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

MASA-OpenCL: Comparação Paralela de Sequências Biológicas Longas em GPU

Marco Antônio Caldas de Figueirêdo Júnior

Brasília
2015

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

MASA-OpenCL: Comparação Paralela de Sequências Biológicas Longas em GPU

Marco Antônio Caldas de Figueirêdo Júnior

Dissertação apresentada como requisito parcial
para conclusão do Mestrado em Informática

Orientadora
Prof.ª Dr.ª Alba Cristina M. A. de Melo

Brasília
2015

Universidade de Brasília — UnB
Instituto de Ciências Exatas
Departamento de Ciência da Computação
Mestrado em Informática

Coordenadora: Prof.ª Dr.ª Alba Cristina M. A. de Melo

Banca examinadora composta por:

Prof.ª Dr.ª Alba Cristina M. A. de Melo (Orientadora) — CIC/UnB
Prof. Dr. Ricardo Pezzuol Jacobi — CIC/UnB
Prof. Dr. Wellington Santos Martins — INF/UFG

CIP — Catalogação Internacional na Publicação

Figueirêdo Júnior, Marco Antônio Caldas de.

MASA-OpenCL: Comparação Paralela de Sequências Biológicas Longas em GPU / Marco Antônio Caldas de Figueirêdo Júnior. Brasília : UnB, 2015.
86 p. : il. ; 29,5 cm.

Dissertação (Mestrado) — Universidade de Brasília, Brasília, 2015.

1. Programação Paralela, 2. Comparação de Sequências Biológicas, 3. Unidades de Processamento Gráfico (GPUs), 4. OpenCL

CDU 004

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

MASA-OpenCL: Comparação Paralela de Sequências Biológicas Longas em GPU

Marco Antônio Caldas de Figueirêdo Júnior

Dissertação apresentada como requisito parcial
para conclusão do Mestrado em Informática

Prof.ª Dr.ª Alba Cristina M. A. de Melo (Orientadora)
CIC/UnB

Prof. Dr. Ricardo Pezzuol Jacobi
CIC/UnB

Prof. Dr. Wellington Santos Martins
INF/UFG

Prof.ª Dr.ª Alba Cristina M. A. de Melo
Coordenadora do Mestrado em Informática

Brasília, 05 de Agosto de 2015

Agradecimentos

Agradeço a Deus pela oportunidade de cumprir mais esta etapa de minha vida.

Agradeço também a meus pais e irmãs, pelo apoio constante e contribuição na formação de minha personalidade. E à minha filha, Gabriela, pela compreensão pelo tempo que tive que dedicar a este trabalho. A família é o alicerce que permite a construção de sonhos maiores.

À minha orientadora, Alba Melo, pelo acompanhamento e suporte sempre precisos, incentivando e indicando a melhor forma de conduzir este trabalho. Aos professores deste Departamento, pela dedicação empregada na árdua tarefa da formação, e aos meus colegas, representados na figura de Edans Sandes, cuja contribuição inestimável foi preponderante para que os resultados fossem atingidos.

Aos amigos queridos, em especial Rafael e Viviane, que me incentivaram e apoiaram neste caminho desde o começo.

E finalmente, mas em especial, à minha esposa Scheila, pelo amor incondicional e pelo carinho dedicado nos momentos de dificuldade. Que esta seja apenas mais uma conquista das nossas vidas.

Resumo

A comparação de sequências biológicas é uma tarefa importante executada com frequência na análise genética de organismos. Algoritmos que realizam este procedimento utilizando um método exato possuem complexidade quadrática de tempo, demandando alto poder computacional e uso de técnicas de paralelização. Muitas soluções têm sido propostas para tratar este problema em GPUs, mas a maioria delas são implementadas em CUDA, restringindo sua execução a GPUs NVidia. Neste trabalho, propomos e avaliamos o MASA-OpenCL, solução desenvolvida em OpenCL capaz de executar a comparação paralela de sequências biológicas em plataformas heterogêneas de computação. O MASA-OpenCL foi testado em diferentes modelos de CPUs e GPUs, avaliando pares de sequências de DNA cujos tamanhos variam entre 10 KBP (milhares de pares de bases) e 47 MBP (milhões de pares de bases), com desempenho superior a outras soluções existentes baseadas em CUDA. A solução obteve um máximo de 179,2 GCUPS (bilhões de células atualizadas por segundo) em uma GPU AMD R9 280X. Até onde temos conhecimento, esta é única solução implementada em OpenCL que realiza a comparação de sequências longas de DNA, e o desempenho alcançado é, até o momento, o melhor já obtido com uma única GPU.

Palavras-chave: Programação Paralela, Comparação de Sequências Biológicas, Unidades de Processamento Gráfico (GPUs), OpenCL

Abstract

The comparison of biological sequences is an important task performed frequently in the genetic analysis of organisms. Algorithms that perform biological comparison using an exact method require quadratic time complexity, demanding high computational power and use of parallelization techniques. Many solutions have been proposed to address this problem on GPUs, but most of them are implemented in CUDA, restricting its execution to NVidia GPUs. In this work, we propose and evaluate MASA-OpenCL, which is developed in OpenCL and capable of performing parallel comparison of biological sequences in heterogeneous computing platforms. The application was tested in different families of CPUs and GPUs, evaluating pairs of DNA sequences whose sizes range between 10 KBP (thousands of base pairs) and 47 MBP (millions of base pairs) with superior performance to other existing solutions based on CUDA. Our solution achieved a maximum of 179.2 GCUPS (billions of cells updated per second) on an AMD R9 280X GPU. As far as we know, this is the only solution implemented in OpenCL that performs long DNA sequence comparison, and the achieved performance is, so far, the best ever obtained on a single GPU.

Keywords: Parallel Programming, Biological Sequence Comparison, Graphical Processor Units (GPUs), OpenCL

Sumário

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

O sequenciamento de organismos vem evoluindo significativamente desde os primeiros trabalhos realizados na década de 1950 [1]. Em especial, o sequenciamento do DNA humano foi alvo de estudo detalhado nos últimos anos, sobretudo considerando-se os desafios enfrentados no Projeto Genoma [2]. Nesse contexto, a análise comparativa entre duas sequências (da mesma espécie ou de espécies diferentes) permite avaliar suas características estruturais, fornecendo subsídios para o desenvolvimento de medicamentos ou novos tratamentos. O grande volume de dados tratados e a necessidade de execução de procedimentos computacionais otimizados para a manipulação das informações contribuíram para o advento da Bioinformática [3], disciplina que integra áreas como Ciência da Computação, Matemática e Biologia com o objetivo de analisar informações biológicas, abrangendo o escopo da Genética.

Dentre as tarefas frequentes realizadas por aplicações em Bioinformática, a comparação de sequências biológicas consiste em calcular um escore que indica a similaridade entre duas sequências de nucleotídeos ou aminoácidos, podendo-se adicionalmente informar a região de similaridade (alinhamento) entre elas [4]. Para tanto, valores são atribuídos para indicar as situações onde ocorrem uma coincidência (*match*) ou divergência (*mismatch*) numa posição de cada sequência, além da penalização caso seja necessário adicionar um espaço (*gap*) em uma das sequências para obter-se um melhor alinhamento. No modelo mais utilizado pelos biólogos, *gaps* consecutivos possuem menor penalização que o primeiro *gap* de uma sequência de espaços, estratégia conhecida como *affine gap model* [5].

1.1 Motivação

Smith-Watterman (SW) [6] é um algoritmo que trata o problema da comparação de duas sequências de tamanhos m e n através de uma solução exata. Consiste na construção de uma matriz de programação dinâmica de tamanho $(m+1)$ x $(n+1)$ e cálculo de uma função de recorrência para obter o escore ótimo de similaridade e o seu respectivo alinhamento. O algoritmo SW possui complexidade de tempo e espaço da ordem $O(mn)$, o que exige alto poder computacional para retornar o resultado da comparação em tempo adequado. Para acelerar a produção de resul-

tados, a adoção de soluções de paralelização em *hardware* e *software* é altamente recomendada.

Os recursos computacionais das placas gráficas (GPUs - *Graphics Processing Units*) vêm sendo muito utilizados na execução de aplicações de propósito geral, devido à possibilidade de uso de muitos núcleos de processamento a custo relativamente mais baixo [7]. A evolução do poder computacional das GPUs pode ser observada na Figura 1.1. No gráfico, foram considerados os picos teóricos de processamento em precisão simples dos modelos com maior desempenho comercializados por cada fabricante entre o final de cada ano e o primeiro trimestre do ano seguinte.

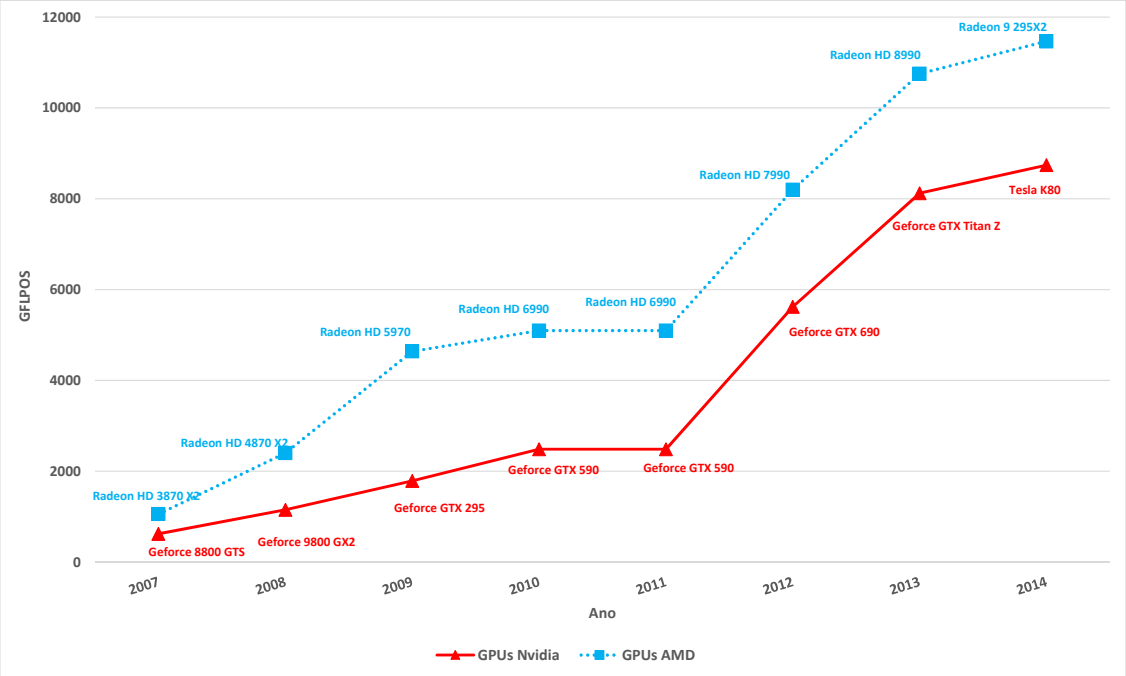


Figura 1.1: Evolução do poder computacional das GPUs (AMD e NVidia)

Como pode ser visto, o desempenho teórico das GPUs teve um enorme crescimento de 2007 a 2014, indo de cerca 500 GFlops (2007) a mais de 11,0 TFlops (2014). É interessante ressaltar também que as GPUs NVidia e AMD possuem desempenho comparável neste período, com uma pequena vantagem para as GPUs AMD. Explorando-se as características do *hardware* com técnicas de paralelismo como o *Single Instruction Multiple Data* (SIMD) combinado ao *Multiple Instruction Multiple Data* (MIMD) obtém-se menores tempos de execução para as aplicações. A programação das aplicações que são executadas em GPUs pode ser realizada em

uma linguagem proprietária de um determinado fabricante de GPU ou em linguagem para plataformas heterogêneas.

Diversas soluções foram propostas nos últimos anos para realizar a comparação ou alinhamento de sequências biológicas em GPU [8] [9] [10] [11] [12] [13] [14] [15] [16] [17] [18]. Apenas algumas delas, contudo, realizam a comparação de sequências genéticas longas [13] [14] [15] [16]. Adicionalmente, a maioria das soluções ([8] [9] [10] [11] [12] [13] [14] [15] [16]) utiliza CUDA na implementação, uma linguagem voltada apenas para GPUs produzidas pela NVidia, não permitindo a execução em plataformas heterogêneas. Os poucos trabalhos que permitem o uso de plataformas heterogêneas usam OpenCL, mas nestes trabalhos são comparadas proteínas [17] ou sequências pequenas de DNA [18].

Em um trabalho recente [19], um *framework* independente de plataforma foi proposto para permitir o desenvolvimento de implementações paralelas que realizam a comparação de sequências em linguagens específicas. Entretanto, não foi proposta nesse trabalho uma implementação que permita a execução em plataformas heterogêneas com OpenCL.

O objetivo desta Dissertação de Mestrado é propor e avaliar uma aplicação paralela que realiza a comparação de sequências longas de DNA baseada em OpenCL, permitindo que a mesma aplicação seja executada em GPUs de diferentes fabricantes, bem como em CPUs ou outros dispositivos. A solução, denominada MASA-OpenCL, baseia-se em uma versão otimizada do algoritmo Smith-Waterman, e é capaz de realizar a comparação de sequências maiores que 30 milhões de pares de bases em diferentes dispositivos.

1.2 Contribuições

As principais contribuições desta Dissertação são as seguintes:

1. Solução eficiente em OpenCL para a execução de comparações paralelas de sequências biológicas longas usando o algoritmo Smith-Waterman em plataformas heterogêneas. Optou-se por utilizar como referência uma solução otimizada já existente desenvolvida em CUDA, adaptando-a para o *framework* OpenCL. Isso permitiu aliar o desempenho original à flexibilidade de execução do mesmo código em diversas plataformas de *hardware* (CPUs e GPUs).

2. Comparação detalhada de desempenho do MASA-OpenCL com soluções em CUDA (GPU) e OpenMP (CPU), permitindo determinar o impacto no desempenho causado pela flexibilidade oferecida pelo OpenCL. Durante os testes com sequências reais, o MASA-OpenCL apresentou desempenho superior em relação às demais soluções testadas, e alguns aspectos do projeto (compilação em 32 bits e uso de memória global para armazenamento das sequências em GPU) puderam ser validadas também nas demais soluções testadas, produzindo melhores desempenhos que as versões originais. Outrossim, obteve-se com o MASA-OpenCL o melhor desempenho reportado para a problema da comparação de sequências longas em uma única GPU.

1.3 Estrutura do Documento

O restante desta Dissertação está organizado da seguinte forma. O Capítulo 2 descreve os conceitos básicos da comparação de sequências biológicas e apresenta alguns algoritmos exatos que tratam o problema. O Capítulo 3 introduz as características das placas gráficas modernas e alternativas para programação em GPUs. O Capítulo 4 apresenta uma revisão bibliográfica de soluções que provêm comparação de sequências em GPU, bem como uma tabela comparativa entre elas. O Capítulo 5 discute os aspectos envolvidos na análise e desenvolvimento da solução MASA-OpenCL, enquanto o Capítulo 6 detalha os testes experimentais e discute os resultados obtidos. Por fim, o Capítulo 7 conclui a Dissertação e lista os trabalhos futuros.

Capítulo 2

Comparação de Sequências Biológicas

O estudo da estrutura dos seres é sem dúvida um aspecto fundamental na Biologia e Genética. Em particular, pode-se destacar a comparação de sequências biológicas como uma das atividades mais relevantes e corriqueiras, buscando-se avaliar e comparar as amostras pesquisadas [4]. A tarefa consiste em comparar duas sequências de nucleotídeos (que compõem o DNA e o RNA) ou de aminoácidos (base de composição das proteínas) buscando a similaridade entre elas [5]. A avaliação deste aspecto permite identificar o nível de semelhança entre as sequências, embasando estudos em áreas tão diversas como a produção de medicamentos ou agricultura transgênica.

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 com macromoléculas biológicas [3]. A execução desta tarefa em tempo adequado demanda alto poder computacional e, por essa razão, seu uso é aliado a algoritmos otimizados e técnicas de paralelização.

No presente capítulo, serão apresentados inicialmente os conceitos básicos de comparação de pares de sequências. A seguir, alguns dos principais algoritmos propostos que produzem uma solução exata ao problema da comparação de sequências serão mostrados. Finalmente, serão mencionados alguns aspectos relativos a estratégias de paralelização destes algoritmos.

2.1 Conceitos Básicos

A comparação de sequências biológicas é uma das operações mais básicas e importantes em Biologia Molecular. Inicialmente, cabe destacar que a tarefa consiste em comparar duas sequências ordenadas representando bases genéticas (Adenina, Citosina, Guanina ou Timina, no caso do DNA) ou de aminoácidos (um conjunto de 20 diferentes substâncias), buscando um alinhamento ótimo [4]. Em cada uma das posições analisadas, pode ocorrer um *match* (quando há uma correspondência entre as bases das duas sequências naquela posição), um *mismatch* (quando há uma divergência) ou ser inserido um *gap* [20] (um espaço, tratado na análise do DNA como sendo a ocorrência de uma deleção) em uma das sequências.

