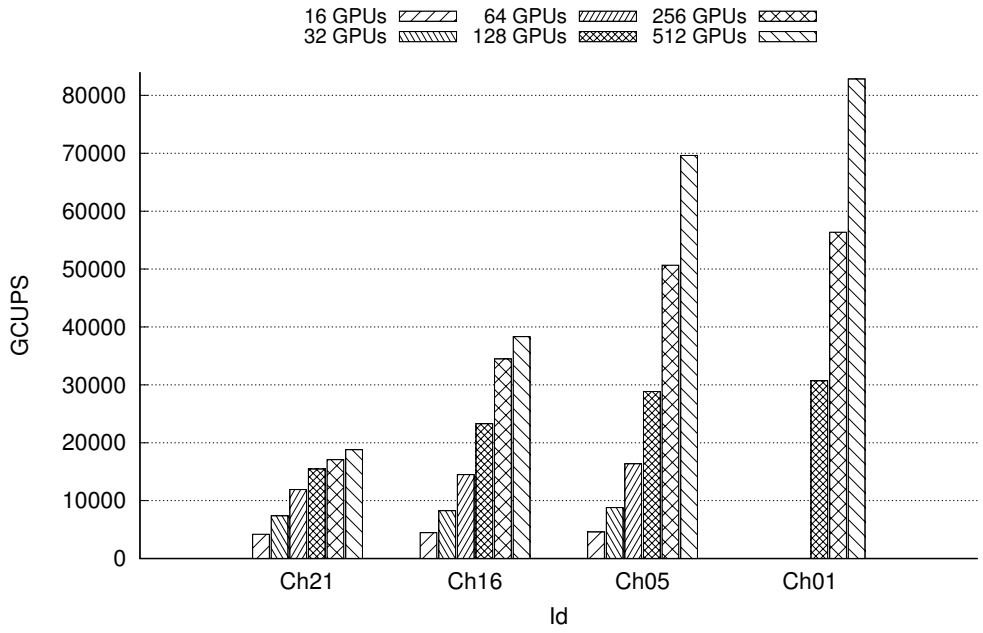


Figura 9.11: GCUPS no ambiente NVidia (16 a 512 V100).

Tabela 9.4: TCUPS e *speedup* no ambiente NVidia (16 a 512 V100).

GPUs (nós)	**Ch21**		**Ch16**		**Ch05**	
	TCUPS	**Spd.**	**TCUPS**	**Spd.**	**TCUPS**	**Spd.**
16 (1)	4,20	1x	4,46	1x	4,60	1x
32 (2)	7,37	1,75x	8,25	1,85x	8,79	1,91x
64 (4)	11,90	2,83x	14,49	3,25x	16,37	3,56x
128 (8)	15,47	3,68x	23,27	5,21x	28,84	6,26x
256 (16)	17,07	5,21x	34,47	7,72x	50,66	11,00x
512 (32)	18,80	6,26x	38,33	8,59x	69,62	15,12x

até 128 GPUs (6,26x, contra um limite de 8x), mas atingiu 15,12x quando o número de *hosts* (GPUs) é aumentado de 1 (16) para 32 (512), o que representa menos da metade do *speedup* teórico no caso (32x).

O melhor desempenho obtido foi de 82.822,8 GCUPS, no caso Ch01 executando em 512 GPUs. Esta comparação foi concluída em cerca de 11 minutos. Este desempenho foi quase 7,99x o GCUPS obtido pelo CUDAlign 4.0 em 384 GPUs (Seção 5.2.1.6) e 2,52x o desempenho reportado pelo BioSEAL (Seção 5.2.1.8) na comparação de duas sequências longas em um ambiente de simulação (Tabela 5.2). Ao nosso melhor conhecimento, este é o melhor desempenho para uma ferramenta de comparação de sequências biológicas com base em variantes do algoritmo SW em GPUs.

9.3 Conclusão do Capítulo

Neste capítulo, foi apresentado e avaliado um *framework* projetado para a comparação de sequências de DNA em múltiplas GPUs com descarte de blocos. Uma nova estratégia de execução com redistribuição de carga de trabalho dinâmica (Dynamic-MultiBP) foi juntada à solução com distribuição estática de colunas com compartilhamento de escore mostrada no Capítulo 8 (Static-MultiBP). Adicionalmente, um módulo flexível (*Decision-Maker*) foi proposto para decidir qual destas duas estratégias deve ser escolhida em um determinado cenário.

As estratégias propostas foram integradas ao MASA-CUDAlign 4.0 (Seção 5.2.1.6), que era uma ferramenta no estado da arte na comparação de sequências longas, tendo sido testada em vários ambientes homogêneos e heterogêneos de GPUs. Em plataformas heterogêneas com 2 GPUs, as soluções foram validadas em diferentes arquiteturas NVidia (Kepler, Maxwell, Pascal e Volta), com prevalência da estratégia dinâmica, e ganhos de desempenho de até 80,3%. As estratégias também foram testadas em ambientes homogêneos (uma plataforma com GPUs da arquitetura Ampere e outras duas com GPUs da arquitetura Volta), onde os ganhos relativos em relação à solução sem descarte de blocos foram menores, mais significativos (até 29,4%). No ambiente homogêneo com mais GPUs, a estratégia estática se mostrou mais eficiente. Finalmente, foi possível executar a estratégia estática com compartilhamento de escore em um grande *cluster* contendo 512 GPUs V100, obtendo-se 82.822,8 GCUPS, o que, ao que se sabe, é o melhor desempenho publicado para uma solução de comparação de sequências em GPU. A Tabela 9.5 compara a solução proposta com outras da literatura para ambientes uniformes (Seção 5.2.1).

Tabela 9.5: MultiBP e outras soluções de comparação de sequências em ambiente uniforme.

Seção	**Tipos de Dispositivos**	**Tamanho Máximo**	**Paralelismo**	**Quantidade de Dispositivos**	**GCUPS**
Ino et al.	GPU	10^3	CUDA	8 GPUs	64,0
SWIPE	CPU	10^3	SSE	12 CPUs	106,2
SW-Rivyera	FPGA	10^3	*Hardware* customizado	128 FPGAs	6.020,0
Swaphi-LS	Intel Xeon Phi	10^7	MPI e IMICS	4 Intel Xeon Phis	111,4
SW-MVM	CPU	10^2	SSE e MPI	128 CPUs	900,0
CUDAlign 4.0	GPU	10^8	CUDA	384 GPUs	10.370,0
CloudSW	CPU em nuvem	10^3	Apache Spark	50 CPUs	529,9
BioSEAL	ReCAM	10^8	Simulação	Simulação	32.800,0
		10^2			90.700,0
MultiBP	**GPU**	10^8	**CUDA**	**512 GPUs**	**82.822,8**

A proposta e os resultados apresentados neste capítulo foram submetidos para revista de referência na área de computação de alto desempenho, e encontra-se em estágio de revisão [26].

Parte III

Conclusão

Capítulo 10

Conclusão e Trabalhos Futuros

10.1 Conclusão

O uso de algoritmos exatos tais como o SW (Seção 2.2.2) e suas variantes na comparação de sequências é uma tarefa ainda muito relevante para a Bioinformática. Devido a uma complexidade quadrática de tempo, esta tarefa pode levar muito tempo para ser concluída, principalmente se duas sequências longas são comparadas, mesmo utilizando-se ferramentas no estado da arte como o MASA-CUDAlign 4.0 (Seção 5.2.1.6). Duas técnicas promissoras utilizadas nos últimos anos nas soluções de comparação de sequências são a execução em múltiplos dispositivos e o uso de técnicas de poda (Seção 2.3). Algumas soluções já foram propostas utilizando múltiplos dispositivos de diversas arquiteturas diferentes, mas limitando-se a duas GPUs (Seção 5.1.6).

