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

MASA-SSE: Comparação de Sequências Biológicas Utilizando Instruções Vetoriais

Phillipe G. Ferreira

Monografia apresentada como requisito parcial
para conclusão do Curso de Engenharia da Computação

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
Curso de Engenharia da Computação

Coordenador: Prof. Dr. Ricardo Zelenovsky

Banca examinadora composta por:

Prof.ª Dr.ª Alba Cristina M. A. de Melo (Orientadora) — CIC/UnB
Prof. Dr. George Luiz Medeiros Teodoro — CIC/UnB
Prof. Dr. Pedro de Azevedo Berger — CIC/UnB

CIP — Catalogação Internacional na Publicação

Ferreira, Phillipe G..

MASA-SSE: Comparação de Sequências Biológicas Utilizando Instruções Vetoriais / Phillipe G. Ferreira. Brasília : UnB, 2015.
47 p. : il. ; 29,5 cm.

Monografia (Graduação) — Universidade de Brasília, Brasília, 2015.

1. Comparação de Sequências Biológicas, 2. Programação Paralela,
3. Instruções Vetoriais

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-SSE: Comparação de Sequências Biológicas Utilizando Instruções Vetoriais

Phillipe G. Ferreira

Monografia apresentada como requisito parcial
para conclusão do Curso de Engenharia da Computação

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

Prof. Dr. George Luiz Medeiros Teodoro
CIC/UnB

Prof. Dr. Pedro de Azevedo Berger
CIC/UnB

Prof. Dr. Ricardo Zelenovsky
Coordenador do Curso de Engenharia da Computação

Brasília, 1 de dezembro de 2015

Dedicatória

Dedico esse trabalho de graduação ao meu falecido pai. Pai, esse trabalho é pra você!

Agradecimentos

Agradeço à todos que me ajudaram a chegar até aqui, especialmente à minha orientadora, Alba, que esteve comigo do início ao fim deste trabalho, sempre disposta a ajudar no que fosse preciso.

Resumo

A comparação de sequências biológicas é uma das operações mais básicas e importantes da Bioinformática. Os métodos exatos de comparação de sequências possuem complexidade quadrática de tempo e por isso soluções paralelas são utilizadas para acelerar a produção de resultados. O *framework* MASA [3] é uma solução paralela flexível e customizável que permite o alinhamento de sequências biológicas em diferentes *hardwares* e *softwares*. Ele foi inicialmente pensado para execução paralela da comparação de sequências em GPUs (*Graphics Processing Units*), porém, atualmente existem duas soluções MASA para CPU: MASA-CPU e MASA-OpenMP. Essas soluções não utilizam instruções vetoriais, deixando de explorar um grande potencial para paralelismo. O presente trabalho de graduação propõe e avalia o MASA-SSE, uma solução em CPU que utiliza as instruções vetoriais SSE da Intel, implementando o algoritmo de Farrar [6], que é considerado o *estado da arte* em comparação de sequências biológicas com instruções vetoriais. Os resultados obtidos a partir da comparação de várias sequências reais de DNA em duas máquinas distintas mostram que o MASA-SSE, executando em uma *thread* e, utilizando instruções vetoriais, possui desempenho superior ao do MASA-OpenMP com quatro *threads*.

Palavras-chave: Comparação de Sequências Biológicas, Programação Paralela, Instruções Vetoriais

Abstract

Biological sequence comparison is one of the most basic and important operations in Bioinformatics. The exact methods that compare two biological sequences have quadratic time complexity and, for this reason, parallel solutions are often used to accelerate the execution. The MASA framework [3] is a flexible and customizable parallel solution for biological sequence comparison which was initially designed for GPU (Graphics Processing Unit) execution but nowadays integrates two CPU solutions: MASA-CPU and MASA-OpenMP. These CPU solutions do not use vector instructions and thus miss the opportunity of exploring a high potential for parallelism. This graduation project proposes and evaluates MASA-SSE, a CPU solution that uses the SSE vector instructions from Intel and implements the Farrar algorithm [6], which is the *state-of-the-art* algorithm for biological sequence comparison with vector instructions. Experimental results obtained with the comparison of real DNA sequences in two different machines show that MASA-SSE, executing with one thread and vector instructions, outperforms MASA-OpemMP, execution with four threads.

Keywords: Biological Sequence Comparison, Parallel Computing, Vector Instructions

Sumário

1 Introdução **1**

2 Comparação de Sequências Biológicas **3**
2.1 Tipos de Sequências Biológicas 3
2.2 Alinhamento de Sequências 5
2.2.1 Alinhamento Global e Alinhamento Local 5
2.3 O Modelo de Escore 6
2.3.1 Modelos de *Gaps* 7
2.4 Algoritmos Exatos de Comparação de Sequências 8
2.4.1 Needleman-Wunsh 8
2.4.2 Smith-Waterman 10
2.4.3 Gotoh 12

3 Comparação de Sequências Biológicas com Instruções Vetoriais **13**
3.1 Modelo SIMD 13
3.2 Intel SSE e AVX 14
3.3 Comparação de Sequências Biológicas com Instruções Vetoriais 15
3.3.1 Wozniak 15
3.3.2 Rognes e Seeberg 16
3.3.3 Farrar 17
3.4 Quadro Comparativo 18

4 MASA (*Mullti-Platform Architecture for Sequence Aligners*) **20**
4.1 Arquitetura MASA 20
4.1.1 MASA-API 21
4.2 Implementações do MASA 22

5 Projeto do MASA-SSE **24**
5.1 Visão Geral 24

6 Resultados Experimentais **28**
6.1 Testes Realizados 28
6.2 Análise dos Resultados 29

7 Conclusão e Trabalhos Futuros **33**

Referências **35**

Lista de Figuras

2.1 Estrutura do DNA [18]. 4
2.2 Alinhamento entre duas sequências de DNA. (a) Alinhamento global. (b) Alinhamento local. 6
2.3 Matriz BLOSUM50 [5]. 7
2.4 Alinhamento local de duas amostras do vírus Varióla com a inserção de um *gap* no início da segunda subsequência. 8
2.5 Primeiro passo do alinhamento global entre u e v. 9
2.6 Alinhamento em H(1,1). 9
2.7 Alinhamento em H(1,2) com ocorrência de um *gap*. 10
2.8 Matriz de similaridade para o alinhamento entre u e v utilizando o algoritmo de Needleman-Wunsch. 10
2.9 Matriz de similaridade para o alinhamento entre u e v utilizando o algoritmo de Smith-Waterman. 11

3.1 Arquitetura SISD e arquitetura SIMD [14]. 14
3.2 Exemplo de programa que faz o uso da extensão AVX2. 15
3.3 Distribuição das células nos vetores em implementações SIMD do algoritmo Smith-Waterman. (a) células distribuídas pela menor diagonal. (b) Células distribuídas em paralelo com a *query sequence* [20]. 17
3.4 Comparação entre a abordagem de Farrar (c) e as demais abordagens (a e b) para o algoritmo de Smith-Waterman paralelizado [19]. 18

4.1 Arquitetura MASA [3]. 21
4.2 Diagrama de classes do MASA [3]. 23

5.1 Arquitetura do MASA-SSE. 25
5.2 Pseudo-código do processamento vetorial do MASA-SSE. 27

Lista de Tabelas

3.1 Quadro comparativo entre as implementações do algoritmo Smith-Waterman para instruções vetoriais. 19

6.1 Sequências de referência. 28
6.2 Quadro comparativo dos resultados com *block pruning* na máquina 1. . . . 29
6.3 Quadro comparativo dos resultados sem *block pruning* na máquina 1. . . . 30
6.4 Quadro comparativo dos resultados com *block pruning* na máquina 2. . . . 30
6.5 Quadro comparativo dos resultados sem *block pruning* na máquina 2. . . . 31

Capítulo 1

Introdução

A Bioinformática é uma área de pesquisa que vem crescendo bastante desde seu surgimento, em meados do século XX, englobando vários aspectos da biologia com modelos matemáticos e computacionais com o intuito de facilitar o estudo dos dados biológicos [25].

A comparação de sequências biológicas é uma das operações básicas na Bioinformática. O resultado da comparação entre duas sequências é dado por uma medida do grau de semelhança conhecida como escore [12]. Normalmente, junto à comparação, também é feito o alinhamento das sequências, com o objetivo de ressaltar suas similaridades e diferenças [5].

Algoritmos exatos para comparação de sequências, como o Smith-Waterman [23], usam programação dinâmica e possuem complexidade de tempo O(nm), onde n e m são os tamanhos das sequências. Devido à grande precisão dos resultados obtidos com algoritmos exatos, pois os cálculos são realizados para todas as células das sequências, o tempo de execução e a quantidade de recursos computacionais necessários podem ser muito altos. Algumas soluções heurísticas foram propostas para solucionar esse problema, mas não serão aprofundadas aqui, pois o foco desse trabalho é em soluções paralelas.

Várias soluções paralelas foram propostas para reduzir o tempo de execução da comparação exata de sequências biológicas. Dentre essas soluções destacam-se o algoritmo de Farrar [6] e o *framework* MASA (*Mullti-Platform Architecture for Sequence Aligners*) [3].

O algoritmo de Farrar é considerado até o presente momento o *estado da arte* entre os algoritmos paralelos de comparação de sequências que usam instruções vetoriais. Além de instruções vetoriais, o algoritmo aplica diversas otimizações visando o ganho de desempenho durante a comparação de sequências biológicas.

O *framework* MASA é uma arquitetura de *software* flexível e customizável que permite a comparação e o alinhamento de sequências biológicas em diferentes plataformas de

hardware e *software*. A sua flexibilidade e facilidade de customização facilita a criação e integração de novos módulos.

O presente trabalho de graduação tem por objetivo propor e avaliar o MASA-SSE, uma solução que integra o algoritmo de Farrar ao *framework* MASA. Na nossa solução, mantivemos a grande maioria das características do algoritmo de Farrar e todas as características do MASA, com exceção do *block pruning* (uma otimização capaz de reduzir a quantidade de dados processados durante a comparação das sequências), devido a grande complexidade para tratar a relação de dependência entre os blocos processador pelo MASA.

Os resultados experimentais obtidos em duas máquinas distintas com sequências de 1K, 10K, 20K, 30K, 40K e 50K mostram que o MASA-SSE executando em uma única *thread* possui desempenho superior à extensão MASA-OpenMP com *block pruning* e executando em 4 *threads*.

O restante deste documento está organizado como se segue. O Capítulo 2 apresenta o problema da comparação de sequências biológicas e algoritmos exatos que são usados para resolvê-lo. O Capítulo 3 traz uma visão geral sobre as arquiteturas vetoriais e algoritmos que as utilizam para comparar sequências. O Capítulo 4 é apresentado o *framework* MASA. O Capítulo 5 descreve o projeto do MASA-SSE. No Capítulo 6 são descritos os testes realizados para medir o desempenho da solução proposta. Finalmente, o Capítulo 7 apresenta a conclusão e sugere trabalhos futuros.

Capítulo 2

Comparação de Sequências Biológicas

Uma tarefa básica da Bioinformática é comparar duas sequências biológicas de maneira a dizer quão parecidas são essas sequências. O resultado dessa comparação é dado através de uma pontuação total (escore) e pode variar de acordo com o processo utilizado para fazer a comparação. Quanto maior o escore, mais próximo na cadeia evolutiva estão as sequências comparadas [12].

Inicialmente, na seção 2.1 são apresentados alguns conceitos básicos sobre DNA, RNA e proteína, em seguida, na seção 2.2 é definido o que é o alinhamento de sequências. Na seção 2.3 são apresentadas as maneiras utilizadas para calcular o escore do alinhamento e, por fim, na seção 2.4 são apresentados alguns algoritmos de comparação de sequências biológicas.