A similaridade entre as sequências é obtida atribuindo-se escores para cada uma das situações citadas no parágrafo anterior, e calculando-se o escore final resultante do somatório dos escores aferidos em cada posição [4]. Os valores escolhidos para cada caso podem variar, mas é comum optar-se por escores positivos para os casos de *match*, e negativos para as situações de *mismatch* e *gaps*. Ademais, visando refletir um cenário que ocorre na natureza, a abertura de novos *gaps* possui penalização maior que a extensão de um *gap* já aberto, modelo denominado *affine gap* [5]. A Figura 2.1 representa um alinhamento simples entre duas sequências pequenas de DNA ($S0$ e $S1$, 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 calculado é igual a $+1$.

```
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 2.1: Exemplo de alinhamento onde o escore final obtido é $+1$

Existem três tipos de alinhamento: global, local e semi-global. 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 as mesmas [21]. Há ainda o alinhamento semi-global, que permite remover somente o prefixo ou o sufixo das sequências a serem alinhadas. Tais formas de alinhamento são aplicadas de duas maneiras principais: 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 de parâmetro possui tamanho restrito [22]. Outra abordagem possível é o alinhamento múltiplo de sequências (*Multiple Sequence Alignment* – MSA) [23], que permite que várias sequências possam ser alinhadas ao mesmo tempo. O MSA está fora do escopo deste trabalho.

A complexidade da comparação de sequências advém do fato que o volume de dados analisados é considerável, em especial na comparação de sequências longas, requerendo alto poder computacional e espaço de armazenamento. Alguns cromossomos humanos, por exemplo, possuem mais de 200 milhões de pares de bases [24], exigindo estratégias que permitam uma solução viável em termos de tempo de processamento e utilização de memória.

2.2 Algoritmos Exatos para Comparação de Sequências

Em termos de algoritmo, a comparação de sequências biológicas pode ser traduzida em um alinhamento entre duas sequências de caracteres, utilizando os alfabetos Σ = {A,T,G,C} para nucleotídeos em DNA, Σ = {A,T,G,U} para nucleotídeos em RNA, e Σ = {A,C,D,E,F,G,H,I,K,L,M,N,P,Q,R,S,T,V,W,Y} para aminoácidos, além de um carácter adicional para representar os *gaps*. A solução consiste em criar uma matriz de programação dinâmica (matriz de similaridade), atribuindo escores a cada um dos cenários de alinhamento possíveis, retornando ao final apenas o escore máximo obtido ou também um ou mais alinhamentos ótimos, que maximizam o escore final. As soluções podem se basear em algoritmos exatos, que buscam sempre um ou mais alinhamentos ótimos, ou heurísticos, que utilizam técnicas que contentam-se com soluções sub-ótimas. Algoritmos deste último grupo não estão contemplados no escopo deste trabalho.

Alguns algoritmos exatos de comparação de sequências biológicas têm sido propostos ao longo dos últimos anos para tratar o problema da comparação de sequências biológicas. A utilização da programação dinâmica é muito frequente entre as soluções propostas, e os desempenhos esperados podem ser comparados através da complexidade de tempo e espaço dessas soluções [25]. Nas subseções 2.2.1 a 2.2.5 alguns destes algoritmos serão detalhados, utilizando algumas definições 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 atribuída a um *gap* constante;
- Em algoritmos que implementam o modelo *affine gap*, G_{first} representa a penalidade atribuída à abertura de um novo *gap* e G_{ext} representa a penalidade da extensão de um *gap* já aberto;
- 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 20x20, chamada matriz de substituição [4].

2.2.1 Needleman-Wunsch - NW

O algoritmo exato de Needleman-Wunsch (NW) [26] 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. Para tanto, inicialmente 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 alinhamento ótimo obtido.

Assumindo os tamanhos das sequências de entrada m e n, a matriz de similaridade A construída através de programação dinâmica terá dimensões $(m+1)$x$(n+1)$, onde cada célula $A_{i,j}$ conterá o escore de melhor alinhamento até aquela posição, ou seja, entre as sub-sequências $S_0[1..i]$ e $S_1[1..j]$. Na inicialização, o valor 0 é intro-

duzido 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 por *gap*.

Para o cálculo de uma posição $A_{i,j}$ ($i, j \neq 0$), três cenários de alinhamento devem ser 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.

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

Durante o preenchimento da matriz, cada célula possui, além do máximo escore calculado até aquela posição, uma referência à posição anterior que gerou este escore. Desta forma, ao final do preenchimento da matriz A, ela pode ser percorrida partindo-se da célula no canto inferior direito e em sentido inverso, seguindo-se os ponteiros armazenados, para retornar o alinhamento obtido. Esse processo é denominado *traceback*. Uma vez que é possível a ocorrência de mais de um caminho possível no *traceback*, mais de um alinhamento ótimo pode ser resultante.

A tarefa de *traceback* não apresenta complexidade de tempo alta, visto que apenas precisa percorrer o alinhamento gerado, sendo proporcional à soma do tamanho das sequências, ou seja, possui um limite superior $O(m+n)$. A complexidade de espaço dessa fase é, no entanto, $O(nm)$. O preenchimento da matriz de similaridade, por outro lado, requer que todas as $(m+1)$x$(n+1)$ posições sejam calculadas, resultando em uma complexidade de tempo e espaço na ordem de $O(nm)$. Na comparação entre duas sequências longas, essa complexidade requer alto poder computacional e grande espaço de memória para que se produza o resultado.

A Figura 2.2 apresenta um exemplo da matriz de similaridade para uma comparação utilizando NW. Os seguintes parâmetros foram adotados: S_0=TAGCTACT, S_1=TAGCTATA, *match*$= +1$, *mismatch*$= -1$, $G = -5$. Os valores dos escores encontram-se no centro da célula. No canto superior direito de cada célula, verifica-se os valores das penalizações atribuídas, enquanto as células com padrão de sombreamento são as que levam ao alinhamento global ótimo, que serão percorridas no *traceback*.

2.2.2 Smith-Waterman - SW

Também baseado em programação dinâmica, o algoritmo de Smith-Waterman (SW) [6] visa identificar alinhamentos locais ótimos. Seu funcionamento é semelhante ao do algoritmo NW (Seção 2.2.1), com algumas modificações que permitem a busca por alinhamentos locais.

Uma das modificações ocorre na definição da equação de recorrência, como pode ser observado na Equação 2.2. Uma entrada com valor zero é adicionada à equação, impedindo que um valor negativo possa ser obtido na determinação do escore

$A_{i,j}$		T	A	G	C	T	A	C	T
	0 0	-5 -5	-10 -5	-15 -5	-20 -5	-25 -5	-30 -5	-35 -5	-40 -5
T	-5 -5	**2** 2	-3 -1	-8 -1	-13 -1	-18 2	-23 -1	-28 -1	-33 -1
A	-10 -5	-3 -1	**4** 2	-1 -1	-6 -1	-11 -1	-16 2	-21 -1	-26 -1
G	-15 -5	-8 -1	-1 -1	**6** 2	1 -5	-4 -1	-9 -1	-14 -1	-19 -1
C	-20 -5	-13 -1	-6 -1	1 -5	**8** 2	3 2	-2 -1	-7 2	-12 -1
T	-25 -5	-18 2	-11 -1	-4 -1	3 2	**10** 2	5 2	0 -1	-5 2
A	-30 -5	-23 -1	-16 2	-9 -1	-2 -1	5 2	**12** 2	7 2	2 -5
T	-35 -5	-28 2	-21 -1	-14 -1	-7 -1	0 2	7 2	**11** -1	9 2
A	-40 -5	-33 -1	-26 2	-19 -1	-12 -1	-5 x	2 2	6 -1	**10** -1

Figura 2.2: Matriz de similaridade utilizando Needleman-Wunsch (NW)

máximo em uma posição. Por consequência, os valores das células na primeira linha e primeira coluna da matriz (posição $A_{i,j}$ onde $i = 0$ ou $j = 0$) também devem ser inicializados com zero. Esta modificação reflete a situação em que, caso seja encontrado um valor negativo ao preencher-se a matriz de similaridade, seria mais indicado começar um novo alinhamento a partir deste ponto, com escore inicial zero.

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

Outra modificação é a execução do *traceback* não necessariamente a partir da célula na posição inferior direita da matriz, mas 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 recuperação do alinhamento deve ser efetuada até uma posição $A_{i,j}$ com valor zero, para que prefixos e sufixos que não levem ao alinhamento local ótimo sejam desconsiderados.

A Figura 2.3 apresenta um exemplo da matriz de similaridade para uma comparação utilizando SW. As sequências de entrada, os valores dos escores e os padrões de preenchimento são os mesmos utilizados na Seção 2.2.1.

A complexidade de processamento e armazenamento desta solução é idêntica ao algoritmo NW - $O(nm)$, o que não é ideal para a comparação de sequências longas. Apesar de permitir que se realize o *traceback* em apenas parte da matriz, no pior caso (quando o valor máximo estiver estiver na última célula da matriz, e o valor zero no início) a complexidade desta tarefa será a mesma do NW.

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

Figura 2.3: Matriz de similaridade utilizando Smith-Waterman (SW)

2.2.3 Gotoh

A principal contribuição do algoritmo de Gotoh [27] é a implementação do modelo *affine gap*, no qual a abertura de novos *gaps* é mais penalizada que a extensão de *gaps* já abertos em uma das sequências. 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 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, uma nova fórmula de recorrência é introduzida, ajustando o algoritmo SW ao modelo de *affine gap*, conforme Equações 2.3, 2.4 e 2.5.

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

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

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

A complexidade de tempo e espaço do algoritmo de Gotoh é também da ordem $O(nm)$, embora utilize o triplo de espaço de armazenamento dos algoritmos NW

(Seção 2.2.1) e SW (Seção 2.2.2), devido à criação de duas matrizes auxiliares com as mesmas dimensões da matriz de similaridade.

2.2.4 Myers-Miller - MM

O algoritmo Myers-Miller (MM) [28] 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 [29], visando obter uma solução 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, determinando-se em cada iteração um ponto médio para a subsequência mais longa (linha ou coluna). Com isso, é realizada uma operação para se obter o ponto médio do alinhamento ótimo contido na linha ou coluna 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 da matriz até este mesmo ponto, em sentido reverso. Neste estágio, o procedimento é chamado a partir deste ponto recursivamente para as duas matrizes resultantes, até a obtenção de problemas triviais. A Figura 2.4 representa um esquema de obtenção do ponto médio ótimo (representado pelas coordenadas $i*$ e $j*$) na matriz de similaridade.

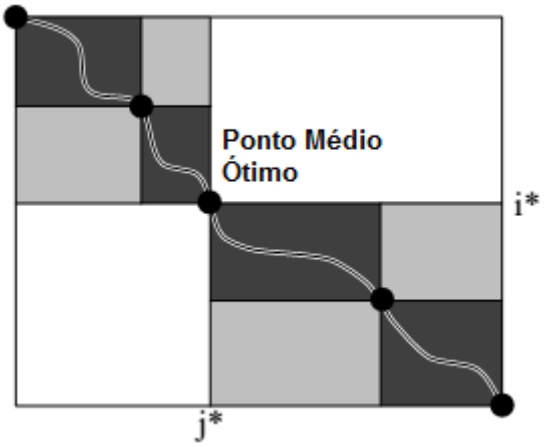


Figura 2.4: Ponto médio ótimo em matriz de similaridade segundo Myers-Miller - adaptado de [28]

A solução proposta por Myers-Miller permitiu a comparação de duas sequências de 62.500 bases utilizando apenas 1 MB de memória [28].

2.2.5 Fickett

Na abordagem de Fickett [30], uma otimização foi proposta para melhorar o desempenho do processamento da matriz de similaridade. Foram realizadas modificações no algoritmo NW (Seção 2.2.1), baseando-se na observação de que sequências muito

similares possuem poucos *gaps*, e portanto o alinhamento ótimo deverá se localizar nas vizinhanças da diagonal principal da matriz de similaridade, como mostrado na Figura 2.5.

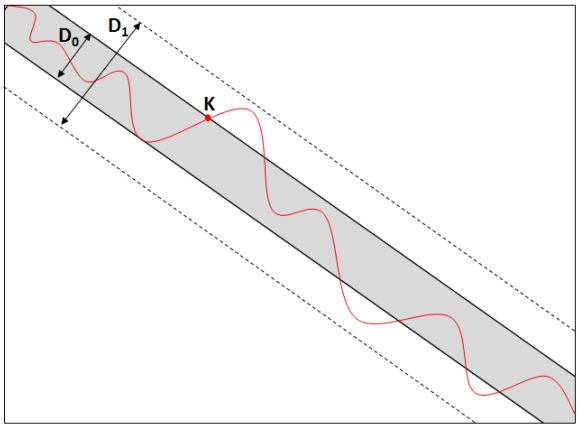


Figura 2.5: Esquema de funcionamento do algoritmo Fickett [31]

Diferentemente das abordagens anteriores, o estudo foi realizado baseando-se no conceito de distâncias de edição, representadas pelas penalizações nos casos de *gaps* e *mismatches*. Desta forma, busca-se a mínima distância em uma matriz d para determinar a sequência mais similar. Para acelerar o processamento, um limite D é estabelecido, e apenas as células d_{ij} com escores menores a este valor D são calculadas, evitando que todos os $(m+1)$x$(n+1)$ elementos da matriz sejam processados.

Para que a matriz d seja preenchida após a determinação do limite D, o primeiro passo é o cálculo dos elementos $d_{1,1}, d_{1,2}, ..., d_{1,L_1}$, de forma que L_1 é o menor índice tal que $d_{1,L_1} \geq D$. Da mesma forma, calculam-se $d_{2,1}, d_{2,2}, ..., d_{2,L_2}$, de forma que L_2 é o menor índice tal que $d_{2,L_2} \geq D$ e $L_2 > L_1$. O processo se repete até que se processem todas as linhas da matriz.

Caso o limite D_0 inicial não permita que o alinhamento ótimo seja encontrado, um novo limite D_1 deve ser escolhido, e as colunas anteriores a K_i e posteriores L_i devem ser calculadas. O processo do incremento do limite D pode se repetir até que o alinhamento seja obtido. A Figura 2.5 mostra um esquema de funcionamento da solução de Fickett: a área na cor cinza mostra a faixa obtida utilizando o limite D_0, e a faixa delimitada pelas linhas tracejadas mostra a faixa expandida com limite D_1 após ter sido alcançado o ponto K que faz parte do alinhamento (representado pela linha curva). Toda a área restante da matriz não é calculada, melhorando o desempenho.

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 - quanto menor número de *gaps*, melhor o desempenho do algoritmo. Espaços de memória adicionais são requeridos para armazenar os valores de K_i e L_i em cada linha, mas que não impactam na complexidade de espaço.

2.3 Abordagens Paralelas

A exigência de alto poder computacional para execução dos algoritmos de comparação exata demanda a utilização de técnicas que possibilitem respostas em tempos adequados. O processamento paralelo [32], por exemplo, que permite que várias operações sejam realizadas simultaneamente, é altamente utilizado. Em especial, a facilidade de tratar múltiplos dados com uma única instrução - definida como arquitetura *Single Instruction Multiple Data* (SIMD) segundo a taxonomia de Fynn [33] - é bastante indicada, permitindo boa paralelização de cálculos envolvendo matrizes.

A paralelização dos algoritmos exatos de alinhamento de sequências se concentra geralmente no cálculo das células da matriz de similaridade, tarefa com maior custo computacional. As equações de recorrências dos algoritmos apresentados na Seção 2.2 possuem a mesma dependência de dados, onde cada posição (i, j) 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* [34], simulando a propagação de uma "onda" em uma determinada direção. Isto permite que a matriz seja percorrida de forma diagonal, a partir do elemento no canto superior esquerdo. Desta forma, as células que compõem uma mesma anti-diagonal podem ser obtidas de forma paralela, devido à independência de dados. A Figura 2.6 ilustra este comportamento, onde células com padrões semelhantes na figura podem ser processadas em paralelo.

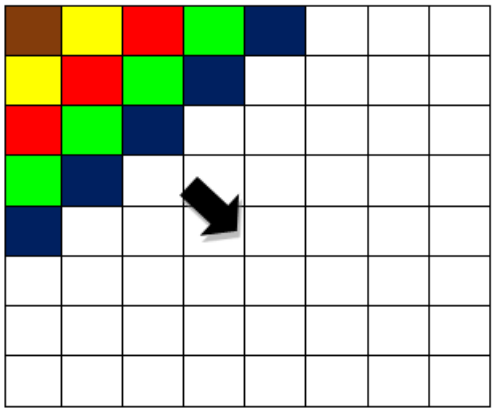


Figura 2.6: Processamento em *wavefront* diagonal de matriz 8x8

2.4 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 o grande volume de dados, a métrica mais usada para medição de desempenho para aplicações de comparação

de sequências biológicas é expressa em milhões de células atualizadas por segundo (MCUPS - *Mega Cells Updated per Second*) ou em bilhões de células atualizadas por segundo (GCUPS - *Giga Cells Updated per Second*).

A métrica CUPS é obtida dividindo-se a quantidade de células da matriz (resultante do produto entre os tamanhos das sequências comparadas) pelo tempo de execução total. 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, o tempo t é multiplicado por 10^9 na Equação 2.6.

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

Capítulo 3

Graphical Processing Unit - GPU

A placa gráfica é certamente um dos componentes com significativa mudança tecnológica na evolução dos computadores. Em 1981, a IBM criou as primeiras placas de vídeo para computadores pessoais [35], com *display* monocromático e com área de 9x14 *pixels*. O padrão seguinte, *Color Graphics Adapter* (CGA), já permitia 16 cores, mas ainda com baixas resoluções.

Os padrões que se sucederam - *Enhanced Graphics Adapter* (EGA), *Video Graphics Array* (VGA) e *Super Video Graphics Array* (SVGA) - apresentaram evoluções na resolução e quantidade de cores oferecidas, e já permitiam alguma programação [36]. A partir da década de 1990, o poder de processamento e a possibilidade de programação das placas gráficas se intensificaram, e o termo *Graphical Processing Unit* (GPU) se consolidou na representação deste novo paradigma.

O crescimento do número de processadores e quantidade/arquitetura de memória disponível nas placas gráficas dedicadas atuais permitem que elas sejam utilizadas para execução de programas não relacionados ao processamento de vídeo, funcionando como unidades de propósito geral, ou *General Purpose Graphics Processing Unit* (GPGPU) [7]. Entre os fabricantes das modernas GPUs, AMD e NVidia se destacam, permitindo utilização do seu alto poder computacional a custos comparativamente mais baixos que as *Central Processing Units* (CPUs). O presente capítulo apresenta inicialmente algumas arquiteturas das GPUs NVidia e AMD. A seguir, discorre-se sobre a programação de placas gráficas com *Compute Unified Device Architecture* (CUDA) e *Open Computing Language* (OpenCL).

3.1 Principais Arquiteturas de GPU

Mesmo considerando-se a arquitetura peculiar, vale ressaltar que as modernas GPUs foram projetadas primordialmente para processamento e exibição de gráficos. Para tanto, componentes foram desenhados para execução de tarefas com esta finalidade, tais como *vertex*, *geometry*, *pixel processing* e *shader*. Com a evolução, as unidades de processamento se tornaram menos especializadas, convertendo-se em unidades lógicas e aritméticas para operações de ponto flutuante. Desta forma, qualquer das funções gráficas pode ser executada por alguma unidade de processamento, e a alta capacidade de processamento pode ser utilizada via programação para resolução de problemas genéricos.

Apesar de compartilhar conceitos similares, a arquitetura das GPUs varia entre os principais fabricantes. As 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 [37].

3.1.1 Arquiteturas AMD/ATI

A *Array Technologies Incorporated* (ATI) se destacou inicialmente no mercado de placas gráficas desenvolvendo soluções integradas para outros fabricantes de computadores, como a IBM, por exemplo. A partir do final da década de 1980, a empresa passou a atuar como fabricante independente, oferecendo soluções de processamento gráfico de alto desempenho não apenas para estações de trabalho e servidores, mas também para o mercado de consoles de jogos. Em 2006, a empresa foi adquirida pela *Advanced Micro Devices* (AMD), e as GPUs mais recentes passaram a ser comercializadas com esta marca.

A AMD HD 6900 [38] é uma placa GPU recente que foi apresentada em 2011, contemplando o processamento não só de aplicações gráficas para computadores, mas também programas de propósitos gerais. É uma arquitetura *Very Large Instruction Word* (VLIW) que inclui *array* para processamento de dados paralelos (DPP, na sigla em inglês), um processador de comandos e um controlador de acesso à memória. A memória não pode ser gravada diretamente por uma aplicação executada no *host*, exigindo que a requisição passe pelo mecanismo de acesso direto à memória para que o dado seja copiado da memória da CPU para a área da GPU. O DPP é organizado como um conjunto de *pipelines* de unidades computacionais, independentes entre si, com capacidade de processar paralelamente dados inteiros ou em ponto flutuante, além de interface individual com a memória. Dependendo do tipo de aplicação, diferentes recursos são alocados, sendo os programas de propósito geral processados pelo *Compute Shader*. Baseada nesta arquitetura, a placa Radeon HD 6990 dual possui 3.072 núcleos de processamento (*cores*), alcançando 5,1 TFlops de desempenho em precisão simples e cerca de 1,3 TFlops em precisão dupla.

A arquitetura *Graphics Core Next* (GCN) sucedeu a arquitetura VLIW, já sendo classificada como uma arquitetura *Reduced Instruction Set Computer* (RISC). As placas baseadas na série AMD Southern Islands Series [39] são baseadas na arquitetura GCN, trazendo melhorias no desempenho e acessibilidade aos recursos computacionais da GPU, com redução no consumo de energia. Conta com barramento de memória de até 384 bits, e desempenho de 8,2 TFlops de processamento de precisão simples e 1,9 TFlops de processamento em precisão dupla no modelo Radeon HD 7990 dual. A arquitetura permite a distribuição de *threads* dentro de unidades computacionais do *array*, cada uma delas contendo lógica de instruções, unidades lógicas/aritméticas escalares e vetoriais com registradores, memória compartilhada e *cache* L1, além de estarem ligadas externamente a um banco de *cache* L2 e a um controlador de memória. Uma aplicação que solicita processamento à GPU é organizada em uma série de comandos, que são divididos em blocos de 64 *threads* processados pelo *array* em *wavefront* [39].

A série sucessora, AMD Sea Islands Series [40] foi lançada em 2013 e possui arquitetura apresentada na Figura 3.1. Como pode ser visto, dados são transferidos da memória do dispositivo ou do *host* através e um controlador de memória (*memory controller*), que gerencia uma arquitetura de *caches* L1 e L2 compartilhadas pelo *array* que estrutura os *cores*. Em relação à série anterior, a Sea Islands Series introduziu melhorias no processamento multi-filas (permitindo até oito *pipelines*, contendo até oito filas cada), endereçamento unificado de sistema e dispositivos, além de gerenciamento de endereços de memória acessados. Exemplo desta arquitetura, o modelo Radeon HD 8990 dual possui mais de 4.000 transistores por placa, com desempenho de 10,7 TFlops de processamento de precisão simples e 2,0 TFlops de precisão dupla.

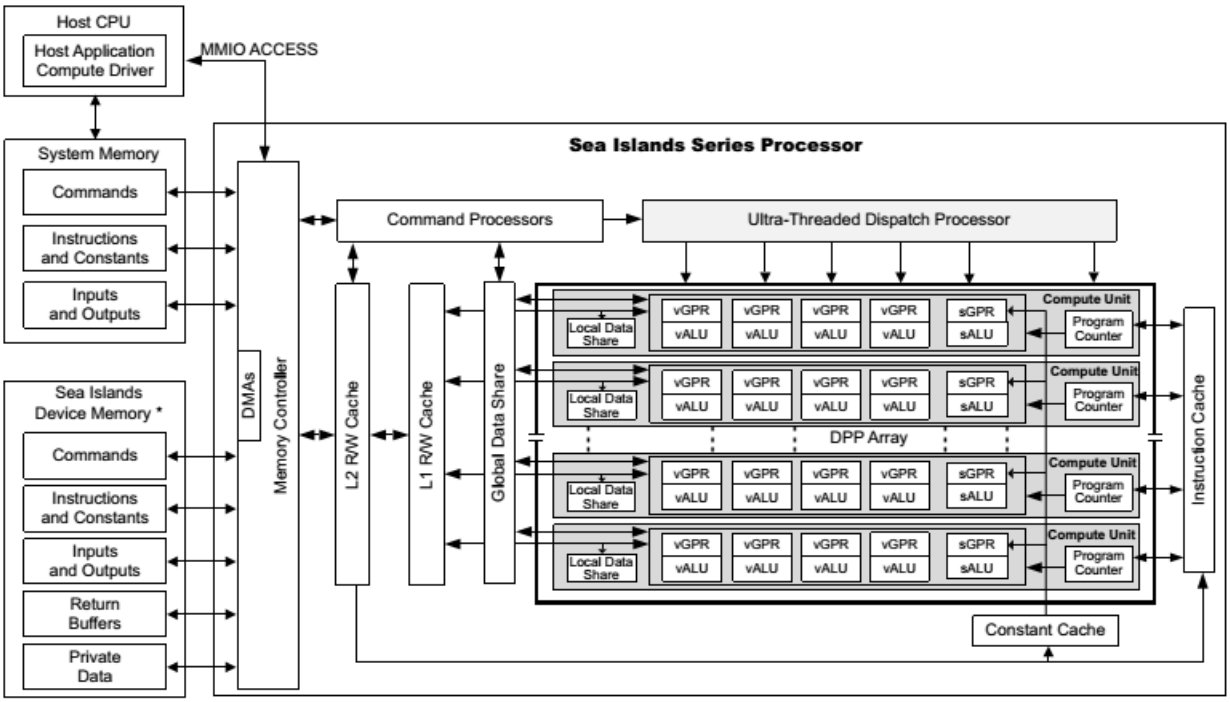


Figura 3.1: Diagrama de blocos da arquitetura AMD Sea Islands [40]

Introduzida nos modelos lançados no segundo semestre de 2013, a série AMD Volcanic Islands [41] é voltada principalmente para o processamento gráfico de jogos em *desktops* e consoles. Apesar de não possuir mudanças significativas em relação à arquitetura GCN utilizada anteriormente, o alto poder de processamento destas GPUs permite que os modelos com esta arquitetura sejam boas alternativas para a execução de aplicações GPGPU. O modelo Radeon R9 295X2 (dual) possui duas GPUs contendo 2.816 *cores*, atingindo cerca de 11,5 TFlops de processamento de precisão simples e 1,4 TFlops de precisão simples.

O modelo Radeon R9 280X, que também pertence à arquitetura Volcanic Islands mas não é uma placa dual, possui 2.048 *cores* e 128 unidades de textura, atingindo mais de 4,1 TFlops de processamento precisão simples e 1,0 TFlops de precisão dupla. O grande número de *cores* permite que muitas *threads* possam ser executadas

simultaneamente, aumentando o paralelismo. A Figura 3.2 ilustra a configuração de uma placa deste modelo, obtida pelo programa GPU-Z. Como pode ser visto, essa GPU (com codinome Tahiti) possui 3 GB de memória DDR5, e barramento (*bus width*) de 384 bits. Os 2.048 núcleos são identificados pelo programa GPU-Z no campo *Shaders*.

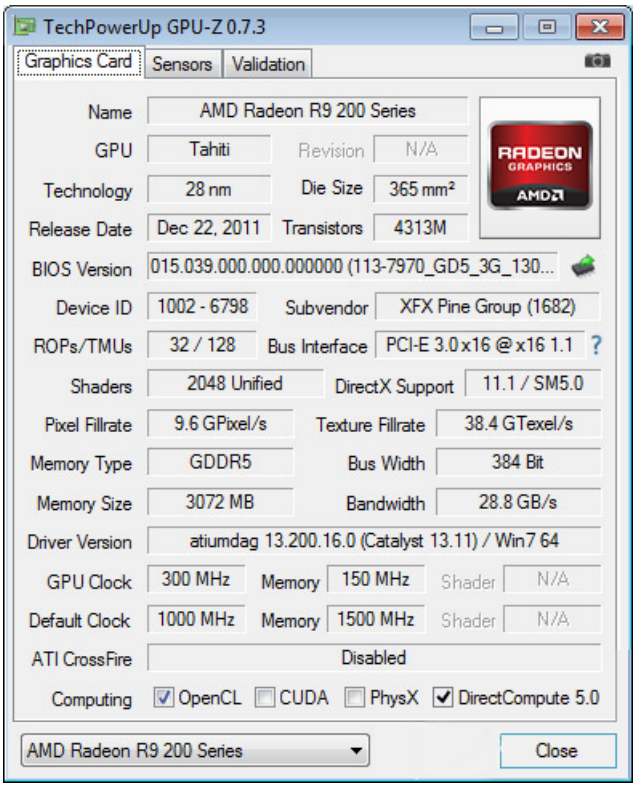


Figura 3.2: Informações coletadas pelo programa GPU-Z para placa R9 280X

3.1.2 Arquiteturas NVidia

A *NVidia Corporation*, empresa americana fundada em 1993, tem se notabilizado nos últimos anos como uma das líderes no mercado de placas gráficas, provendo soluções para consoles de jogos, estações de trabalho e processamento gráfico de alto desempenho. As arquiteturas recentes têm apresentado bons resultados no processamento de aplicações GPGPU, seja utilizando linguagem de programação proprietária ou multiplataforma.

Além de componentes que possibilitam a comunicação placa gráfica/*host* e de distribuição de trabalho entre os componentes, as GPUs NVidia atuais possuem um *array* de processadores denominado *Texture Processing Cluster* (TPC), que acessa áreas de memória (DRAM). Tipicamente, cada um dos TPCs possui um ou mais *Streaming Multiprocessors* (SM), contendo área de *cache* de instruções (*Instruction Cache* - I-Cache), área de *cache* de constantes (*Constant Cache* - C-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) [42]. Um esquema ilustrativo da arquitetura de um TPC pode ser visto na Figura 3.3.

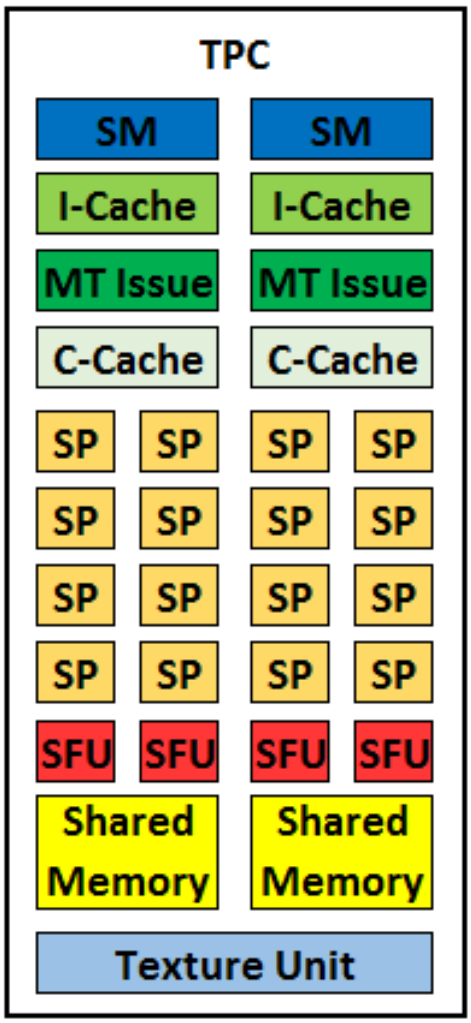


Figura 3.3: Arquitetura de TPC contendo 2 SMs, 16 SPs e 4 SFUs [42]

A arquitetura Tesla [43] foi introduzida em 2006, em conjunto com sua plataforma de desenvolvimento - CUDA. Já era possível nesta arquitetura um alto poder de processamento paralelo, contendo no mínimo 128 núcleos e desempenho de 518 GFlops na GPU utilizada em estações de trabalho, modelo C870. Recentemente, alguns aceleradores foram comercializados com esta mesma nomenclatura [44], embora utilizem outras arquiteturas.

A arquitetura Fermi [45] consolidou a estratégia iniciada pela NVidia com a arquitetura G80, ao apresentar soluções que contemplassem de forma unificada tanto o processamento gráfico como o processamento paralelo de propósito geral. A arquitetura possui 32 núcleos por SM, avançada hierarquia de *cache* L1 e L2, além de permitir execução de aplicações *kernel* concorrentes. Os modelos GTX 460 e GTX

470, lançados em 2010, utilizam esta arquitetura. Também da arquitetura Fermi, o modelo GTX 580 possui 32 SPs e 4 SFUs em cada um dos seus 16 SMs, atingindo desempenho superior a 1,5 TFlops de processamento de precisão simples.

Com o advento da arquitetura Kepler [46], o paralelismo na execução de instruções se tornou dinâmico, passando a GPU a ter maior capacidade de gerenciar o trabalho e sincronizar resultados sem envolver a CPU, além de facilitar a programação. A conexão de interface entre o *host* e a GPU passa a contar com 32 conexões simultâneas e gerenciadas por *hardware*. Adicionalmente, uma nova funcionalidade permite que várias GPUs conectadas em rede possam trocar dados sem requerer acesso à memória da CPU. Estas características podem ser encontradas, por exemplo, nos modelos Geforce GTX 670 e GTX 680. Este último modelo atinge cerca de 3,1 TFlops no processamento gráfico de precisão simples, a partir dos seus 1.536 núcleos. A Geforce GTX Titan Black, modelo com apenas uma GPU mais recentemente disponibilizado, também possui arquitetura Kepler.

A Tabela 3.1 apresenta algumas características de placas representativas das arquiteturas citadas, mostrando como a evolução nos componentes tem contribuído para o desempenho das GPUs.

Arquitetura	**Modelo**	**Núcleos**	**Memória (GB)**	**GFlops Precisão Simples**	**GFlops Precisão Dupla**
Tesla	C870	128	1,5	518	43
Fermi	GTX 460	336	1,0	907	75
Fermi	GTX 470	448	1,2	1.088	136
Fermi	GTX 580	512	1,5	1.581	197
Kepler	GTX 670	1.344	2,0	2.460	102
Kepler	GTX 680	1.536	2,0	3.090	129
Kepler	GTX Titan	2.880	6,0	5.121	1.881

Tabela 3.1: Comparação entre algumas placas NVidia

No final do ano 2014, a Nvidia lançou a sua nova arquitetura (Maxwell), presente por exemplo nas placas GTX 980 e 970. De acordo com publicações da NVidia, as próximas arquiteturas da empresa se denominam Pascal e Volta. Esta última, em particular, possui uma inovação na tecnologia de memória, possuindo módulos empilhados verticalmente sobre o mesmo substrato da GPU, possibilitando um barramento de 1 TB/s [47].

3.2 Programação em GPUs

A arquitetura altamente paralelizável das modernas GPUs torna-se um atrativo para a utilização deste recurso na execução de programas de propósito geral que demandem elevado poder computacional. O conceito de GPGPU se consolidou nos últimos anos, e pesquisadores vêm se dedicando não apenas ao estudo de aplicações, mas também em como tornar este ambiente mais eficiente [48] [49] [50].

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) [51], desenvolvida pela NVidia. A vantagem desta abordagem é a possibilidade de programação explorando mais diretamente os recursos da GPU, aproximando-se da arquitetura do *hardware*. Por outro lado, soluções muito específicas podem requerer novo esforço de codificação caso um modelo de GPU traga mudanças de arquitetura, como será o caso da futura arquitetura Volta da NVidia (Seção 3.1.2).

Uma alternativa a esta decisão de projeto é a adoção de *frameworks* de programação paralela para ambientes heterogêneos, como por exemplo *Open Computing Language* (OpenCL) [52] ou o OmpSs [53]. No caso do OpenCL, os algoritmos são implementados em uma linguagem não proprietária, e o código será compilado em tempo de execução no dispositivo final, tornando-o portável para diversas plataformas com mínima interferência. O que se questiona neste caso é se esta solução terá desempenho comparável ao de uma solução desenvolvida especificamente para uma dada plataforma.

A ATI (atualmente AMD) possuía uma plataforma de desenvolvimento específica para suas placas, chamada ATI Stream [54]. Entretanto, observou-se que a empresa abandonou esta estratégia, tendo recentemente optado por disponibilizar ferramentas de desenvolvimento baseadas em OpenCL [55].

3.2.1 *Compute Unified Device Architecture* - CUDA

A arquitetura de *hardware* e *software* CUDA foi desenvolvida pela NVidia em 2006 para possibilitar a implementação de aplicações GPGPU em suas placas gráficas [56]. Os componentes da arquitetura podem ser observados na Figura 3.4.

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

A camada subsequente 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), que por sua vez possuem um componente adicional que funciona como uma máquina virtual, portando um código gerado para diferentes modelos de GPUs. Um programador pode desenvolver aplicações diretamente sobre o PTX [58], com a utilização da linguagem CUDA C [59].

Como alternativa, a arquitetura oferece *drivers* que permitem integração com outras ferramentas de desenvolvimento, tais como o OpenCL e o DirectX. Ademais, a aplicação pode ser desenvolvida em outras linguagens (como Java, Python, Fortran ou C++) utilizando funções CUDA C, e rodar sobre a plataforma de execução (*runtime*) disponível na arquitetura.

Quanto ao modelo de programação, CUDA C estende a linguagem C, possibilitando ao programador definir funções (denominadas *kernels*) para execução em uma quantidade definida de *threads* em paralelo. Cada umas das *threads* recebe

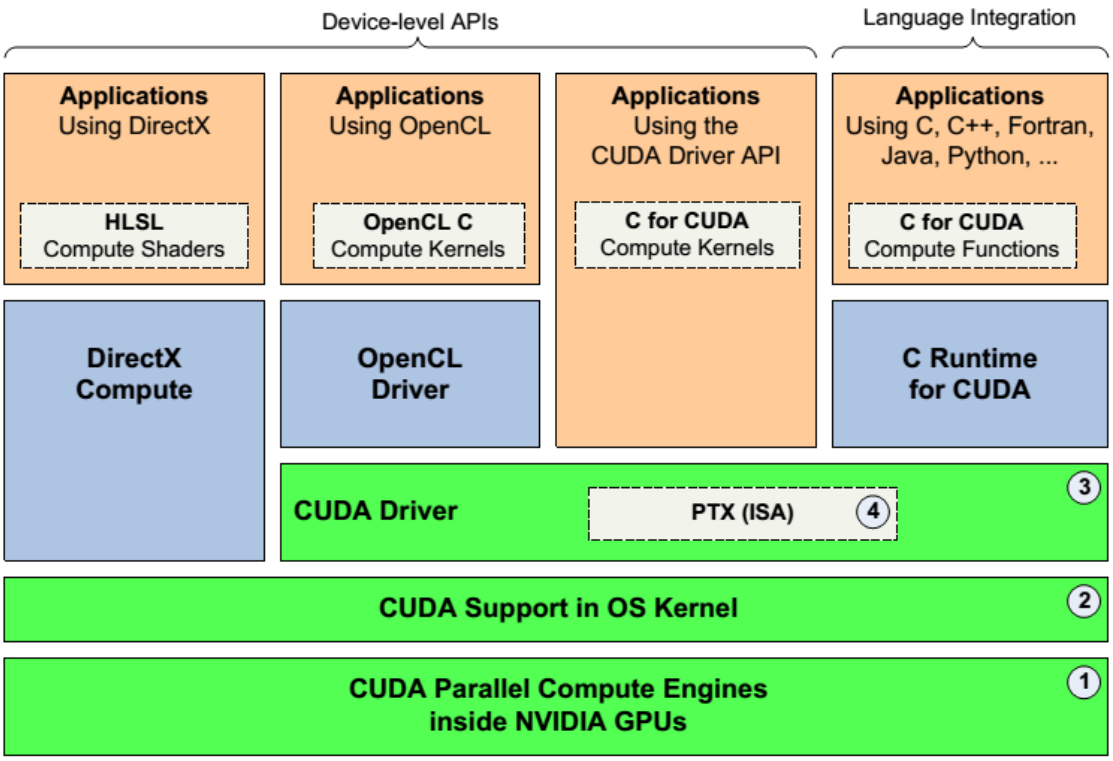


Figura 3.4: Modelagem dos componentes da arquitetura CUDA [57]

um identificador único (`threadIdx`), que pode ser referenciado dentro do *kernel*. O *kernel* é composto por um vetor de até três dimensões, compondo um bloco. Todas as *threads* de um mesmo bloco devem residir no mesmo processador e compartilhar a mesma memória, o que impõe uma restrição ao número de *threads* por bloco [59].

A arquitetura de memória disponível em CUDA oferece diferentes localidades e formas de acesso às *threads* [59], a saber:

- **Local**: acesso restrito apenas à *thread* onde está localizada;
- **Compartilhada**: possibilita a comunicação rápida entre *threads* de um mesmo bloco;
- **Global**: memória de acesso permitido a todos os blocos e *threads*;
- **Textura**: utilizada caso as *threads* possuam padrão comum de acesso à memória, permitindo um acesso acelerado a uma memória global de apenas leitura;
- **Constante**: memória somente para leitura de pequena capacidade que pode ser lida rapidamente por todas as *threads*.

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

[59] representa de forma resumida o código CUDA C para soma de duas matrizes (A e B) de dimensões NxN, com resultado armazenado na matriz C. A função paralelizada é definida nas linhas 2 a 7, e chamada na linha 17.

Programa 3.1 Exemplo de código em CUDA C [59]