Nesta Tese de Doutorado, foram propostas e avaliadas estratégias de distribuição de carga de trabalho na comparação de sequências longas de DNA em múltiplos dispositivos com técnica de descarte de blocos (BP). As soluções propostas são voltadas para a execução em GPUs produzidas pela NVidia, e foram incorporadas ao MASA-CUDAlign 4.0, que representava o estado da arte para o alinhamento de sequências longas em múltiplas GPUs.

Inicialmente, foi feito um estudo que fundamentou o projeto das soluções apresentadas nesta Tese. O objetivo deste estudo era realizar uma análise quantitativa de soluções de comparação de sequências em GPUs através de técnicas estatísticas [27]. Técnicas como a avaliação da média e comparação pareada foram utilizadas para validar os ganhos obtidos pelo MASA-OpenCL (Seção 5.1.8) em relação ao MASA-CUDAlign. Ademais, experimentos permitiram propor e validar uma equação de regressão linear múltipla para prever, com ótimo coeficiente de determinação ($R^2 = 0,999$) e baixa taxa de erro (média de 5,9%), o tempo de execução de uma comparação para uma dada sequência em uma dada GPU.

A seguir, foram realizados testes experimentais utilizando múltiplos dispositivos em ambiente contendo 2 CPUs e 2 GPUs para avaliar o desempenho de soluções MASA (Seção 5.1.7) em ambientes heterogêneos ou híbridos. Nos testes utilizando dois dispositivos (2 GPUs ou 1 CPU e 1 GPU), verificou-se que o aspecto que mais interferia nos resultados era uma distribuição de carga inadequada. Na execução em 4 dispositivos (2 CPUs e 2 GPUs), verificou-se que a ordem de alocação dos dispositivos na execução não era relevante para o desempenho, desde que a distribuição de colunas estivesse balanceada.

A primeira estratégia proposta para a comparação em múltiplas GPUs com técnicas de descarte de blocos foi o Static-MultiBP [28]. Nesta versão, a estratégia permite a comparação de sequências longas utilizando-se *block pruning* (Seção 2.3.3) em múltiplas GPUs em ambiente homogêneo ou heterogêneo, mas com distribuição estática de trabalho entre os dispositivos. A estratégia foi testada em ambientes de médio porte (até 4 GPUs NVidia P100), comparando sequências de até 63 MBP e obtendo 694,8 GCUPS. Nos testes em ambiente heterogêneo, houve ganhos de desempenho de até 80,9% em sequências similares. Em ambiente homogêneo, os ganhos também foram significativos (até 17,4%), com bom *speedup*. Até onde temos conhecimento, esta foi a primeira solução que permitiu o uso de técnicas de poda em mais de 2 GPUs.

A proposta de estratégias de BP em múltiplas GPUs evoluiu então para a criação de um *framework* (MultiBP) que incorporasse a estratégia estática a uma outra baseada em reavaliação dinâmica da carga de trabalho [26], dividindo a matriz DP e criando ciclos de execução. A estratégia permitiu considerar o padrão não uniforme de processamento gerado pelo BP, bem como corrigir, ao longo dos ciclos, uma eventual distribuição de colunas inadequada informada pelo usuário. Adicionalmente, um módulo de decisão foi adicionado ao *framework*, visando indicar uma estratégia mais adequada em cada execução. As estratégias foram testadas em ambientes heterogêneos e homogêneos. De forma geral, a estratégia dinâmica se mostrou mais adequada nos ambientes heterogêneos, com ganhos de até 80,3% e obtendo até 527,6 GCUPS em um ambiente com 2 GPUs (P100 e V100). Já a estratégia estática apresentou melhores desempenhos no ambiente homogêneo, atingindo até 2.521,5 GCUPS em ambiente com 8 GPUs V100, e um máximo de 82.822,8 GCUPS obtido na execução em 512 GPUs V100. Salvo entendimento contrário, este é o melhor desempenho na comparação de sequências longas de DNA (pelo menos 1 MBP) em qualquer dispositivo, bem como o maior GCUPS já obtido para uma solução de comparação de sequências de qualquer tamanho em GPU.

Observando a Tabela 10.1 e comparando os resultados do MultiBP com outros dois trabalhos de maior relevância na comparação de sequências com pelo menos 1 MBP - MASA-CUDAlign 4.0 (Seção 5.2.1.6) e BioSEAL (Seção 5.2.1.8) -, podemos ver que as estratégias propostas nesta Tese evoluem o estado da arte do problema de comparação

Tabela 10.1: Comparação entre o MultiBP, MASA-CUDAlign 4.0 e BioSEAL.

Solução	Saída	BP	Distribuição	Ambiente	GCUPS
MultiBP	Escore	Sim	Estát./Dinâm.	512 X V100	82.822
MASA-CUDAlign 4.0	Escore e Alinh.	Não	Estát.	384 X M2090	10.370
BioSEAL	Escore e Alinh.	Não	Estát.	Simulação ReCAM	32.800

de sequências longas de DNA em quatro aspectos. Primeiro, devido à possibilidade de execução do BP em múltiplos dispositivos, uma vez que o MASA-CUDAlign 4.0 executa em várias GPUs mas sem descarte de blocos, e que este recurso também não é utilizado no BioSEAL. Segundo, uma nova estratégia de distribuição de carga de trabalho dinâmica é proposta. Como terceiro avanço, ao se comparar com o MASA-CUDAlign 4.0 em relação à quantidade de GPUs utilizadas, as soluções MultiBP foram executadas em até 512 GPUs com arquiteturas mais recentes da NVidia (Tesla V100), enquanto o MASA-CUDAlign 4.0 foi testado em 384 GPUs M2090. Finalmente, o desempenho obtido pela estratégia MultiBP é quase 8x o valor dos GCUPS obtidos pelo MASA-CUDAlign 4.0, e 2,52x o desempenho do BioSEAL na comparação de duas sequências longas de DNA, que foi testado em um simulador com a memória ReCAM [1].

Avaliando os resultados obtidos, podemos verificar que o *framework* MultiBP contribuiu no avanço de soluções de comparação de sequências longas de DNA em vários aspectos, e suas estratégias são aplicáveis a um grande número de configurações de ambientes de GPU, sejam com plataforma homogênea ou heterogênea, de pequeno, médio ou grande porte. As soluções desenvolvidas ao longo desta Tese contemplam todos os objetivos específicos planejados (Seção 1.3), principalmente o relacionado à adoção de uma estratégia dinâmica para distribuição de carga de trabalho. Planejamos a disponibilização do código fonte das soluções projetadas no repositório público do projeto (Anexo I) tão logo o artigo submetido seja revisado e publicado, para que o MultiBP possa ser utilizado por profissionais da área de Bioinformática.

10.2 Trabalhos Futuros

Nesta Tese de Doutorado, o objetivo foi investigar a incorporação da técnica de BP em múltiplas GPUs para a obtenção do melhor escore ótimo com estratégias de distribuição de carga de trabalho. Desta forma, a Tese não abordou a recuperação do alinhamento, o que demandaria modificações na fase de *traceback* do MASA-CUDAlign 4.0 (Seção 5.2.1.6). Como um trabalho futuro, pretendemos então adicionar esta funcionalidade ao *framework* MultiBP, possibilitando que o alinhamento possa ser recuperado.

[1] O BioSEAL reportou um desempenho de 90.700 GCUPS, mas umas das sequências não é longa.

