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

Comparação de Sequências Biológicas Longas em FPGA usando Particionamento

Andressa Sousa da Silveira

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
2017

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

Comparação de Sequências Biológicas Longas em FPGA usando Particionamento

Andressa Sousa da Silveira

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. Ricardo Pezzuol Jacobi
CIC/UnB

Prof. Dr. Ricardo Pezzuol Jacobi
Coordenador do Curso de Engenharia da Computação

Brasília, 03 de Fevereiro de 2017

Dedicatória

Para Rondinele Prado, por nunca desistir de me mostrar que eu sou capaz.

Agradecimentos

Agradeço aos meus pais, que sempre lutaram para que eu alcançasse a minha formação acadêmica. Às minhas irmãs, por serem minha inspiração. Aos meus amigos que me deram apoio nos momentos de estresse. E à minha orientadora Alba Cristina Melo, pela dedicação e competência que tornaram possível a conclusão deste trabalho.

Resumo

Uma importante operação da Bioinformática é a comparação de sequências biológicas, que tem como resultado o escore ótimo determinando o grau de similaridade entre duas sequências. O escore ótimo é obtido por meio de algoritmos exatos que calculam uma matriz de programação dinâmica, possuindo complexidade quadrática $O(mn)$, onde m e n são os tamanhos das sequências.

As soluções em FPGA para a comparação de sequências biológicas atingiram altíssimo desempenho, porém essas soluções exigem que poucos milhares de caracteres das sequências sejam comparados a cada vez, ou seja, a comparação deve ser particionada.

O objetivo do presente trabalho de graduação é propor, implementar e avaliar uma solução em FPGA para a comparação exata de sequências biológicas com particionamento. Foram propostos dois circuitos de particionamento (v.1 e v.2), onde o circuito v.1 recebe a coluna, linha e diagonal intermediárias da matriz de programação dinâmica, calcula uma parte da matriz e produz o escore. O circuito v.2, da mesma maneira, recebe a coluna, linha e diagonal intermediárias da matriz de programação dinâmica, calcula uma parte da matriz e produz o escore, linha, coluna e diagonal finais da partição da matriz que foi calculada.

Os resultados experimentais mostram uma boa ocupação do FPGA com elementos de processamento. Além disso, a execução do circuito de particionamento v.1 na plataforma XD2000i para a comparação de sequências biológicas de 20 caracteres cada apresenta tempos bastante satisfatórios (até $948\mu s$).

Palavras-chave: Bioinformática, comparação de sequência biológicas, particionamento, matriz de programação dinâmica, Smith-Waterman, Gotoh, FPGA, XD2000i, CUDAlign, MASA

Abstract

An important operation in Bioinformatics is the biological sequence comparison, which has as result the best score that determines how similar are two sequences. The best score is obtained with the computation of the dynamic programming matrix by exact algorithms, whose complexities are $O(mn)$, where m and n are the size of the sequences.

Solutions in FPGA for biological sequence comparison reached great performances, however these solutions require a few thousand characters of the sequences being compared at a time, in other words, the comparison must be partitioned.

The objective of this graduation work is to propose, implement and evaluate a solution in FPGA for the biological sequence comparison with partitioning. Two partitioning circuits (v.1 and v.2) were proposed, where circuit v.1 receives the intermediate column, row and diagonal from the dynamic programming matrix, computes a part of the matrix and produces the score. In the same way, the circuit v.2 receives the intermediate column, row and diagonal from the dynamic programming matrix, computes a part of the matrix and produces the score, last column, row and diagonal of the computed partition of the matrix.

The experimental results show a great occupation of the FPGA with processing elements. Besides, the execution of the partitioning circuit v.1 in the XtremeData XD2000i^{TM} for the biological sequence comparison with both sequences having 20 characters present very satisfactory times (up to $948\mu s$).

Keywords: Bioinformatics, biological sequence comparison, partitioning, dynamic programming matrix, Smith-Waterman, Gotoh, FPGA, XD2000i, CUDAlign, MASA

Sumário

1 Introdução **1**

2 Comparação de Sequências Biológicas **3**
2.1 Alinhamento e Escore . . . 3
2.2 Needleman-Wunsh (NW) . . . 4
2.3 Smith-Waterman (SW) . . . 6
2.4 Gotoh . . . 7
2.5 Myers-Miller (MM) . . . 8