2.1 Tipos de Sequências Biológicas

O composto orgânico que carrega as características hereditárias de todos os seres vivos e de alguns vírus é conhecido como ácido desoxirribonucleico, ou DNA (*deoxyribonucleic acid*). O DNA consiste em um par de moléculas helicoidais ligadas entre si, formando uma dupla hélice. Cada uma das hélices é composta por vários nucleotídeos, que consistem em bases nitrogenadas ligadas com uma pentose (molécula de açúcar) e um fosfato. [5]

As bases nitrogenadas são classificadas em dois grupos: as purinas, onde se encaixam a adenina (A) e a guanina (G) e as pirimidinas, citosina (C) e timina (T). Essas bases se ligam através de pontes de hidrogênio para formar o DNA, assim, a base púrica adenina se liga com a base pirimídica timina, da mesma forma que a guanina se liga com a citosina. Assim, se em uma fita de DNA têm-se uma adenina, na outra, nessa mesma posição, haverá uma timina, por exemplo.

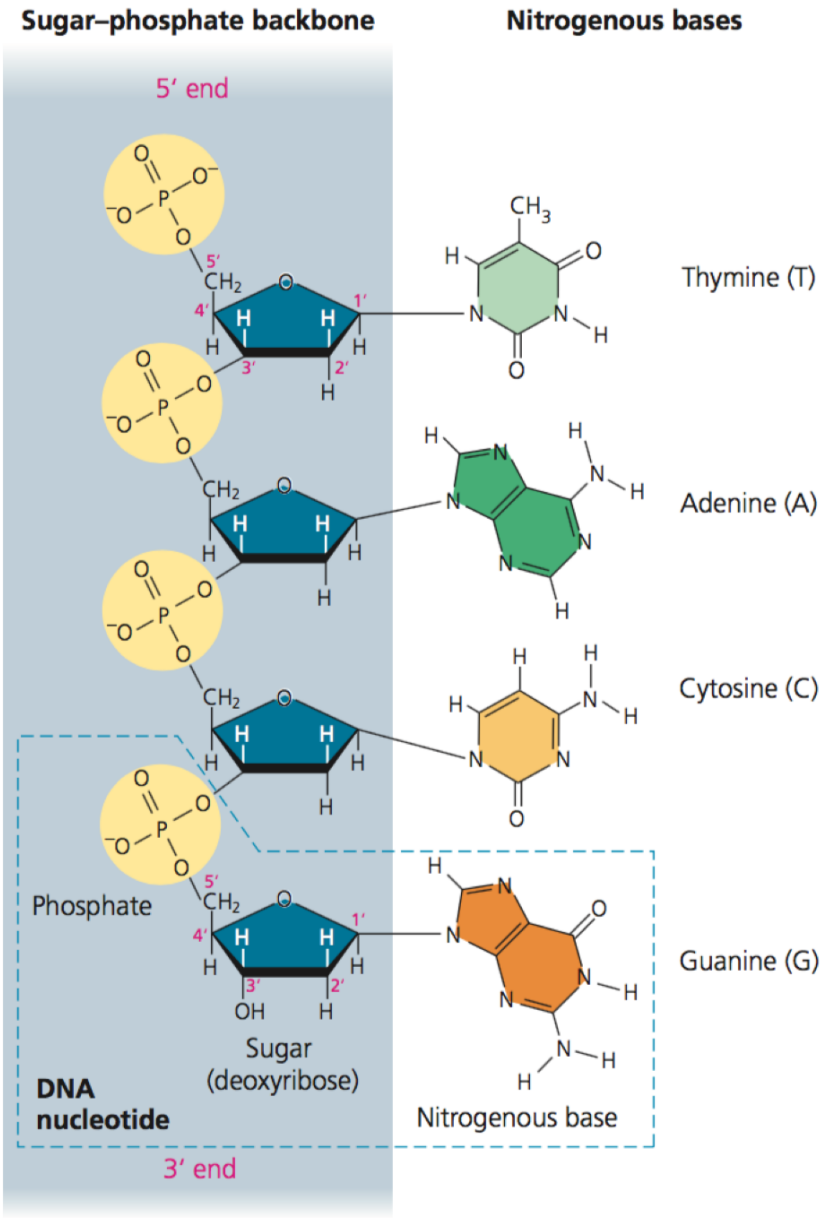

Figura 2.1: Estrutura do DNA [18].

O ácido ribonucleico, ou RNA (*ribonucleic acid*), possui a sua composição parecida com a do DNA, porém, é formado por um filamento simples de nucletídeos, de maneira que as bases se pareiam umas com as outras no mesmo filamento. A pirimidina uracila (U) está presente no lugar da timina (presente no DNA). Outra diferença estrutural é que o RNA possui o açúcar ribose em seus nucletídeos ao invés da desoxirribose que está presente nos nucleotídeos do DNA.

O RNA é responsável pela síntese de proteínas na célula. O processo de produção do RNA a partir do DNA é conhecido como transcrição e a síntese de proteínas é conhecida

como tradução e é feitas a partir de comandos do RNA.

Vários tipos de RNA são produzidos a partir do DNA, de maneira mais simples: o RNA ribossômico, que participa na produção de ribossomos; o RNA transportador, que transporta aminoácidos até os ribossomos para produção de proteínas; e o RNA mensageiro, que possui as informações responsáveis pela síntese das proteínas.

As proteínas estão presentes em todos os seres vivos e são constituídas por cadeias de aminoácidos. Até 20 tipos diferentes de aminoácidos podem ser encontrados nas sequências e eles se ligam através de ligações peptídicas.

A proteína é fundamental na vida dos seres vivos e sua função está ligada diretamente a sua forma e qualquer erro em sua estrutura ou síntese pode desencadear distúrbios metabólicos no ser vivo.

2.2 Alinhamento de Sequências

O alinhamento de sequências é definido como uma forma de comparar duas sequências de DNA, RNA ou proteína de modo a identificar similaridades presentes em ambas as sequências. Define-se similaridade neste contexto como sequências significantes de bases nitrogenadas ou aminoácidos que existem nas duas sequências comparadas, de maneira a verificar mudanças evolutivas e relações estruturais ou funcionais entre as sequências biológicas [5].

2.2.1 Alinhamento Global e Alinhamento Local

Alinhar duas sequências biológicas na prática significa comparar cada um dos resíduos presentes em ambas e encontrar equivalências entre as duas. Esse procedimento pode ser realizado de maneira global ou local.

Mais utilizado em sequências muito parecidas e com tamanhos próximos ou equivalentes, o alinhamento global consiste em alinhar todos os resíduos das duas sequências, de maneira a compará-las como um todo. A Figura 2.2a mostra mostra um exemplo de alinhamento global.

O alinhamento local consiste em comparar subsequências das sequências principais e é utilizado para comparar sequências que possuem geralmente tamanhos diferentes nas quais suspeita-se de que hajam partes similares. A Figura 2.2b mostra um possível alinhamento local para as mesmas sequências da Figura 2.2a.

Há ainda o alinhamento híbrido, ou alinhamento semi-global, no qual se procura o melhor alinhamento possível que inclua o começo e o fim de uma determinada sequência.

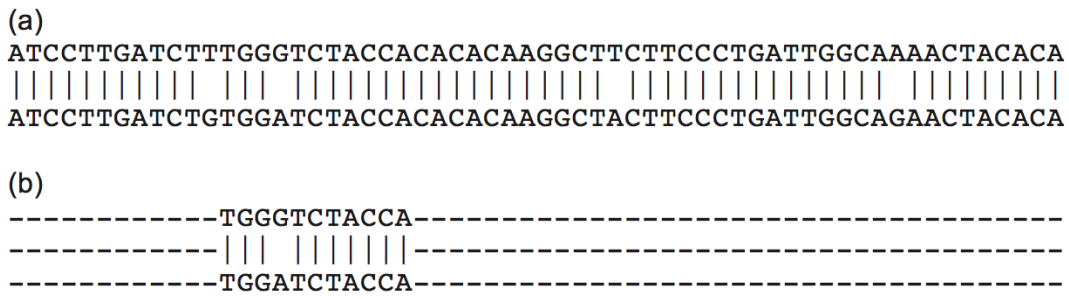

Figura 2.2: Alinhamento entre duas sequências de DNA. (a) Alinhamento global. (b) Alinhamento local.

2.3 O Modelo de Escore

As mutações dos genes nos seres vivos faz com que as sequências biológicas diferenciem-se umas das outras. É muito comum no processo de alinhamento adicionar ou remover resíduos para que se obtenha um escore mais preciso. Também leva-se em conta a substituição de resíduos, que ocorre quando por exemplo compara-se dois resíduos que se ligariam caso pertencessem a uma mesma molécula.

Para cada par alinhado é definido um valor e o escore total consiste na soma de cada par de resíduo alinhado mais as inserções e remoções de resíduos, também chamados de *gaps* [5].

Um dos desafios para determinar o grau de similaridade entre duas sequências é encontrar o melhor alinhamento entre ambas. Pareando os resíduos de cada sequência é possível verificar se o par é equivalente (*match*) ou se é diferente (*mismatch*) e assim atribuir uma determinada pontuação, que pode ser positiva ou negativa, para cada par de resíduo comparado.

Para determinar o valor exato de cada *match* e *mismatch* em comparação de proteínas, existem matrizes de substituição que foram propostas a partir de evidências biológicas e estudos baseados em comparação de proteínas intimamente relacionadas (matriz PAM) ou em alinhamentos locais observados entre proteínas (matriz BLOSUM). A Figura 2.3 mostra a matriz BLOSUM62, utilizada para o alinhamento de proteínas.

Para comparação de sequências de DNA, é mais comum utilizar um escore positivo para matches e um escore negativo para mismatches e gaps. Trocar duas bases púricas (adenina e guanina) ou duas pirimidinas (citosina e timina) são consideradas mutações mais comuns e são penalizadas menos do que trocar uma purina com uma pirimidina ou vice-versa [5].

	A	R	N	D	C	Q	E	G	H	I	L	K	M	F	P	S	T	W	Y	V
A	**5**	−2	−1	−2	−1	−1	−1	0	−2	−1	−2	−1	−1	−3	−1	1	0	−3	−2	0
R	−2	**7**	−1	−2	−4	1	0	−3	0	−4	−3	3	−2	−3	−3	−1	−1	−3	−1	−3
N	−1	−1	**7**	2	−2	0	0	0	1	−3	−4	0	−2	−4	−2	1	0	−4	−2	−3
D	−2	−2	2	**8**	−4	0	2	−1	−1	−4	−4	−1	−4	−5	−1	0	−1	−5	−3	−4
C	−1	−4	−2	−4	**13**	−3	−3	−3	−3	−2	−2	−3	−2	−2	−4	−1	−1	−5	−3	−1
Q	−1	1	0	0	−3	**7**	2	−2	1	−3	−2	2	0	−4	−1	0	−1	−1	−1	−3
E	−1	0	0	2	−3	2	**6**	−3	0	−4	−3	1	−2	−3	−1	−1	−1	−3	−2	−3
G	0	−3	0	−1	−3	−2	−3	**8**	−2	−4	−4	−2	−3	−4	−2	0	−2	−3	−3	−4
H	−2	0	1	−1	−3	1	0	−2	**10**	−4	−3	0	−1	−1	−2	−1	−2	−3	2	−4
I	−1	−4	−3	−4	−2	−3	−4	−4	−4	**5**	2	−3	2	0	−3	−3	−1	−3	−1	4
L	−2	−3	−4	−4	−2	−2	−3	−4	−3	2	**5**	−3	3	1	−4	−3	−1	−2	−1	1
K	−1	3	0	−1	−3	2	1	−2	0	−3	−3	**6**	−2	−4	−1	0	−1	−3	−2	−3
M	−1	−2	−2	−4	−2	0	−2	−3	−1	2	3	−2	**7**	0	−3	−2	−1	−1	0	1
F	−3	−3	−4	−5	−2	−4	−3	−4	−1	0	1	−4	0	**8**	−4	−3	−2	1	4	−1
P	−1	−3	−2	−1	−4	−1	−1	−2	−2	−3	−4	−1	−3	−4	**10**	−1	−1	−4	−3	−3
S	1	−1	1	0	−1	0	−1	0	−1	−3	−3	0	−2	−3	−1	**5**	2	−4	−2	−2
T	0	−1	0	−1	−1	−1	−1	−2	−2	−1	−1	−1	−1	−2	−1	2	**5**	−3	−2	0
W	−3	−3	−4	−5	−5	−1	−3	−3	−3	−3	−2	−3	−1	1	−4	−4	−3	**15**	2	−3
Y	−2	−1	−2	−3	−3	−1	−2	−3	2	−1	−1	−2	0	4	−3	−2	−2	2	**8**	−1
V	0	−3	−3	−4	−1	−3	−3	−4	−4	4	1	−3	1	−1	−3	−2	0	−3	−1	**5**

Figura 2.3: Matriz BLOSUM50 [5].