Adicionalmente, a solução de distribuição dinâmica de carga de trabalho assume que a quantidade de GPUs disponíveis se mantém inalterada ao longo do processamento. Planejamos que esta alocação possa também ser dinâmica, reavaliando a necessidade de aumento ou diminuição das GPUs utilizadas em cada *breakpoint*. Isto pode ser particularmente útil em um ambiente de computação em nuvem, por exemplo: uma vez que os provedores costumam cobrar por tempo de uso de recurso, é possível determinar um certo limite desejável para o tempo de execução da comparação, e verificar, após cada ciclo, se os recursos alocados podem ser desligados ou se há necessidade de alocar novas GPUs para que se atinja o desempenho esperado. A solução então poderia ser integrada a *scripts* de gerenciamento das máquinas virtuais em nuvem para que a alocação dinâmica de recursos pudesse ocorrer. A execução em nuvem requer ajustes na forma de comunicação entre os módulos do MultiBP e no gerenciamento dos recursos.

A adição de novas heurísticas ao módulo de decisão utilizado na solução é também um dos trabalhos planejados. O módulo *Decision-Maker* (Seção 9.1.5) do MultiBP poderia, por exemplo, prever o cenário em que muitas GPUs de alto poder computacional estão sendo utilizadas, para indicar de forma antecipada a execução da estratégia estática em vez da estratégia dinâmica, já que esta não se mostrou mais eficiente neste cenário. Outro aspecto que será objeto de análise futura é a adaptação do módulo dinâmico a outros cenários de execução, como por exemplo em uma situação em que o *buffer* de comunicação pudesse ser incrementado de forma que não houvesse retenção, o que exige que a equação de redistribuição de colunas seja modificada para avaliar outros parâmetros de execução.

Esta Tese se concentrou na integração das estratégias de comparação de sequências longas em múltiplas GPUs, utilizando como base a MASA-CUDAlign 4.0. As estratégias propostas, contudo, podem ser também implementadas em outras extensões MASA (Seção 5.1.7) para outros tipos de *hardware*, como o MASA-OpenMP ou MASA-Phi. Isto permitiria que a solução pudesse ser executada em múltiplas CPUs ou Intel Xeon Phis, respectivamente, ou ainda em um ambiente híbrido, aumentando a capacidade de processamento. Pretendemos, portanto, projetar o descarte de blocos em múltiplos dispositivos também em outras extensões MASA, ajustando o *framework* proposto para que o usuário indique as plataformas de *hardware* desejadas.

Por fim, um outro trabalho futuro almejado é uma análise teórica do comportamento do BP em múltiplos dispositivos, tal como realizado para um dispositivo em [109]. Isto permitirá que o padrão de descarte de blocos em um dado ambiente possa ser estimado, dando mais subsídios para melhorar a eficiência dos algoritmos de redistribuição de carga de trabalho, pois seria possível predizer como uma determinada redistribuição de carga afetaria o BP, e, consequentemente, o desempenho.

Referências

[1] S. F. Altschul, W. Gish, W. Miller, E. W. Myers, and D. J. Lipman. Basic local alignment search tool. *Journal of molecular biology*, 215(3):403–410, 1990. 19

[2] M. Amaris, G. Lucarelli, C. Mommessin, and D. Trystram. Generic algorithms for scheduling applications on hybrid multi-core machines. In *European Conference on Parallel Processing*, pages 220–231. Springer, 2017. 42

[3] AMD. Placa gráfica amd radeon rx 6900 xt. https://www.amd.com/pt/products/graphics/amd-radeon-rx-6900-xt, Dec 2020. 30

[4] Y. Azar. On-line load balancing. In *Online Algorithms*, pages 178–195. Springer, 1998. 46, 47

[5] R. B. Batista, A. Boukerche, and A. C. M. A. Melo. A parallel strategy for biological sequence alignment in restricted memory space. *Journal of Parallel and Distributed Computing*, 68(4):548–561, 2008. 3, 48

[6] S. Batzoglou. The many faces of sequence alignment. *Briefings in bioinformatics*, 6(1):6–22, 2005. 11

[7] A. D. Baxevanis and B. F. Ouellette. Bioinformatics: a practical guide to the analysis of genes and proteins. 2004. 10, 11

[8] N. Buchbinder and J. S. Naor. Fair online load balancing. In *Proceedings of the eighteenth annual ACM symposium on Parallelism in algorithms and architectures*, pages 291–298. ACM, 2006. 47

[9] D. R. Butenhof. *Programming with POSIX threads*. Addison-Wesley Professional, 1997. 37

[10] G. Caffarena, C. Pedreira, C. Carreras, S. Bojanic, and O. Nieto-Taladriz. Fpga acceleration for dna sequence alignment. *Journal of Circuits, Systems, and Computers*, 16(02):245–266, 2007. 3, 48

[11] H. Casanova, A. Legrand, D. Zagorodnov, and F. Berman. Heuristics for scheduling parameter sweep applications in grid environments. In *Heterogeneous Computing Workshop, 2000.(HCW 2000) Proceedings. 9th*, pages 349–363. IEEE, 2000. 43

[12] T. L. Casavant and J. G. Kuhl. A taxonomy of scheduling in general-purpose distributed computing systems. *IEEE Transactions on software engineering*, 14(2):141–154, 1988. 4, 42, 45

[13] C. Chen and B. Schmidt. An adaptive grid implementation of dna sequence alignment. *Future Generation Computer Systems*, 21(7):988–1003, 2005. 4, 70

[14] D. E. Culler, J. P. Singh, and A. Gupta. *Parallel computer architecture: a hardware/software approach*. Morgan Kaufmann, 1999. 26

[15] L. Dagum and R. Menon. Openmp: an industry standard api for shared-memory programming. *Computational Science & Engineering, IEEE*, 5(1):46–55, 1998. 33

[16] T. Davidović and T. G. Crainic. Parallel local search to schedule communicating tasks on identical processors. *Parallel Computing*, 48:1–14, 2015. 43

[17] A. Davidson. A fast pruning algorithm for optimal sequence alignment. In *BIBE 2001*, pages 49–56. IEEE, 2001. 4, 19, 20

[18] A. Duran, E. Ayguadé, R. M. Badia, J. Labarta, L. Martinell, X. Martorell, and J. Planas. Ompss: a proposal for programming heterogeneous multi-core architectures. *Parallel Processing Letters*, 21(02):173–193, 2011. 56

[19] R. Durbin, S. R. Eddy, A. Krogh, and G. Mithison. *Biological sequence analysis: probabilistic models of proteins and nucleic acids*. Cambridge University Press, 1998. 12

[20] M. Farrar. Striped smith–waterman speeds database searches six times over other simd implementations. *Bioinformatics*, 23(2):156–161, 2006. 3, 48, 62, 66

[21] M. Farrar. Striped smith–waterman speeds database searches six times over other simd implementations. *Bioinformatics*, 23(2):156–161, 2007. 50

[22] D. G. Feitelson. Packing schemes for gang scheduling. In *Workshop on Job Scheduling Strategies for Parallel Processing*, pages 89–110. Springer, 1996. 45

[23] D. G. Feitelson and A. M. Weil. Utilization and predictability in scheduling the ibm sp2 with backfilling. In *Parallel Processing Symposium, 1998. IPPS/SPDP 1998. Proceedings of the First Merged International and Symposium on Parallel and Distributed Processing 1998*, pages 542–546. IEEE, 1998. 44

[24] J. W. Fickett. Fast optimal alignment. *Nucleic acids research*, 12(1Part1):175–179, 1984. 4, 18

[25] M. A. C. Figueiredo, F. d. O. Edans, and A. C. M. A. Melo. Parallel megabase dna sequence comparison with opencl. In *High Performance Computing (HiPC), 2015 IEEE 22nd International Conference on*, pages 436–445. IEEE, 2015. 3, 7, 48, 57, 76, 80, 84