```
 1: // Definição do Kernel que executa a soma, utilizando modificador __global__
 2: __global__ voidMatAdd(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: intmain()
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 *Open Computing Language* - OpenCL

OpenCL é um *framework* para programação paralela em ambiente heterogêneo desenvolvido pelo Khronos Group [60]. A solução permite que um mesmo código-fonte possa ser executado em diferentes plataformas, possibilitando que o desenvolvedor defina explicitamente a plataforma, o contexto e a distribuição do trabalho no *hardware* disponível, explorando a concorrência na execução de instruções através de 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 melhor 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 [61], onde o *host* (arquitetura alvo da aplicação) 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. A Figura 3.5 representa o modelo de plataforma.

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. Durante a execução, cada instância de *kernel* é identificada em um espaço de endereçamento como um *work-item*, que são orga-

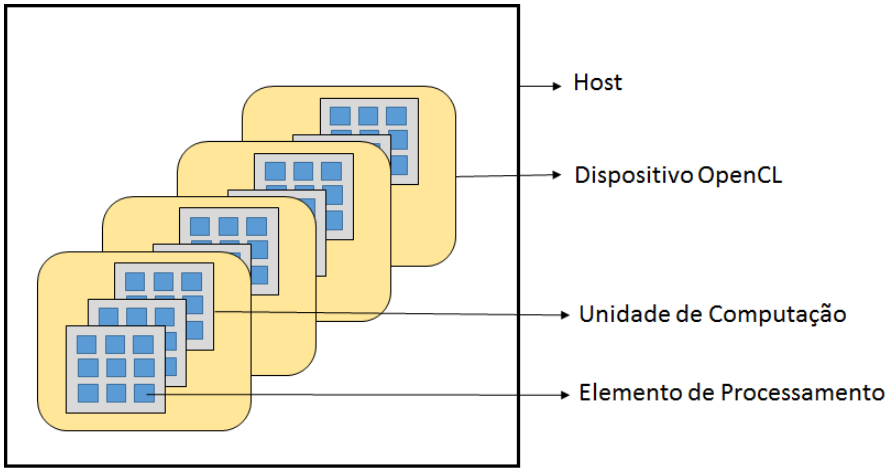


Figura 3.5: Modelo de plataforma do OpenCL [61]

nizados em um grupo que garante a execução concorrente, denominado de *work-group*. Adicionalmente, no *host* é definido um contexto, que representa um ambiente (dispositivos, *kernels*, objetos de programa e de memória) onde a aplicação será executada. Para permitir a execução em ambiente heterogêneo, o programa-objeto é compilado a partir do fonte em tempo de execução, com base no contexto escolhido. Finalmente, para permitir a execução de *kernels* em elementos de processamento, comandos de transferência de memória e comandos de sincronização são usados, bem como uma fila (*command-queue*) é implementada para controlar o fluxo de execução, permitindo processamento de comandos em ordem e fora de ordem [61].

O *framework* OpenCL também define um modelo de memória, com dois tipos de objetos: objetos *buffer* (bloco contíguo de memória acessível aos programadores a partir de ponteiros) e objetos de imagem (restrito para armazenamento de imagens, que podem ser manipuladas através de funções específicas). Cinco regiões de memória são definidas [61]:

- ***Host***: visível apenas para o *host*;
- **Global**: permite acesso de leitura e escrita a todos os *work-items* dentro de todos os *work-groups*;
- **Constante**: região da memória global que permanece constante durante a execução de um *kernel*, sendo acessíveis apenas para leitura pelos *work-items*;
- **Local**: região local a um *work-group*, podendo ser utilizada para alocar variáveis que são compartilhadas pelos *work-items*;
- **Privada**: região privada a um *work-item* específico.

Na definição do modelo de programação, o OpenCL permite ao programador adequar a execução ao algoritmo projetado, permitindo maior flexibilidade. Os dois modelos propostos são: o paralelismo de dados, provido tanto entre *work-items*

dentro de um *work-group* como entre diferentes *work-groups*; e o paralelismo de tarefas, permitido, por exemplo, quando *kernels* são submetidos à execução concorrente fora de ordem. Devido às características do *framework*, algumas limitações podem ser impostas no projeto da solução para adequar o algoritmo aos modelos de programação existentes.

A Figura 3.6 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:

- ◇ Inicialmente, realiza-se uma consulta ao *host* sobre plataformas disponíveis;
- ◇ Dentre as plataformas retornadas, o *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 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.

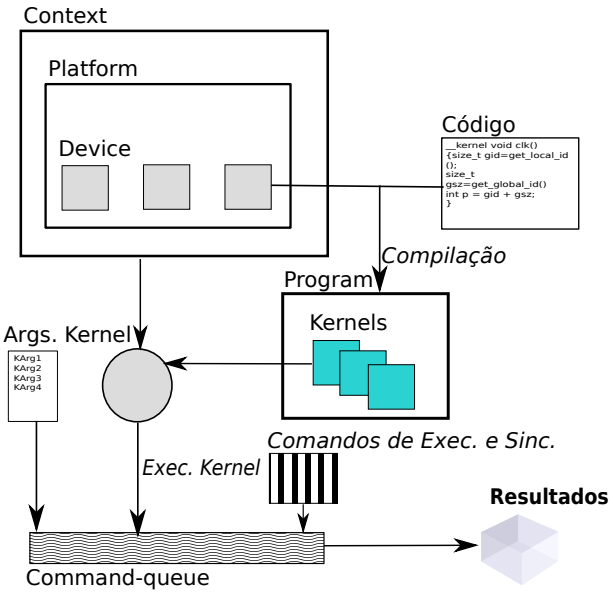


Figura 3.6: Esquema da execução de programa OpenCL em um dispositivo

As instruções utilizadas para implementar as etapas da execução de um programa em OpenCL podem ser observadas na listagem de trecho do programa 3.2, que executa o programa `convolve`, cujo código encontra-se no arquivo `Convolution.cl`. Instruções adicionais e de tratamento de erro foram omitidas para simplificar o entendimento.

Observando o código, na linha 2 uma consulta é realizada ao *host*, que retorna a quantidade de plataformas disponíveis. Depois da alocação dinâmica de memória (linha 5), os *ids* existentes são consultados e armazenados na linha 6. A quantidade de plataformas é utilizada como parâmetro no *loop* na linha 10, contendo instruções para que se identifique um dispositivo do tipo GPU, conforme observado nas linhas 11 a 21. As instruções 24 a 27 são utilizadas para criar um contexto de acordo com algumas propriedades definidas. Nas linhas 30 a 32, o arquivo que contém o código fonte que será compilado e executado é aberto, e o descritor armazenado serve como parâmetro para a criação do programa (linha 35), que é compilado (linha 38) para criação do *kernel* (linha 41). As instruções restantes (linhas 44 a 58) são utilizadas para o acesso à memória, criação da fila de execução e enfileiramento do código compilado para execução paralela.

Diferentemente da plataforma CUDA, que pode gerar códigos otimizados para as arquiteturas da NVidia, o OpenCL deve produzir um código genérico o suficiente para permitir a portabilidade em várias arquiteturas, o que pode impactar o desempenho. Contudo, apesar das diferenças em algumas nomenclaturas utilizadas e no paradigma de programação, pode-se identificar a equivalência nos termos adotados entre as duas plataformas, como ilustrado na Tabela 3.2.

Item	**CUDA**	**OpenCL**
Terminologia	*Thread*	*Work Item*
	Thread Block	*Work Group*
Área de memória	*Global*	*Global*
	Constant	*Constant*
	Shared	*Local*
	Local	*Private*
Qualificadores de funções	`_global_`	`_kernel`
	`_device_`	Não necessário
Qualificadores de variáveis	`_constant_`	`_constant`
	`_device_`	`_global`
	`_shared_`	`_local`
Indexadores	`gridDim`	`get_num_groups()`
	`blockDim`	`get_local_size()`
	`blockIdx`	`get_group_id()`
	`threadIdx`	`get_group_id()`
	`blockIdx * blockDim + threadIdx`	`get_global_id()`
	`blockDim * gridDim`	`get_global_size()`
Sincronização	`__syncthreads()`	`barrier()`

Tabela 3.2: Comparação entre termos CUDA e OpenCL [59] [62]

Programa 3.2 Exemplo de código em OpenCL [61]

```
// Consulta plataformas e armazena na variável "numPlatforms"
errNum = clGetPlatformIDs(0, NULL, &numPlatforms);

// Aloca espaço e armazena "ids" de plataformas
platformIDs = (cl_platform_id *)alloca(sizeof(cl_platform_id) * numPlatforms);
errNum = clGetPlatformIDs(numPlatforms, platformIDs, NULL);
deviceIDs = NULL;

// Para cada plataforma, pesquisa por dispositivos em GPU
for (i = 0; i < numPlatforms; i++)
 { errNum = clGetDeviceIDs(platformIDs[i],CL_DEVICE_TYPE_GPU,0,NULL,&numDevices);
   if (errNum != CL_SUCCESS && errNum != CL_DEVICE_NOT_FOUND)
    { checkErr(errNum, "clGetDeviceIDs");
    }
   else if (numDevices > 0)
    { deviceIDs = (cl_device_id *)alloca(sizeof(cl_device_id) * numDevices);
      errNum = clGetDeviceIDs(platformIDs[0], CL_DEVICE_TYPE_GPU, numDevices,&deviceIDs[0], NULL);
      checkErr(errNum, "clGetDeviceIDs");
      break;
    }
 }

// Seta propriedades do contexto
cl_context_properties contextProperties[] =
  { CL_CONTEXT_PLATFORM,(cl_context_properties)platformIDs[i], 0 };

// Cria contexto
context = clCreateContext(contextProperties, numDevices, deviceIDs,&contextCallback, NULL, &errNum);

// Abre arquivo "Convolution.cl" com código a ser executado no kernel
std::ifstream srcFile("Convolution.cl");
std::string srcProg(std::istreambuf_iterator<char>(srcFile),(std::istreambuf_iterator<char>()));
const char * src = srcProg.c_str();

// Cria programa a partir do código existente no arquivo com descritor "src"
program = clCreateProgramWithSource(context, 1, &src, &length, &errNum);

// Compila programa
errNum = clBuildProgram(program, numDevices, deviceIDs, NULL, NULL, NULL);

// Cria kernel
kernel = clCreateKernel(program, "convolve", &errNum);

// Cria buffer
maskBuffer = clCreateBuffer(context, CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,sizeof(cl_uint) *
    maskHeight * maskWidth,static_cast<void *>(mask), &errNum);

// Cria a command queue
queue = clCreateCommandQueue(context, deviceIDs[0], 0, &errNum);

// Seta argumentos do kernel
errNum = clSetKernelArg(kernel, 0, sizeof(cl_mem), &inputSignalBuffer);
errNum |= clSetKernelArg(kernel, 1, sizeof(cl_mem), &maskBuffer);
errNum |= clSetKernelArg(kernel, 2, sizeof(cl_mem), &outputSignalBuffer);
errNum |= clSetKernelArg(kernel, 3, sizeof(cl_uint), &inputSignalWidth);
errNum |= clSetKernelArg(kernel, 4, sizeof(cl_uint), &maskWidth);

// Enfileira kernel para execução
errNum = clEnqueueNDRangeKernel(queue,kernel,1,NULL,globalWorkSize, localWorkSize,0,NULL,NULL);
errNum = clEnqueueReadBuffer(queue, outputSignalBuffer, CL_TRUE, 0,sizeof(cl_uint)
    * outputSignalHeight * outputSignalHeight,outputSignal, 0, NULL, NULL);
```

Capítulo 4

Comparação Paralela de Sequências Biológicas em GPU

A tarefa de comparação paralela de sequências pode exigir um alto poder computacional, a fim de produzir resultados em tempo aceitável, o que demanda, além de algoritmos otimizados, uma arquitetura de *hardware* com alto poder de paralelização. Vários trabalhos vêm propondo soluções em diferentes arquiteturas, seja utilizando plataformas CPU *multicore* [63] [64] [65] ou plataformas híbridas [66] [67] [68], buscando utilizar os recursos computacionais para paralelização do processamento.

Nas últimas décadas, entretanto, as modernas GPUs (Capítulo 3) vêm sendo muito utilizadas para a execução de algoritmos paralelos de comparação de sequências biológicas, aproveitando sua arquitetura peculiar e custos comparativamente mais reduzidos. As Seções 4.1 a 4.8 apresentam os trabalhos mais relevantes que tratam o problema nessa plataforma. O desempenho obtido por cada solução está expresso em GCUPS (Seção 2.4).

4.1 SW-CUDA

SW-CUDA [8] é uma das primeiras soluções que utilizam CUDA (Seção 3.2.1) na implementação do algoritmo SW (Seção 2.2.2), mas já incorporando o modelo de *affine gap* (Gotoh - Seção 2.2.3). A solução foi comparada com soluções de alinhamento de sequências em outras plataformas, apresentando desempenho compatível. Os testes foram realizados utilizando duas placas NVidia Geforce 8800 GTX, comparando poucas sequências de aminoácidos (no máximo, 511 resíduos) contra o banco de dados de proteínas Swiss-Prot 51.3, que possuía mais de 250.000 proteínas.

O algoritmo provê um acesso sequencial à matriz de substituição (Seção 2.2) para explorar a área de *cache* da memória de textura da GPU. Adicionalmente, múltiplas *threads* (450 blocos, com 64 *threads* cada) são executadas paralelamente, sendo cada uma delas responsável por computar o alinhamento completo da sequência buscada com uma sequência existente na base de dados, ordenadas em função do tamanho. Não está claro no artigo qual o tipo de resultado produzido, mas pelas comparações propostas subtende-se que apenas o escore é retornado.

Os testes foram realizados executando em apenas uma e nas duas placas existentes, obtendo resultado duas vezes melhor no segundo cenário. O melhor desempenho foi obtido comparando-se uma sequência de tamanho 127 contra a base de dados selecionada, obtendo-se 3,6 GCUPS.

4.2 Ligowski e Rudnicki

Com a proposta de melhorar o desempenho da comparação de sequências em GPU, a solução de Ligowski e Rudnicki [9] também implementa o algoritmo de Gotoh (Seção 2.2.3) em plataforma CUDA. Os testes foram realizados na placa gráfica NVidia Geforce 9800 GX2, que possui duas GPUs integradas e é capaz de executar 768 *threads* concorrentemente.

O algoritmo proposto visa minimizar a comunicação com a memória principal, executando concorrentemente parte em CPU (apenas o laço de controle) e parte em GPU de forma paralelizada, realizando a busca de uma sequência de aminoácidos em um banco de dados de referência. As sequências do banco de dados são ordenadas previamente pelo tamanho (maiores sequências primeiro) e organizadas em blocos. A sequência a ser buscada (*query*) é comparada através de *kernels* CUDA com as sequências do banco de dados, retornando o maior escore de alinhamento produzido por cada sequência. O alinhamento produzido pelas sequências com maior escore pode ser obtido por um algoritmo executando na CPU, não fazendo parte da análise dos resultados.

Os testes foram realizados utilizando um subconjunto da base de dados 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, atingindo 14,5 GCUPS. Os autores atribuem o resultado próximo ao limite teórico devido ao acesso à memória principal apenas na inicialização do laço, além da utilização de 4.096 *threads*.

4.3 CUDASW++

O projeto CUDASW++ implementa o algoritmo SW com *affine gap* (Seção 2.2.3) em GPUs para comparação de proteínas. Na versão 1.0 [10], a paralelização é obtida através da criação de várias tarefas, e em cada uma delas a sequência de referência é comparada com uma das sequências do banco de dados. A paralelização é otimizada através de dois métodos: inter-tarefas e intra-tarefas.

Na metodologia inter-tarefas, 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 mais rápida e voltada a sequências menores.

Na metologia intra-tarefas, 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 2.3), o que permite a comparação de sequências maiores. Otimizações também foram realizadas na forma de acesso à memória global, agrupando as *threads* de acordo com a arquitetura da GPU.

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

A versão 2.0 do CUDASW++ [11] apresenta duas implementações do algoritmo SW com *affine gap*: uma otimização do modelo inter-tarefas e uma variação do padrão de processamento em tiras. Na primeira abordagem, duas modificações foram realizadas no algoritmo inicial: *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 dados; 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 [64], particionando as sequências e armazenando-as em área de textura.

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 inter-tarefas otimizada; e 17,8 GCUPS (GTX 280) e 29,9 GCUPS (GTX 295), para a variante do modelo *striped*.

Em CUDASW++ 3.0 [12], os autores propõem uma solução de processamento híbrido para a comparação de sequências de proteínas contra um subconjunto da base de dados Swiss-Prot 2012_11, acoplando instruções *Single Instruction Multiple Data* (SIMD) de GPU e CPU. Foi implementada também a distribuição estática da carga de trabalho da comparação de sequências entre as plataformas, de acordo com a capacidade de processamento. Em testes com apenas uma GPU, a solução obteve 83,3 GCUPS executando em placa NVidia GTX 680. Melhores resultados foram obtidos quando executando em plataforma híbrida, o que está fora do escopo de avaliação desta dissertação.

4.4 CUDAlign

A comparação de duas sequências longas de DNA através de uma variante do algoritmo SW com o modelo de *affine gap* (Seção 2.2.3) executando em GPU utilizando CUDA é a principal contribuição do CUDAlign 1.0 [13]. A solução produz o escore final ótimo e as coordenadas do alinhamento ótimo na matriz de similaridade. Para tanto, as células da matriz de programação dinâmica (A) são agrupadas em blocos, que são processados diagonalmente e ajustados de forma que a quantidade de blocos executados concorrentemente (B) e o número de *threads* executados por bloco (T) possam ser configurados de acordo com a arquitetura da GPU utilizada. Considerando duas sequências de tamanhos m e n, o tamanho de cada bloco é calculado definindo-se dois parâmetros, R e C, tais que: $C = \frac{n}{B}$ e $R = \alpha * T$, sendo α

uma constante que define a quantidade de linhas da matriz que são processadas por cada *thread*, o que otimiza o processamento da matriz. Tais blocos são então agrupados em um *grid* (G) contendo $\frac{m}{R}$x$\frac{n}{C}$ blocos, permitindo que a matriz A seja processada explorando ao máximo o paralelismo.

Adicionalmente, algumas técnicas foram propostas para otimizar a execução do algoritmo. Na delegação de células, 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* configuradas, num aprimoramento do processamento em *wavefront* (Seção 2.3). Além disso, uma subdivisão da fase de cálculo das matrizes de programação dinâmica é realizada para forçar uma sincronização no processamento dos blocos, evitando assim que uma dependência de dados gere uma leitura incorreta. Finalmente, a utilização dos registradores e áreas de memória global e compartilhada da GPU é otimizada na solução de acordo com o uso esperado no algoritmo, permitindo que longas sequências possam ser comparadas mesmo com placas gráficas com menos recursos.

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

O CUDAlign 2.0 [69] recupera o alinhamento ótimo entre duas sequências longas de DNA com um algoritmo que combina Gotoh (Seção 2.2.3) e Myers-Miller (Seção 2.2.4). Executa-se em seis estágios, sendo os três inciais em GPU e os três finais em CPU. O primeiro estágio equivale ao CUDAlign 1.0 modificado para salvar algumas linhas da matriz de similaridade em disco.

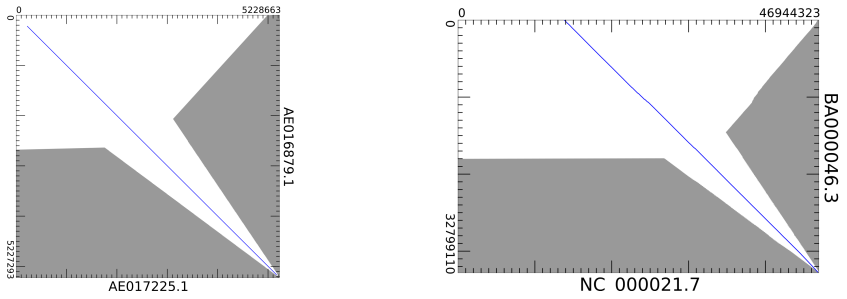
Na versão CUDAlign 2.1 [14], os autores utilizaram como base a versão 2.0 [69] da solução, e adicionalmente propuseram uma técnica de descarte de blocos (*Block Pruning* - BP). A otimização é baseada na observação matemática que, em algum ponto do processamento da matriz de programação dinâmica, algumas células não são capazes de produzir um escore final melhor que o maior escore atual, desta forma elas podem ser descartadas. Para se avaliar cada célula, a Equação 4.1 é utilizada, calculando-se o máximo escore que pode ser obtido a partir de uma célula em uma determinada posição ($H_{max}(i,j)$), baseando-se no seu escore atual ($H(i,j)$) e o escore incremental ($H_{inc}(i,j)$) que pode ser alcançado caso todas as células seguintes tenham casos de *match*. Caso este valor seja menor ou igual ao máximo escore já obtido naquele momento, a célula pode ser descartada.

$$H_{max}(i,j) = H(i,j) + H_{inc}(i,j) \tag{4.1}$$

Este conceito foi extrapolado para blocos de células, avaliando-se as células com coordenadas no canto superior esquerdo e inferior direito de cada bloco. Desta forma, todas as células dentro de um determinado bloco identificado podem ser descartadas ao mesmo tempo.