2.3.1 Modelos de *Gaps*

A inclusão ou remoção de resíduos na sequência significa manipulá-la de maneira a considerar ou desconsiderar determinadas mutações genéticas. Evidentemente os gaps devem computar um escore negativo no cálculo do escore total, por isso dois tipos básicos de penalidades são propostos [5]:

- *Linear gap*: um *gap* de tamanho g é penalizado por uma constante d:

$$y(g) = -gd \tag{2.1}$$

- *Affine gap*: leva em conta que um *gap* longo é mais propício de acontecer do que vários *gaps* pequenos, por isso, considera-se d como a penalidade por abrir um *gap* (*gap-open*) e e como a penalidade por extendê-lo (*gap-extension*).

$$y(g) = -d - (g-1)e \tag{2.2}$$

A Figura 2.4 ilustra o alinhamento de duas amostras diferentes de DNA do vírus Varíola, com a inclusão de um *gap* no início da segunda subsequência com o intuito de alinhar vários outros resíduos subsequentes.

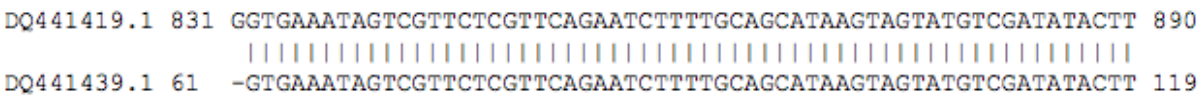

Figura 2.4: Alinhamento local de duas amostras do vírus Varióla com a inserção de um *gap* no início da segunda subsequência.

2.4 Algoritmos Exatos de Comparação de Sequências

Quando permite-se a inclusão de gaps nas sequências para encontrar alinhamentos locais, torna-se bastante custoso alinhar qualquer sequência de tamanho razoável, pois o número de alinhamentos possíveis entre duas sequências de tamanho n cresce conforme o representado na Equação 2.3.

$$\binom{2n}{n} = \frac{(2n)!}{(n!)^2} \simeq \frac{2^{2n}}{\sqrt{\pi n}} \tag{2.3}$$

Evidentemente, torna-se inviável calcular todos os alinhamentos possíveis em tempo hábil. Por esse motivo, alguns algoritmos de programação dinâmica foram propostos para encontrar o melhor alinhamento possível entre duas sequências.

2.4.1 Needleman-Wunsh

O primeiro algoritmo de alinhamento de sequências foi proposto por Needleman e Wunsch em 1970 [13]. Trata-se de um algoritmo de programação dinâmica para encontrar o melhor alinhamento global ótimo entre duas sequências.

O primeiro passo consiste em montar uma matriz de similaridade H(i,j), sendo cada um dos índices uma sequência. Primeiramente inicializa-se H(0,0) = 0, e, consequentemente, para H(i,0) e H(0,j), assume-se que seus valores são múltiplos da penalidade de *gap*, isto é, H(i,0) = -id e H(0,j) = -jd. Depois, calcula-se H(i,j) recursivamente, preenchendo toda a matriz da esquerda para a direita, de cima para baixo, na diagonal. Quando H(i - 1,j - 1), H(i - 1,j) e H(i,j - 1) são conhecidos, é possível calcular H(i,j) e existem apenas três possíveis valores para H(i,j):

- x_i estar diretamente alinhado com y_j;
- x_i estar alinhado com um *gap*;
- y_j estar alinhado com um *gap*.

Assim, têm-se a Equação 2.4 para o escore de H(i,j).

$$H(i,j) = max \begin{cases} H(i-1, j-1) + s(x_i, y_j) \\ H(i-1, j) - d \\ H(i, j-1) - d \end{cases} \tag{2.4}$$

onde s(x_i, y_j) é o valor para *match* quando $x_i = y_j$ ou *mismatch*, quando $x_i \neq y_j$ e d é o valor do *gap*, no caso de DNA ou RNA, ou a pontuação da matriz de substituição, caso contrário.

Tomando como exemplo as sequências u = GCATCC e v = GCAAC, primeiramente escolhe-se um escore para *match* = +1, *mismatch* = -1 e *gaps* = -2.

Para a primeira linha e primeira coluna, isto é, H(i,0) e H(0,j), são preenchidos os valores referentes aos *gaps* (Figura 2.5).

		G	C	A	T	C	C
	0	-2	-4	-6	-8	-10	-12
G	-2						
C	-4						
A	-6						
A	-8						
C	-10						

Figura 2.5: Primeiro passo do alinhamento global entre u e v.

Para H(1,1), acontece um *match* e o menor valor valor anterior é H(0,0) = 0, portanto, H(1,1) = 0 + 1 = 1.

		G
	0	-2
G	-2	↖1

Figura 2.6: Alinhamento em H(1,1).

Já para H(1,2), o maior valor não vem da diagonal, e sim da esquerda, significando o alinhamento com um *gap*, portanto, H(1,2) = 1 - 2 = -1.

Para cada alinhamento, é mantido um ponteiro indicando de onde veio cada um dos alinhamentos. Dessa forma, é montada a matriz de similaridade ilustrada na Figura 2.8. As setas em azul significam *match*, as setas em vermelho *mismatch* e, em laranja, *gap*.

	G	C
	-2	-4
G	↖1	←-1

Figura 2.7: Alinhamento em H(1,2) com ocorrência de um *gap*.

		G	C	A	T	C	C
	0	←-2	←-4	←-6	←-8	←-10	←-12
G	↑-2	↖1	←-1	←-3	←-5	←-7	←-9
C	↑-4	↑-1	↖2	← 0	←-2	↖←-4	↖←-6
A	↑-6	↑-3	↑ 0	↖3	← 1	←-1	←-3
A	↑-8	↑-5	↑-2	↖↑ 1	↖2	↖← 0	↖←-2
C	↑-10	↑-7	↖↑-4	↖↑-1	↖↑0	↖3	↖←1

Figura 2.8: Matriz de similaridade para o alinhamento entre u e v utilizando o algoritmo de Needleman-Wunsch.

Com a matriz de similaridade montada, é feito o *traceback*. As setas são seguidas do final da matriz até o início e é dado o melhor alinhamento global. Note que podem existir vários alinhamentos ótimos, Neste exemplo, os alinhamentos globais ótimos são indicados pelas setas coloridas na Figura 2.8. Uma seta na diagonal representa um *match* ou *mismatch*. Uma seta para cima significa um *gap* na sequência de cima e uma seta para a esquerda significa um *gap* na sequência da lateral. Assim, um dos alinhamentos ótimos entre as duas sequências é:

u = GCATCC

v = GCAAC−

Em relação a performance do algoritmo, é importante notar que se duas sequências de tamanho n e m são alinhadas, a quantidade de memória necessária é da ordem de $O(nm)$, assim como o tempo necessário, resultando em um algoritmo de ordem $O(nm)$.

2.4.2 Smith-Waterman

Uma situação mais comum do que comparar duas sequências completas, é comparar subsequências de ambas, de maneira a encontrar similaridades entre dois genomas comple-

tamente distintos, por exemplo. Para isso, o algoritmo de alinhamento local foi proposto por Smith e Waterman em 1981. [23]

O algoritmo é bastante parecido com o Needleman-Wunsch, tendo a sua principal diferença o fato de não existirem valores negativos na matriz, conforme a Equação 2.5.

$$H(i,j) = max \begin{cases} 0 \\ H(i-1, j-1) + s(x_i, y_j) \\ H(i-1, j) - d \\ H(i, j-1) - d \end{cases} \tag{2.5}$$

Tomando as mesmas sequências u = GCATCC e v = GCAAC como exemplo, neste caso, a matriz de similaridade ficaria conforme a Figura 2.9.

		G	C	A	T	C	C
	0	0	0	0	0	0	0
G	0	↖1	0	0	0	0	0
C	0	0	↖2	0	0	↖1	↖1
A	0	0	0	↖3	←1	0	0
A	0	0	0	↖↑ 1	↖2	0	0
C	0	0	↖ 1	0	0	↖3	↖←1

Figura 2.9: Matriz de similaridade para o alinhamento entre u e v utilizando o algoritmo de Smith-Waterman.

É possível notar alguns alinhamentos locais na matriz de similaridade. Sempre que um 0 é encontrado, corresponde a um novo alinhamento. Para fazer o *traceback*, diferentemente de como é feito no Needleman-Wunsch, onde se inicia em H(n,m), procura-se na matriz de similaridade o maior valor de H(i,j) e o *traceback* é iniciado a partir daí, até que se encontre um valor 0 na matriz. Assim, é encontrado o melhor alinhamento local possível.

No caso da Figura 2.9, os melhores alinhamentos locais são

`u = GCATC u = GCA`

`v = GCAAC v = GCA`

Como foram escolhidas sequências pequenas e parecidas para facilitar o entendimento dos algoritmos, os alinhamentos locais são parecidos com o melhor alinhamento global,

mas essa não é uma regra e na prática é bastante comum que os melhores alinhamentos locais sejam bastante distintos dos alinhamentos globais.

2.4.3 Gotoh

Uma maneira mais precisa biologicamente de se alinhar sequências tanto de maneira global quanto local foi proposta em 1982 por Gotoh [9]. O algoritmo se baseia no uso de *affine gaps* ao invés de *linear gap* (Seção 2.3.1).

Para cada posição (n,m) da matriz de similaridade mantém-se os valores para: x_i alinhado com y_j (H(i - 1,j - 1) + s(x_i, y_j)), um *gap* alinhado com y_j (E(i,j)) e um *gap* alinhado com x_i (F(i,j)). Assim, a Equação 2.5 ficaria da seguinte maneira:

$$H(i,j) = max \begin{cases} 0 \\ H(i-1,j-1) + s(x_i, y_j) \\ E(i,j) \\ F(i,j) \end{cases} \tag{2.6}$$

onde E(i,j) e F(i,j) são dados por:

$$E(i,j) = max \begin{cases} E(i,j-1) - e \\ H(i,j-1) - d \end{cases} \tag{2.7}$$

$$F(i,j) = max \begin{cases} F(i-1,j) - e \\ H(i-1,j) - d \end{cases} \tag{2.8}$$

Portanto, primeiro calculam-se os elementos E(i,j) e F(i,j) para então encontrar H(i,j).