[26] M. A. C. Figueiredo, J. Oliveira, E. F. O. Sandes, G. Teodoro, and A. C. Melo. Parallel fine-grained comparison of long dna sequences in homogeneous and heterogeneous gpu platforms with pruning. *submitted to a prestigious journal*, 2020. 7, 8, 136, 139, 153

[27] M. A. C. Figueiredo, E. F. O. Sandes, G. N. Rodrigues, G. L. Teodoro, and A. C. M. de Melo. Masa-opencl: Parallel pruned comparison of long dna sequences with opencl. *Concurrency and Computation: Practice and Experience*, 31(11):e5039, 2019. 7, 8, 76, 79, 85, 138, 153

[28] M. A. C. Figueiredo, E. F. O. Sandes, G. Teodoro, and A. C. Melo. Parallel comparison of huge dna sequences in multiple gpus with block pruning. In *2020 28th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*, pages 22–29. IEEE, 2020. 6, 8, 95, 111, 139, 153

[29] R. J. Francis, J. Rose, and Z. G. Vranesic. *Field-programmable gate arrays*, volume 180. Springer, 1992. 29

[30] M. Fulekar. *Bioinformatics: applications in life and environmental sciences*. Springer Science & Business Media, 2009. 1

[31] A. R. Galper and D. L. Brutlag. *Parallel similarity search and alignment with the dynamic programming method*. Knowledge Systems Laboratory, Medical Computer Science, Stanford University, 1990. 3, 48

[32] W. Gentzsch. Sun grid engine: Towards creating a compute power grid. In *Cluster Computing and the Grid, 2001. Proceedings. First IEEE/ACM International Symposium on*, pages 35–36. IEEE, 2001. 70

[33] G. Giribet and W. C. Wheeler. On gaps. *Molecular phylogenetics and evolution*, 13(1):132–143, 1999. 11

[34] O. Gotoh. An improved algorithm for matching biological sequences. *J Mol Biol*, 162(3):705–708, December 1982. 2, 15

[35] R. L. Graham. Bounds for certain multiprocessing anomalies. *Bell Labs Technical Journal*, 45(9):1563–1581, 1966. 4, 39, 40, 41, 42

[36] W. Gropp, E. Lusk, N. Doss, and A. Skjellum. A high-performance, portable implementation of the mpi message passing interface standard. *Parallel computing*, 22(6):789–828, 1996. 33

[37] K. Group. Opencl overview - the khronos group inc. `http://www.khronos.org/opencl/`, May 2014. 35

[38] P. E. Hart, N. J. Nilsson, and B. Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE transactions on Systems Science and Cybernetics*, 4(2):100–107, 1968. 19

[39] D. S. Hirschberg. A linear space algorithm for computing maximal common subsequences. *Communications of the ACM*, 18(6):341–343, 1975. 17

[40] D. T. Hoang and D. P. Lopresti. Fpga implementation of systolic sequence alignment. In *International Workshop on Field Programmable Logic and Applications*, pages 183–191. Springer, 1992. 3, 48

[41] F. Ino, Y. Kotani, Y. Munekawa, and K. Hagihara. Harnessing the power of idle gpus for acceleration of biological sequence alignment. *Parallel Processing Letters*, 19(04):513–533, 2009. 4, 59

[42] F. Ino, Y. Munekawa, and K. Hagihara. Sequence homology search using fine grained cycle sharing of idle gpus. *IEEE Transactions on Parallel and Distributed Systems*, 23(4):751–759, 2012. 60

[43] R. K. Jain. *The art of computer systems performance analysis*. John Wiley & Sons, 2008. 77, 78, 82

[44] J. Jeffers and J. Reinders. *Intel Xeon Phi coprocessor high-performance programming*. Newnes, 2013. 29

[45] R. Kaplan, L. Yavits, R. Ginosar, and U. Weiser. A resistive cam processing-in-storage architecture for dna sequence alignment. *IEEE Micro*, 37(4):20–28, 2017. 64

[46] R. Kaplan, L. Yavits, and R. Ginosasr. Bioseal: In-memory biological sequence alignment accelerator for large-scale genomic data. In *SYSTOR*, pages 36–48. ACM, 2020. 64

[47] A. Khalafallah, H. F. Elbabb, O. Mahmoud, and A. Elshamy. Optimizing smith-waterman algorithm on graphics processing unit. In *Computer Technology and Development (ICCTD), 2010 2nd International Conference on*, pages 650–654. IEEE, 2010. 3, 48, 50

[48] J. Kleinberg and E. Tardos. *Algorithm Design*. Pearson, 2005. 44

[49] G. Komaki and V. Kayvanfar. Grey wolf optimizer algorithm for the two-stage assembly flow shop scheduling problem with release time. *Journal of Computational Science*, 8:109–120, 2015. 42

[50] R. E. Korf. Depth-first iterative-deepening: An optimal admissible tree search. *Artificial intelligence*, 27(1):97–109, 1985. 19

[51] M. Korpar and M. Šikić. Sw#–gpu-enabled exact alignments on genome scale. *Bioinformatics*, 29(19):2494–2495, 2013. 3, 5, 21, 48, 54, 55

[52] H. Lan, W. Liu, Y. Liu, and B. Schmidt. Swhybrid: A hybrid-parallel framework for large-scale protein sequence database search. In *Parallel and Distributed Processing Symposium (IPDPS), 2017 IEEE International*, pages 42–51. IEEE, 2017. 4, 67

[53] H. Lan, W. Liu, B. Schmidt, and B. Wang. Accelerating large-scale biological database search on xeon phi-based neo-heterogeneous architectures. In *Bioinformatics and Biomedicine (BIBM), 2015 IEEE International Conference on*, pages 503–510. IEEE, 2015. 4

[54] E. L. Lawler, J. K. Lenstra, A. H. R. Kan, and D. B. Shmoys. Sequencing and scheduling: Algorithms and complexity. *Handbooks in operations research and management science*, 4:445–522, 1993. 41

[55] S. Lee, T. Lee, Y. Noh, and S. Kim. Ranked k-spectrum kernel for comparative and evolutionary comparison of exons, introns, and cpg islands. *IEEE/ACM Trans Comp Biology and Bioinformatics*, early access:1–10, 2019. 1

[56] K. Li and K.-H. Cheng. Job scheduling in a partitionable mesh using a two-dimensional buddy system partitioning scheme. *IEEE Transactions on Parallel and Distributed Systems*, 2(4):413–422, 1991. 44

[57] L. Ligowski and W. Rudnicki. An efficient implementation of smith waterman algorithm on gpu using cuda, for massively parallel scanning of sequence databases. In *Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, pages 1–8. IEEE, 2009. 3, 48, 49

[58] Y. Liu, D. L. Maskell, and B. Schmidt. Cudasw++: optimizing smith-waterman sequence database searches for cuda-enabled graphics processing units. *BMC research notes*, 2(1):73, 2009. 3, 48, 49

[59] Y. Liu and B. Schmidt. Swaphi: Smith-waterman protein database search on xeon phi coprocessors. In *Application-specific Systems, Architectures and Processors (ASAP), 2014 IEEE 25th International Conference On*, pages 184–185. IEEE, 2014. 4, 61

[60] Y. Liu, B. Schmidt, and D. L. Maskell. Cudasw++ 2.0: enhanced smith-waterman protein database search on cuda-enabled gpus based on simt and virtualized simd abstractions. *BMC research notes*, 3(1):93, 2010. 50

[61] Y. Liu, T.-T. Tran, F. Lauenroth, and B. Schmidt. Swaphi-ls: Smith-waterman algorithm on xeon phi coprocessors for long dna sequences. In *Cluster Computing (CLUSTER), 2014 IEEE International Conference on*, pages 257–265. IEEE, 2014. 4, 61