Esta técnica melhora significativamente o desempenho do algoritmo, principalmente se duas sequências bem similares forem comparadas, o que permitiu um

ganho de desempenho de mais de 50% em relação à mesma execução sem o descarte de blocos, como pode ser observado na Figura 4.1.



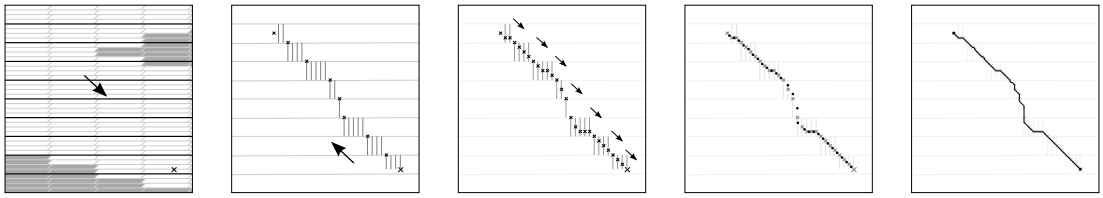
(a) (b)

Figura 4.1: Matriz de programação dinâmica em duas comparações de DNA com CUDAlign 2.1 [14]: (a) 5 MBP x 5 MBP e (b) 33 MBP x 47 MBP. A área cinza (53,7% e 48,1%, respectivamente) representa os blocos descartados. A linha diagonal indica o alinhamento ótimo obtido.

Na versão 2.1, o CUDAlign também é executado em seis estágios, tal como na versão 2.0, conforme ilustrado na Figura 4.2. As tarefas executadas em cada estágio podem são detalhadas a seguir:

- No estágio 1, as matrizes de programação dinâmica são processadas usando a abordagem *wavefront* em forma de paralelogramo e dados de colunas especiais são armazenados em disco. Blocos que não podem gerar um escore maior que o corrente em um dado momento são descartados (técnica de *Block Pruning*), sendo gerados como saída o escore ótimo e as suas coordenadas;
- No estágio seguinte, um alinhamento semi-global é realizado no sentido reverso de forma otimizada a partir do ponto onde o escore máximo ocorre, usando as colunas especiais salvas em disco. É obtida uma lista de coordenadas dos pontos finais do alinhamento ótimo, denominadas *crosspoints*;
- No estágio 3, mais *crosspoints* são obtidos, com a diferença que partições são definidas com pontos de início e fim do alinhamento;
- A seguir, o algoritmo MM (Seção 2.2.4) 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;
- A concatenação dos resultados obtidos em cada partição anterior para obter o alinhamento final é o objetivo do estágio 5, sendo os resultados gerados em formato binário;
- Finalmente, uma visualização da representação binária do alinhamento pode ser obtida (opcionalmente) no estágio 6.

Os testes do CUDAlign 2.1 foram realizados num ambiente com GPU NVidia GTX 560 Ti, obtendo melhor resultado na comparação de duas sequências contendo 33 e 47 milhões de pares de bases, atingindo 52,8 GCUPS.



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

Figura 4.2: Estágios de execução do CUDAlign 2.1 [14]

Na versão 3.0 [15], a solução foi adaptada para permitir a execução do estágio 1 do CUDAlign 2.1 em múltiplas GPUs. Para tanto, 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 e duas de comunicação. A comunicação é realizada através de *sockets* internos caso as GPUs estejam no mesmo *host*, ou através de *sockets* TCP pela rede de interconexão caso as GPUs residam em *hosts* diferentes. *Buffers* circulares utilizados tanto como entrada como saída são responsáveis por armazenar os dados das colunas da matriz de programação dinâmica antes e após o processamento em uma GPU. Adicionalmente, os autores sugerem uma fórmula de predição do tempo de execução da solução de acordo com dados das GPUs alocadas, o que é útil no processo de alocação de recursos. Por executar em múltiplas GPUs, os resultados obtidos por esta versão estão fora do escopo desta dissertação.

4.4.1 *Framework* MASA

O principal objetivo do *framework Multi-Platform Architecture for Sequence Aligners* (MASA) [19] 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 baseado em uma solução de processamento paralelo, permitindo a implementação de soluções específicas.

O processamento em seis estágios proposto pelo CUDAlign 2.1 foi utilizado como base para o desenvolvimento do MASA. Para tanto, o código foi dividido em 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 funções estatísticas; e as funções dependentes de plataforma, contendo a estratégia de processamento paralelo da matriz de programação dinâmica e o módulo de descarte de blocos (BP), que devem ser implementadas de acordo com a plataforma selecionada. A integração entre estes módulos pode ser observada na Figura 4.3.

Como estratégia de paralelização, duas abordagens são propostas: o método diagonal, onde o processamento inicia-se no canto superior esquerdo da matriz e propaga-se diagonalmente, permitindo o paralelismo no processamento das células de um mesmo *wavefront*; e o método de fluxo de dados, propagando-se de forma genérica entre nós que representam blocos de células. De forma similar, a técnica

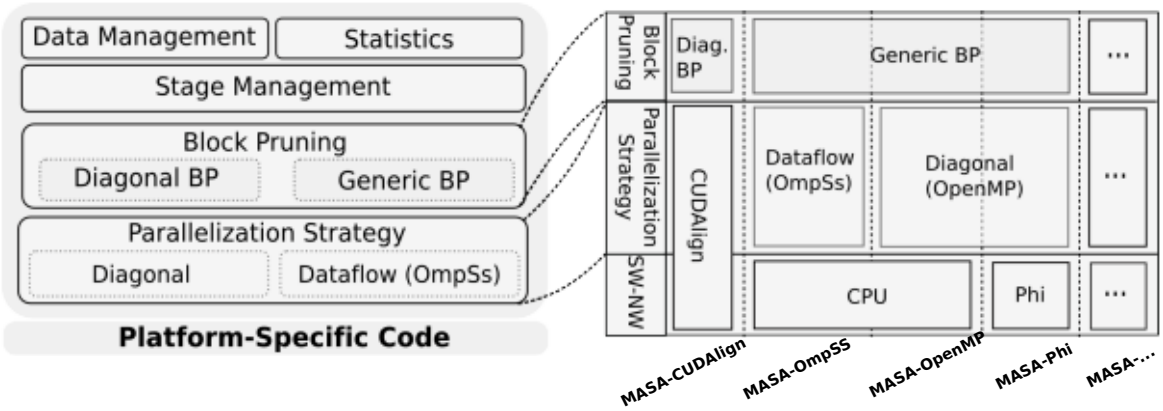


Figura 4.3: Arquitetura MASA [19]

de descarte de blocos também é proposta para execução de forma diagonal ou genérica, evitando o cálculo desnecessário de blocos que não podem contribuir para o escore ótimo.

Para a criação de uma extensão específica, 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 soluções em CUDA (derivada do CUDAlign), OpenMP [70] (para *multicore* CPU e Intel Phi [71]) e OmpSs [72], confirmando a flexibilidade do *framework* MASA. Entretanto, as extensões desenvolvidas ainda são voltadas para uma determinada arquitetura, não tendo sido apresentada uma solução que possa ser executada independente da plataforma.

Comparações envolvendo sequências de tamanhos variando entre 10 KBP e 47 MBP foram utilizadas nos testes das extensões propostas. Exceto pelas menores sequências testadas (que foram processadas mais rapidamente pela solução MASA-OmpSs), o MASA-CUDAlign apresentou o melhor desempenho, chegando a 56,16 GCUPS no processamento dos seis estágios da comparação envolvendo duas sequências de 10 MBP (57,43 GCUPS no processamento do primeiro estágio), executando em GPU Nvidia Tesla M2090.

4.5 SW#

A solução SW# [16] é desenhada em três fases: resolução, localização e reconstrução. Na fase de resolução, utiliza as mesmas técnicas apresentadas no CUDAlign 2.1 (Seção 4.4) para execução do algoritmo MM (Seção 2.2.4) na comparação de sequências longas de DNA em plataforma CUDA. Como otimização, os autores sugerem a divisão desta fase em dois subproblemas após a execução da primeira fase do algoritmo MM, sendo obtidos o ponto e o escore médio da matriz. Isto permite que o problema possa ser distribuído para duas placas gráficas neste ponto

de execução, acelerando a localização dos pontos finais de alinhamento. Na fase de localização, o algoritmo SW modificado (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 é 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 4.4, sendo: (a) resolução, (b) localização, e (c) reconstrução.

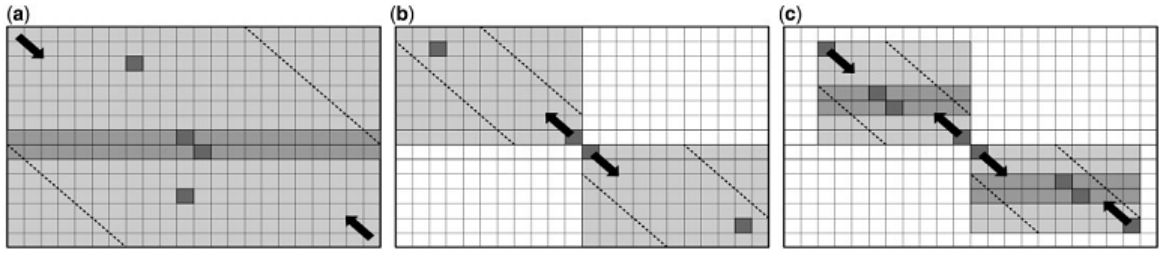


Figura 4.4: Fases da solução SW# [16]

Os resultados obtidos foram semelhantes ao CUDAlign 2.1 em uma única placa gráfica, sendo até mesmo mais lento nos testes com sequências mais longas. A melhoria de desempenho (medido em tempo de execução) foi observada apenas com a utilização da placa NVidia Geforce GTX 690 dual. Baseando-se no tamanho da sequência testada e no tempo de execução, calcula-se que a solução obteve 65,2 GCUPS. Como vantagem em relação ao CUDAlign, esta abordagem não necessita de espaço em disco adicional para armazenamento de linhas/colunas especiais para produzir um alinhamento ótimo.

4.6 SW 2.0

O algoritmo Gotoh (Seção 2.2.3) também é utilizado como base na solução SW 2.0 [17], desenvolvida em OpenCL (Seção 3.2.2), que realiza a comparação de uma sequência de referência de proteína com 128 símbolos contra as 262.144 sequências existentes na base de dados Swiss-Prot 2010_07, com suporte ao modelo de *affine gap*. A máxima paralelização é obtida fazendo com que cada *thread* (implementado no OpenCL como um *work-item*) calcule um alinhamento ótimo entre a sequência *query* e uma das sequências da base de dados.

Outra otimização relevante para o desempenho da solução determina que todas as *threads* de um mesmo *work-group* acessem a memória global em espaços contíguos, padrão de acesso denominado *coalesced*. Em vez de armazenar os símbolos da sequência *query*, são armazenados na área de memória de textura da GPU os escores obtidos na comparação da sequência de pesquisa contra todos os possíveis

símbolos do alfabeto (para proteínas, 20 caracteres), e eles são lidos em conjunto de quatro para se ajustar ao tamanho do barramento da memória (32 bits).

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

4.7 Razmyslovich *et. al*

A solução proposta por Razmyslovich *et. al* [18] 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. Não está claro no trabalho se o modelo *affine gap* foi implementado na solução.

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. 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 de valores da matriz e uma contendo os dados que podem ser transferidos, como ilustrado na Figura 4.5.

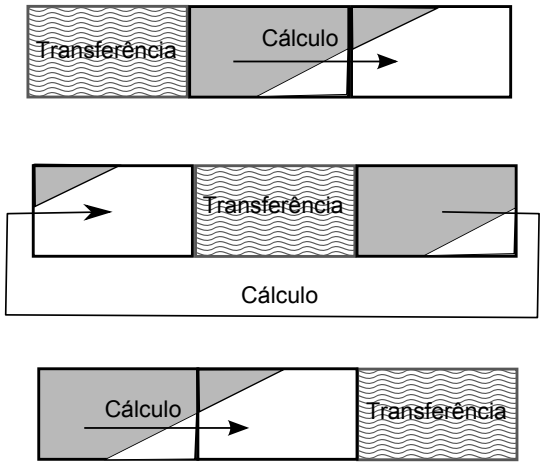


Figura 4.5: Modelo de *buffer* em anel da solução Razmyslovich *et. al* [18]

Duas versões da solução foram propostas e testadas em uma GPU NVidia Geforce 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 utilizadas como referência, possibilitando a pesquisa concorrente de até 600 sequências contendo os 36 nucleotídeos do Illumina. A versão que provê um dos alinhamentos ótimos calculados foi testada com 40 comparações simultâneas. O trabalho não informa explicitamente os resultados de desempenho obtidos.

4.8 Tabela Comparativa

A Tabela 4.1 apresenta uma comparação entre alguns aspectos das soluções apresentadas neste capítulo. Os artigos são referenciados preferencialmente pelo nome dado pelo(s) autor(es). As seções relativas aos termos abordados neste trabalho também são referenciadas entre parênteses.

A coluna "Tipo de Solução" indica se a solução realiza a comparação de proteínas, geralmente de tamanhos reduzidos e comparados contra uma base de dados existentes, ou de sequências de DNA, que em geral possuem tamanho mais significativo. A coluna "Tipo de Saída" informa se a solução exibe como saída de dados apenas o escore gerado ou também o alinhamento obtido, como discutido na Seção 2.2.

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), descritas no Capítulo 3.

A coluna "GCUPS" representa o desempenho no melhor caso obtido nos testes, medido em bilhões de células atualizadas por segundo (GCUPS - Seção 2.4). Finalmente, a coluna "Ambiente de Teste" detalha as placas gráficas utilizadas nos testes da solução proposta.

Avaliando-se os trabalhos mais relevantes que abordam a comparação de sequências biológicas em uma GPU (Tabela 4.1), observa-se que poucas 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 (Seção 3.2.1), restringindo o uso da aplicação para placas de vídeo produzidas pela NVidia (Seção 3.1.2) - apenas uma das soluções foi testada em uma GPU produzida pela AMD (Seção 3.1.1). As soluções existentes que utilizam a linguagem OpenCL (Seção 3.2.2) são focadas na comparação de sequências pequenas, inexistindo, a nosso conhecimento, uma aplicação que realize a comparação de sequências longas de DNA desenvolvida em OpenCL. Adicionalmente, o máximo desempenho reportado por qualquer das aplicações para processamento em uma GPU foi de 83,3 GCUPS.

Artigo	**Tipo de Solução**	**Tipo de Saída**	**Algoritmo**	**Plataforma**	**GCUPS**	**Ambiente de Teste**
SW-CUDA (4.1)	Proteínas	Escore	Gotoh (2.2.3)	CUDA (3.2.1)	3,6	NVidia Geforce 8800 GTX
Ligowsky e Rudinick (4.2)	Proteínas	Escore	Gotoh	CUDA	14,5	NVidia Geforce 9800 GX2 dual
CUDASW++ 1.0 (4.3)	Proteínas	Escore	Gotoh	CUDA	16,1	NVidia Geforce GTX 295 dual
CUDASW++ 2.0 (4.3)	Proteínas	Escore	Gotoh	CUDA	29,9	NVidia Geforce GTX 295 dual
CUDASW++ 3.0 (4.3)	Proteínas	Escore	Gotoh	CUDA	83,3	NVidia Geforce GTX 680
CUDAlign 1.0 (4.4)	DNA	Escore	Gotoh	CUDA	20,4	NVidia Geforce GTX 280
CUDAlign 2.1 (4.4)	DNA	Escore + Alinhamento	Gotoh + MM (2.2.4)	CUDA	52,8	NVidia Geforce GTX 560 Ti
SW# (4.5)	DNA	Escore + Alinhamento	Gotoh + MM	CUDA	65,2	NVidia Geforce GTX 690
SW 2.0 (4.6)	Proteína	Escore	Gotoh	OpenCL (3.2.2)	66,0	AMD/ATI HD 850
Razmyslovich et al (4.7)	DNA	Escore	SW	OpenCL	n.d.	NVidia Geforce GTX 260

Tabela 4.1: Artigos sobre soluções de comparação de sequências biológicas em GPU

Capítulo 5

Projeto do MASA-OpenCL

Como observado no Capítulo 4, várias soluções vêm sendo propostas para tratar o problema da comparação de sequências em ambiente de GPU. A maioria delas, contudo, utilizam linguagem proprietária (CUDA), e poucas implementam a comparação de sequências longas.

O objetivo desta dissertação é propor uma ferramenta para comparação de sequências biológicas longas de DNA em ambiente heterogêneo chamada MASA-OpenCL, possibilitando que ela seja executada em GPUs (Capítulo 3) de diversos fabricantes, ou mesmo em outras arquiteturas - tais como CPUs e placas co-processadoras ou circuitos programáveis como o *Field Programmable Gate Array* (FPGA) [73] - com poucas modificações no código-fonte. Devido a estes requisitos, foi escolhido o *framework* OpenCL (Seção 3.2.2) para implementação, para que a portabilidade do código em diferentes arquiteturas possa ser testada e avaliada.

Dentre as ferramentas avaliadas, o CUDAlign (Seção 4.4) apresenta desempenho muito bom na comparação de sequências longas em GPU. Em particular, a versão 2.1 produz o escore ótimo e retorna o alinhamento gerado na execução em uma GPU, com execução em seis estágios, sendo os três iniciais realizados em GPU. Ademais, a portabilidade de aplicações CUDA para OpenCL já foi objeto de estudo de alguns trabalhos [74] [75], servindo como parâmetros para a avaliação de uma aplicação implementada em OpenCL.

Por outro lado, a arquitetura MASA (Seção 4.4.1) foi implementada usando as mesmas abordagens do CUDAlign, oferecendo um *framework* com a infra-estrutura necessária para o processamento das sequências, e requerendo que apenas as funções de processamento da matriz de similaridade baseadas em variantes do algoritmo Smith-Waterman (Seção 2.2.2) sejam implementadas na linguagem específica. Desta forma, as otimizações já existentes no CUDAlign - como o processamento em *wavefront* (Seção 2.3), descarte de blocos e delegação de células - podem também ser utilizadas pelas extensões do MASA.

As Seções 5.1 a 5.5 detalham as fases envolvidas no desenvolvimento do MASA-OpenCL.

5.1 Análise da Implementação do MASA-CUDAlign

Para que o código original CUDA do MASA-CUDAlign pudesse ser portado para a linguagem OpenCL, o passo inicial foi a análise das funções existentes no programa relacionadas ao processamento dos estágios, além da avaliação geral do código e classes existentes. Em paralelo, foi necessário identificar todas as chamadas a funções providas pela plataforma CUDA, pois elas tiveram que ser adaptadas para a nova plataforma heterogênea.

Em particular, o escopo definido para o MASA-OpenCL foi a implementação do primeiro estágio do MASA-CUDAlign, que calcula o escore ótimo, retornando seu valor e posição na matriz de similaridade. Este estágio é o que possui maior impacto de desempenho na comparação de sequências biológicas, sendo também o que permite maior paralelização. Ademais, a maioria dos trabalhos avaliados na bibliografia (Capítulo 4) produz o escore como resultado final, ratificando a relevância deste estágio. O desenvolvimento dos demais estágios da solução e a obtenção do alinhamento ótimo são previstos como trabalhos futuros.

Visando adequar o MASA-CUDAlign ao escopo delimitado, uma versão simplificada da solução foi obtida. Neste código otimizado, apenas o primeiro estágio do programa é executado, inibindo a chamada aos procedimentos do segundo estágio, e consequentemente aos demais estágios. Além disso, o armazenamento de linhas especiais em disco também foi desabilitado, visto que não é necessário para realizar-se o cálculo do escore ótimo. Com exceção destas mudanças, todas as demais otimizações existentes no MASA-CUDAlign persistem nesta versão simplificada, incluindo a otimização do descarte de blocos (*Block Pruning* - BP).

Após avaliação do código, observa-se que o *framework* MASA e sua extensão CUDAlign foram desenvolvidos em linguagem C++, baseando-se nos conceitos de orientação a objeto. O código é bastante modularizado, tendo sido identificadas 34 classes envolvidas no processamento do primeiro estágio. Algumas das classes disponíveis e suas funcionalidades podem ser vistas na Tabela 5.1.

Classe	**Funcionalidade**
`AbstractAligner`	Classe abstrata que realiza o procedimento de alinhamento
`AbstractDiagonalAligner`	Classe abstrata que processa uma diagonal de blocos
`BlockAlignerParameters`	Classe que contém os parâmetros utilizados no alinhamento
`Configs`	Classe com funcionalidades de configuração da aplicação
`BlockPruningDiagonal`	Classe que implementa o procedimento de descarte de blocos (BP)
`Job`	Classe que gerencia os estágios e status do trabalho de alinhamento

Tabela 5.1: Algumas classes plataforma-independentes do MASA

Outro aspecto relevante é o nível de parametrização da aplicação, tanto em constantes existentes no código fonte como em argumentos informados na linha de comando de sua execução, que permitem que sejam escolhidos alguns parâmetros para o processamento das sequências. Todas as opções de compilação e execução estão também disponíveis no MASA-OpenCL.

A análise detalhada do código das funções indicou a necessidade de modificação em pelo menos 15 métodos ou procedimentos que utilizavam funções disponibilizadas pela linguagem CUDA. Estes procedimentos tiveram que ser modificados para que pudessem ser executados na plataforma OpenCL.

Adicionalmente, as funções específicas em CUDA que são executadas na GPU precisavam ser codificadas em OpenCL, para que pudessem ser executadas em ambientes heterogêneos. Foram identificadas quatro funções *kernel*: `kernel_initialize_busH_ungapped` é responsável por reinicializar a informação de BP nos blocos processados, e as outras três (chamadas `kernel_single_phase`, `kernel_long_phase` e `kernel_short_phase`) são chamadas para processar a diagonal externa ativa, dependendo dos parâmetros de processamento. Outras 15 funções auxiliares do tipo *device*, conforme a definição CUDA (Seção 3.2.1), foram identificadas no arquivo que implementa os procedimentos CUDA. O relacionamento entre estas funções pode ser observado na Figura 5.1, e representa o esforço de desenvolvimento requerido para a construção da extensão MASA em código específico.

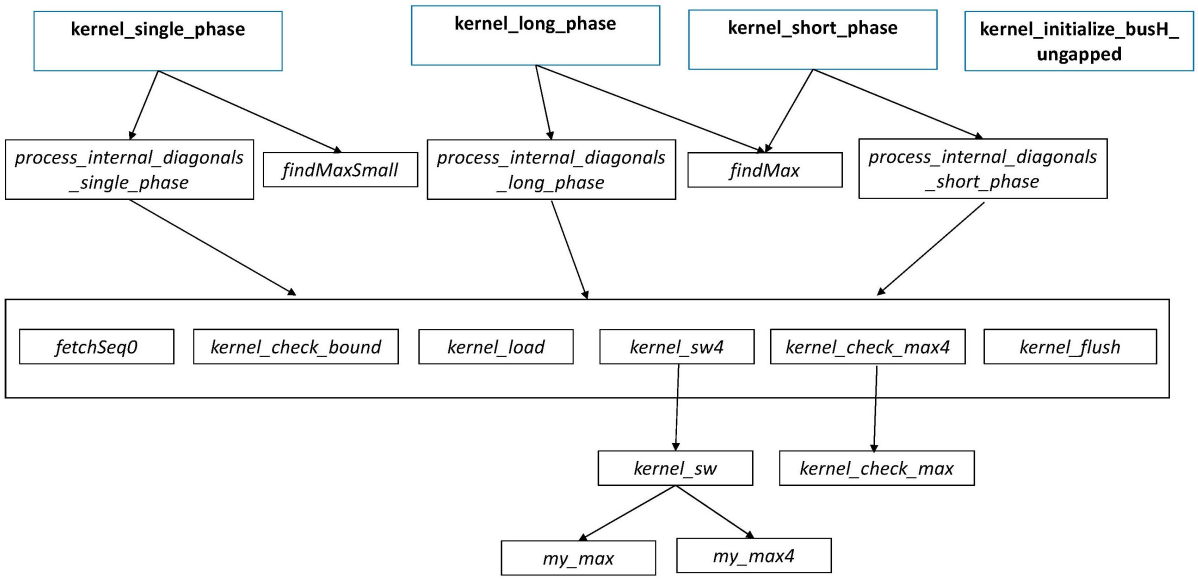


Figura 5.1: Funções do CUDAlign 2.1 executadas em GPU

Conforme identificado após a fase de análise, quase todas as funções CUDA mapeadas possuíam procedimentos equivalentes em OpenCL, ou poderiam ser reescritas na nova linguagem para produzir resultados similares. A exceção identificada foi a função `cuMemGetInfo` existente em CUDA para obtenção da quantidade de memória utilizada pelo programa, que não possui semelhante em OpenCL, visto que o gerenciamento da memória não é disponibilizado na linguagem. Contudo,

esta função é utilizada no MASA-CUDAlign apenas para fins informativos, não impactando no processamento do estágio.

5.2 Arquitetura da Solução MASA-OpenCL

Após a conclusão do mapeamento de todo o código do MASA-CUDAlign, foi possível identificar o escopo do projeto do MASA-OpenCL e sua interface com o *framework* MASA, propiciando a definição da arquitetura da solução. A opção pelo desenvolvimento da aplicação como uma extensão do MASA mostrou-se a alternativa mais adequada, uma vez que a utilização da arquitetura de classes existente simplificaria a implementação, objetivando manter o bom desempenho já obtido pelo MASA-CUDAlign na comparação de sequências longas.