É importante notar que mesmo se utilizando de três matrizes para fazer o alinhamento, o algoritmo se mantém na ordem de $O(nm)$.

Capítulo 3

Comparação de Sequências Biológicas com Instruções Vetoriais

Existem várias maneiras de otimizar a comparação de sequências biológicas. Neste capítulo será abordado a otimização a partir da programação paralela de alto desempenho em processadores que dão suporte a isso.

Os supercomputadores vetoriais surgiram entre as década de 1970 com o CDC Star-100 e o Texas Instruments ASC. Esses computadores eram capazes de processar vários dados (ou um vetor) com uma única instrução, seguindo o modelo SIMD (Single Instruction Multiple Data) da taxonomia de Flynn [7].

Na década de 1990, os computadores desktop possuíam cada vez mais poder de processamento para atender as demandas de aplicações mais pesadas. Como consequência, os fabricantes passaram a disponibilizar extensões SIMD em seus processadores, tais como a extensão MMX da Intel [16] ou o conjunto de instruções VIS do Sun UltraSPARC [11].

3.1 Modelo SIMD

A arquitetura SIMD descreve um modelo de operação em computadores para a resolução de problemas intensos que podem ser resolvidos de maneira paralela, como é o caso da computação biológica. O modelo consiste numa única instrução ser capaz de processar múltiplos dados simultaneamente e se baseia no processamento de vetores.

Inicialmente, apenas os supercomputadores conhecidos por processadores *array* utilizavam o modelo SIMD para processamento. Posteriormente, a arquitetura se tornou mais frequente nos computadores pessoais, com conjuntos de instruções específicas. Em 1997 a Intel foi pioneira ao trazer as instruções MMX em sua arquitetura x86, no Pentium MMX [16], seguida pelas extensões SSE, SSE2, SSE3, SSE4, AVX e AVX2.

A Figura 3.1 mostra um comparativo entre o modelo SISD (*Single Instruction Single Data*), onde uma única instrução opera sobre um único dado escalar e o modelo SIMD, operando sobre um vetor de escalares.

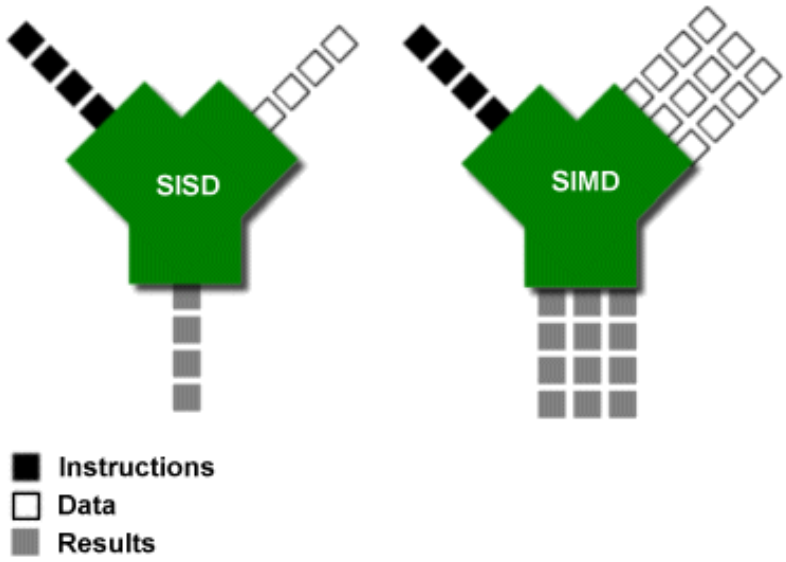

Figura 3.1: Arquitetura SISD e arquitetura SIMD [14].

3.2 Intel SSE e AVX

O conjunto de instruções SSE (*Streaming SIMD Extensions*) foi proposto pela Intel em 1999 e introduzido aos processadores Pentium III [17] como resposta ao *3DNow!* lançado pela AMD no ano anterior [15].

Logo na primeira versão do SSE, é adicionado o suporte a operações com ponto flutuante de precisão simples, contando com 8 registradores de 128 bits, nomeados de XMM0 a XMM7. Também foi mantido o suporte a extensão MMX, para as operações com números inteiros [17].

O SSE2 foi introduzido no Pentium 4, trazendo o suporte a operações com ponto flutuante de precisão dupla e operações SIMD sobre dados do tipo inteiro de 8, 16 e 32 bits utilizando os registradores XMM [2].

Ainda no Pentium 4, na família Prescott, foi lançado o SSE3, de maneira a incrementar o conjunto de instruções com novas instruções que auxiliam no processamento de digitais e modelos 3D [2].

Em 2006 a Intel lançou o SSE4 nos processadores Intel Core, adicionando 54 novas instruções SIMD à arquitetura, sendo 47 instruções presentes no SSE4.1 lançado inicial-

mente na família Penryn [8], seguidas de mais 7 instruções lançadas posteriormente no SSE4.2, a partir da família Nehalem [22].

Mais recentemente, a partir da família Sandy Bridge, foi disponibilizada a extensão AVX (*Advanced Vector Extensions*), dobrando o tamanho dos registradores XMM para 256 bits, e renomeando-os para YMM0 a YMM7 [21]. Os registradores YMM no entanto, não são capazes de fazer operações com números inteiros, limitação essa que foi resolvida com o lançamento do AVX2 na recente família Haswell, onde a maioria das instruções SSE e AVX para inteiros foram expandidas para 256 bits [10].

A Figura 3.2 mostra um exemplo de programa utilizando a extensão AVX2 para calcular a diferença entre dois vetores. O tipo `__m256` refere-se a um vetor de 256 *bits* (contendo 8 inteiros de 32 *bits*, nesse caso).

```
// diferenca_avx.c
#include <immintrin.h>
#include <stdio.h>

int main() {
   __m256i vector_1 = _mm256_set_epi32(2,4,6,8,10,12,14,16);
   __m256i vector_2 = _mm256_set_epi32(1,3,5,7,9,11,13,15);

   __m256i result = _mm256_sub_epi32(vector_1, vector_2);

   return 0;
}
```

Figura 3.2: Exemplo de programa que faz o uso da extensão AVX2.

3.3 Comparação de Sequências Biológicas com Instruções Vetoriais

3.3.1 Wozniak

Uma das primeiras implementações do algoritmo Smith-Waterman com as otimizações de Gotoh usando instruções SIMD foi proposta por Wozniak em 1997 [26]. Foi utilizado o conjunto de instruções VIS (*Visual Instruction Set*) do processador Sun UltraSPARC [11].

O maior obstáculo para utilizar o algoritmo de Smith-Waterman com instruções vetoriais é a interdependência do cálculo entre as células. Se os elementos (i, j) não forem devidamente escolhidos ou se a interdependência não for tratada após o processamento, é inviável resolver o algoritmo em paralelo.

Para calcular H(i, j), cada célula (i, j) depende dos valores de (i - 1, j), (i - 1, j - 1) e (i, j - 1). Com base nisso, nota-se que é possível calcular os valores das menores diagonais (ou anti-diagonais) da matriz em paralelo (Figura 3.3a), visto que estes valores não dependem uns dos outros. A implementação se baseia nesse fato para realizar o paralelismo. Os registradores VIS de 64 *bits* do UltraSPARC são divididos de maneira a calcular 8 células em paralelo, com precisão de até 8 *bits*.

Comparando sequências de até 5217 aminoácidos com o banco de dados da Swiss-Prot [1] em um servidor Sun UltraSPARC 167 MHz com 12 processadores, o algoritmo conseguiu atingir 0.19 GCUPS [26].

3.3.2 Rognes e Seeberg

Em 2000, Rognes e Seeberg propuseram uma implementação do algoritmo Smith-Waterman otimizado com Gotoh para as instruções SIMD MMX/SSE da Intel [20].

Com o objetivo de simplificar e agilizar o carregamento dos vetores para a memória, estes são carregados em paralelo com a *query sequence* (Figura 3.3b) ao invés de serem carregados seguindo a menor diagonal, conforme a Figura 3.3a. A desvantagem é que a dependência dos dados entre os vetores deve ser tratada pelo algoritmo.

Uma das otimizações que o algoritmo utiliza chama-se SWAT (Green, 1993). A otimização é utilizada com *affine gap* e se baseia no fato de que o elemento F(i, j) (Equação 2.8) normalmente possui valor zero e não influencia no cálculo de H(i, j) (Equação 2.6). Assim, F(i, j) só é calculado quando necessário, diminuindo consideravelmente o tempo de processamento.

Outra otimização realizada utilizada pelo algoritmo consiste em dividir os registradores MMX de 64 *bits* em oito unidades de 8 *bits* cada, de maneira a aumentar o número de operações em paralelo, diminuindo a precisão dos cálculos de escore para 8 *bits* (0 a 255).

Também foi proposto o *query profile* para otimizar as comparações com a matriz de substituição. O *query profile* consiste em calcular uma matriz P para cada *query sequence*, dada uma determinada matriz de substituição, de maneira a otimizar as buscas ao calcular o escore.

Em relação a performance, o algoritmo foi capaz de atingir 0.156 GCUPS num Pentium III 500 MHz comparando sequências entre 189 a 567 aminoácidos com o banco de dados da SwissProt [1], demonstrando na época uma excelente performance num processador barato de propósito geral [20].

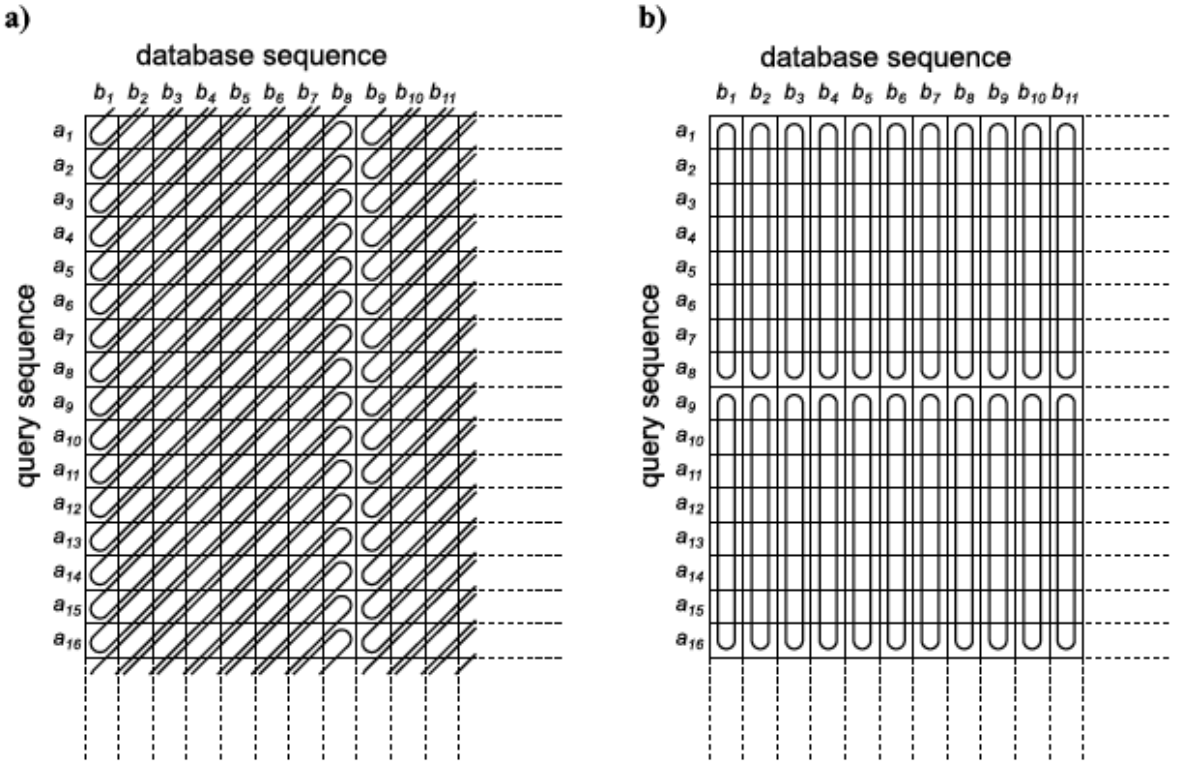

Figura 3.3: Distribuição das células nos vetores em implementações SIMD do algoritmo Smith-Waterman. (a) células distribuídas pela menor diagonal. (b) Células distribuídas em paralelo com a *query sequence* [20].