[62] Y. Liu, A. Wirawan, and B. Schmidt. Cudasw++ 3.0: accelerating smith-waterman protein database search by coupling cpu and gpu simd instructions. *BMC bioinformatics*, 14(1):117, 2013. 4, 66, 69

[63] M. Livny and M. Melman. Load balancing in homogeneous broadcast distributed systems. In *Proceedings of the Computer Network Performance Symposium*, pages 47–55, 1982. 46

[64] D. Luebke, M. Harris, N. Govindaraju, A. Lefohn, M. Houston, J. Owens, M. Segal, M. Papakipos, and I. Buck. Gpgpu: General-purpose computation on graphics hardware. In *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*, SC '06, New York, NY, USA, 2006. ACM. 29

[65] N. M. Luscombe, D. Greenbaum, and M. Gerstein. What is bioinformatics? a proposed definition and overview of the field. *Methods of information in medicine*, 40(4):346–358, 2001. 1, 10

[66] S. Maleki, M. Musuvathi, and T. Mytkowicz. Parallelizing dynamic programming through rank convergence. *ACM SIGPLAN Notices*, 49(8):219–232, 2014. 4, 62

[67] S. A. Manavski and G. Valle. Cuda compatible gpu cards as efficient hardware accelerators for smith-waterman sequence alignment. *BMC bioinformatics*, 9(Suppl 2):S10, 2008. 4

[68] C. Y. McLean, P. L. Reno, A. A. Pollen, A. I. Bassan, T. D. Capellini, C. Guenther, V. B. Indjeian, X. Lim, D. B. Menke, B. T. Schaar, et al. Human-specific loss of regulatory dna and the evolution of human-specific traits. *Nature*, 471:216–219, 2011. 1

[69] F. Mendonca. *Multi-purpose Efficient Resource Allocation for Parallel Systems*. PhD thesis, Université de Grenoble, 5 2017. 45

[70] F. M. Mendonca and A. C. M. A. de Melo. Biological sequence comparison on hybrid platforms with dynamic workload adjustment. In *Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), 2013 IEEE 27th International*, pages 501–509. IEEE, 2013. 4, 66, 69

[71] X. Meng and V. Chaudhary. A high-performance heterogeneous computing platform for biological sequence analysis. *IEEE Transactions on Parallel and Distributed Systems*, 21(9):1267–1280, 2010. 4, 65, 69

[72] I. A. Moschakis and H. D. Karatza. Multi-criteria scheduling of bag-of-tasks applications on heterogeneous interlinked clouds with simulated annealing. *Journal of Systems and Software*, 101:1–14, 2015. 43

[73] D. W. Mount. Sequence and genome analysis. *New York: Cold Spring*, 2004. 1, 2, 10, 12

[74] A. W. Mu'alem and D. G. Feitelson. Utilization, predictability, workloads, and user runtime estimates in scheduling the ibm sp2 with backfilling. *IEEE Transactions on Parallel and Distributed Systems*, 12(6):529–543, 2001. 44

[75] Y. Munekawa, F. Ino, and K. Hagihara. Accelerating smith-waterman algorithm for biological database search on cuda-compatible gpus. *IEICE TRANSACTIONS on Information and Systems*, 93(6):1479–1488, 2010. 4

[76] A. Munshi, B. Gaster, T. G. Mattson, and D. Ginsburg. *OpenCL programming guide*. Pearson Education, 2012. 36

[77] E. W. Myers and W. Miller. Optimal alignments in linear space. *Comput. Appl. Biosci.*, 4(1):11–17, March 1988. 2, 17, 18

[78] S. B. Needleman and C. D. Wunsch. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *Journal of molecular biology*, 48(3):443–453, 1970. 2, 13

[79] NVidia. Nvidia cuda architecture. http://developer.download.nvidia.com/compute/cuda/docs/CUDA_Architecture_Overview.pdf, Apr 2009. 33

[80] NVidia. Cuda c programing guide v6.0. http://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf, Feb 2014. 34, 35

[81] NVidia. Cuda c programming guide. *NVidia Corporation*, Feb 2014. 33

[82] NVidia. Paralell thread execution isa - application guide. http://docs.nvidia.com/cuda/pdf/ptx_isa_4.0.pdf, Feb 2014. 34

[83] NVidia. Nvidia tesla v100 gpu architecture. https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf, Aug 2017. 32

[84] NVidia. Nvidia a100 tensor core gpu architecture. https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/nvidia-ampere-architecture-whitepaper.pdf, Jun 2020. 31

[85] NVidia. Cuda c++ programming guide. https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html, Jan 2021. 35

[86] T. F. Oliver, B. Schmidt, and D. L. Maskell. Reconfigurable architectures for bio-sequence database scanning on fpgas. *IEEE Transactions on Circuits and Systems II: Express Briefs*, 52(12):851–855, 2005. 3, 48

[87] C. H. Papadimitriou and K. Steiglitz. *Combinatorial optimization: algorithms and complexity*. Courier Corporation, 1998. 43

[88] D. A. Patterson and J. L. Hennessy. *Computer organization and design: the hardware/software interface*. Elsevier, 2014. 30, 31

[89] S. Rajko and S. Aluru. Space and time optimal parallel sequence alignments. *IEEE Transactions on Parallel and Distributed Systems*, 15(12):1070–1081, 2004. 3, 48

[90] R. V. Rasmussen and M. A. Trick. Round robin scheduling–a survey. *European Journal of Operational Research*, 188(3):617–636, 2008. 43

[91] D. Razmyslovich, G. Marcus, M. Gipp, M. Zapatka, and A. Szillus. Implementation of smith-waterman algorithm in opencl for gpus. In *Parallel and Distributed Methods in Verification, 2010 Ninth International Workshop on, and High Performance Computational Systems Biology, Second International Workshop on*, pages 48–56. IEEE, 2010. 3, 48, 51

[92] K. Reinert, T. H. Dadi, M. Ehrhardt, H. Hauswedell, S. Mehringer, R. Rahn, J. Kim, C. Pockrandt, J. Winkler, E. Siragusa, et al. The seqan c++ template library for efficient sequence analysis: A resource for programmers. *Journal of biotechnology*, 261:157–168, 2017. 121

[93] T. Rognes. Faster smith-waterman database searches with inter-sequence simd parallelisation. *BMC bioinformatics*, 12(1):221, 2011. 4, 60

[94] T. Rognes and E. Seeberg. Six-fold speed-up of smith–waterman sequence database searches using parallel processing on common microprocessors. *Bioinformatics*, 16(8):699–706, 2000. 3, 48

[95] E. Rucci, A. De Giusti, M. Naiouf, G. Botella, C. García, and M. Prieto-Matias. Smith-waterman algorithm on heterogeneous systems: A case study. In *Cluster Computing (CLUSTER), 2014 IEEE International Conference on*, pages 323–330. IEEE, 2014. 4, 69

[96] E. Rucci, C. García, G. Botella, A. De Giusti, M. Naiouf, and M. Prieto-Matías. An energy-aware performance analysis of swimm: Smith–waterman implementation on intel's multicore and manycore architectures. *Concurrency and Computation: Practice and Experience*, 27(18):5517–5537, 2015. 4, 73

[97] E. Rucci, C. Garcia, G. Botella, A. De Giusti, M. Naiouf, and M. Prieto-Matias. Accelerating smith-waterman alignment of long dna sequences with opencl on fpga. In *International Conference on Bioinformatics and Biomedical Engineering*, pages 500–511. Springer, 2017. 3, 48

[98] E. Rucci, C. Garcia, G. Botella, A. E. De Giusti, M. Naiouf, and M. Prieto-Matias. Oswald: Opencl smith–waterman on altera's fpga for large protein databases. *The International Journal of High Performance Computing Applications*, page 1094342016654215, 2016. 4, 67, 69