A estratégia de implementação, portanto, consistiu em desenvolver uma nova implementação do *framework* MASA utilizando a linguagem OpenCL, chamada de MASA-OpenCL. A linguagem foi utilizada na implementação da estratégia de paralelização e do algoritmo de Smith-Waterman (Seção 2.2.2), enquanto a técnica de descarte de blocos foi executada de forma diagonal. Considerando a modelagem da arquitetura, esta extensão foi adicionada à Figura 4.3 ilustrada anteriormente, como pode ser visto na Figura 5.2.

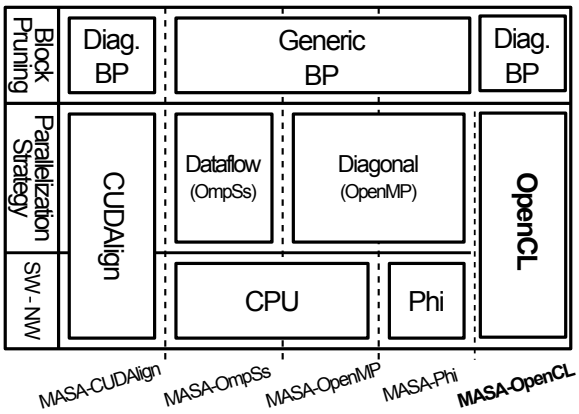


Figura 5.2: Arquitetura MASA com extensão MASA-OpenCL

Além da migração das funções identificadas, a extensão MASA-OpenCL deveria possuir novas funções utilizadas para realizar a chamada dos *kernels* existentes. Este requisito deve-se ao fato que, nas chamadas CUDA, um simples modificador é adicionado na chamada da função para que o compilador identifique que um procedimento deste tipo seja executado, além dos parâmetros desejados. Já na linguagem OpenCL, algumas etapas adicionais de preparação são requeridas, como observado na Seção 3.2.2.

5.3 Teste de Ferramenta de Migração de Código

Com a disseminação do OpenCL como linguagem para paralelização de execução em ambientes heterogêneos, alguns trabalhos abordaram ferramentas para conversão automática de códigos CUDA para OpenCL. O objetivo dessas ferramentas é agilizar o desenvolvimento, fazendo um mapeamento e reescrita automática das funções para gerar um código-fonte compilável na nova arquitetura, tendo em vista que os programas em CUDA são restritos à arquitetura NVidia. Dentre estas ferramentas, pode-se citar o SWAN [76] e o CU2CL [77].

Visando avaliar a viabilidade destas ferramentas, foi realizado um teste da execução do CU2CL para conversão do código CUDA existente no MASA-CUDAlign. Para tanto, foi necessária a instalação de alguns pacotes adicionais na estação de desenvolvimento, pois o CU2CL utiliza o *framework* de compilação CLANG [78]. Os testes, contudo, não produziram o resultado esperado, apresentando falha durante a execução. Isto ocorreu pois o programa CU2CL requer que o código CUDA esteja numa estrutura específica, o que não ocorre no MASA-CUDAlign.

Devido ao insucesso no teste, a opção pela conversão automática do código CUDA para OpenCL foi descartada, pelos seguintes motivos:

- Avaliou-se que o esforço para adequar a estrutura do código CUDA do MASA-CUDAlign ao exigido pelo CU2CL causaria alto impacto no processo de desenvolvimento;
- O MASA-CUDAlign é voltado para uma aplicação bem específica, com código não trivial, diferente da suíte de programas utilizados nos testes reportados no trabalho que propõe o CU2CL [77];
- A conversão automática poderia gerar códigos não otimizados, prejudicando a avaliação de desempenho;
- Finalmente, a ferramenta não possui uma abrangência total, deixando de converter várias instruções e modelos do CUDA, como foi observado pelos próprios autores em trabalho posterior [79].

Desta forma, optou-se por realizar a portabilidade do código da forma tradicional, buscando-se identificar as funcionalidades da linguagem OpenCL equivalentes às funções implementadas, para que não houvesse mudança substancial no projeto inicial, o que prejudicaria a comparação entre a solução proposta e a estrutura do MASA-CUDAlign.

5.4 Desenvolvimento da Solução MASA-OpenCL

Um dos objetivos deste trabalho é avaliar a portabilidade da solução MASA-OpenCL em ambientes heterogêneos, avaliando-se o esforço de mudança no código OpenCL em *hardwares* distintos. Para tanto, optou-se pelo desenvolvimento de código OpenCL que permitisse a execução em GPUs (NVidia e AMD) e CPUs. O detalhamento dos ambientes utilizados nos testes são mostrados na Seção 6.2.1.

Um dos pré-requisitos para o desenvolvimento da ferramenta é a escolha da versão da biblioteca OpenCL a ser utilizada na codificação. O Khronos Group já disponibiliza a versão 2.0 dos cabeçalhos OpenCL [60]. A nova versão possui algumas melhorias importantes em relação à versão 1.2, como funções de criação e manipulação de áreas de memória compartilhada a partir do *host*, paralelismo de *kernels* e novas funções atômicas. Contudo, optou-se pela adoção da versão 1.2 do *framework*, pois esta é a versão suportada pelos *hardwares* selecionados para teste. Da mesma forma, apesar de uma biblioteca na linguagem C++ estar disponível, optou-se pela versão desenvolvida na linguagem C, que já vem sendo testada por desenvolvedores há mais tempo, minimizando os riscos de incompatibilidade.

O próximo passo no desenvolvimento foi o mapeamento inicial das principais modificações a serem realizadas no pseudo-código do MASA-CUDAlign para produzir a solução de alinhamento MASA-OpenCL. Estas linhas estão marcadas com asteriscos no Algoritmo 1. O método `InitializeStructures()` (linha 2) teve que ser modificado para incluir a nova estrutura com ponteiros para as áreas globais de memória alocadas pelas funções OpenCL, além de outras variáveis globais utilizadas ao longo do código, tais como o contexto OpenCL e o endereço do dispositivo selecionado para execução do programa. O método `Aligner::processBlock()` (linha 12) também sofreu mudanças para incluir uma chamada para a nova função `cl_launch_external_diagonals` que efetivamente processa a diagonal ativa em determinado momento e chama as funções *kernel*. Algumas classes (tais como `IsBlockPruned`, `dispatchScore` e `EntryPoint`) foram herdadas da superclasse MASA, declarada na parte do código que é plataforma-independente.

Algoritmo 1 Implementação MASA-OpenCL - funções principais

```
1: procedure ALIGNER::ALIGNPARTITION(partition)
2:     InitializeStructures() ***
3:     for d = 0 to (gridH + gridW + 1) do
4:         for bx = max(0, d − (gridH − 1)) to min(d, gridW − 1) do
5:             by := d − bx
6:             AlignBlock(gridBlock[bx][by])
7:         end for
8:     end for
9: end procedure

10: procedure ALIGNER::ALIGNBLOCK(block)
11:     if Not MASA::IsBlockPruned(block) then
12:         block.score := Aligner::processBlock(block) ***
13:         MASA::dispatchScore(block.score)
14:     end if
15: end procedure

16: procedure MAIN(args)
17:     MASA::EntryPoint(args, new Aligner)
18: end procedure
```

O pseudo-código das funções de inicialização da estrutura e processamento de blocos são detalhadas no Algoritmo 2. As funções nativas OpenCL são executadas no procedimento `OpenCLAligner::Initialize` para identificar e alocar variáveis globais OpenCL (`platform`, `context`, `device` e `command-queue`) nas linhas 2 a 5. Na linha 6, todos os *buffers* de memória global são alocados, e os ponteiros para estas áreas são alocados em uma estrutura global. O código-fonte OpenCL é então compilado nas linhas 7 e 8, e a referência para o código binário gerado é

armazenado. Durante o processamento da diagonal externa, as funções *kernel* são inicializadas com os respectivos argumentos (linhas 11 e 12). O número de blocos existentes no *grid* que está sendo processado é avaliado na linha 13, e as funções *kernel* apropriadas são inseridas na fila de comandos (`command-queue`) nas linhas 14, 16 e 17, para posterior execução.

Algoritmo 2 Implementação MASA-OpenCL - funções de processamento de blocos

```
1: procedure OPENCLALIGNER::INITIALIZE
2:     platform := clSetupPlatform()
3:     context := CreateContext(platform)
4:     device := clSetupDevice(platform)
5:     commandqueue := clCreateCommandQueue(context, device)
6:     clAllocateGlobalStructures()
7:     program = clCreateProgramWithSource(context, sourcecode)
8:     clBuildProgram(program)
9: end procedure

10: procedure CL_LAUNCH_EXTERNAL_DIAGONALS(grid, opstructure)
11:     kernel := clCreateKernel(program, kernelname)
12:     clSetKernelArg(kernel, arguments)
13:     if blocks = 1 then
14:         ExecuteKernel(commandqueue, cl_kernel_single_phase)
15:     else
16:         ExecuteKernel(commandqueue, cl_kernel_long_phase)
17:         ExecuteKernel(commandqueue, cl_kernel_short_phase)
18:     end if
19: end procedure
```

5.4.1 Aspectos da Conversão de Código para OpenCL

Como citado na Seção 5.1, foram identificadas funções existentes na biblioteca OpenCL 1.2 equivalentes às principais funções CUDA utilizadas no código simplificado MASA-CUDAlign utilizado como base para a migração, e é possível utilizar a correlação entre os termos e estruturas utilizados pelas duas plataformas (Tabela 3.2) para realizar a portabilidade do código. Entretanto, alguns requisitos do OpenCL exigiram vários ajustes na programação, tais como:

- **Área de memória de textura:** em OpenCL, a área de memória de textura (que apresenta, em geral, menor tempo de acesso) é utilizada de forma restrita e apenas para manipulação de imagens. Esta característica é justificada pois o OpenCL, por definição, deve gerar um código portável para ambientes heterogêneos, e a área de textura é restrita às GPUs. Na código do MASA-CUDAlign esta área é utilizada para a alocação das sequências, que são armazenadas na inicialização do programa e são acessadas apenas para leitura durante o processamento. Devido a esta restrição, as sequências tiveram que ser armazenadas em memória global na solução MASA-OpenCL;
- **Conversão implícita de ponteiros vetoriais:** uma das funções *device* executadas no MASA-CUDAlign utiliza uma conversão implícita entre um ponteiro inteiro vetorial para um ponteiro inteiro (de `int4*` para `int*`). Esta operação não foi permitida pelo compilador OpenCL, e variáveis locais tiveram que ser criadas na respectiva função para realizar a conversão explicitamente, acessando-se o componente desejado do vetor;

- **Declaração de variáveis globais:** o MASA-CUDAlign realiza a declaração de matrizes do tipo `shared` dentro da área global do arquivo que armazena as funções CUDA. Isto não é permitido pela biblioteca OpenCL 1.2, que permite apenas que uma variável do tipo `constant` (que deve ser inicializada na declaração, e acessada posteriormente apenas para leitura) seja declarada desta forma, o que não atenderia aos requisitos. Para contornar o problema, as variáveis foram declaradas dentro do escopo das funções `device`, e repassadas como parâmetro para os demais procedimentos onde eram requeridas;
- **Restrição da função de sincronização:** a instrução utilizada em OpenCL para sincronização de *threads* (chamada `barrier`) exige que o código da função seja inteiramente executado por todas as *threads*, ou seja, não são permitidas instruções condicionais que modifiquem o fluxo de instruções de apenas algumas *threads*. Desta forma, o código teve que ser reescrito para evitar estas ocorrências onde existiam originalmente. Um exemplo das modificações feitas em uma das funções *kernel* pode ser visto no trecho de programa 5.1, onde o teste condicional realizado na função original MASA-CUDAlign (linhas 4 a 6) foi retirado para permitir que a função de processamento da diagonal fosse executada por todas a *threads* (linhas 14 a 16). Adicionalmente, a variável `max` teve que ser reinicializada para um valor mínimo (linha 16), para que as *threads* pudessem executar corretamente.

Programa 5.1 Ajuste de código em função *kernel*

```
 1: // Código original MASA-CUDAlign - com instrução condicional
 2: __global__ void kernel_long_phase()
 3: { // ...
 4:    if (i < i1) {
 5:      process_internal_diagonals_long_phase();
 6:   }
 7:   // ...
 8: }

 9: // ************************************************** //

10: // Código modificado MASA-OpenCL - sem instrução condicional
11: __kernel void cl_kernel_long_phase()
12: {
13: // ...
14:   cl_process_internal_diagonals_long_phase();
15:   if (!flush_id) {
16:      max = -INF;
17:   }
18: // ...
19: }
```

Outro aspecto relevante identificado no desenvolvimento da solução MASA-OpenCL está relacionado aos parâmetros utilizados na chamada das funções *kernel*. Segundo a sintaxe utilizada pela API de tempo de execução CUDA, a chamada a uma função *kernel* deve possuir, além dos argumentos normalmente exigidos pela referida função, dois parâmetros principais informados dentro dos delimitadores `<<< ... >>>`: o `grid_size`, indicando o número de blocos a serem processados;

e o `block_size`, que informa a quantidade de *threads* que devem ser executadas. Um terceiro parâmetro opcional pode ser passado informando o tamanho da memória compartilhada. No código do MASA-CUDAlign, a quantidade de *threads* a ser utilizada pelas funções é fixada no código (*hard-coded*) por questões de desempenho, visando otimizar a execução do programa à arquitetura da GPU utilizada, o que requer um esforço de recompilação caso uma arquitetura NVidia diferente seja utilizada.

Comparativamente, a linguagem OpenCL exige dois parâmetros na chamada das funções *kernel*: o `global_size`, que indica o tamanho total do trabalho a ser executado; e o `local_size`, que representa a quantidade de *threads* concorrentes a serem processadas. Por definição, o máximo valor do `local_size` é limitado por duas informações: o número máximo de *threads* permitidas pela função *kernel* específica e a quantidade de recursos computacionais do dispositivo escolhido (ou máximo `work_size`). A biblioteca OpenCL oferece duas funções para coletar estes dados: `clGetKernelWorkGroupInfo` e `clGetDeviceInfo`, respectivamente. Com isso, o valor de `local_size` pode ser ajustado em tempo de execução após a compilação do programa OpenCL e antes da chamada a cada função, sem requerer que o programa *host* seja modificado e recompilado, mesmo em dispositivos com arquiteturas diferentes.

O bom desempenho do algoritmo também foi um requisito que norteou a implementação da solução MASA-OpenCL. A otimização na alocação de memória e a atenção às melhores práticas utilizadas no desenvolvimento em OpenCL foram observadas durante toda a fase de codificação. Em especial, um aspecto avaliado foi a implementação do *loop unrolling*, técnica de otimização frequente em compiladores que reduz o tempo de execução nas instruções geradas em código binário ao paralelizar a execução dos laços mais críticos. Em OpenCL, esta técnica pode ser programada através de uma diretiva de pré-compilação, utilizando a instrução `#pragma unroll` no código fonte. Contudo, no código no MASA-CUDAlign esta otimização já é implementada explicitamente nos laços das funções *device* que processam as diagonais da matriz, ajustando o processamento à quantidade de blocos e *threads* utilizados. Desta forma, a mesma abordagem foi mantida no código do MASA-OpenCL, otimizando o paralelismo das instruções.

5.4.2 Desenvolvimento das Versões para CPU Intel e GPU NVidia

Visando a melhor organização do código, optou-se por adicionar um novo arquivo fonte ao projeto contendo funções específicas para chamar as funções *kernel* desenvolvidas em OpenCL. Enquanto na API CUDA a chamada a uma função deste tipo é realizada diretamente, em OpenCL alguns procedimentos de preparação e execução são necessários (Seção 3.2.2), como por exemplo a chamada às funções `clEnqueueWriteBuffer`, `clCreateKernel`, `clSetKernelArg`, `clEnqueueNDRangeKernel` e `clFinish`. A Figura 5.3 representa resumidamente o modelo de execução das funções *kernel* da solução MASA-OpenCL: a super-classe `AbstractAligner` da arquitetura MASA é especializada para construir os métodos/funções que realizam o alinhamento em OpenCL, e na função

`cl_launch_external_diagonals` a quantidade de blocos a serem processados é avaliada para executar as funções *kernel* correspondentes.

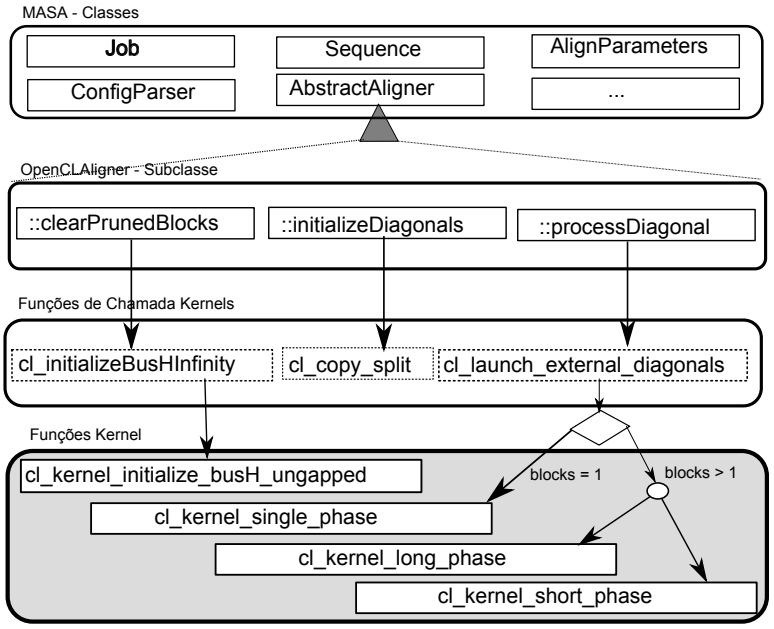


Figura 5.3: Modelo de execução de funções *kernel* no MASA-OpenCL

Também por decisão de projeto, as funções OpenCL necessárias para identificar plataformas (`clGetPlatformIDs`) e dispositivos (`clGetDeviceIDs`), além da compilação do código OpenCL das funções *kernel* (`clBuildProgram`) e criação da fila de comandos (`clCreateCommandQueue`) foram implementados no método que realiza a inicialização da classe herdada da arquitetura MASA. Esta observação é importante pois a compilação do código fonte OpenCL ocorre em tempo de execução, e ao realizar-se estas tarefas apenas na inicialização do programa minimiza-se o impacto durante o efetivo processamento da matriz de programação dinâmica. Cabe ressaltar que esta abordagem é possível para uma solução em única CPU/GPU: para a construção de uma versão em múltiplas CPUs ou GPUs (ou mesmo híbrida), este procedimento deveria ser reexecutado cada vez que um novo dispositivo fosse selecionado.

A primeira arquitetura selecionada no desenvolvimento para testes foi uma CPU Intel. As restrições impostas pela plataforma OpenCL citadas anteriormente foram os únicos aspectos relevantes observados na implementação desta primeira versão, aliado à complexidade inerente à migração de um código existente para uma nova plataforma/linguagem. Visando a avaliação da portabilidade da solução, optou-se por manter o código praticamente inalterado entre a versão original para GPU e a versão para CPU do MASA-OpenCL, apesar da existência de ferramenta que otimiza um código OpenCL projetado para GPUs visando sua execução em CPU com múltiplos núcleos [80]. Os resultados obtidos serão apresentados e discutidos no Capítulo 6.

O mesmo código desenvolvido para CPUs Intel pôde ser testado com sucesso em GPUs NVidia. A única modificação necessária é a escolha da plataforma desejada, que é informada como parâmetro na função `clGetDeviceIDs`: em vez da constante `CL_DEVICE_TYPE_CPU`, que faz com que a função procure por uma arquitetura tipo CPU, foi informado o parâmetro `CL_DEVICE_TYPE_GPU`, que força a busca por uma GPU. Na versão final do MASA-OpenCL, o tipo de plataforma desejado pode ser informado como argumento na linha de comando de execução de programa (CPU ou GPU, sendo que `CL_DEVICE_TYPE_GPU` é selecionado por *default* caso nenhuma opção seja indicada), mantendo o código totalmente portável entre as duas plataformas. O programa foi testado com sucesso em duas GPUs NVidia de diferentes arquiteturas, sem requerer nenhuma outra mudança.

5.4.3 Desenvolvimento das Versões para CPU e GPU AMD

Para os testes da portabilidade do código OpenCL em diferentes plataformas e fabricantes, o código construído inicialmente foi recompilado em uma máquina possuindo CPU e GPU da AMD - detalhes são fornecidos na Seção 6.2.1. Nestes ambientes, foi identificado um problema na construção da classe `Job` da arquitetura MASA, responsável pelo controle do processo de alinhamento: uma variável deveria ter sido inicializada com `Null` para garantir sua correta avaliação. Este problema era mascarado em outras arquiteturas pois a pilha era inicializada automaticamente pelo compilador, o que não ocorreu na versão executando em AMD. Vale salientar que esta foi a única modificação que teve que ser realizada na porção plataforma-independente do MASA durante todo o desenvolvimento da solução MASA-OpenCL.

Uma outra modificação que teve que ser realizada para este ambiente foi na sintaxe utilizada no acesso de variáveis que fazem parte de uma estrutura referenciada por um ponteiro. Enquanto na versão para CPU Intel e GPUs NVidia uma instrução do tipo `pos->x` é permitida, o compilador da AMD para OpenCL aceita apenas a sintaxe `(*pos).x`. É importante salientar que esta modificação foi exigida apenas pelo OpenCL na compilação do código das funções *kernel*, pois não há restrição à utilização da sintaxe original no código C/C++ da aplicação.

Finalmente, outra característica identificada neste ambiente está relacionada com a definição das variáveis compartilhadas auxiliares utilizadas no processamento das sequências. Enquanto nos ambientes CPU Intel e GPUs NVidia elas puderam ser declaradas nas funções adicionais executadas no dispositivo, neste ambiente elas só foram permitidas dentro do escopo das funções *kernel*, sendo necessário passá-las como parâmetro para as demais funções.

Para validação e otimização do código OpenCL desenvolvido neste ambiente (CPU e GPU AMD), foi utilizada a ferramenta AMD CodeXL [81]. Além de permitir a depuração do código, o programa também permite realizar um *profile* da aplicação desenvolvida, identificando eventuais problemas de desempenho e sugerindo modificações para ajustar o código às melhores práticas de desenvolvimento do OpenCL. A ferramenta identificou poucos pontos de melhoria no MASA-OpenCL, ratificando as decisões de projeto realizadas.

Mesmo considerando que as restrições citadas não foram identificadas na versão inicial desenvolvida para CPUs Intel e GPUs NVidia, cabe ressaltar que a nova versão desenvolvida para dispositivos AMD é totalmente compatível com os ambientes anteriores, pois as modificações realizadas também são aceitas pelos demais compiladores sem gerar erros. Desta forma, pôde-se observar que, de fato, o OpenCL possui um alto grau de portabilidade entre os ambientes analisados, um dos fatores que favorecem a adoção deste *framework*.

5.5 Resumo do Projeto do MASA-OpenCL

Baseando-se nas decisões de projeto e desenvolvimento discutidas, algumas características principais podem ser identificadas na solução MASA-OpenCL. Inicialmente, cabe destacar que a solução foi projetada como uma extensão da arquitetura MASA, utilizando como base o *framework* OpenCL. O objetivo foi desenvolver uma ferramenta portável para diversas plataformas. Foi utilizada na implementação a versão 1.2 do OpenCL, e a biblioteca de funções existentes nesta versão em linguagem C.

Adicionalmente, cabe destacar que o foco da aplicação é o cálculo do escore ótimo entre duas sequências longas de DNA, provendo também suas coordenadas na matriz de programação dinâmica. A solução possui boa flexibilidade, permitindo que sejam escolhidos parâmetros (como plataforma, quantidade de blocos ou quantidade de *threads*) a serem informados na linha de comando de execução do programa, que alteram a forma de processamento da matriz de programação dinâmica.

Foi possível gerar um código único capaz de ser executado em CPUs e GPUs. Em verdade, o código pode ser ajustado para executar em qualquer plataforma disponível em um determinado *host*, bastando para tanto modificar o parâmetro informado na chamada da função `clGetDeviceIDs` para `CL_DEVICE_TYPE_ALL`, requerendo a realização de ajustes nos parâmetros de execução ao *hardware* existente.

Finalmente, cabe destacar que a forma que a solução foi projetada permite que ela possa ser aprimorada em versões futuras com muito bom reuso das funções implementadas, como, por exemplo, para possibilitar o desenvolvimento dos demais estágios necessários para prover o alinhamento ótimo, ou a ampliação para execução em múltiplos dispositivos.

Capítulo 6

Resultados Experimentais

Neste capítulo estão descritos os testes de execução realizados para avaliar o desempenho e a portabilidade do MASA-OpenCL, ressaltando as condições de execução e resultados obtidos. Na Seção 6.1, descreve-se as versões dos programas utilizados para as compilações das soluções. Na Seção 6.2, os ambientes e sequências de DNA utilizados nos testes são detalhados, enquanto na Seção 6.3 são apresentados e discutidos os resultados obtidos nos testes.

6.1 Informações de Compilação

A compilação do código C/C++ utilizou a versão do compilador instalado em cada ambiente. Para os testes em GPUs NVidia, foi utilizado o compilador `gcc` 4.6.3, uma vez que esta versão já havia sido utilizada anteriormente na compilação de outras extensões MASA. Ademais, esta versão é a mais atual que é compatível com o pacote CUDA instalado no ambiente. Nesta plataforma, foi ainda necessária a recompilação das soluções MASA-CUDAlign (Seção 4.4.1) e SW# (Seção 4.5) utilizando-se CUDA, visando uma comparação de desempenho entre as soluções, tendo sido utilizado o compilador `nvcc` 4.2. Foram realizados testes com as versões em 32 e 64 bits do ambiente de desenvolvimento (SDK) CUDA, forçando-se a compatibilidade para arquitetura (*compute capability*) 2.0 (ou sm_20) e, quando a arquitetura da GPU permitia, a versão 3.0 da arquitetura foi forçada na compilação. Para tanto, o parâmetro de compilação `-gpu-architecture` foi utilizado.

Nos testes em CPUs e na GPU da AMD, foi utilizado o compilador `gcc` 4.8.2, mais recente disponível para o sistema operacional Linux. Os parâmetros de compilação foram mantidos idênticos aos utilizados nas GPUs NVidia (opções `-O3 -DUNIX`), para permitir uma comparação equitativa.