3 CUDAlign **10**
3.1 Histórico . . . 10
3.1.1 CUDAlign 1.0 . . . 10
3.1.2 CUDAlign 2.0 . . . 11
3.1.3 CUDAlign 2.1 . . . 13
3.1.4 CUDAlign 3.0 . . . 13
3.1.5 CUDAlign 4.0 . . . 14
3.1.6 Arquitetura MASA . . . 14
3.2 Multi-platform Architecture for Sequence Aligners (MASA) . . . 14
3.2.1 Visão geral . . . 15
3.2.2 Como integrar novas implementações . . . 17

4 Plataforma XD2000i^{TM} **19**
4.1 Componentes da plataforma XD2000i . . . 19
4.2 Projeto do *design* de referência . . . 20
4.3 Aplicações do módulo XD2000i . . . 22
4.4 FPGA *Bridge* . . . 23
4.5 Protocolo FIFO da AFU . . . 23
4.5.1 Interface FIFO tipo AFU . . . 24
4.5.2 Interface FIFO tipo DRV . . . 24
4.6 Configuração da plataforma XD2000i . . . 25

4.6.1 Estabelecimento da comunicação com a FPGA *Bridge* 25
4.6.2 Programação da FPGA de aplicação 26

5 O Projeto do Circuito de Particionamento 27
5.1 Solução de Colletti (2016) 27
5.1.1 Método *diagonal wavefront* DW 27
5.1.2 Estrutura sistólica 28
5.1.3 Elemento de processamento 28
5.1.4 Cálculo da equação de recorrência 30
5.1.5 Cálculo do escore máximo local 31
5.1.6 Unidade de controle 32
5.2 Circuito de Particionamento proposto v.1 32
5.3 Circuito de Particionamento proposto v.2 35
5.4 Integração com a plataforma XD2000i 38

6 Resultados Experimentais 39
6.1 Sequências utilizadas 39
6.2 Simulação 40
6.3 Síntese do circuito de particionamento v.1 44
6.4 Execução do circuito de particionamento v.1 na plataforma XD2000i 45

7 Conclusão 47

Referências 48

Lista de Figuras

2.1 Alinhamento entre as sequências $S_0 = ATCACGA$ e $S_1 = ACATGC$ e seu escore. 4
2.2 Alinhamento global ótimo entre as sequências $S_0 = ATGAGATCCAGTAT$ e $S_1 = AGATTTCCAGTT$. 5
2.3 Matriz de programação dinâmica gerada pela comparação entre as sequências $S_0 = ATGAGATCCAGTAT$ e $S_1 = AGATTTCCAGTT$ com o algoritmo NW. 6
2.4 Alinhamento local ótimo entre as sequências $S_0 = ATGAGATCCAGTAT$ e $S_1 = AGATTTCCAGTT$. 7
2.5 Matriz de programação dinâmica gerada pela comparação entre as sequências $S_0 = ATGAGATCCAGTAT$ e $S_1 = AGATTTCCAGTT$ com o algoritmo SW. 7
2.6 O algoritmo Myers-Miller realizado com dois níveis de recursão. Fonte: [22]. 9

3.1 CUDAlign 2.0 executado em seis estágios. Fonte: [22]. 12
3.2 Arquitetura MASA. Fonte: [22]. 15
3.3 Processamento em partições coordenado pelo módulo Gerenciamento dos Estágios. Fonte: [22]. 16
3.4 Extensões MASA com os componentes MASA utilizados em cada uma. Fonte: [22]. 18

4.1 Componentes da plataforma XD2000i. Fonte: [12]. 20
4.2 Design de referência da FPGA de aplicação. Fonte: [12]. 21
4.3 Fluxo de dados do sistema. Fonte: [12]. 21

5.1 O método *diagonal wavefront*. Fonte: [4]. 28
5.2 A estrutura sistólica linear unidirecional. Fonte: [4]. 29
5.3 Estrutura interna inicial do elemento de processamento. Fonte: [4]. 29
5.4 Circuito combinacional para o cálculo do valor na diagonal com as pontuações de *match* e *mismatch*. Fonte: [4]. 30