3.3.3 Farrar

Em 2006, Farrar propôs novas otimizações no algoritmo Smith-Waterman com instruções SIMD, utilizando a extensão SSE2 da Intel [6].

Os registradores SIMD SSE2 de 128 *bits* são divididos em segmentos de 8 *bits*, permitindo que até 16 células de 8 *bits* sejam calculadas em paralelo. Caso o escore seja maior que 255, o registrador é dividido em 8 unidades de 16 *bits* e então o escore é recalculado, podendo chegar até 65535.

A principal mudança em relação a implementação proposta por Rognes e Seeberg (Seção 3.3.2) é a maneira como as células são carregadas no vetor para serem processadas em paralelo. Farrar manteve o carregamento do vetor em paralelo com a *query sequence*, porém de maneira particionada, conforme mostrado na Figura 3.4c, cada célula do segmento é colocada no vetor baseado no tamanho do segmento. Dessa maneira, a dependência entre as células é reduzida e o tempo computacional é aumentado consideravelmente, pois as dependências são tratadas apenas uma vez em cada iteração do *loop* exterior, com uma operação de deslocamento (*shift*).

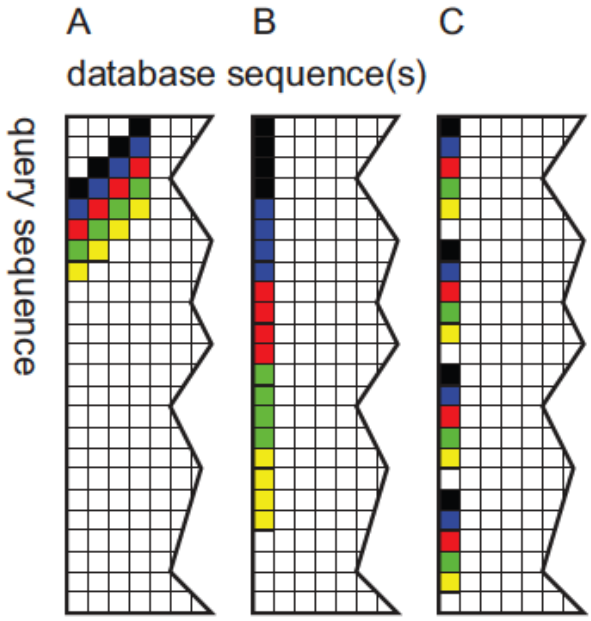

Figura 3.4: Comparação entre a abordagem de Farrar (c) e as demais abordagens (a e b) para o algoritmo de Smith-Waterman paralelizado [19].

O algoritmo de Farrar também faz o uso do SWAT (nomeando-o de *Lazy-F loop*), de modo que o cálculo de F(i, j) seja feito fora do *loop* interior, para que F seja calculado apenas quando necessário.

Comparando sequências entre 143 e 567 aminoácidos com o banco de dados da Swiss-Prot [1] utilizando um Xeon Core 2 Duo 2 GHz com 2 GB de RAM, o algoritmo atingiu picos de 3 GCUPS [6], mostrando uma performance de 2 a 8 vezes maior que as propostas já conhecidas até então.

3.4 Quadro Comparativo

A Tabela 3.1 apresenta a comparação entre os algoritmos apresentados na Seção 3.3. Todos os resultados referem-se a comparação de proteínas.

É notável que, com o passar dos anos, a evolução dos processadores representou um avanço considerável na comparação de sequências biológicas. Além de instruções vetoriais mais completas e robustas, otimizações nos algoritmos que as utilizam foram essenciais para o ganho de desempenho. Enquanto [26] foi capaz de processar 0,19 GCUPS utilizando 12 unidades de processamento, [20] foi capaz de chegar a 0,156 GCUPS com apenas uma única unidade de processamento.

Tabela 3.1: Quadro comparativo entre as implementações do algoritmo Smith-Waterman para instruções vetoriais.

		Plataforma			Algoritmo		
Ref.	Ano	Hardware	SIMD	Nº U.P.	Algo.	Otimiz.	GCUPS
[26]	1997	Sun UltraSPARC 167 MHz	VIS	12	Gotoh	paralelismo diagonal	0,19
[20]	2000	Intel Pentium III 500 MHz	MMX	1	Gotoh	SWAT, *query profile*, vetor SW paralelo com a *query sequence*	0,156
[6]	2006	Intel Xeon Core 2 Duo 2 GHz	SSE2	2	Gotoh	SWAT, *query profile*, *striped* SW	3,00

A forma particionada utilizada por [6] para carregar os segmentos a serem processados em paralelo foi crucial para o ganho de desempenho na aplicação. Utilizando a extensão SSE2 e duas unidades de processamento, essa implementação foi capaz de processar 3,00 GCUPS. REF3 atualmente é utilizada como base para novas implementações do algoritmo Smith-Waterman em paralelo, tanto em CPU quanto em GPU.

Capítulo 4

MASA (*Mullti-Platform Architecture for Sequence Aligners*)

Nesse capítulo será apresentado o MASA (*Mullti-Platform Architecture for Sequence Aligners*) [3], uma arquitetura de software especializada para alinhamento de sequências biológicas baseada no CUDAlign 2.1 [4]. Sua principal característica é o fato dela ser flexível e customizável para várias arquiteturas de hardware e software, sendo capaz de processar comparações de sequências tanto em CPU, quanto em GPU, por exemplo.

4.1 Arquitetura MASA

A implementação do MASA propõe uma arquitetura dividida em 5 módulos, conforme a Figura 4.1a. Os módulos *Data Management*, *Statistics* e *Stage Management* são genéricos, portanto, são utilizados em todas as implementações. Os módulos *Block Pruning* e *Parallelization Strategy* são customizáveis de acordo com a estratégia de paralelização e alinhamento utilizada.

Data Management: esse módulo é executado assim que a implementação do MASA é inicializada e ele é responsável por lidar com os dados como por exemplo as sequências de entrada, parâmetros do passados pelo usuário, alinhamento ótimo e *score*.

Statistics: esse módulo é responsável apenas por obter estatísticas como o tempo de execução, quantidade de memória e disco utilizados, percentual de blocos alinhados etc.

Stage Management: esse módulo coordena a execução dos estágios MASA. Os estágios 1 a 3 dividem a matriz DP para ser calculada pela parte dependente da plataforma, como GPU, CPU ou FPGA. No estágio 1, é considerada apenas uma única partição da matriz DP que possui o mesmo tamanho da matriz. Nos estágios 2 e 3 essa única partição é subdividida em partições menores de acordo com as linhas e colunas especiais.

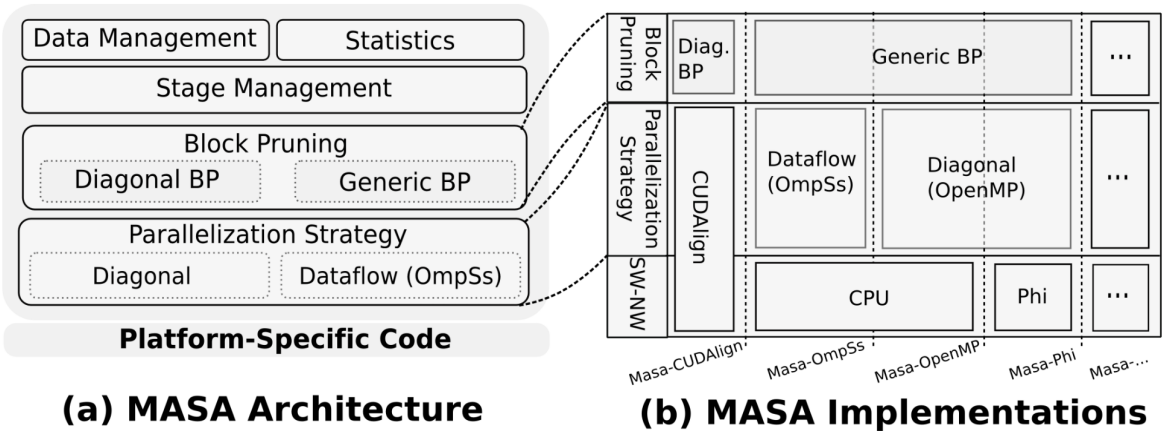

Figura 4.1: Arquitetura MASA [3].

Block Pruning: esse módulo é responsável por aplicar a otimização de Block Pruning. Uma solução independente de plataforma é utilizada pelo MASA (Diagonal BP). Duas células (k_s e k_e) na mesma diagonal indicam a janela que não é descartável são mantidas em memória para cada diagonal. As células fora da janela $[k_s..k_e]$ são descartadas, otimizando assim o desempenho. O MASA provê também uma implementação genérica do Block Pruning, chamada Generic BP, que pode ser utilizada por novas implementações de alinhadores que utilizam diferentes estratégias de paralelização, como, por exemplo, *dataflow*.

Parallelization Strategy: esse módulo é responsável por definir a estratégia de paralelização utilizada para calcular as células (i, j) da matriz DP. O MASA conta com duas estratégias de paralelização: diagonal e *dataflow*. A estratégia em diagonal calcula os blocos que pertencem a mesma diagonal em paralelo, iniciando da ponta superior esquerda. A limitação dessa estratégia é a sincronização entre os pontos no final de cada diagonal computada. A estratégia em *dataflow* visa otimizar o tempo de sincronização existente na estratégia em diagonal. Cada nó do fluxo de dados é um bloco de células e os blocos são computados em paralelo quando suas dependências já tiverem sido computadas.

4.1.1 MASA-API

O MASA foi desenvolvido utilizando a linguagem C++. A Figura 4.2 ilustra a API do MASA num diagrama de classes. A interface entre os estágios do MASA e as implementações dos alinhadores se dá através da classe IAligner. Qualquer implementação de alinhador que for utilizar o MASA deve implementar os métodos virtuais da classe IAligner. O método *initialize()* é chamado uma vez para inicializar o uso dos recursos necessários para o alinhamento, assim como o finalize() é chamado para finalizar o uso

desses recursos. Cada estágio do MASA chama os métodos *setSequences()* e *unsetSequences()* para definir a direção e o intervalo da sequência que será utilizado no estágio. Para fazer o alinhamento dos intervalos das sequências, o método *alignPartition()* é utilizado.

A classe AbstractAligner encapsula os métodos da classe IManager e inicializa os operações de block pruning. As classes AbstractBlockAligner e AbstractDiagonalAligner herdam de AbstractAligner e implementam diferentes tipos de alinhadores.