Como descrito no Capítulo 5, a versão da biblioteca OpenCL escolhida para o desenvolvimento da solução foi a 1.2, visando ampliar a possibilidade de execução do programa em diversos ambientes sem requerer nova codificação. Com o mesmo objetivo, a biblioteca em linguagem C desta versão da plataforma foi escolhida para o desenvolvimento do MASA-OpenCL.

6.2 Informações de Execução

Os testes realizados na solução MASA-OpenCL possuíam dois objetivos principais: avaliar a portabilidade do código OpenCL, executando-o em diferentes plataformas e fabricantes; e mensurar o desempenho da solução, comparando-a com testes semelhantes realizados por outras soluções de comparação de sequências biológicas longas. As seções 6.2.1 a 6.2.3 detalham as condições de execução.

6.2.1 Ambientes Utilizados

Foram escolhidos quatro ambientes para testes da solução MASA-OpenCL, que permitiram a avaliação de quatro CPUs de diferentes modelos e fabricantes, além de três GPUs (duas da NVidia e uma da AMD).

- **Ambiente I**: Estação de usuário (Notebook) dedicada, utilizada para testes exclusivamente da CPU Intel. Neste ambiente, os testes foram realizados fora do ambiente gráfico, para que a utilização de recursos gráficos não comprometessem os resultados. A máquina possui as seguintes especificações de *hardware* e *software*:
  - CPU: Intel I7 4500U, quatro núcleos, 1.80 GHz
  - *Cache*: L1 - 128 KB, L2 - 512 KB , L3 - 4096 KB
  - Memória: 8 GB, 1600 MHz
  - Disco rígido: 1 TB, 5400 RPM
  - Sistema operacional: Linux, distribuição Ubuntu LTS 12.04, 64 bits
- **Ambiente II**: Máquina dedicada para experimentos em GPU NVidia, localizada no LAICO (LAboratório de sistemas Integrados e COncorrentes) do Departamento de Ciências da Computação da Universidade de Brasília (CIC/UnB), com as especificações abaixo:
  - CPU: Intel I7 3770, quatro núcleos, 3.40 GHz
  - *Cache*: L1 - 128 KB, L2 - 1024 KB , L3 - 8192 KB
  - Memória: 8 GB, 1333 MHz
  - Disco rígido: 1 TB, 7200 RPM
  - Sistema operacional: Linux, distribuição Ubuntu LTS 12.04, 64 bits
  - GPU: NVidia Geforce GTX 680
- **Ambiente III**: Máquina dedicada para experimentos em GPU NVidia, também localizada no LAICO, com as especificações abaixo:
  - CPU: Intel I7 2600K, quatro núcleos, 3.40 GHz
  - *Cache*: L1 - 128 KB, L2 - 1024 KB , L3 - 8192 KB
  - Memória: 8 GB, 1333 MHz
  - Disco rígido: 1 TB, 7200 RPM

- Sistema operacional: Linux, distribuição Ubuntu LTS 12.04, 64 bits

- GPU: NVidia Geforce GTX 580

- **Ambiente IV**: Máquina dedicada para experimentos em CPU e GPU AMD, também localizada no LAICO, com as especificações abaixo:

  - CPU: AMD FX-8350, quatro núcleos, 4.00 GHz

  - *Cache*: L1 - 384 KB, L2 - 8192 KB , L3 - 8192 KB

  - Memória: 8 GB, 667 MHz

  - Disco rígido: 1 TB, 5400 RPM

  - Sistema operacional: Linux, distribuição Ubuntu LTS 14.04, 64 bits

  - GPU: AMD R9 280X

Os ambientes de CPU utilizados são de diferentes fabricantes, visando avaliar a portabilidade do código OpenCL. As principais características de cada CPU podem ser vistas na Tabela 6.1.

Característica	**FX-8350**	**I7 4500U**	**I7 3770**	**I7 2600K**
Fabricante	AMD	Intel	Intel	Intel
Threads de CPU	8	4	4	4
Clock por núcleo (GHz)	4.0	1.8	3.4	3.4
Cache L1 (KB)	384	128	128	128
Velocidade de memória (MHz)	1866	1600	1333	1333

Tabela 6.1: Comparação entre CPUs utilizadas nos testes

Três dos ambientes utilizados possuíam placas de vídeo dedicadas para testes de aplicações GPGPU. As principais características de cada GPU podem ser vistas na Tabela 6.2.

Característica	**Geforce GTX 580**	**Geforce GTX 680**	**Radeon 280X**
Fabricante	NVidia	NVidia	AMD
Nome do projeto	GF110	GK104	Tahiti XTL
Memória global (MB)	1536	2048	3072
Clock (MHz)	772	1006	850
Núcleos de processamento	512	1536	2048
Unidades de textura	64	128	128
Barramento (bit)	384	256	384

Tabela 6.2: Configurações de GPUs utilizadas nos testes

6.2.2 Sequências Comparadas

Sequências de diferentes tamanhos foram selecionadas para avaliar a execução do MASA-OpenCL, variando de milhares de pares de bases (KBP) a milhões de pares de bases (MBP), refletindo os mesmos casos de testes sugeridos em [19]. As sequências foram obtidas no site do *National Center for Biotechnology Information* (NCBI), distribuídas em pares de comparações e avaliadas em sua similaridade através do escore obtido no alinhamento ótimo. A Tabela 6.3 resume as informações relativas às comparações usadas nos testes de execução, onde a coluna "Escore" representa o valor do escore (ótimo) da região de maior similaridade entre as sequências, baseando-se nos parâmetros utilizados no processamento da matriz de programação dinâmica.

Comp.	**Sequência 1**		**Sequência 2**		**Escore**
	Id de Acesso	*Tamanho*	*Id de Acesso*	*Tamanho*	
10K	AF133821.1	10K	AY352275.1	10K	5091
50K	NC_001715.1	57K	AF494279.1	57K	52
150K	NC_000898.1	162K	NC_007605.1	172K	18
500K	NC_003064.2	543K	NC_000914.1	536K	48
1M	CP000051.1	1M	AE002160.2	1M	88353
3M	BA000035.2	3M	BX927147.1	3M	4226
5M	AE016879.1	5M	AE017225.1	5M	5220960
7M	NC_005027.1	7M	NC_003997.3	5M	172
10M	NC_017186.1	10M	NC_014318.1	10M	10235188
23M	NT_033779.4	23M	NT_037436.3	25M	9063
47M	NC_000021.7	47M	BA000046.3	33M	27206434

Tabela 6.3: Sequências selecionadas para testes

6.2.3 Parâmetros de Execução

Da mesma forma que na solução MASA-CUDAlign, o MASA-OpenCL permite que alguns parâmetros sejam informados para modificar a forma com que a matriz de programação dinâmica é processada, seja com alterações em constantes no código-fonte ou em argumentos informados na linha de comando de execução do programa. De forma geral, os testes foram realizados com os mesmos valores de parâmetros utilizados pelo MASA-CUDAlign em [19], visando equiparar os cenários de comparação.

A quantidade de linhas da matriz processadas por cada *thread* (α) foi mantida em 4, visto que este fator apresentou os melhores valores nos testes experimentais. Ademais, o código das funções que processam os blocos já encontrava-se otimizado para este valor, e a mudança nesta constante exigiria uma modificação relevante no algoritmo.

O número de registradores por *kernel* (R) também foi mantido livre, conforme o código original, de forma otimizada ao número de *threads* alocados. Os testes iniciais foram realizados habilitando a opção de descarte de blocos (*Block Pruning*

- BP), visando atingir melhores resultados. Testes posteriores foram realizados inibindo-se o BP para avaliar o ganho gerado pela técnica em GPUs.

Para o valor do número de blocos (B) processados simultaneamente, alguns testes foram realizados em GPUs variando-se a quantidade máxima de blocos, sem que houvesse diferenças perceptíveis de desempenho. Desta forma, a quantidade padrão de blocos utilizadas no MASA-CUDAlign foi mantida: 512. Caso desejado, este valor pode ser escolhido através do argumento `-blocks` na linha de execução do programa. Cabe ressaltar ainda que este valor pode ser reduzido pelo algoritmo durante a execução do programa, caso o parâmetro exceda o máximo número de blocos permitidos no *grid* construído.

Da mesma forma, o número de *threads* (T) foi definido como 128, mantendo-se o valor adotado por padrão na solução original. Este valor pode ser modificado durante a compilação, ajustando-se o parâmetro `THREADS_COUNT` no arquivo de configuração. Visando uma melhor análise do comportamento do programa em CPUs, este valor foi modificado para realizar diferentes testes nesta plataforma, tendo sido efetuados experimentos com 8, 16, 32, 64, 128 e 256 *threads*.

Finalmente, os valores dos escores utilizados para obtenção dos alinhamentos foram os mesmos adotados na solução MASA-CUDAlign: *match*: $+1$; *mismatch*: -3; primeiro *gap*: -5; extensão de *gap*: -2.

6.3 Resultados Obtidos

As sequências selecionadas foram testadas nos ambientes escolhidos em pelo menos três execuções individuais, apresentando desvio-padrão desprezível em cada caso de teste. Desta forma, os valores médios obtidos foram adotados nas análises. A portabilidade do código em plataformas heterogêneas foi avaliada baseada no esforço de ajuste no código fonte para obtenção dos resultados esperados, enquanto que o desempenho foi quantificado em GCUPS (Seção 2.4), utilizando estatísticas disponíveis na arquitetura MASA.

Conforme explicado na Seção 5.1, apenas o primeiro estágio do alinhamento de sequências (que retorna o escore ótimo obtido) foi inicialmente implementado no MASA-OpenCL, portanto apenas os tempos de execução relativos à inicialização do programa e execução efetiva do estágio 1 foram considerados na análise. Na apresentação dos dados, os melhores resultados obtidos em cada comparação estão identificados em **negrito** para melhor visualização.

6.3.1 Resultados em CPUs

O objetivo principal dos testes nos ambientes em CPU não foi a obtenção de desempenhos significativos, uma vez que os ambientes de CPUs utilizados (Tabela 6.1) não possuíam alto poder computacional. Em vez disso, o propósito primordial era a avaliação da portabilidade da solução MASA-OpenCL neste ambiente, considerando que tratava-se de CPUs de diferentes fabricantes e que o código da solução que serviu como base para a criação do MASA-OpenCL era desenvolvida em CUDA e voltada exclusivamente a GPUs NVidia.

Considerando este aspecto, apenas os casos de testes com sequências de tamanhos até 7 MBP (10K a 7M, conforme Tabela 6.3) foram testados, visto que as sequências maiores demandariam muitas horas para execução, e os casos escolhidos já foram suficientes para avaliar a execução da solução em CPUs. Para este teste inicial, foram escolhidas as CPUs Intel I7 4500U (Ambiente I) e AMD FX-8350 (Ambiente IV). Visando a avaliação do impacto da quantidade de *threads* utilizadas na execução do programa para estas configurações de *hardware*, foram realizadas execuções com diferentes valores de T, no intervalo de 8 a 256, sem que este aspecto afetasse de forma relevante os resultados obtidos em cada processador, como pode ser observado nas Tabelas 6.4 e 6.5, que mostram os resultados dos testes com sequências com tamanho de até 1 MBP em cada um dos processadores.

Comp.	**Intel I7 4500U** *GCUPS*					
	8T	**16T**	**32T**	**64T**	**128T**	**256T**
10K	0,019	0,039	0,044	0,057	0,079	**0,083**
50K	0,256	0,353	0,342	0,394	0,498	**0,568**
150K	0,553	0,736	0,728	0,816	0,855	**0,867**
500K	0,797	0,987	0,993	1,022	1,035	**1,058**
1M	0,942	1,023	1,179	1,199	1,212	**1,254**

Tabela 6.4: Resultados de testes em CPU - Intel I7 4500U

Comp.	**AMD FX-8350** *GCUPS*					
	8T	**16T**	**32T**	**64T**	**128T**	**256T**
10K	0,113	0,125	0,131	0,135	**0,138**	0,136
50K	**0,368**	0,358	0,347	0,338	0,268	0,215
150K	**0,418**	0,407	0,399	0,392	0,374	0,355
500K	**0,429**	0,419	0,412	0,410	0,405	0,399
1M	**0,486**	0,470	0,462	0,457	0,452	0,452

Tabela 6.5: Resultados de testes em CPU - AMD FX-8350

Contudo, refletindo a diferença nas arquiteturas dos processadores Intel e AMD escolhidos, os melhores desempenhos foram alcançados com diferentes parâmetros para cada CPU: 256 *threads* para o processador Intel I7 4500U e 8 *threads* para o processador AMD FX-8350. Os melhores resultados obtidos para cada CPU podem ser visualizados na Tabela 6.6.

Os resultados obtidos foram compatíveis com o poder computacional de CPUs, em especial considerando-se que as CPUs selecionadas possuem arquiteturas indicadas para estações de trabalho. Comparando-se com outros resultados reportados por outras extensões MASA [19], os resultados apresentam também padrão semelhante, produzindo melhor resultado quando duas sequências similares (com alto valor no escore ótimo) são comparadas, como no caso da comparação 5M. A CPU Intel obteve os melhores resultados para quase todas as comparações testadas, exceto a comparação 10K.

Comp.	Intel I7 4500U *GCUPS*	AMD FX-8350 *GCUPS*
10K	0,083	**0,113**
50K	**0,568**	0,368
150K	**0,867**	0,418
500K	**1,058**	0,429
1M	**1,254**	0,486
3M	**1,178**	0,434
5M	**2,570**	0,910
7M	**1,184**	0,436

Tabela 6.6: Resultados de testes em CPUs com *threads* otimizadas

Se forem avaliadas apenas as características de cada *hardware* (Tabela 6.1), o resultado esperado seria oposto ao obtido, considerando especialmente a maior quantidade de núcleos e maior *clock* da CPU AMD. Contudo, os melhores resultados alcançados pela CPU I7 podem ser explicados devido a uma característica implementada pelo fabricante na SDK OpenCL (a partir da versão 1.1) nos mais recentes processadores Intel: um módulo de vetorização implícita automática. A técnica baseia-se em produzir instruções em linguagem intermediária que realizam operações lógicas e aritméticas em vários elementos de um vetor ou matriz simultaneamente (instruções SIMD - *Single Instruction Multiple Data*). Na aplicação MASA-OpenCL, esta característica melhora o paralelismo no cálculo da matriz de programação dinâmica, permitindo melhor desempenho no processamento das sequências pelas funções *kernel*. Esta conclusão baseia-se não apenas nos resultados comparativos obtidos, mas na saída fornecida pelo comando de compilação do OpenCL neste ambiente, que reporta que as quatro funções *kernel* implementadas no MASA-OpenCL foram vetorizadas com sucesso no processador Intel.

Para ratificar esta observação, uma nova versão do programa MASA-OpenCL foi gerada inibindo-se a vetorização das funções *kernel*. Para simular este comportamento foi adicionado na declaração das funções o modificador `__attribute__((vec_type_hint(int4)))`, que informa ao compilador que o referido *kernel* já encontra-se vetorizado. Como pode observado na Figura 6.1, a versão não vetorizada do programa obteve resultado em média 57% inferior à versão vetorizada na CPU I7, possuindo valores de GCUPS inferiores às execuções na CPU AMD, exceto na comparação 10K.

Em outro teste, a solução MASA-OpenCL foi executada nas duas outras CPUs disponíveis, que possuem maior poder computacional: I7 3770 (Ambiente II) e I7 2600K (Ambiente III). O programa foi compilado utilizando-se 128 *threads*, visto que o tamanho do trabalho simultâneo é ajustado no código por funções OpenCL no momento do enfileiramento da execução dos *kernels*, como citado na Seção 5.4.1.

Visando ter um parâmetro para avaliação do desempenho, a extensão MASA-OpenMP [19] desenvolvida baseada no *framework* MASA também foi testada no mesmo ambiente. A solução é desenvolvida utilizando a interface de programação de aplicativo (API) *Open Multi-Processing* (OpenMP) [70], que permite a paralelização em *threads* conforme a quantidade de núcleos disponíveis na CPU (nos ambi-

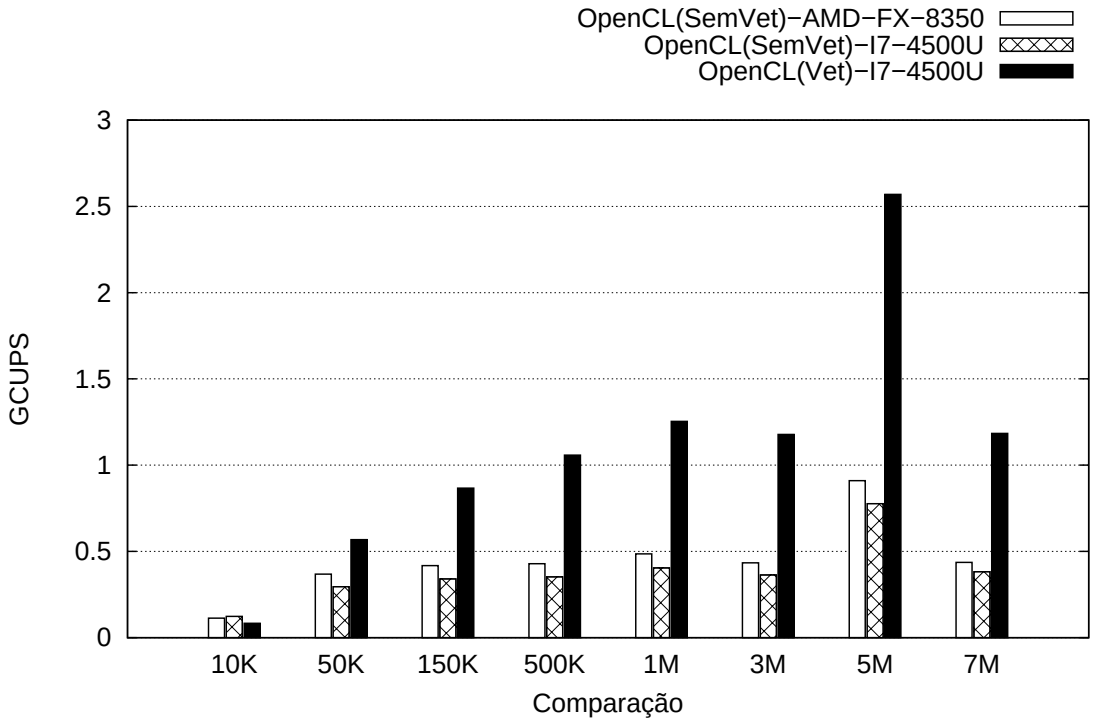


Figura 6.1: Resultado de comparações em CPUs: Intel I7 4500U (256 *threads*, versões com *kernel* vetorizado e não vetorizado) e AMD FX-8350 (8 *threads*)

entes testados foram usados oito *threads*). A aplicação MASA-OpenMP foi gerada em duas versões: compilada em 64 bits e compilada em 32 bits (usando a opção `-enable-32bit` disponível na configuração). A execução foi realizada utilizando-se os parâmetros `-no-flush -stage-1`, que forçam a aplicação a executar apenas o estágio 1, tornando o seu comportamento equivalente ao do MASA-OpenCL. As comparações até 7M foram usadas como entradas para os testes. Os resultados podem ser observados na Tabela 6.7.

Comp.	**I7 3770** ***GCUPS***			**I7 2600K** ***GCUPS***		
	OpenMP		**OpenCL**	**OpenMP**		**OpenCL**
	32 bits	***64 bits***	***32 bits***	***32 bits***	***64 bits***	***32 bits***
10K	**0,904**	0,895	0,083	0,660	**0,784**	0,105
50K	0,958	**0,971**	0,742	0,875	**0,916**	0,789
150K	1,017	1,018	**1,175**	0,928	0,960	**1,471**
500K	1,038	1,035	**1,774**	0,947	0,976	**2,077**
1M	1,182	1,178	**2,202**	1,078	1,111	**2,500**
3M	1,047	1,043	**2,118**	0,955	0,984	**2,295**
5M	2,457	2,424	**4,588**	2,241	2,309	**5,009**
7M	1,042	1,047	**2,145**	0,954	0,983	**2,344**

Tabela 6.7: Resultados de testes em CPUs - MASA-OpenMP e MASA-OpenCL

Como pode ser observado, as versões em 32 e 64 bits da solução MASA-OpenMP não apresentam diferença significativa de desempenho entre si. Além disso, exceto pelas menores sequências comparadas, a solução MASA-OpenCL apresenta desempenho consistentemente superior, chegando a um ganho de 145% na comparação 5M no processador I7 2600K.

A principal conclusão dos testes em CPU, contudo, é a confirmação da alta portabilidade da linguagem OpenCL, visto que o mesmo código fonte pôde ser executado em CPUs de diferentes modelos e fabricantes (bem como nas GPUs fabricadas pela NVidia e AMD) com mínimos ajustes, como mencionado na Seção 5.4.3.

6.3.2 Resultados em GPUs

A maioria das soluções presentes na literatura (Tabela 4.1) realiza a comparação/alinhamento de sequências pequenas. A nosso conhecimento, apenas o SW# (Seção 4.5) e o CUDAlign (Seção 4.4) comparam sequências longas de DNA em GPU, possibilitando uma comparação justa com a solução MASA-OpenCL. Desta forma, estas duas soluções foram selecionadas para a comparação de desempenho, restringindo a avaliação ao estágio que retorna apenas o escore ótimo calculado.

Visando uma avaliação mais detalhada, duas versões da solução SW# foram geradas: compilada em 64 bits (arquitetura original proposta pelos autores) e compilada em 32 bits. Os programas foram testados nas duas GPUs NVidia disponíveis, mantendo-se os parâmetros de execução originais (128 *threads* e 480 blocos). Para forçar a utilização de apenas uma GPU e restringir a execução para o cálculo do escore ótimo, as opções `-score -cards 0` foram utilizadas nos testes. Apenas as comparações envolvendo sequências de até 10 MBP foram utilizadas na avaliação, pois o programa SW# abortou durante a execução das comparações 23M e 47M em ambas as GPUs. Os resultados obtidos podem ser observados na Tabela 6.8.

Comp.	**GTX 580** ***GCUPS***			**GTX 680** ***GCUPS***		
	SW#		**OpenCL**	**SW#**		**OpenCL**
	64 bits	***32 bits***	***32 bits***	***64 bits***	***32 bits***	***32 bits***
10K	**2,2**	1,5	0,6	2,4	**2,5**	2,0
50K	**18,0**	17,0	9,4	**21,3**	19,4	15,7
150K	24,6	**27,8**	22,7	32,2	32,8	**34,0**
500K	30,5	32,2	**36,7**	39,0	43,2	**47,6**
1M	35,6	36,0	**43,4**	44,4	50,4	**56,5**
3M	32,9	34,1	**42,1**	41,5	47,9	**52,9**
5M	63,2	67,9	**82,6**	79,3	89,3	**106,7**
7M	33,1	34,2	**42,5**	41,8	48,3	**53,0**
10M	63,1	67,7	**82,8**	78,9	89,0	**106,5**

Tabela 6.8: Resultado de comparações em GPUs NVidia: SW# e MASA-OpenCL

A compilação em 32 bits da solução SW# apresentou melhores resultados entre as duas versões testadas. Contudo, exceto nas sequências menores, o MASA-OpenCL apresenta maiores GCUPS, com ganhos de até 22% em relação ao SW#

compilado em 32 bits. As Figuras 6.2 e 6.3 representam a comparação de desempenho entre as soluções em cada uma das GPUs testadas.

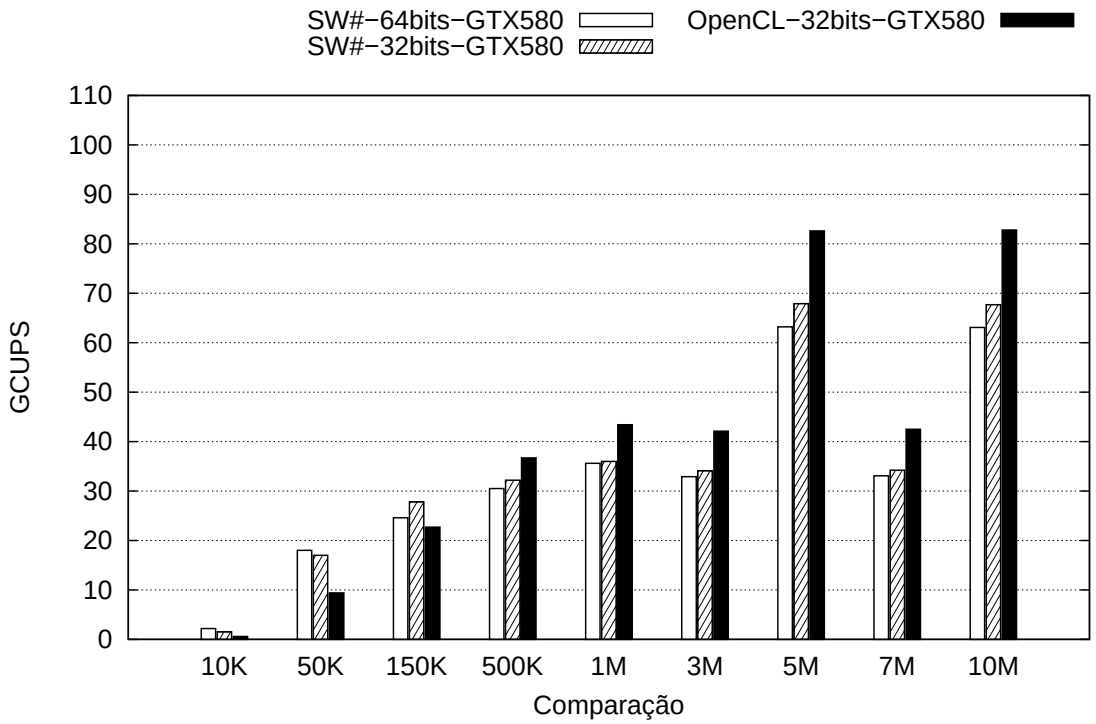


Figura 6.2: Resultado de comparações SW# e MASA-OpenCL na NVidia GTX 580

A primeira GPU escolhida para testes comparativos entre as soluções MASA-CUDAlign e MASA-OpenCL foi a Geforce GTX 680 (Ambiente II). Para tanto, o código do MASA-OpenCL foi compilado e executado em ambiente não gráfico, de forma que a placa gráfica ficou dedicada para a execução do programa. Para uma avaliação comparativa do desempenho obtido, o código da versão simplificada (apenas executando o primeiro estágio, sem salvar colunas especiais em disco) do MASA-CUDAlign foi compilado no mesmo ambiente, mantendo os mesmos parâmetros de execução entre as duas soluções. Todos os casos de comparação (Tabela 6.3) foram testados individualmente em cada uma das aplicações.