[99] J. Sanders and E. Kandrot. *CUDA by example: an introduction to general-purpose GPU programming*. Addison-Wesley Professional, 2010. 33

[100] E. F. O. Sandes. Comparação paralela de sequências biológicas longas utilizando unidades de processamento gráfico (gpus). Master's thesis, Universidade de Brasília, 2012. 19

[101] E. F. O. Sandes and A. C. M. A. de Melo. Smith-waterman alignment of huge sequences with gpu in linear space. In *Parallel & Distributed Processing Symposium (IPDPS), 2011 IEEE International*, pages 1199–1211. IEEE, 2011. 53, 54

[102] E. F. O. Sandes and A. C. M. A. de Melo. Retrieving smith-waterman alignments with optimizations for megabase biological sequences using gpu. *Parallel and Distributed Systems, IEEE Transactions on*, 24(5):1009–1021, 2013. 4, 20, 21, 22, 54

[103] E. F. O. Sandes and A. C. M. A. Melo. Cudalign: using gpu to accelerate the comparison of megabase genomic sequences. In *Proceedings of the 15th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 137–146, 2010. 3, 48, 52

[104] E. F. O. Sandes, G. Miranda, X. Martorell, E. Ayguade, G. Teodoro, and A. C. M. 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. 4, 63

[105] E. F. O. Sandes, G. Miranda, X. Martorell, E. Ayguade, G. Teodoro, and A. C. M. A. Melo. Masa: a multiplatform architecture for sequence aligners with block pruning. *ACM Transactions on Parallel Computing (TOPC)*, 2(4):28, 2016. 5, 7, 21, 22, 55, 56

[106] E. F. O. Sandes, G. Miranda, A. C. Melo, X. Martorell, and E. Ayguade. Fine-grain parallel megabase sequence comparison with multiple heterogeneous gpus. In *Proceedings of the 19th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 383–384, 2014. 63

[107] E. F. O. Sandes, G. Miranda, A. C. M. A. Melo, X. Martorell, and E. Ayguade. Cudalign 3.0: Parallel biological sequence comparison in large gpu clusters. *14th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, May 2014. 4, 62, 63

[108] E. F. O. Sandes, C. G. Ralha, and A. C. M. de Melo. An agent-based solution for dynamic multi-node wavefront balancing in biological sequence comparison. *Expert Systems with Applications*, 41(10):4929–4938, 2014. 4, 71, 72, 73

[109] E. F. O. Sandes, G. L. Teodoro, M. E. M. Walter, X. Martorell, E. Ayguade, and A. C. Melo. Formalization of block pruning: Reducing the number of cells computed in exact biological sequence comparison algorithms. *The Computer Journal*, pages 1–27, 2017. 22, 23, 24, 25, 122, 141

[110] S. Schwartz, W. J. Kent, A. Smit, Z. Zhang, R. Baertsch, R. C. Hardison, D. Haussler, and W. Miller. Human–mouse alignments with blastz. *Genome Research*, 13:103–107, 2003. 1

[111] D. B. Shmoys, J. Wein, and D. P. Williamson. Scheduling parallel machines on-line. *SIAM journal on computing*, 24(6):1313–1331, 1995. 46

[112] J. Singh and I. Aruni. Accelerating smith-waterman on heterogeneous cpu-gpu systems. In *Bioinformatics and Biomedical Engineering,(iCBBE) 2011 5th International Conference on*, pages 1–4. IEEE, 2011. 4, 69

[113] T. F. Smith and M. S. Waterman. Identification of common molecular subsequences. *J Mol Biol*, 147(1):195–197, March 1981. 2, 14

[114] A. S. Tanenbaum and N. Machado Filho. *Sistemas operacionais modernos*, volume 3. Prentice-Hall, 1995. 37

[115] A. N. Tantawi and D. Towsley. Optimal static load balancing in distributed computer systems. *Journal of the ACM (JACM)*, 32(2):445–465, 1985. 45

[116] H. Topcuoglu, S. Hariri, and M.-y. Wu. Performance-effective and low-complexity task scheduling for heterogeneous computing. *IEEE transactions on parallel and distributed systems*, 13(3):260–274, 2002. 44

[117] T. Velte, A. Velte, and R. Elsenpeter. *Cloud computing, a practical approach*. McGraw-Hill, Inc., 2009. 64

[118] P. Virouleau, F. Broquedis, T. Gautier, and F. Rastello. Using data dependencies to improve task-based scheduling strategies on numa architectures. In *European Conference on Parallel Processing*, pages 531–544. Springer, 2016. 43

[119] F. Wang, S. Oral, G. Shipman, O. Drokin, T. Wang, and I. Huang. Understanding lustre filesystem internals. *Oak Ridge National Laboratory, National Center for Computational Sciences, Tech. Rep*, 2009. 120

[120] L. Wang, Y. Chan, X. Duan, H. Lan, X. Meng, and W. Liu. Xsw: Accelerating biological database search on xeon phi. In *Parallel & Distributed Processing Symposium Workshops (IPDPSW), 2014 IEEE International*, pages 950–957. IEEE, 2014. 3, 48

[121] Y.-T. Wang et al. Load sharing in distributed systems. *IEEE Transactions on computers*, 100(3):204–217, 1985. 45

[122] M. S. Waterman and M. Eggert. A new algorithm for best subsequence alignments with application to trna-rrna comparisons. *Journal of molecular biology*, 197(4):723–728, 1987. 70

[123] L. Wienbrandt. Bioinformatics applications on the fpga-based high-performance computer rivyera. In *High-Performance Computing Using FPGAs*, pages 81–103. Springer, 2013. 4, 60

[124] L. Wienbrandt. The fpga-based high-performance computer rivyera for applications in bioinformatics. In *Conference on Computability in Europe*, pages 383–392. Springer, 2014. 4, 61

[125] B. Xu, C. Li, H. Zhuang, J. Wang, Q. Wang, and X. Zhou. Efficient distributed smith-waterman algorithm based on apache spark. In *2017 IEEE 10th International Conference on Cloud Computing (CLOUD)*, pages 608–615. IEEE, 2017. 64

[126] A. B. Yoo, M. A. Jette, and M. Grondona. Slurm: Simple linux utility for resource management. In *Workshop on job scheduling strategies for parallel processing*, pages 44–60. Springer, 2003. 8, 133

[127] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, I. Stoica, et al. Spark: Cluster computing with working sets. *HotCloud*, 10(10-10):95, 2010. 64

[128] P. Zhang, G. Tan, and G. R. Gao. Implementation of the smith-waterman algorithm on a reconfigurable supercomputing platform. In *Proceedings of the 1st international workshop on High-performance reconfigurable computing technology and applications: held in conjunction with SC07*, pages 39–48. ACM, 2007. 3, 48