A classe AbstractBlockAligner calcula a matriz utilizando a estratégia de blocos através da classe GenericBP. Os blocos são processados pela classe AbstractBlockProcessor. A subclasse CPUBlockProcessor calcula Smith-Waterman e Needleman-Wunsch em CPU (sem utilizar SSE). Outras implementações poderiam ser adicionadas herdando da classe AbstractBlockProcessor.

A classe AbstractDiagonalAligner utiliza a classe DiagonalBP para calcular a matriz DP. Nessa estratégia, a classe CUDAligner é responsável por calcular as diagonais na GPU.

4.2 Implementações do MASA

Até o presente momento, existem quatro implementações do MASA, que utilizam diferentes modelos de programação (OpenMP, OmpSs e CUDA) e focados em diferentes *hardwares* (*multicore*, GPU e Intel Phi).

O MASA-OpenMP/CPU utiliza o OpenMP para calcular os blocos da mesma diagonal em paralelo. *Threads* independentes processam cada diagonal em paralelo e o OpenMP controla a utilização de cada núcleo do processador. Essa implementação embora funcional, não se aproveita das instruções vetoriais do processador para otimizar o desempenho.

O MASA-OpenMP/Intel Phi utiliza a mesma estratégia de paralelização adotada no MASA-OpenMP/CPU, portanto, também não utiliza as instruções SIMD de 512 bits do processador, resultando num desempenho limitado.

As outras implementações consistem no MASA-OmpSs/CPU, que faz o uso do OmpSs na estratégia de paralelização em *dataflow* e no MASA-CUDAlign, que é utilizada em GPU.

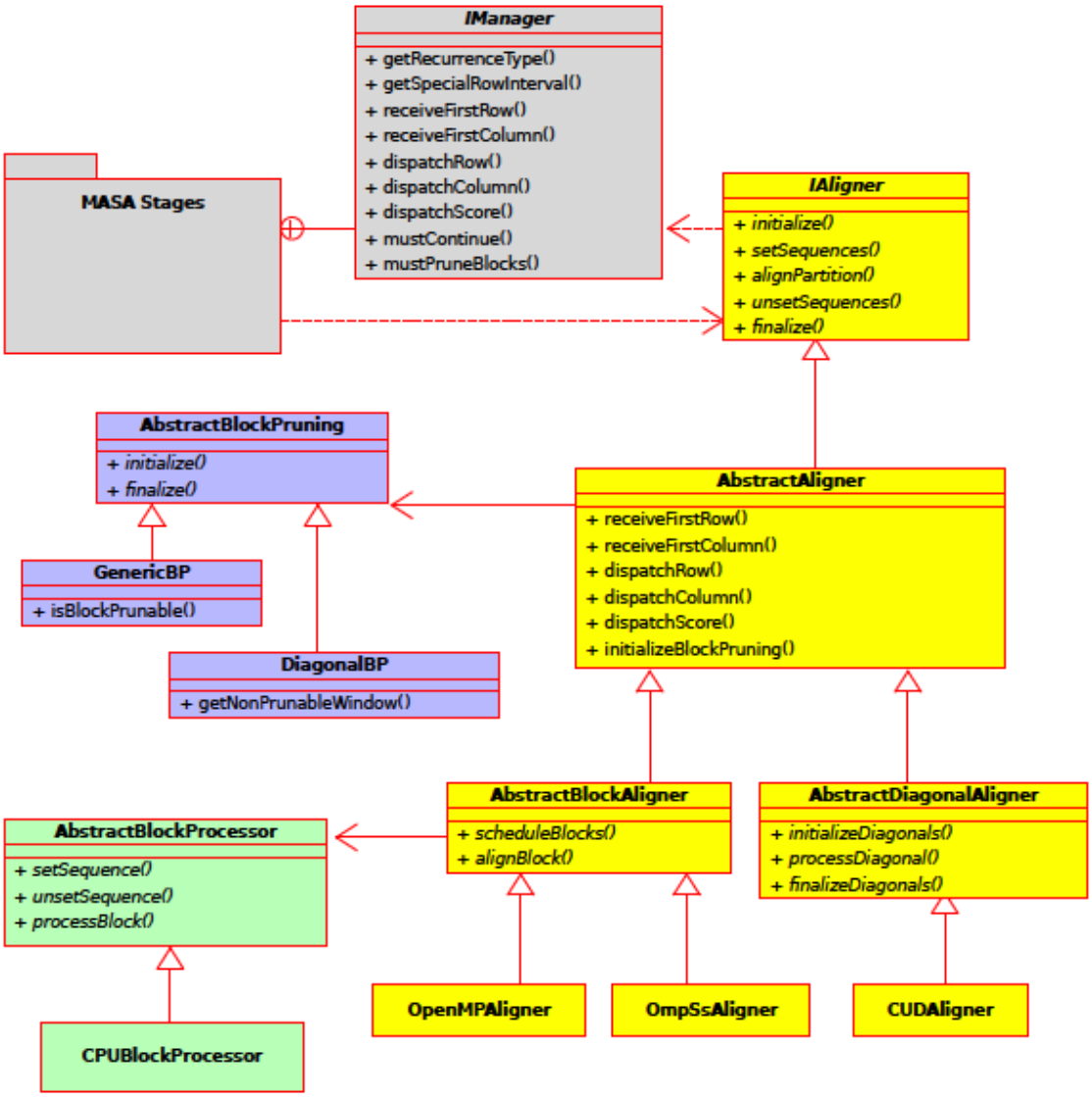

Figura 4.2: Diagrama de classes do MASA [3].

Capítulo 5

Projeto do MASA-SSE

5.1 Visão Geral

No projeto do MASA-SSE, pensou-se incialmente em estender o MASA-Core baseado no MASA-OpenMP (Figura 4.1b), de maneira que fosse mantida a paralelização realizada pelo OpenMP, calculando diversos blocos em diferentes *threads*, porém, cada bloco seria otimizado com instruções vetoriais. Dessa maneira, seria possível manter as otimizações de *Block Pruning* do MASA e obter o máximo desempenho de cada núcleo do processador, para o caso de processadores *multi-core* e *many-core*.

Determinamos, então, que o trecho otimizado com instruções vetoriais seria limitado a alinhamentos locais e seguiria a implementação proposta por Farrar (Seção 3.3.3), adaptada para nucleotídeos. No MASA-SSE, as implementações *SSW Library* [27] e SWPS3 [24] foram utilizadas como referência. Ambas se baseiam na implementação do Farrar.

Ao analisarmos a implementação do Farrar em detalhes, ficou claro que a adição de *threads* OpenMP iria incluir um novo nível de paralelismo (*threads* + instruções vetoriais), o que dificultaria bastante a programação, pois o tratamento da relação de recorrência entre blocos implementada atualmente pelo MASA não funcionaria e deveria ser reescrita de maneira a suportar o processamento vetorial das células. Sendo assim, optou-se por utilizar como base o Aligner presente no MASA-CPU Serial. A classe SSEAligner implementa a interface da classe AbstractBlockAligner (Figura 4.2) e reescreve os métodos *scheduleBlocks()* e *alignBlock()* de maneira simplificada, de forma que o tamanho do *grid* na solução vetorial fique fixado em 1x1, ou seja, não há divisão da computação em blocos. Uma consequência dessa decisão é a impossibilidade de manter a estratégia de *block pruning* originária do MASA.

O esquema mostrado na Figura 5.1 ilustra onde o MASA-SSE se encaixa na atual estrutura do MASA.

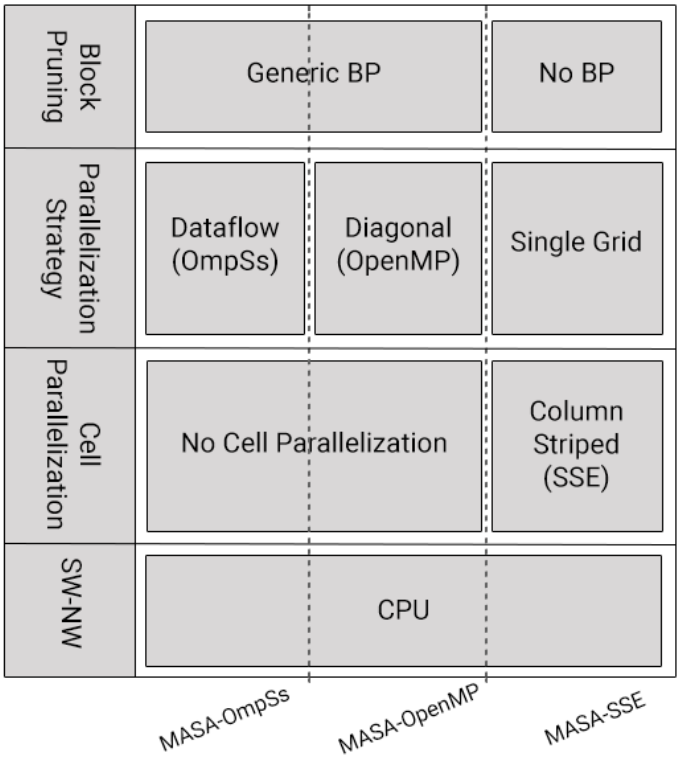

Figura 5.1: Arquitetura do MASA-SSE.

A classe SSEBlockProcessor foi criada, implementando a interface da classe abstrata AbstractBlockProcessor. O método *processBlock()* presente no CPUBlockProcessor foi completamente reescrito de maneira a suportar o processamento vetorial.

Chamado por SSEAligner::*alignBlock()*, o método *processBlock()* recebe como entrada o bloco a ser alinhado e retorna o escore do alinhamento local e sua posição (i, j). Durante sua execução, o método *generateQp()* gera o *query profile* (Seção 3.3.3) para os *matches* e *mismatches* de todo o bloco. O *query profile* é ajustado a partir do valor de abertura de um *gap*, somando-se *DNA_GAP_OPEN* em todos os elementos do *query profile*, com o intuito de trabalhar-se apenas com números sem sinal.

Para suportar escores entre 0 e 65535, os vetores XMM são divididos em 8 unidades de 16 *bits*, possibilitando o cálculo do escore de até 8 células em paralelo. O processador, para tanto, deve suportar a extensão SSE4.1, pois o algoritmo faz o uso da instrução PMAXUW (*_mm_max_epu16*), presente nessa extensão.

Por conta da dependência de H(i - 1, j - 1), dois *buffers* são utilizados para armazenar os valores de H, sendo um para armazenar os valores antigos de H e outro para os novos. Para resolver a dependência entre os segmentos de uma coluna e outra, os valores do último vetor H são deslocados 16 *bits* à esquerda durante o *loop* exterior, fazendo com que

estes valores fiquem alinhados com o próximo segmento. Dessa maneira, não é necessário nenhuma outra operação no *loop* interior para inserir o valor de H no próximo segmento.

O pseudo-código da parte vetorizada é mostrado na Figura 5.2.

Incialmente as matrizes de programação dinâmica E e F são inicializadas com zero no *loop* exterior. Durante esse *loop*, a interdependência dos valores de H é resolvida através de um *shift* para a esquerda de 2 *bytes*, ocorrendo também a troca dos valores entre os dois *buffers*.

No *loop* interior, utilizando instruções vetoriais, os valores da matriz H são somados ao *query profile*. É calculado o máximo entre as matrizes H, E e F e esse valor é salvo em *vMaxColumn*. O valor calculado para H é salvo em um dos *buffers* também. A seguir, são calculados os novos valores de E e F considerando as penalidades de *gap*. Esse *loop* é processado para cada um dos segmentos da coluna *i*.