Os primeiros resultados mostraram um ganho consistente de cerca de 35% de desempenho em favor da solução MASA-OpenCL. Apesar de existirem trabalhos confirmando que, sob condições justas de comparação, a linguagem OpenCL apresenta desempenho comparável à CUDA [75], este resultado significativamente superior não era esperado, uma vez que o esforço de compilação em tempo de execução do OpenCL é uma tarefa adicional não existente na solução em CUDA. Outrossim, era esperado que, a princípio, a linguagem CUDA geraria códigos mais otimizados para GPUs NVidia, por ser uma solução linguagem específica para aplicações em GPUs produzidas por esse fabricante, o que, no mínimo, contribuiria para resultados equivalentes entre as duas soluções.

Uma vez que as linguagens CUDA e OpenCL geram um código intermediário (instruções PTX) no processo de compilação em GPU NVidia, a comparação

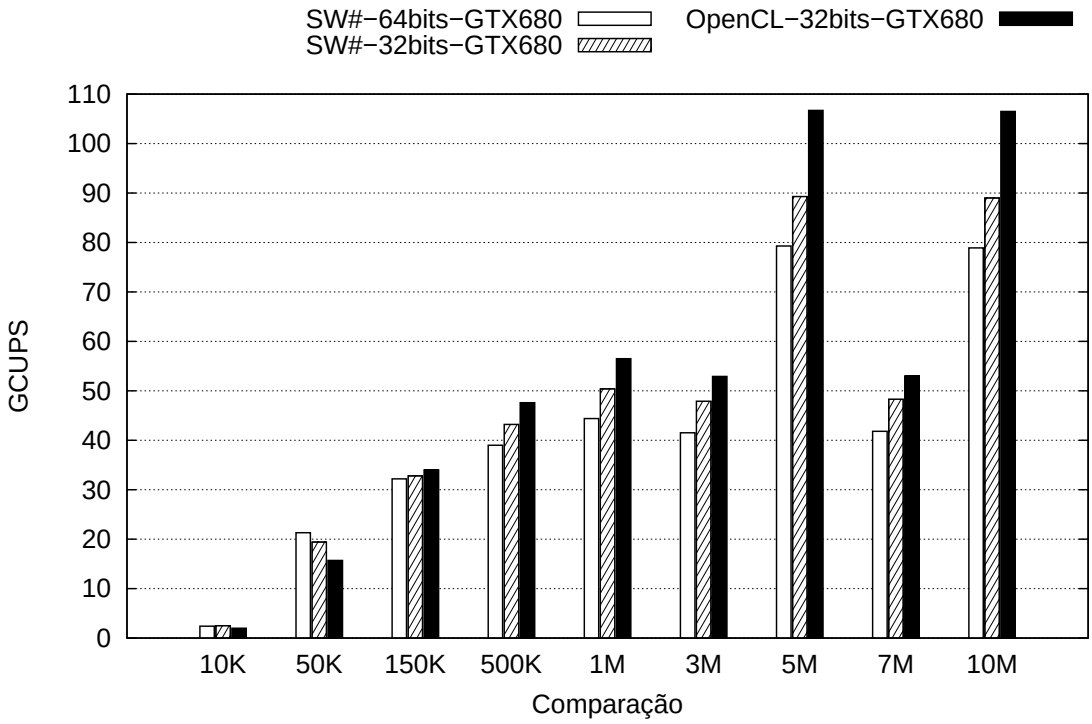


Figura 6.3: Resultado de comparações SW# e MASA-OpenCL na NVidia GTX 680

dos códigos gerados foi a melhor opção para investigar as causas do resultado obtido. Para tanto, o código PTX gerado pela compilação do MASA-CUDAlign foi obtido utilizando-se o comando `nvcc -ptx` sobre o arquivo que continha as funções *kernel* escritas em CUDA. Para a obtenção do código gerado pela compilação OpenCL, um programa simplificado que apenas compilava as funções *kernel* foi criado, e o parâmetro `CL_PROGRAM_BINARIES` foi informado na execução da instrução `clGetProgramInfo`, permitindo que o código com as instruções PTX pudesse ser salvo em um arquivo texto.

Ao comparar os dois arquivos, foi identificado que o código OpenCL gerava apenas instruções 32 bits, enquanto o código PTX gerado pela compilação CUDA priorizava instruções 64 bits. Por exemplo, enquanto o código gerado pelo OpenCL gerava a instrução `ld.param.u32` para carregamento de valor em um ponteiro, o código CUDA produzia a instrução `ld.param.u64`. Isto ocorre pois a compilação OpenCL é sempre realizada em 32 bits, enquanto que a compilação CUDA pode ser feita em plataformas 32 ou 64 bits, devendo a opção ser especificada na instrução de compilação.

Como pode ser observado, ponteiros 64 bits e suas respectivas operações tendem a requerer mais recursos no processamento em GPUs do que equivalentes em 32 bits, o que poderia produzir as diferenças de desempenho. Para verificar este impacto, o código do MASA-CUDAlign foi recompilado utilizando o parâmetro de configuração `-enable-32bit`. Os testes foram então repetidos com a nova versão do programa, mostrando um desempenho 20% superior à versão 64 bits, em todas as comparações. Comportamento semelhante já havia sido identificado nos testes

das versões compiladas em 32 e 64 bits da solução SW# (Tabela 6.8).

Ainda objetivando identificar possíveis razões para a diferença de desempenho, o código do MASA-CUDAlign foi modificado para substituir a alocação das sequências: em vez da área de memória de textura utilizada originalmente em CUDA, foi utilizada a área de memória global. Esta era a única diferença significativa entre as arquiteturas das duas soluções, uma vez que, como citado na Seção 5.4.1, o OpenCL utiliza a área de textura apenas em instruções de manipulação de imagens, o que exigiu que o local de armazenamento tivesse que ser modificado no desenvolvimento do MASA-OpenCL. Após realizadas as mudanças no código, a aplicação foi novamente recompilada em 32 bits e testada.

As comparações das sequências em CUDA utilizando a memória global da GPU obtiveram valores de GCUPS 10% superiores à versão que utilizava a área de textura. Os resultados podem ser explicados pelas otimizações implementadas pela NVidia no *cache* L1 das placas com arquitetura Fermi e Kepler. Mesmo que o acesso à área de textura seja mais rápido do que o acesso à memória global, a vantagem pode ser compensada caso os dados requeridos residam na *cache*, e dependendo ainda da forma com que os dados são acessados. Esta característica já foi observada em um trabalho que avalia uma outra aplicação GPGPU [82], onde o acesso à memória global foi mais rápido que à memória de textura. Estas mesmas condições são observadas na solução MASA-CUDAlign, pois as sequências são inteiramente carregadas durante a inicialização, e apenas uma das sequências é acessada sequencialmente em conjuntos de quatro caracteres.

Com as modificações realizadas, a versão sem textura e compilada em 32 bits do MASA-CUDAlign apresentou desempenho comparável ao obtido pelo MASA-OpenCL, mas ainda um pouco inferior na maioria das comparações. Esta diferença (cerca de 3%, em média) pode ser atribuída a otimizações de desempenho realizadas pelo processo de compilação OpenCL. Os resultados (expressos em GCUPS) obtidos para a GPU Geforce GTX 680 discutidos nos parágrafos anteriores podem ser observados na Figura 6.4.

As estatísticas implementadas no MASA-CUDAlign e herdadas pelo MASA-OpenCL permitem que os tempos de execução (ou *wallclock*) de cada uma das fases do processamento do primeiro estágio do alinhamento das sequências possam ser comparados. A Tabela 6.9 mostra os tempos obtidos (expressos em milissegundos) para cada caso de teste. As seguintes fases são avaliadas:

- **Sequências**: as sequências envolvidas na comparação são carregadas na memória da CPU, para posterior processamento pela GPU;
- **Inicialização**: são realizados os processos de inicialização de estruturas e áreas de memória. Na solução MASA-OpenCL, nesta fase ocorre a compilação em tempo de execução das funções *kernel*;
- **Execução**: contempla todas as funções necessárias para calcular as matrizes de programação dinâmica e obter o escore ótimo.

Como pode-se observar, os tempo necessários para o carregamento das sequências é um pouco maior na solução MASA-OpenCL. A compilação em tempo de execução requerida pela linguagem OpenCL não impacta significativamente o tempo

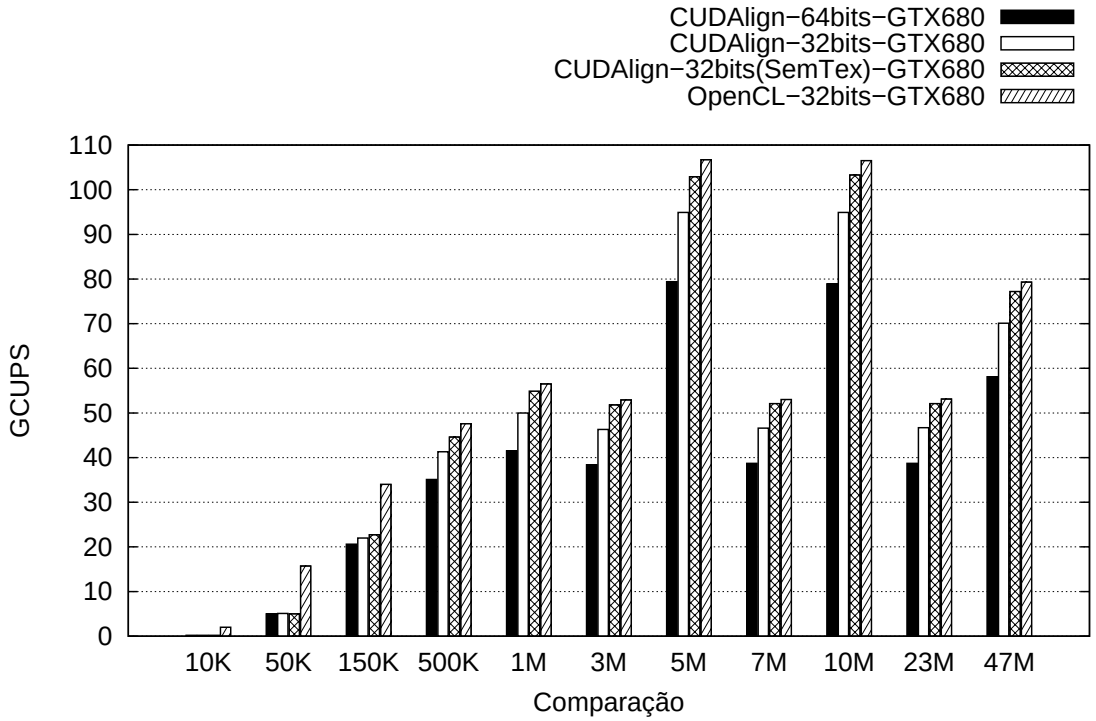


Figura 6.4: Resultado de comparações na NVidia GTX 680 - 128 *threads*

Comp.	MASA-CUDAlign			MASA-OpenCL		
	Seq.	**Inicial.**	**Execução**	**Seq.**	**Inicial.**	**Execução**
10K	0,6	515,9	14,2	0,7	43,3	19,0
50K	1,1	519,5	126,1	3,0	50,1	148,8
150K	7,2	518,6	702,7	8,1	45,7	761,2
500K	16,1	509,3	6.003,3	16,7	38,1	6.112,9
1M	27,5	505,8	19.880,1	28,1	38,1	19.844,0
3M	64,5	503,8	198.794,3	80,6	38,7	195.289,1
5M	96,8	516,3	264.918,7	121,9	38,7	256.307,3
7M	114,5	506,9	715.656,6	137,3	38,2	702.328,2
10M	188,3	521,2	1.013.905,8	217,0	38,5	981.201,9
23M	409,4	507,0	10.835.397,0	501,9	38,4	10.533.671,0
47M	699,9	510,5	19.943.354,0	844,8	38,2	19.336.776,0

Tabela 6.9: Tempos das fases de sequência, inicialização e execução (ms)

dos processos de inicialização, sendo aproximadamente constante para todas as sequências. Na realidade, o tempo requerido para execução destas tarefas é mais acentuado na solução MASA-CUDAlign, pois existe uma função adicional que pesquisa as GPUs existentes para identificar a que possui maior capacidade de processamento CUDA. Esta função não foi implementada na solução MASA-OpenCL, tendo em vista que ela deve ser portável para arquiteturas heterogêneas. Em vez disso, a primeira GPU disponível é selecionada. Na fase de execução efetiva do

primeiro estágio, o MASA-OpenCL processa o algoritmo em menor tempo, determinando um melhor desempenho em quase todas as comparações.

Para confirmar os resultados e análises realizadas, outra GPU da NVidia (Geforce GTX 580, instalada no Ambiente III) foi selecionada para testes, repetindo-se as mesmas ações realizadas na GPU GTX 680. Foram compilados e testados, portanto, as seguintes versões do programa: MASA-CUDAlign original, compilado em 64 bits; MASA-CUDAlign original, mas compilado em 32 bits; MASA-CUDAlign modificado para não utilizar a memória de textura, compilado em 32 bits; e, finalmente, o MASA-OpenCL, compilado em 32 bits.

Nesta GPU, a comparação 47M (envolvendo as sequências contendo 47 MBP e 33 MBP) apresentou erro na execução no MASA-OpenCL e não pôde ser realizada. Isto ocorreu pois a GPU possui uma limitação na quantidade de memória global disponível (1536 MB), sendo insuficiente para a quantidade de memória requerida pela solução MASA-OpenCL. Em testes empíricos, verificou-se que a GPU é capaz de processar sequências de até 32 MBP utilizando a aplicação, limitação que não ocorre nas demais GPUs testadas. Vale ressaltar que o MASA possui funcionalidades de particionamento de sequências, o que evitaria que este erro ocorresse. Contudo, esta opção não está disponível na versão simplificada utilizada como base no desenvolvimento, visto que requer que o salvamento de linhas/colunas especiais estivesse habilitado.

Os resultados obtidos na GPU Geforce GTX 580 confirmam as mesmas observações realizadas nos testes iniciais: a versão do MASA-CUDAlign sem utilizar a área de textura e compilada em 32 bits apresenta melhores resultados dentre as versões testadas desta aplicação. Contudo, a solução MASA-OpenCL também apresentou melhores resultados para todas as sequências testadas nesta GPU, com um ganho médio em torno de 8%. Os valores de GCUPS obtidos nesta GPU (Figura 6.5) foram inferiores aos reportados nos testes com a GPU Geforce GTX 680, o que era esperado, dado a diferença de configuração entre as placas (Tabela 6.2).

Visando avaliar o desempenho e a portabilidade da solução MASA-OpenCL em uma GPU AMD, a solução foi testada na placa de vídeo AMD Radeon R9 280X (Ambiente IV), sem requerer mudanças significativas no código implementado. Neste ambiente, não foi possível realizar testes do MASA-CUDAlign, uma vez que a linguagem CUDA não é compatível com esta arquitetura. A expectativa era que os resultados apresentados fossem superiores aos obtidos pela solução MASA-OpenCL nos testes na GPU NVidia GTX 680, devido ao maior número de núcleos disponíveis para processamento (2.048), o que aumentaria o paralelismo.

Os resultados obtidos neste primeiro teste, contudo, não apresentaram melhorias tão significativas de desempenho, mostrando, inclusive, menores GCUPS em algumas comparações. Isto ocorreu pois foi mantida a quantidade padrão de *threads* dos testes anteriores (128), o que limitava a utilização dos recursos da placa R9 280X, que possibilita que mais *threads* possam ser executados simultaneamente. Todos os testes foram realizados considerando diferentes versões dos programas, como detalhado na Tabela 6.10. Os resultados obtidos são compilados na Tabela 6.11.

Para testar o MASA-OpenCL na placa AMD com maior número de *threads*, o parâmetro `THREADS_COUNT` foi ajustado para 256 (máximo número de *threads*

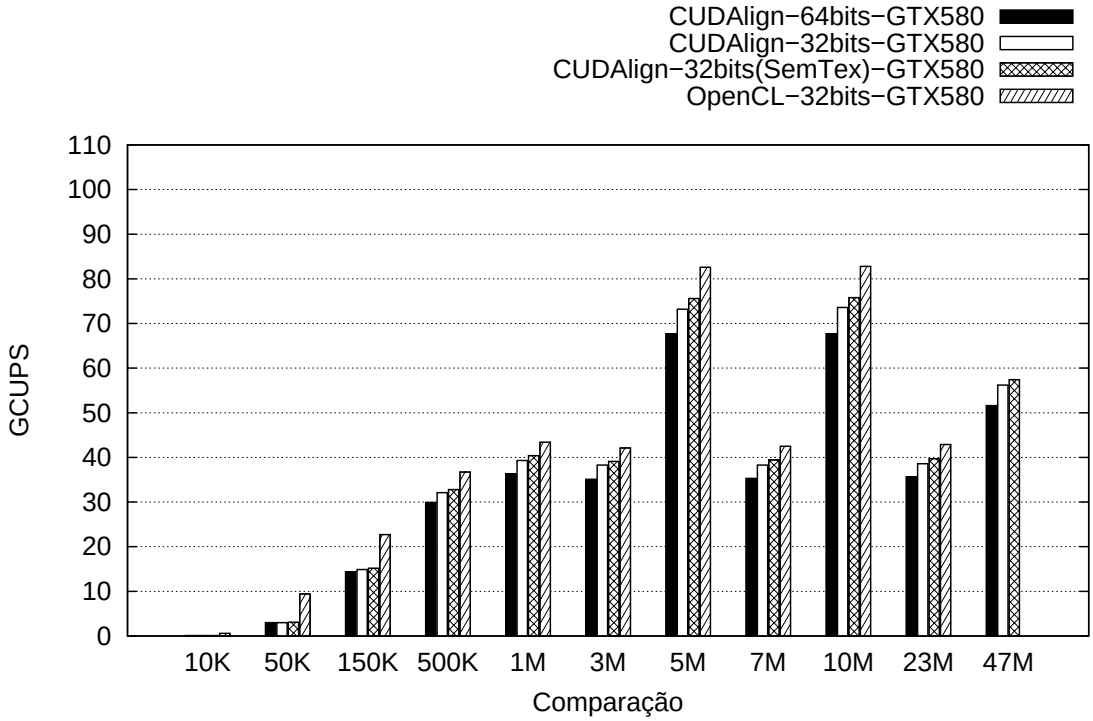


Figura 6.5: Resultado de comparações na NVidia GTX 580 - 128 *threads*

Versão	**Solução**	**Compilação**	**Alocação de Sequências**
V1	MASA-CUDAlign	64 bits	Memória de Textura
V2	MASA-CUDAlign	32 bits	Memória de Textura
V3	MASA-CUDAlign	32 bits	Memória Global
V4	MASA-OpenCL	32 bits	Memória Global

Tabela 6.10: Versões de programas testados em GPUs

permitido para a execução da solução nesta GPU), sendo então o programa recompilado. Com a nova versão, os resultados obtidos foram muito superiores (entre 60% e 90%) aos conseguidos com a GPU NVidia para as sequências com tamanho acima de 1 MBP, uma vez que as comparações menores não utilizam todos os recursos disponíveis na placa da AMD testada. O pico de desempenho obtido foi de 179,2 GCUPS, na comparação 10M. Salvo melhor entendimento, este é o melhor desempenho obtido por uma solução que provê o escore ótimo na comparação de sequências biológicas com algoritmo exato utilizando apenas uma GPU.

Uma versão do MASA-OpenCL utilizando 256 *threads* também foi testada na GPU NVidia GTX 680, sem prover melhores resultados, confirmando que o ganho na placa AMD decorre de sua arquitetura. Os resultados dos testes utilizando 256 *threads* nas duas GPUs podem ser vistos na Tabela 6.12.

A Figura 6.6 mostra os melhores resultados obtidos por cada uma das soluções utilizando 128 *threads*, além dos resultados da compilação utilizando 256 *threads* do MASA-OpenCL na GPU AMD R9 280X. Como pode ser observado, os testes na

Comp.	NVidia GTX 580				NVidia GTX 680				AMD
	V1	V2	V3	V4	V1	V2	V3	V4	V4
10K	0,1	0,1	0,1	0,6	0,2	0,2	0,2	**2,0**	0,1
50K	3,0	3,0	3,1	9,4	5,0	5,0	5,0	**15,7**	2,2
150K	14,4	14,9	15,2	22,7	20,6	22,1	22,7	**34,0**	10,3
500K	29,9	32,1	32,8	36,7	35,1	41,2	44,6	**47,6**	32,8
1M	36,3	39,3	40,4	43,4	41,5	49,9	54,9	**56,5**	51,6
3M	35,1	38,3	39,1	42,1	38,4	46,3	51,8	52,9	**64,9**
5M	67,7	73,2	75,6	82,6	79,4	94,8	102,9	**106,7**	105,7
7M	35,3	38,3	39,4	42,5	38,7	46,5	52,1	53,0	**69,2**
10M	67,7	73,6	75,8	82,8	78,9	94,8	103,3	106,5	**111,2**
23M	35,6	38,6	39,7	42,9	38,7	46,4	52,1	53,1	**74,7**
47M	51,6	56,2	57,4	n.d.	58,1	70,1	77,2	79,3	**97,4**

Tabela 6.11: Resultado de testes em GPUs utilizando 512 blocos e 128 *threads*

Comp.	NVidia GTX 680 OpenCL	AMD R9 280X OpenCL
10K	0,1	**0,2**
50K	**16,6**	4,5
150K	**34,4**	16,9
500K	**46,4**	46,0
1M	55,1	**73,0**
3M	51,7	**89,5**
5M	109,5	**171,8**
7M	51,9	**97,1**
10M	108,9	**179,2**
23M	52,4	**101,2**
47M	79,3	**144,0**

Tabela 6.12: Resultado de testes da solução MASA-OpenCL usando 256 *threads*

GPU AMD obtiveram os maiores GCUPS nas comparações envolvendo sequências mais longas, enquanto utilizando a técnica de BP.

6.3.3 Avaliação do *Block Pruning* em GPUs

A técnica de descarte de blocos (BP) proposta inicialmente no CUDAlign 2.1 (Seção 4.4) foi também implementada na solução MASA-OpenCL, e a sua contribuição para o desempenho foi avaliada em testes experimentais nas GPUs. Apenas as comparações possuindo sequências com pelo menos 1 MBP foram analisadas, uma vez que neste cenário as vantagens do uso do BP são mais relevantes. Para que a técnica de BP fosse inibida nos testes, o parâmetro `-no-block-pruning` foi informado na linha de comando de execução.

A primeira constatação é que a linguagem escolhida para implementação do MASA-OpenCL não interfere na forma com que a técnica de BP afeta o proces-

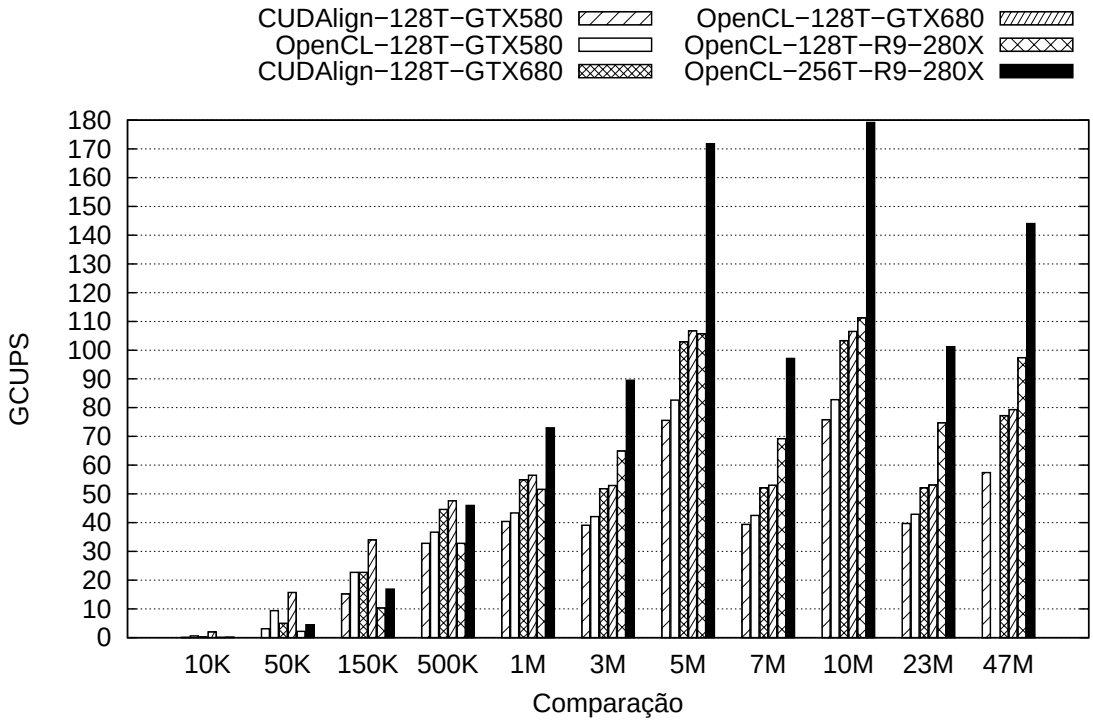


Figura 6.6: Resultado de testes com BP número de *threads* (T) otimizado

samento dos blocos, visto que os resultados são semelhantes a outros obtidos por outras implementações MASA [19]. Inibindo-se a execução do BP, o número de células atualizadas por segundo tende a se estabilizar nas comparações com sequências maiores (acima de 3 MBP), e não apresenta relação com o grau de similaridade entre as sequências, como pode ser observado na Figura 6.7, que compara os GCUPS obtidos na execução do MASA-OpenCL utilizando 128 *threads* sem o BP habilitado nos três ambientes de GPU testados. Comparando-se as respectivas séries com os resultados obtidos utilizando-se o BP (Figura 6.6), observa-se que, especialmente nas comparações envolvendo sequências maiores, o número de GCUPS continua aumentando ao se utilizar o BP, principalmente nas comparações envolvendo sequências muito similares (5M, 10M e 47M).