Anexo I

Repositório GitHub Contendo Códigos e *Scripts*

Repositório MultiBP: https://github.com/Marcoacfbr/

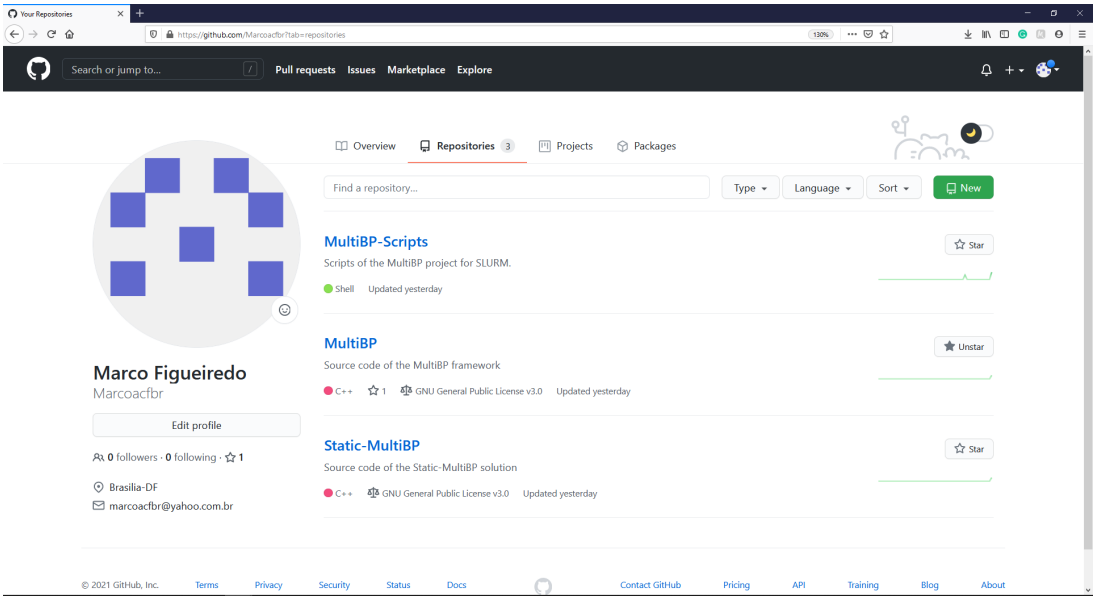


Figura I.1: Repositório GitHub contendo código-fonte das soluções e *scripts* criados para execução do MultiBP em ambiente SLURM.

Anexo II

Artigos Decorrentes desta Tese

II.1 Artigo completo publicado em periódico internacional

MASA-OpenCL: Parallel pruned comparison of long DNA sequences with OpenCL. Concurrency and Computation: Practice and Experience, v. 31, n. 11, p. e5039, 2019 [27].

II.2 Artigo completo publicado em conferência internacional

Parallel Comparison of Huge DNA Sequences in Multiple GPUs with Block Pruning. In: 2020 28th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP). IEEE, 2020. p. 22-29 [28].

II.3 Artigo completo submetido a periódico internacional (em processo de revisão)

Parallel fine-grained comparison of long dna sequences in homogeneous and heterogeneous gpu platforms with pruning. IEEE Transactions on Parallel and Distributed Systems, 2020 [26].

II.4 Primeira página dos artigos

(em ordem de publicação)

Received: 28 October 2017 | Revised: 24 August 2018 | Accepted: 17 September 2018
DOI: 10.1002/cpe.5039

WILEY

RESEARCH ARTICLE

MASA-OpenCL: Parallel pruned comparison of long DNA sequences with OpenCL

Marco Antonio C. de Figueiredo Jr. | **Edans F. de Oliveira Sandes** | **Genaina N. Rodrigues** | **George L. M. Teodoro** | **Alba Cristina M. A. de Melo**

Department of Computer Science, University of Brasilia (UnB), Brasília, DF, Brazil

Correspondence
Marco Antonio C. de Figueiredo Jr., Department of Computer Science, University of Brasilia (UNB), Brasília, DF, Brazil.
Email: marcoacf@aluno.unb.br

Summary

Biological sequence comparison is often used as an auxiliary task in the analysis of genetic material. Pairwise comparison algorithms like Smith-Waterman evaluate two strings representing sequences of proteins, DNA or RNA to obtain optimal alignment between them. Many applications have been proposed to address the sequence comparison problem, prioritizing the use of graphics cards and proprietary languages such as CUDA. In this paper, we propose and evaluate MASA-OpenCL, an OpenCL solution for comparing long DNA sequences that is based on the MASA sequence alignment framework, with pruning capability proportional to the similarity of the sequences compared. The results of MASA-OpenCL were compared to its CUDA counterpart (MASA-CUDAlign) and, in most cases, MASA-OpenCL achieved better performance. In order to better understand the behavior of MASA-OpenCL, we performed a statistical analysis considering 11 comparisons of sequences with high, medium and low similarity in 4 GPUs. As a result, we obtained a multiple linear regression model that considers (a) the sizes of the sequences, (b) the similarity between them, (c) the computational power of the GPU, and (d) the GPU memory bandwidth. We used this model to predict the performance in two other GPUs, with low error rates.

KEYWORDS

OpenCL, pairwise sequence alignment, regression model, Smith-Waterman

1 | INTRODUCTION

The interdisciplinary field Bioinformatics[1] is becoming more relevant each day. In particular, the pairwise sequence comparison operation is a very frequent task, evaluating DNA, RNA, or protein samples to provide a score that represents the similarity between the sequences. Alternatively, an optimal sequence alignment can also be informed as result. An alignment can be global (all the characters of the sequences are represented in the final alignment), local (covering just some part of the sequences), or semi-global (the head/tail of one sequence is not evaluated).

Several algorithms have been proposed to deal with the sequence comparison problem, varying from heuristic approaches (such as BLAST[2]) to exact solutions (like Needleman-Wunsch (NW),[3] Smith-Watterman (SW),[4] Gotoh,[5] or Myers-Miller (MM)[6]). Heuristic algorithms usually execute faster but have worse accuracy than exact methods. The algorithms that provide exact solutions are usually based on Dynamic Programming (DP), calculating a DP matrix to find the optimal score/alignment. Algorithms NW, SW, and Gotoh have $O(nm)$ time and space complexity, where n and m are the sequences lengths.

The challenge presented by exact algorithms resides in the high computational requirements to provide a solution in reasonable time, especially for long DNA sequences that often have thousands (KBP) or millions of Base Pairs (MBP). A better performance can be achieved using parallelism, which often requires High-Performance Computing (HPC) platforms such as Graphics Processing Units (GPUs). The performance of such algorithms is usually measured in billions of cells (or Gigacells) Updated per Second (GCUPS).

Most of GPU sequence comparison approaches proposed in other works[7-12] are based on Compute Unified Device Architecture (CUDA),[13] implementing some SW algorithm variant to compare protein sequences with up to 5,478 amino acids,[11,12] or DNA sequences with up to 249 millions

Parallel Comparison of Huge DNA Sequences in Multiple GPUs with Block Pruning

Marco Figueiredo Jr. — *Univ. of Brasilia* — 160063027@aluno.unb.br
Edans Sandes — *Univ. of Brasilia* — marcoacf@sarah.br edans.sandes@gmail.com
George Teodoro — *Univ. Fed. de Minas Gerais* — george@dcc.ufmg.br
Alba C. M. A. Melo — *Univ. of Brasilia* — alves@unb.br

***Abstract*—Sequence comparison is a task performed in several Bioinformatics applications daily all over the world. Algorithms that retrieve the optimal result have quadratic time complexity, requiring a huge amount of computing power when the sequences compared are long. In order to reduce the execution time, many parallel solutions have been proposed in the literature. Nevertheless, depending on the sizes of the sequences, even those parallel solutions take hours or days to complete. Pruning techniques can significantly improve the performance of the parallel solutions and a few approaches have been proposed to provide pruning capabilities for sequence comparison applications. This paper proposes and evaluates a variant of the block pruning approach that runs in multiple GPUs, in homogeneous or heterogeneous environments. Experimental results obtained with DNA sequences in two testbeds show that significant performance gains are obtained with pruning, compared to its non-pruning counterpart, achieving the impressive performance of 694.8 GCUPS (Billions of Cells Updated per Second) for four GPUs.**

***Index Terms*—bioinformatics, DNA alignment, GPU, pruning**

I. Introduction

Bioinformatics produces solutions that are used by various fields of study, such as medicine and biology [1]. Biological sequence comparison operations are executed several times daily all over the world, either in stand-alone mode or incorporated into Bioinformatics applications to solve complex problems such as evolutionary relationship determination and drug design. Due to their quadratic time complexity, sequence comparison algorithms that retrieve the optimal result can take a lot of time. In order to reduce the execution time of such algorithms, parallel solutions have been proposed in the literature over the last decades.