Após o processamento de todo o *loop* interior, *vF* é deslocado 2 *bytes* à esquerda para resolver a dependência com a próxima coluna e verifica-se F é maior que H - (*DNA_GAP_OPEN* + *DNA_GAP_EXT*). Caso positivo, F pode influenciar no valor de H, portanto, seus valores são recalculados até que todos os valores de F sejam menores que H - (*DNA_GAP_OPEN* + *DNA_GAP_EXT*).

Os valores calculados presentes em *vMaxColumn* são então comparados com o maior escore já encontrado até o presente momento e o maior escore é atualizado, caso necessário.

```
func SSEAligner::alignBlock:
      ...
      processBlock(i0, j0, i1, j1);
      ...
endfunc

func SSEBlockProcessor::processBlock:

max := 0;
refLen := i1 - i0;

// Calcula o total de segmentos necessários
// para processar toda a query.
segLen := (length(j1 - j0) + 7) / 8;

for i := 0 ... refLen
  // Inicializa o vetor F com zero.
  // vMaxColumn armazena os maiores
  // escores da coluna i.
  vE, vF, vMaxColumn := <0,...,0>;

  // Ajusta o último valor de H para ser
  // usado no próximo  segmento.
  vH := vHStore[segLen - 1];
  vH := _mm_slli_si128(vH, 2);

  // Troca os dois buffers de H
  swap(vHLoad, vHStore);

  for j := 0 ... segLen
      // Soma vH ao query profile vP[i][j].
      vH := _mm_adds_epu16(vH,
_mm_load_si128(vP + j));
      vH := _mm_subs_epu16(vH, vBias);

      // Atualiza vH com o máximo entre vH e
      // os valores de E e F.
       vE := _mm_load_si128(vE + j);
       vH := _mm_max_epu16(vH, vE);
      vH := _mm_max_epu16(vH, vF);

       // Salva os valores de vH maiores que
      // os presentes em vMaxColumn.
       vMaxColumn :=
_mm_max_epi16(vMaxColumn, vH);

       // Salva os valores de vH
       _mm_store_si128(vHStore + j, vH);

      // Calcula os novos valores de E e F
      // baseado nas penalidades de gap.
            vH := _mm_subs_epu16(vH, vGapO);
            vE := _mm_subs_epu16(vE, vGapE);
            vE := _mm_max_epu16(vE, vH);
            _mm_store_si128(pvE + j, vE);
            vF := _mm_subs_epu16(vF, vGapE);
            vF := _mm_max_epi16(vF, vH);

      // Carrega em vH o próximo H a ser
      // processado.
            vH := _mm_load_si128(vHLoad +
j);
  endfor;

  // Lazy-F loop.
  j := 0;
  vH := _mm_load_si128(vHStore + j);
  vF := _mm_slli_si128(vF, 2);
  vTemp := _mm_subs_epu16(vH, vGapO);
  vTemp := _mm_subs_epu16(vF, vTemp);
  vTemp := _mm_cmpeq_epi16(vTemp, vZero);
  cmp := _mm_movemask_epi8(vTemp);

  while cmp != 0xffff
      vH := _mm_max_epu16(vH, vF);
       vMaxColumn :=
_mm_max_epu16(vMaxColumn, vH);
       _mm_store_si128(vHStore + j, vH);
       vF := _mm_subs_epu16(vF, vGapE);
       j++;
       if j >= segLen
          j := 0;
           vF := _mm_slli_si128(vF, 2);
       endif
       vH := _mm_load_si128(vHStore + j);
       vTemp := _mm_subs_epu16(vH, vGapO);
       vTemp := _mm_subs_epu16(vF, vTemp);
       vTemp := _mm_cmpeq_epi16(vTemp,
vZero);
       cmp := _mm_movemask_epi8(vTemp);
  endwhile

  vMaxScore := _mm_max_epu16(vMaxScore,
  vMaxColumn);

  temp := 0;
  temp := MAX8(vMaxScore);

  if temp > max
      max = temp;
  endif

endfor

endfunc
```

Figura 5.2: Pseudo-código do processamento vetorial do MASA-SSE.

Capítulo 6

Resultados Experimentais

6.1 Testes Realizados

Devido a limitação de 65535 no tamanho máximo do escore suportado pela implementação do MASA-SSE, foram selecionadas sequências de tamanho entre 1K e 50K para a realização dos testes. As sequências utilizadas como referência estão descritas na Tabela 6.1. Os resultados para as comparações de tamanho 10K e 50K referem-se às próprias sequências, os demais resultados referem-se a trechos truncados das sequências mostrados na Tabela 6.1.

Tabela 6.1: Sequências de referência.

Sequência 1			Sequência 2		
Accession	Tam.	Nome	Accession	Tam.	Nome
AF133821.1	10035	*HIV-1 isolate MB2059 from Kenya, complete genome*	AY352275.1	10280	*HIV-1 isolate SF33 from USA, complete genome*
NC_024791.1	50440	*Vibrio phage ICP2_2013_A Haiti, complete genome*	KM224878.1	50250	*Vibrio phage ICP2_2011_A, complete genome*

A pontuação utilizada para *match* é 1, e *mismatch*, -3. O peso de abertura de um *gap* é 3 e estendê-lo custa 2. Para o MASA-CPU Serial e o MASA-OpenMP o tamanho do bloco é definido pelo próprio MASA de acordo com o tamanho das sequências e limitado em 1024x1024 também pelo MASA.

Foram feitos testes em duas máquinas distintas. A Tabela 6.2 mostra os resultados na máquina 1, que contém um processador Intel Core I5-5257U, com 2 núcleos e 4 *threads*, 8 GB de memória RAM LPDDR3 a 1866 MHz e sistema operacional Mac OS X 10.11.1.

Já a Tabela 6.4 mostra os resultados na máquina 2, que contém um processador Intel Core I5-4670K, com 4 núcleos e 4 *threads*, 8 GB de memória RAM DDR3 a 1333 MHz e sistema operacional Ubuntu 15.10.

Todos os resultados referem-se ao estágio 1 do MASA, onde é apenas calculado o escore do alinhamento das sequências. Tanto a Tabela 6.2 quando a Tabela 6.4 trazem o comparativo dos resultados do MASA-SSE com o MASA-CPU Serial e o MASA-OpenMP. São apresentados tanto o tempo total de execução, como a métrica MCUPS (Milhões de Células Processadas por Segundo). Cabe ressaltar que o MASA-CPU Serial e o MASA-OpenMP estavam com o *block pruning* habilitado. Os valores destacados em negrito representam o melhor resultado. Para cada caso de teste, foram realizadas 3 medidas e aferida a média entre elas.

Tabela 6.2: Quadro comparativo dos resultados com *block pruning* na máquina 1.

Tamanho		MASA Ext.	Desempenho		*Pruned*
Sequência 1	Sequência 2		Tempo	MCUPS	
1K (AF133821.1)	1K (AY352275.1)	MASA-CPU Serial	<0,1	41,8	**4,7%**
		MASA-CPU OpenMP	<0,1	49,8	
		MASA-SSE	<0,1	**115,6**	0%
10K (AF133821.1)	10K (AY352275.1)	MASA-CPU Serial	1,5	69,3	**22,7%**
		MASA-CPU OpenMP	0,7	142,3	
		MASA-SSE	**0,5**	**187,8**	0%
20K (NC_024791.1)	20K (KM224878.1)	MASA-CPU Serial	5,7	68,5	**20,5%**
		MASA-CPU OpenMP	2,4	160,3	
		MASA-SSE	**1,6**	**238,7**	0%
30K (NC_024791.1)	30K (KM224878.1)	MASA-CPU Serial	12,3	71,0	**21,9%**
		MASA-CPU OpenMP	4,9	178,9	
		MASA-SSE	**3,7**	**239,7**	0%
40K (NC_024791.1)	40K (KM224878.1)	MASA-CPU Serial	19,7	78,8	**26,3%**
		MASA-CPU OpenMP	8,3	186,4	
		MASA-SSE	**6,4**	**244,2**	0%
50K (NC_024791.1)	50K (KM224878.1)	MASA-CPU Serial	34,3	73,9	**24,0%**
		MASA-CPU OpenMP	14,0	181,7	
		MASA-SSE	**9,7**	**259,5**	0%

6.2 Análise dos Resultados

Conforme observa-se na Tabela 6.3 e na Tabela 6.5, onde os testes foram feitos sem o *block pruning*, fica evidente o ganho de desempenho do MASA-SSE em relação ao MASA-CPU Serial e ao MASA-OpenMP. Para as sequências comparadas, nota-se um ganho de até 4,4 vezes na máquina 1 e 4,6 vezes na máquina 2 em relação ao MASA-CPU Serial e de até

Tabela 6.3: Quadro comparativo dos resultados sem *block pruning* na máquina 1.

Tamanho		MASA Ext.	Desempenho		*Pruned*
Sequência 1	Sequência 2		Tempo	MCUPS	
1K (AF133821.1)	1K (AY352275.1)	MASA-CPU Serial	<0,1	44,4	0%
		MASA-CPU OpenMP	<0,1	60,9	
		MASA-SSE	<0,1	**115,6**	
10K (AF133821.1)	10K (AY352275.1)	MASA-CPU Serial	1,8	57,1	0%
		MASA-CPU OpenMP	0,9	118,8	
		MASA-SSE	**0,5**	**187,8**	
20K (NC_024791.1)	20K (KM224878.1)	MASA-CPU Serial	6,9	57,2	0%
		MASA-CPU OpenMP	2,9	133,8	
		MASA-SSE	**1,6**	**238,7**	
30K (NC_024791.1)	30K (KM224878.1)	MASA-CPU Serial	15,1	57,6	0%
		MASA-CPU OpenMP	6,4	136,7	
		MASA-SSE	**3,7**	**239,7**	
40K (NC_024791.1)	40K (KM224878.1)	MASA-CPU Serial	26,8	58,1	0%
		MASA-CPU OpenMP	11,3	137,9	
		MASA-SSE	**6,4**	**244,2**	
50K (NC_024791.1)	50K (KM224878.1)	MASA-CPU Serial	43,4	58,4	0%
		MASA-CPU OpenMP	18,4	137,7	
		MASA-SSE	**9,7**	**259,5**	

Tabela 6.4: Quadro comparativo dos resultados com *block pruning* na máquina 2.

Tamanho		MASA Ext.	Desempenho		*Pruned*
Sequência 1	Sequência 2		Tempo	MCUPS	
1K (AF133821.1)	1K (AY352275.1)	MASA-CPU Serial	<0,1	68,3	**4,7%**
		MASA-CPU OpenMP	<0,1	124,8	
		MASA-SSE	<0,1	**271,1**	0%
10K (AF133821.1)	10K (AY352275.1)	MASA-CPU Serial	1,2	88,8	**22,7%**
		MASA-CPU OpenMP	0,4	256,7	
		MASA-SSE	**0,3**	**334,23**	0%
20K (NC_024791.1)	20K (KM224878.1)	MASA-CPU Serial	4,4	87,8	**20,5%**
		MASA-CPU OpenMP	1,4	274,1	
		MASA-SSE	**1,1**	**339,3**	0%
30K (NC_024791.1)	30K (KM224878.1)	MASA-CPU Serial	9,1	95,9	**21,9%**
		MASA-CPU OpenMP	2,8	316,1	
		MASA-SSE	**2,5**	**348,2**	0%
40K (NC_024791.1)	40K (KM224878.1)	MASA-CPU Serial	15,6	99,7	**26,3%**
		MASA-CPU OpenMP	4,6	339,2	
		MASA-SSE	**4,4**	**349,6**	0%
50K (NC_024791.1)	50K (KM224878.1)	MASA-CPU Serial	26,4	96,1	**24,0%**
		MASA-CPU OpenMP	7,7	331,2	
		MASA-SSE	**7,2**	**349,9**	0%

Tabela 6.5: Quadro comparativo dos resultados sem *block pruning* na máquina 2.