A Tabela 6.13 apresenta o ganho de desempenho obtido com a implementação do técnica de BP no processamento do primeiro estágio das soluções MASA-CUDAlign e MASA-OpenCL nas três GPUs avaliadas. Como citado, o ganho proporcionado pelo descarte de blocos é mais significativo nas comparações com maior escore ótimo calculado. Em algumas das comparações, mais da metade das células da matriz de programação dinâmica foram descartadas, refletindo num ganho de aproximadamente 50% no tempo de processamento nas GPUs NVidia. Para cálculo do ganho, utilizou-se a Equação 6.1, onde: $GCUPS_{SemBP}$ é o desempenho obtido sem a técnica de BP habilitada, enquanto $GCUPS_{BP}$ é o valor de GCUPS utilizando a técnica de descarte de blocos.

$$Ganho = 1 - \frac{GCUPS_{SemBP}}{GCUPS_{BP}} \tag{6.1}$$

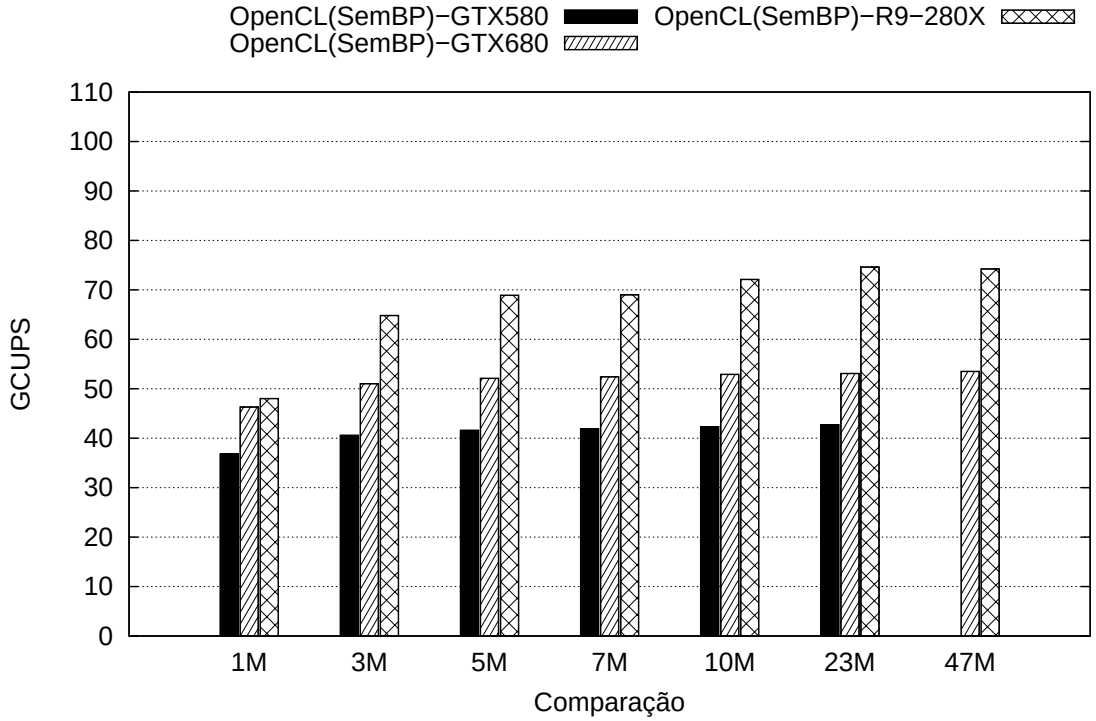


Figura 6.7: Resultado de testes da solução MASA-OpenCL em GPUs sem BP

Comp.	Ganho BP (%)					
	NVidia 580		NVidia 680		AMD R9	
	CUDAlign 128T	OpenCL 128T	CUDAlign 128T	OpenCL 128T	OpenCL 128T	OpenCL 256T
1M	17,1%	15,2%	**18,4%**	18,1%	7,0%	17,7%
3M	3,6%	3,6%	3,7%	3,6%	0,2%	**5,4%**
5M	49,1%	49,6%	51,0%	**51,2%**	34,8%	46,9%
7M	1,5%	1,4%	2,5%	1,1%	0,3%	**4,8%**
10M	48,4%	48,9%	**50,3%**	**50,3%**	35,2%	46,0%
23M	0,5%	0,5%	0,6%	0,0%	0,1%	**0,9%**
47M	31,0%	n.d.	**32,9%**	32,5%	23,8%	29,7%

Tabela 6.13: Ganho gerado pela técnica de BP em GPUs

Como pode ser observado na Figura 6.8, que compara o ganho com a técnica de BP nas execuções do MASA-OpenCL, o percentual de blocos descartados é semelhante entre as placas gráficas Nvidia utilizadas nos testes. Ademais, o ganho obtido na GPU AMD R9 280X utilizando 128 *threads* é inferior aos resultados alcançados nas GPUs NVidia, só sendo superior quando 256 *threads* são usadas durante a execução na GPU AMD. O resultado confirma que a arquitetura da GPU R9 280X é melhor explorada quando esta quantidade maior de *threads* é utilizada.

Os resultados obtidos com a técnica de *Block Pruning* na solução MASA-OpenCL são comparáveis aos valores alcançados com a solução MASA-CUDAlign, ratifi-

cando a relevância desta técnica na otimização do processamento da matriz de programação dinâmica.

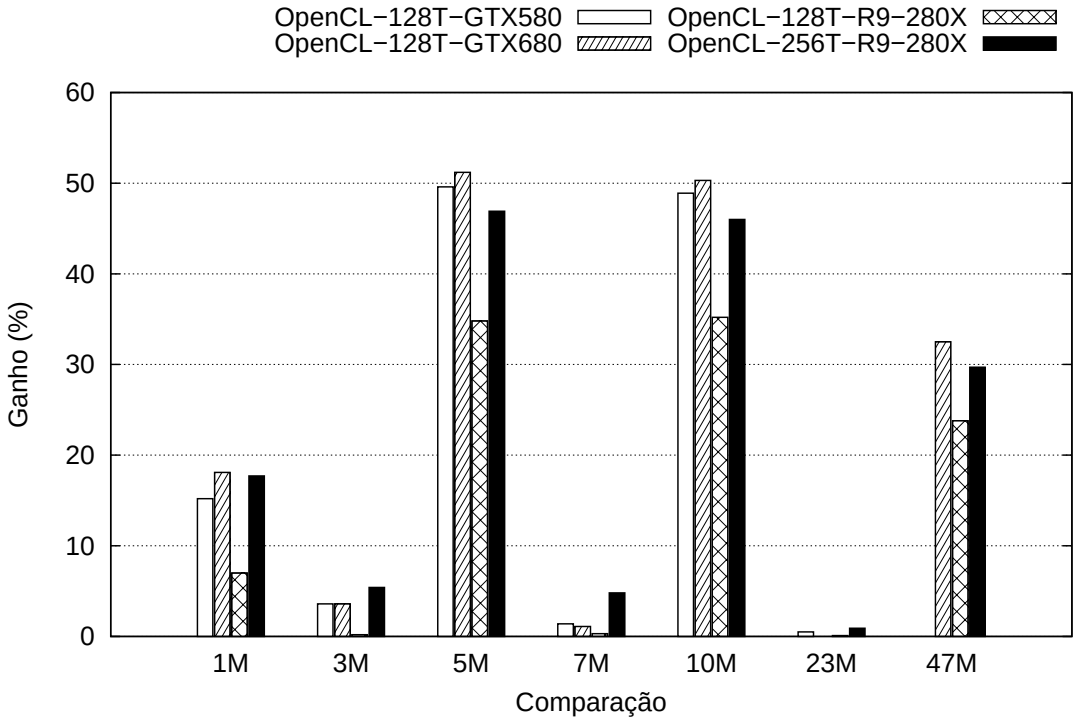


Figura 6.8: Ganho de desempenho obtido com a técnica de BP no MASA-OpenCL

Capítulo 7

Conclusão e Trabalhos Futuros

7.1 Conclusão

Nesta Dissertação, foi proposto e avaliado o MASA-OpenCL, uma solução de comparação exata de sequências longas de DNA em GPU. O MASA-OpenCL foi desenvolvido com base no *framework* MASA (Seção 4.4.1) e na linguagem OpenCL (Seção 3.2.2), retornando o escore ótimo e as coordenadas na matriz de programação dinâmica construída. Apesar de voltada primordialmente para execução em GPUs de diferentes fabricantes, a solução permite a sua execução em ambientes heterogêneos, tendo sido testada com sucesso em CPUs.

As principais contribuições introduzidas pela solução CUDAlign (Seção 4.4) - tais como a delegação de células, processamento em *wavefront* e técnica de descarte de blocos (ou *Block Pruning* - BP) - foram incorporadas ao MASA-OpenCL, contribuindo para o bom desempenho. Em especial, a técnica de BP pôde ser avaliada detalhadamente na solução proposta, apresentando comportamento similar ao obtido originalmente.

Durante a fase de projeto (Capítulo 5), alguns desafios inerentes ao processo de portar um código para uma linguagem voltada a plataformas heterogêneas foram enfrentados, como a reestruturação das funções para permitir a compilação em tempo de execução do OpenCL e o uso de barreiras globais de sincronização. Adicionalmente, características impostas pela linguagem OpenCL tiveram que ser incorporadas na solução, como o uso da memória global para armazenamento das sequências, em detrimento do uso da área de textura.

Durante a fase de testes (Capítulo 6), a solução pôde ser executada em CPUs de diferentes modelos e fabricantes, obtendo, em geral, desempenho superior (pico de 5,0 GCUPS) se comparado a outra solução que realiza a comparação de sequências nessa plataforma - o MASA-OpenMP (Seção 4.4.1). Nos testes realizados em GPUs NVidia e AMD, o MASA-OpenCL pôde ser comparado com outras duas soluções implementadas em CUDA - o MASA-CUDAlign (Seção 4.4.1) e o SW# (Seção 4.5). Exceto por alguns casos de teste com sequências menores, o MASA-OpenCL apresentou melhor desempenho, com ganhos de até 35%, possuindo a vantagem adicional de permitir a execução em GPUs fabricadas pela AMD. O relevante ganho de desempenho da solução MASA-OpenCL em relação ao MASA-CUDAlign motivaram uma pesquisa detalhada, sendo identificados alguns aspectos que contribuí-

ram para esta melhoria, tais como a compilação em 32 bits e o armazenamento das sequências em memória global (em vez da área de memória de textura). Estes aspectos foram também avaliados na execução do MASA-CUDAlign em GPUs NVidia, melhorando, de fato, o desempenho em relação à versão original.

O MASA-OpenCL obteve um máximo desempenho de 179,2 GCUPS na GPU AMD Radeon R9 280X, utilizando-se 256 *threads* e 512 blocos. Pelo que temos conhecimento, este é o melhor resultado reportado para a comparação de sequências longas em uma única GPU. Outrossim, não encontramos artigos que apresentem outras soluções que realizam a comparação de sequências longas de DNA (acima de 30 milhões de pares de bases) desenvolvidas em OpenCL, ratificando a relevância do MASA-OpenCL.

7.2 Trabalhos Futuros

A versão implementada do MASA-OpenCL realiza a comparação de duas sequências longas de DNA, retornando apenas o escore ótimo que representa a similaridade entre elas. Entre os trabalhos futuros, espera-se desenvolver uma solução que informe também o alinhamento ótimo entre as duas sequências, ou mesmo todos os alinhamentos possíveis com escore acima de um valor pré-determinado. Para atingir este objetivo, os demais estágios previstos no *framework* MASA [19] deverão ser implementados utilizando a linguagem OpenCL, mantendo a solução portável para diferentes plataformas.

Adicionalmente, é desejável que a solução MASA-OpenCL possa ser executada em múltiplas GPUs, aumentando o paralelismo e o desempenho da execução. Esta funcionalidade já é prevista por outras soluções de alinhamento de sequências [15] [16], contudo elas são baseadas em uma linguagem específica para execução em GPUs NVidia (CUDA), não sendo opções viáveis para plataformas heterogêneas. Para tanto, as funções *kernel* OpenCL devem ser recompiladas em tempo de execução a cada novo dispositivo selecionado, armazenando os resultados parciais para que possam ser processados em múltiplas GPUs.

Outrossim, planeja-se a execução do MASA-OpenCL em outros dispositivos que ofereçam *drivers* OpenCL, tais como FPGA e co-processador Intel Phi, avaliando-se a portabilidade e desempenho da solução nestes ambientes. Possíveis otimizações de código também devem ser analisadas para explorar as características dessas plataformas de *hardware*.

Finalmente, pretende-se executar o MASA-OpenCL em plataformas híbridas, utilizando a característica intrínseca do OpenCL de permitir a identificação de todas as unidades de processamento existentes. Para tanto, o procedimento de seleção do dispositivo de execução deve ser modificado para contemplar este requisito, realizando a distribuição do trabalho de acordo com a capacidade de processamento disponível.

Referências

[1] Gautam B Singh. *Introduction to Bioinformatics*. Springer, 2015. 1

[2] Francis S Collins, Michael Morgan, and Aristides Patrinos. The human genome project: lessons from large-scale biology. *Science*, 300(5617):286–290, 2003. 1

[3] Nicholas M. Luscombe, Dov Greenbaum, and Mark Gerstein. What is bioinformatics? a proposed definition and overview of the field. *Methods of information in medicine*, 40(4):346–358, 2001. 1, 5

[4] David W. Mount. Sequence and genome analysis. *New York: Cold Spring*, 2004. 1, 5, 6, 7

[5] Andreas D. Baxevanis and BF Francis Ouellette. Bioinformatics: a practical guide to the analysis of genes and proteins. 2004. 1, 5, 6

[6] T. F. Smith and M. S. Waterman. Identification of common molecular subsequences. *J Mol Biol*, 147(1):195–197, March 1981. 1, 8

[7] David Luebke, Mark Harris, Naga Govindaraju, Aaron Lefohn, Mike Houston, John Owens, Mark Segal, Matthew Papakipos, and Ian 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. 2, 15

[8] Svetlin A Manavski and Giorgio Valle. Cuda compatible gpu cards as efficient hardware accelerators for smith-waterman sequence alignment. *BMC bioinformatics*, 9(Suppl 2):S10, 2008. 3, 28

[9] Lukasz Ligowski and Witold 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, 29

[10] Yongchao Liu, Douglas L Maskell, and Bertil Schmidt. Cudasw++: optimizing smith-waterman sequence database searches for cuda-enabled graphics processing units. *BMC research notes*, 2(1):73, 2009. 3, 29

[11] Yongchao Liu, Bertil Schmidt, and Douglas 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. 3, 30

[12] Yongchao Liu, Adrianto Wirawan, and Bertil Schmidt. Cudasw++ 3.0: accelerating smith-waterman protein database search by coupling cpu and gpu simd instructions. *BMC bioinformatics*, 14(1):117, 2013. 3, 30

[13] Edans F. O. Sandes and Alba Cristina de Melo. Cudalign: using gpu to accelerate the comparison of megabase genomic sequences. In *ACM Sigplan Notices*, volume 45, pages 137–146. ACM, 2010. 3, 30

[14] Edans F. O. Sandes and Alba Cristina 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. 3, 31, 32, 33

[15] Edans F. O. Sandes, Guillermo Miranda, Alba Cristina de Melo, Xavier Martorell, and Eduard 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. 3, 33, 71

[16] Matija Korpar and Mile Šikić. Sw#–gpu-enabled exact alignments on genome scale. *Bioinformatics*, 29(19):2494–2495, 2013. 3, 34, 35, 71

[17] Ayman Khalafallah, Hosain Fath 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, 35

[18] Dzmitry Razmyslovich, Guillermo Marcus, Markus Gipp, Marc Zapatka, and Andreas 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, 36

[19] Edans F. O. Sandes, Guillermo Miranda, Xavier Martorell, Eduard Ayguade, George T. Medeiros, and Alba Cristina de Melo. Masa: a multi-platform architecture for sequence aligners with block pruning. In *ACM Transation on Parallel Processing, Submitted*. Euro-Par, 2014. 3, 33, 34, 54, 56, 57, 67, 71

[20] Gonzalo Giribet and Ward C. Wheeler. On gaps. *Molecular phylogenetics and evolution*, 13(1):132–143, 1999. 5

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

[22] Richard Durbin, Sean R. Eddy, Anders Krogh, and Graeme Mithison. *Biological sequence analysis: probabilistic models of proteins and nucleic acids*. Cambridge University Press, 1998. 6

[23] Robert C. Edgar and Serafim Batzoglou. Multiple sequence alignment. *Current opinion in structural biology*, 16(3):368–373, 2006. 6

[24] J. Craig Venter, Mark D. Adams, Eugene W. Myers, Peter W. Li, Richard J. Mural, Granger G. Sutton, Hamilton O. Smith, Mark Yandell, Cheryl A. Evans, Robert A. Holt, et al. The sequence of the human genome. *science*, 291(5507):1304–1351, 2001. 6

[25] Thomas H. Cormen. *Algoritmos: teoria e prática*. Elsevier, 2nd edition, 2002. 7

[26] Saul B. Needleman and Christian 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. 7

[27] Osamu Gotoh. An improved algorithm for matching biological sequences. *J Mol Biol*, 162(3):705–708, December 1982. 10

[28] Eugene W. Myers and Webb Miller. Optimal alignments in linear space. *Comput. Appl. Biosci.*, 4(1):11–17, March 1988. 11

[29] Daniel S. Hirschberg. A linear space algorithm for computing maximal common subsequences. *Communications of the ACM*, 18(6):341–343, 1975. 11

[30] James W. Fickett. Fast optimal alignment. *Nucleic acids research*, 12(1Part1):175–179, 1984. 11

[31] Edans F. O. Sandes. Comparação paralela de sequências biológicas longas utilizando unidades de processamento gráfico (gpus). Master's thesis, Universidade de Brasília, 2012. 12

[32] David E Culler, Jaswinder Pal Singh, and Anoop Gupta. *Parallel computer architecture: a hardware/software approach*. Morgan Kaufmann, 1999. 13

[33] Michael Flynn. Some computer organizations and their effectiveness. *Computers, IEEE Transactions on*, 100(9):948–960, 1972. 13

[34] Gregory F. Pfister. *In search of clusters*. Prentice-Hall, Inc., 1998. 13

[35] Greg Goth. Ibm pc retrospective: There was enough right to make it work. *Computer*, 44(8):26–33, 2011. 15

[36] Richard F. Ferraro. *Programmer's Guide to the EGA, VGA, and Super VGA Cards*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 3rd edition, 1994. 15

[37] Ying Zhang, Lu Peng, Bin Li, Jih-Kwon Peir, and Jianmin Chen. Architecture comparisons between nvidia and ati gpus: Computation parallelism and data communications. In *Workload Characterization (IISWC), 2011 IEEE International Symposium on*, pages 205–215. IEEE, 2011. 16

[38] AMD. Hd 6900 series instruction set architecture. Nov 2011. 16

[39] AMD. Southern islands series instruction set architecture. Dec 2012. 16

[40] AMD. Sea islands series instruction set architecture. Jul 2013. 17

[41] AMD. Graphics core next architecture, generation 3. Mar 2015. 17

[42] David A. Patterson and John L. Hennessy. *Computer organization and design: the hardware/software interface*. Elsevier, 2014. 19

[43] NVidia. Nvidia tesla gpu computing technical brief. May 2007. 19

[44] NVidia. Nvidia tesla k40 gpu acelerator board specification. Jul 2013. 19

[45] NVidia. Nvidia fermi compute architecture whitepaper. Jul 2009. 19

[46] NVidia. Nvidia kepler gk110 architecture whitepaper. Jul 2012. 20

[47] Ryan Smith. Nvidia updates gpu roadmap; unveils pascal architecture for 2016. `http://www.anandtech.com/show/7900/NVidia-updates-gpu-roadmap-unveils-pascal-architecture-for-2016`, May 2014. 20

[48] Abbas Rahimi, Amirali Ghofrani, Miguel Angel, Kwang-Ting Cheng, Luca Benini, and Rajesh K Gupta. Energy-efficient gpgpu architectures via collaborative compilation and memristive memory-based computing. 2014. 20

[49] Amir Banari, Christian Janßen, Stephan T Grilli, and Manfred Krafczyk. Efficient gpgpu implementation of a lattice boltzmann model for multiphase flows with high density ratios. *Computers & Fluids*, 2014. 20

[50] Adwait Jog, Evgeny Bolotin, Zvika Guz, Mike Parker, Stephen W Keckler, Mahmut T Kandemir, and Chita R Das. Application-aware memory system for fair and efficient execution of concurrent gpgpu applications. In *Proceedings of Workshop on General Purpose Processing Using GPUs*, page 1. ACM, 2014. 20

[51] NVidia. Cuda c programming guide. *NVidia Corporation*, Feb 2014. 21

[52] John E Stone, David Gohara, and Guochun Shi. Opencl: A parallel programming standard for heterogeneous computing systems. *Computing in science & engineering*, 12(3):66, 2010. 21

[53] Alejandro Duran, Eduard Ayguadé, Rosa M badia, Jesús Labarta, Luis martinell, Xavier Martorell, and Judit Planas. Ompss: a proposal for programming heterogeneous multi-core architectures. *Parallel Processing Letters*, 21(2):173–193, 2011. 21

[54] Amr Bayoumi, Michael Chu, Yasser Hanafy, Patricia Harrell, and Gamal Refai-Ahmed. Scientific and engineering computing using ati stream technology. *Computing in Science & Engineering*, 11(6):92–97, 2009. 21

[55] AMD Accelerated Parallel Processing. Opencl programming guide. Nov 2013. 21

[56] Jason Sanders and Edward Kandrot. *CUDA by example: an introduction to general-purpose GPU programming*. Addison-Wesley Professional, 2010. 21

[57] NVidia. Nvidia cuda architecture. Apr 2009. 22

[58] NVidia. Paralell thread execution isa - application guide. Feb 2014. 21

[59] NVidia. Cuda c programing guide v6.0. Feb 2014. 21, 22, 23, 26

[60] Khronos group - opencl. `http://www.khronos.org/opencl/`, May 2014. 23, 44

[61] Aaftab Munshi, Benedict Gaster, Timothy G Mattson, and Dan Ginsburg. *OpenCL programming guide*. Pearson Education, 2012. 23, 24, 27

[62] CMSoft. Opencl tutorial, 2014, accessed Feb 10, 2015. 26

[63] Torbjørn Rognes and Erling Seeberg. Six-fold speed-up of smith–waterman sequence database searches using parallel processing on common microprocessors. *Bioinformatics*, 16(8):699–706, 2000. 28

[64] Michael Farrar. Striped smith–waterman speeds database searches six times over other simd implementations. *Bioinformatics*, 23(2):156–161, 2007. 28, 30

[65] Torbjørn Rognes. Faster smith-waterman database searches with inter-sequence simd parallelisation. *BMC bioinformatics*, 12(1):221, 2011. 28

[66] Jaideeep Singh and Ipseeta 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. 28

[67] M Affan Zidan, Talal Bonny, and Khaled N Salama. High performance technique for database applications using a hybrid gpu/cpu platform. In *Proceedings of the 21st edition of the great lakes symposium on Great lakes symposium on VLSI*, pages 85–90. ACM, 2011. 28

[68] Yongchao Liu, Adrianto Wirawan, and Bertil Schmidt. Cudasw++ 3.0: accelerating smith-waterman protein database search by coupling cpu and gpu simd instructions. *BMC bioinformatics*, 14(1):117, 2013. 28

[69] Edans F. O. Sandes and Alba Cristina 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. 31

[70] Leonardo Dagum and Ramesh Menon. Openmp: an industry standard api for shared-memory programming. *Computational Science & Engineering, IEEE*, 5(1):46–55, 1998. 34, 57

[71] James Jeffers and James Reinders. *Intel Xeon Phi coprocessor high-performance programming*. Newnes, 2013. 34

[72] Alejandro Duran, Eduard Ayguadé, Rosa M Badia, Jesús Labarta, Luis Martinell, Xavier Martorell, and Judit Planas. Ompss: a proposal for programming heterogeneous multi-core architectures. *Parallel Processing Letters*, 21(02):173–193, 2011. 34

[73] Robert J Francis, Jonathan Rose, and Zvonko G Vranesic. *Field-programmable gate arrays*, volume 180. Springer, 1992. 39

[74] Kamran Karimi, Neil G Dickson, and Firas Hamze. A performance comparison of cuda and opencl. *unpublished*, 2010. 39

[75] Jianbin Fang, Ana Lucia Varbanescu, and Henk Sips. A comprehensive performance comparison of cuda and opencl. In *Parallel Processing (ICPP), 2011 International Conference on*, pages 216–225. IEEE, 2011. 39, 60

[76] Matt J Harvey and Gianni De Fabritiis. Swan: A tool for porting cuda programs to opencl. *Computer Physics Communications*, 182(4):1093–1099, 2011. 43

[77] Gabriel Martinez, Mark Gardner, and Wu-chun Feng. Cu2cl: A cuda-to-opencl translator for multi-and many-core architectures. In *Parallel and Distributed Systems (ICPADS), 2011 IEEE 17th International Conference on*, pages 300–307. IEEE, 2011. 43

[78] Chris Lattner. Llvm and clang: Next generation compiler technology. In *The BSD Conference*, pages 1–2, 2008. 43

[79] Paul Sathre, Mark Gardner, and Wu-chun Feng. Lost in translation: Challenges in automating cuda-to-opencl translation. In *Parallel Processing Workshops (ICPPW), 2012 41st International Conference on*, pages 89–96. IEEE, 2012. 43

[80] WEN Mei, Da-fei HUANG, Chang-qing XUN, and CHEN Dong. Improving performance portability for gpu-specific opencl kernels on multi-core/many-core cpus by analysis-based transformations. *Frontiers*, 1. 48

[81] AMD. Amd codexl quick start guide. 2014. 49

[82] Naoya Maruyama and Takayuki Aoki. Optimizing stencil computations for nvidia kepler gpus. In *Proceedings of the 1st International Workshop on High-Performance Stencil Computations, Vienna*, pages 89–95, 2014. 62