The type of parallelism provided by Graphical Processor Units (GPUs) makes these devices a very good alternative to run sequence comparisons [2] [3]. CUDAlign 4.0 [3] is a state-of-the art tool that compares huge DNA sequences in multiple GPUs and obtains the optimal result, combining the Gotoh [4] and the Myers-Miller [5] algorithms. Using 384 GPUs, it was able to compare the homologous human x chimpanzee chromosomes 5 (180 Million of Base Pairs – MBP – each) in 53 minutes, computing a matrix of 33.04 Petacells at 10.37 TCUPS (Trillions of Cells Updated per Second). In an earlier version for one GPU (CUDAlign 2.1 [6]), the block pruning (BP) strategy was proposed to avoid the computation of parts of the matrix that surely will not lead to the optimal solution, with good results for one GPU. Further versions of CUDAlign present pruning capabilities only for single GPU executions. SW# [7] implemented the original MM algorithm and extended the block pruning strategy [6] to be used in two GPUs, but the performance was just a little better than the execution of CUDAlign in one GPU [7]. As far as we know, there is no work in the literature that obtains the optimal result with pruning using more than two GPUs. Other works use CPUs [8], FPGAs [9] or hybrid environment [10], but they are outside the scope of the paper.

This paper proposes and evaluates Multi-BP, an adaptation of block pruning for multiple GPUs. It is based on static distribution and dynamic sharing of pruning information, leading to considerable performance gains in medium-sized GPU environments. Multi-BP combines the multi-GPU CUDAlign version [3] and the pruning technique proposed in [6]. The challenges to design Multi-BP were the following: (a) ensure that Multi-BP will not affect the performance in single GPU executions; (b) adapt the calculation of the index of each GPU block of cells and the evaluation of the pruning window to a multiple GPU environment; (c) disseminate the pruning information obtained by each GPU to all others with low overhead; and (d) adjust the pruning technique to the heterogeneous GPU environments, considering that the DP matrix might not be partitioned evenly among the GPUs.

Experimental results obtained with real DNA sequences with sizes varying from 1 to 63 MBP in two computing environments show that very good gains were attained with Multi-BP. The execution time of the comparison of chromosome 20 (human x chimpanzee) in the same heterogeneous environment (GTX 980 Ti + GTX 680) was reduced from 8h17min (without Multi-BP) to 4h55min (with Multi-BP).

The remainder of this paper is organized as follows. In Section II we present the pairwise sequence alignment problem and in Section III we discuss pruning approaches and the block pruning technique. Section IV discusses solutions that execute biological sequence comparisons in multiple GPUs. Section V describes the design of Multi-BP and Section VI details the experiments. Finally, Section VII concludes the paper.

II. Pairwise Biological Sequence Comparison

The field of Bioinformatics [1] demands continuous processing improvements. Due to the huge volume of data and performance requirements, new parallel algorithms and tools are proposed regularly, aiming to provide faster executions. In particular, the alignment of biological sequences (proteins,

Parallel Fine-Grained Comparison of Long DNA Sequences in Homogeneous and Heterogeneous GPU Platforms with Pruning

Marco Figueiredo Jr., Joao Paulo Navarro, Edans Sandes, George Teodoro, and Alba C. M. A. Melo, *Senior Member, IEEE*

Abstract—The parallelization of Smith-Waterman sequence comparison tools for long DNA sequences has been a big challenge over the years, requesting the use of several devices and sophisticated optimizations. Pruning is one of these optimizations, which can reduce considerably the amount of computation. This paper proposes MultiBP, a sequence comparison solution in multiple GPUs with block pruning. Two MultiBP strategies are proposed. In static score-sharing, workload is statically distributed to the GPUs, and the best score is sent to neighbor GPUs to simulate a global view. In the dynamic strategy, execution is divided into cycles and workload is dynamically assigned, according to the GPUs processing rate. MultiBP was integrated to MASA-CUDAlign and tested in homogeneous and heterogeneous platforms, with different NVidia GPU architectures. The best results in our homogeneous and heterogeneous platforms were mostly obtained by the static and dynamic approaches, respectively. We also show that our decision module is able to select the best strategy in most cases. Finally, the comparison of the human and chimpanzee chromosomes 1 in a cluster with 512 V100 NVidia GPUs took 11 minutes and obtained the impressive rate of 82,822 GCUPS which is, to our knowledge, the best performance for SW tools in GPUs.

Index Terms—GPU, sequence comparison, pruning, workload assignment, heterogeneous platforms.

1 Introduction

Biological sequence comparison is one of the most important operations in Bioinformatics. Smith-Waterman (SW) [1] is a popular algorithm that compares two sequences, giving as output the optimal local alignment. Two widely used variants of SW are Gotoh [2], which computes the optimal alignment with the affine gap model, and Myers-Miller (MM) [3], which computes the alignment in linear space. Smith-Waterman and its variants compute a dynamic programming (DP) matrix using recurrence relations, with quadratic time complexity and data dependencies on the left, up, and diagonal neighbor cells.

Particularly, whole chromosome comparisons can be used to help elucidate aspects related to the evolution of species, including human evolution. The comparison of such Megabase sequences is particularly challenging and the use of SW-based tools for this kind of problem is often considered unfeasible. In [4], a thorough study compared the human chromosomes with mammals' homologous chromosomes. Pairwise comparisons were done with BLASTZ, which is a heuristic method that uses SW in some of its steps, since using it in the whole comparison was considered unfeasible by the authors [5]. In [6], a heuristic alignment-free approach is used to compare chromosomes of several species (human, chimpanzee, cow, dog, etc.). The authors state that an alignment-free tool approach was chosen because it was unfeasible to obtain SW alignments, but they acknowledge that SW alignments are more accurate.

The comparison of huge sequences with SW in reasonable time requires sophisticated algorithmic and parallelization techniques applied to multiple devices. SWAPHI-LS [7] and MASA-CUDAlign 4.0 [8] are tools that can compare whole chromosomes with SW variants. With MASA-CUDAlign 4.0, the comparison of the human and chimpanzee homologous chromosomes 1 (249 Millions of Base Pairs - MBP - and 228 MBP, respectively) took about 2 hours and attained 10,370 GCUPS (10.3 TCUPS). This has been, to our knowledge, the best SW performance for GPU-based platforms.

Pruning techniques can be used to improve performance, avoiding the calculation of DP cells that cannot lead to the optimal result. Block pruning was proposed in CUDAlign 2.1 [9] and it is a lightweight pruning technique that does the pruning test by blocks of diagonals. Block Pruning was extended to 2 GPUs [10], but it is not applicable to more GPUs. A theoretical study of Block Pruning [11] showed that the pruning rate can be as high as 66%.

The design of a Block Pruning solution for multiple GPUs is challenging since the pruning shape (i.e., the DP matrix cells which are not computed) is not known before the beginning of the computation and it is obtained while we compute. In addition, dynamic workload assignment for this problem takes a non-negligible time and we must guarantee that the benefits of pruning are not overcome by the overhead of dynamic assignment. In this paper, we propose a Multi-GPU pruning technique called MultiBP, which was integrated to CUDAlign 4.0, and our contributions are: (a) Static-MultiBP: a static strategy where the left GPU periodically sends its current best score to its right neighbor; (b) Dynamic-MultiBP: a dynamic pruning strategy that divides the computation into cycles and assigns workload to each GPU at the end of the cycle, considering the pruning behavior inferred by the GPU's processing rate; and (c) Decision-Maker: a module that can be used to decide which strategy to use. This paper extends the work presented in