Tamanho		MASA Ext.	Desempenho		*Pruned*
Sequência 1	Sequência 2		Tempo	MCUPS	
1K (AF133821.1)	1K (AY352275.1)	MASA-CPU Serial	<0,1	71,3	0%
		MASA-CPU OpenMP	<0,1	68,2	
		MASA-SSE	<0,1	**271,1**	
10K (AF133821.1)	10K (AY352275.1)	MASA-CPU Serial	1,3	74,4	0%
		MASA-CPU OpenMP	0,5	209,8	
		MASA-SSE	**0,3**	**334,23**	
20K (NC_024791.1)	20K (KM224878.1)	MASA-CPU Serial	5,2	74	0%
		MASA-CPU OpenMP	1,6	245,3	
		MASA-SSE	**1,1**	**339,3**	
30K (NC_024791.1)	30K (KM224878.1)	MASA-CPU Serial	11,6	75,2	0%
		MASA-CPU OpenMP	3,4	254,5	
		MASA-SSE	**2,5**	**348,2**	
40K (NC_024791.1)	40K (KM224878.1)	MASA-CPU Serial	20,5	75,9	0%
		MASA-CPU OpenMP	5,9	263,5	
		MASA-SSE	**4,4**	**349,6**	
50K (NC_024791.1)	50K (KM224878.1)	MASA-CPU Serial	33,4	75,8	0%
		MASA-CPU OpenMP	9,5	266,3	
		MASA-SSE	**7,2**	**349,9**	

1,9 vezes na máquina 1 e 1,3 vezes na máquina 2 em relação ao MASA-OpenMP. Esse ganho é consequência do nível de paralelismo imposto pelo SSE no cálculo das células e mostrou-se crescente no caso do MASA-SSE.

Conforme observa-se na Tabela 6.2 e na Tabela 6.4, o MASA-SSE mostrou um desempenho superior em todos os casos de teste, em ambas as máquinas. Nesse comparativo, considera-se o *block pruning*. Nesse caso, observa-se que, além do desempenho do MASA-SSE ser crescente, com seu maior valor nas sequências de 50K, na máquina 1 o MASA-SSE saiu-se 3,1 vezes mais rápido que o MASA-CPU Serial e 1,3 vezes mais rápido que o MASA-OpenMP. Já na máquina 2, o MASA-SSE mostrou-se até 3,5 vezes mais rápido que o MASA-CPU Serial e 1,03 vezes mais rápido que o MASA-OpenMP. Vale ressaltar que, nesse caso de teste, houve um percentual de 26,3% de *block pruning* no MASA-CPU Serial e no MASA-OpenMP.

O percentual de *block pruning* é um fator que pode influenciar bastante no desempenho do MASA-OpenMP e MASA-CPU Serial, principalmente em processadores que possuem vários núcleos. Para os casos analisados, o MASA-SSE é mais rápido, mesmo em processadores com 4 núcleos e admitindo-se aproximadamente 25% de *block pruning*. O número de núcleos do processador também pode fazer com que o MASA-OpenMP apresente diferentes resultados, podendo ser maiores ou menores que os apresentados pelo MASA-SSE.

Para as sequências de tamanho 1K, todas as soluções demonstram um desempenho baixo. Por tratar-se de sequências muito pequenas, não existe paralelismo suficiente para que as soluções alcancem bom desempenho.

Nota-se ainda que, na máquina 2, embora o processador seja mais antigo, da geração *Haswell*, o desempenho de todas as implementações foram superiores, em especial no MASA-OpenMP, que faz o uso de múltiplas *threads*. Como o processador possui 4 núcleos, observa-se um desempenho quase 2 vezes maior, nesse caso. Já para o MASA-SSE, nota-se uma diferença entre 1,4 a 2,3 vezes maior na máquina 2, que pode ser explicada por uma maior frequência de operação do processador (3,4 GHz contra 2,7 GHz), além de diversas outras otimizações, por tratar-se de um processador *desktop*.

Capítulo 7

Conclusão e Trabalhos Futuros

O presente trabalho de graduação propôs e avaliou o MASA-SSE, uma solução que usa instruções vetoriais para a comparação de sequências biológicas. A solução baseou-se no algoritmo de Farrar (Seção 3.3.3) e foi adaptada para o *framework* MASA (Capítulo 4). A extensão SSE4.1 da Intel [8] foi utilizada para fazer o processamento vetorial e otimizações como o *query profile*, *Lazy-F loop* e processamento *striped*, presentes no algoritmo do Farrar foram integradas ao MASA-SSE, porém, não foi possível manter o *block pruning*, presente no MASA (Capítulo 4).

Os resultados experimentais obtidos em 2 máquinas distintas com sequências de DNA reais mostram que o MASA-SSE, usando apenas uma *thread*, possui desempenho superior ao MASA-OpenMP usando 4 *threads* em 4 núcleos nos casos de teste selecionados. Isso foi possível pois as instruções vetoriais SSE utilizadas permitem que até 8 instruções lógicas e aritméticas sejam executadas ao mesmo tempo, seguindo o modelo SIMD (Seção 3.1).

Devido à limitação do tamanho das sequências no MASA-SSE, a implementação ainda não tem muita aplicação na comparação efetiva de DNAs, limitando-se a uma pequena gama de DNAs ou a sequências truncadas. Atualmente, o MASA-CUDAlign é a solução mais viável para a comparação de DNA, devido ao desempenho superior em GPU.

Como trabalhos futuros, sugerimos:

- Integrar o MASA-SSE com o MASA-OpenMP, criando uma solução que explore tanto o paralelismo em nível de *threads* quanto em instruções vetoriais. Para tanto, propomos a adaptação do padrão *striped* para o padrão de execução paralela de blocos do MASA.
- Criar o MASA-AVX2, que utilize a extensão AVX2 da Intel. A extensão possui vetores maiores do que os presentes no SSE, permitindo que um maior paralelismo seja explorado, como por exemplo, processar 16 células em paralelo, ao invés de 8,

como implementado no MASA-SSE. Além disso, é possível otimizar o código para escores de até 32 *bits*, permitindo escores maiores dos que os suportados atualmente.

Referências

[1] Amos Bairoch, Brigitte Boeckmann, Serenella Ferro, and Elisabeth Gasteiger. Swiss-prot: juggling between evolution and stability. *Briefings in bioinformatics*, 5(1):39–55, 2004. 16, 18

[2] Darrell Boggs, Aravindh Baktha, Jason Hawkins, Deborah T Marr, J Alan Miller, Patrice Roussel, Ronak Singhal, Bret Toll, and KS Venkatraman. The microarchitecture of the intel pentium 4 processor on 90nm technology. *Intel Technology Journal*, 8(1), 2004. 14

[3] Edans F. de O. Sandes, Guilermo Miranda, Xavier Martorell, Eduard Ayguade, George Teodoro, and Alba C. M. A. de Melo. Masa: a multi-platform architecture for sequence aligners with block pruning, 2015. vi, vii, x, 1, 20, 21, 23

[4] Edans Flavius de O. Sandes and Alba Cristina M.A. de Melo. Retrieving smith-waterman alignments with optimizations for megabase biological sequences using gpu. *Parallel and Distributed Systems, IEEE Transactions on*, 24(5):1009–1021, May 2013. 20

[5] Richard Durbin. *Biological sequence analysis: probabilistic models of proteins and nucleic acids*. Cambridge university press, 1998. x, 1, 3, 5, 6, 7

[6] Michael Farrar. Striped smith–waterman speeds database searches six times over other simd implementations. *Bioinformatics*, 23(2):156–161, 2007. vi, vii, 1, 17, 18, 19

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

[8] Varghese George, Sanjeev Jahagirdar, Chao Tong, Ken Smits, Satish Damaraju, Scott Siers, Ves Naydenov, Tanveer Khondker, Sanjib Sarkar, and Puneet Singh. Penryn: 45-nm next generation intel® core™ 2 processor. In *Solid-State Circuits Conference, 2007. ASSCC'07. IEEE Asian*, pages 14–17. IEEE, 2007. 15, 33

[9] Osamu Gotoh. An improved algorithm for matching biological sequences. *Journal of Molecular Biology*, 162(3):705 – 708, 1982. 12

[10] Per Hammarlund, Rajesh Kumar, Randy B Osborne, Ravi Rajwar, Ronak Singhal, Reynold D'Sa, Robert Chappell, Shiv Kaushik, Srinivas Chennupaty, Stephan Jourdan, et al. Haswell: The fourth-generation intel core processor. *IEEE Micro*, (2):6–20, 2014. 15

[11] Leslie Kohn, Guillermo Maturana, Marc Tremblay, A Prabhu, and G Zyner. The visual instruction set (vis) in ultrasparc. In *compcon*, page 462. IEEE, 1995. 13, 15

[12] David W Mount. Sequence and genome analysis. *Bioinformatics: Cold Spring Harbour Laboratory Press: Cold Spring Harbour*, 2, 2004. 1, 3

[13] 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. 8

[14] C. NVidia. *NVIDIA CUDA Compute Unified Device Architecture - Programming Guide*. NVIDIA: Santa Clara, CA, 2007. x, 14

[15] Stuart Oberman, Greg Favor, and Fred Weber. Amd 3dnow! technology: Architecture and implementations. *Micro, IEEE*, 19(2):37–48, 1999. 14

[16] Alex Peleg, Sam Wilkie, and Uri Weiser. Intel mmx for multimedia pcs. *Communications of the ACM*, 40(1):24–38, 1997. 13

[17] Srinivas K Raman, Vladimir Pentkovski, and Jagannath Keshava. Implementing streaming simd extensions on the pentium iii processor. *IEEE micro*, (4):47–57, 2000. 14

[18] Jane B Reece, Lisa A Urry, Michael L Cain, Steven A Wasserman, Peter V Minorsky, and Robert B Jackson. *Campbell Biology*. Benjamin Cummings, 10 edition, 2013. x, 4

[19] Torbjørn Rognes. Faster smith-waterman database searches with inter-sequence simd parallelisation. *BMC bioinformatics*, 12(1):221, 2011. x, 18

[20] 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. x, 16, 17, 18, 19

[21] Efraim Rotem, Alon Naveh, Avinash Ananthakrishnan, Doron Rajwan, and Eliezer Weissmann. Power-management architecture of the intel microarchitecture code-named sandy bridge. *IEEE Micro*, (2):20–27, 2012. 15

[22] Ronak Singhal. Inside intel next generation nehalem microarchitecture. In *Hot Chips*, volume 20, 2008. 15

[23] T.F. Smith and M.S. Waterman. Identification of common molecular subsequences. *Journal of Molecular Biology*, 147(1):195 – 197, 1981. 1, 11

[24] Adam Szalkowski, Christian Ledergerber, Philipp Krähenbühl, and Christophe Dessimoz. Swps3–fast multi-threaded vectorized smith-waterman for ibm cell/be and× 86/sse2. *BMC Research Notes*, 1(1):107, 2008. 24

[25] Hugo Verli et al. Bioinformática da biologia à flexibilidade molecular. *Porto Alegre, Brasil*, 1, 2014. 1

[26] Andrzej Wozniak. Using video-oriented instructions to speed up sequence comparison. *Computer applications in the biosciences: CABIOS*, 13(2):145–150, 1997. 15, 16, 18, 19

[27] Mengyao Zhao, Wan-Ping Lee, Erik P Garrison, and Gabor T Marth. Ssw library: An simd smith-waterman c/c++ library for use in genomic applications. *PLOS One*, 2013. 24