5.5 Circuito combinacional para o cálculo dos valores na linha superior e coluna à esquerda com a pontuação de *gap*. Fonte: [4]. 30
5.6 Circuito implementado por Carvalho com cinco elementos de processamento. Fonte: [5]. 31
5.7 Circuito da Figura 5.6 encapsulado. Fonte: [5]. 31
5.8 Estrutura sequencial para o cálculo do escore máximo local. Fonte: [5]. . . 32
5.9 Máquina de estados implementada na unidade de controle. Fonte: [5]. . . . 33
5.10 (a) Cálculo sem particionamento. (b) Cálculo com particionamento. As áreas com o cinza mais escuro representam a parte da matriz de programação dinâmica sendo calculada no momento. Com isso, nota-se que a matriz da Figura 5.10(a), de tamanho 20×20, deve ser calculada por inteiro de uma vez. Já na Figura 5.10(b), pode-se observar que a matriz está dividida em várias partes delimitadas pelas partições das sequências biológicas sendo comparadas. Cada uma das partes pode ser calculada por vez, o que determina um tamanho ilimitado para as sequências. 34
5.11 Formato das entradas e saídas do circuito de particionamento v1. As áreas hachuradas representam bits ignorados. 35
5.12 Circuito combinacional para determinar os valores da linha superior e diagonal de acordo com o valor registrado pelo contador dentro de um PE. O contador começa a ser incrementado apenas quando FLAG_IN é setado. HORI representa o valor na coluna à esquerda do PE no ciclo de *clock* anterior. DATA_NOV representa o valor calculado no interior do PE no ciclo de *clock* anterior. DIAG_IN e ROW_CEL_IN representam, caso trata-se dos PEs 2 a n, o valor de uma célula da linha inicial da matriz de programação dinâmica. Caso trata-se do PE1, DIAG_CEL é a diagonal inicial da matriz de programação dinâmica. LINHA e DIAG serão utilizados posteriormente nos circuitos combinacionais apresentados nas Figuras 5.4 e 5.5. 36
5.13 Formato das entradas e saídas do circuito de particionamento v2. 36
5.14 Obtenção da última coluna, linha e diagonal da matriz de programação dinâmica resultante da comparação entre as sequências S_0, de tamanho $n = 5$, e S_1, de tamanho $m = 5$. A última coluna e linha serão obtidas a partir do momento em que contador da unidade de controle registrar que o ciclo de *clock* corrente é maior que n e m, respectivamente. A última diagonal é obtida quando o contador registrar que o ciclo de *clock* corrente é igual a $m + n = 10$. 37
5.15 O módulo que customiza o projeto de design de referência. 38

6.1 Resultado da simulação do circuito de particionamento v.1 (instantes 430 ns a 470 ns) realizada com a sequência de consulta S_C(a) e a sequência de banco de dado S_{BD}(a) mostradas na Tabela 6.2. 41
6.2 Resultado da simulação do circuito de particionamento v.1 realizada com a sequência de consulta S_C(b) e a sequência de banco de dado S_{BD}(b) mostradas na Tabela 6.2. 41
6.3 Resultado da simulação do circuito de particionamento v.1 realizada com a sequência de consulta S_C(c) e a sequência de banco de dado S_{BD}(c) mostradas na Tabela 6.2. 42
6.4 Resultado da simulação do circuito de particionamento v.2 (instantes 430 ns a 470 ns) realizada com a sequência de consulta S_C(a) e a sequência de banco de dado S_{BD}(a) mostradas na Tabela 6.2. 42
6.5 Resultado da simulação do circuito de particionamento v.2 realizada com a sequência de consulta S_C(a) e a sequência de banco de dado S_{BD}(a) mostradas na Tabela 6.2. 43
6.6 Resultado da simulação do circuito de particionamento v.2 realizada com a sequência de consulta S_C(b) e a sequência de banco de dado S_{BD}(b) mostradas na Tabela 6.2. 43
6.7 Resultado da simulação do circuito de particionamento v.2 realizada com a sequência de consulta S_C(c) e a sequência de banco de dado S_{BD}(c) mostradas na Tabela 6.2. 44
6.8 Resultado da síntese no Quartus II 8.1 [1] do circuito de particionamento v.1 implementado com 1024 PEs. 45
6.9 Resultado da execução do circuito de particionamento v.1 na plataforma XD2000i utilizando a sequência de consulta S_C(a) e a sequência de banco de dado S_{BD}(a) mostradas na Tabela 6.2. 46
6.10 Resultado da execução do circuito de particionamento v.1 na plataforma XD2000i utilizando a sequência de consulta S_C(b) e a sequência de banco de dado S_{BD}(b) mostradas na Tabela 6.2. 46
6.11 Resultado da execução do circuito de particionamento v.1 na plataforma XD2000i utilizando a sequência de consulta S_C(c) e a sequência de banco de dado S_{BD}(c) mostradas na Tabela 6.2. 46

Lista de Tabelas

6.1 Sequências biológicas geradas de forma sintética. 39
6.2 Partições das sequências biológicas mostradas na Tabela 6.1. 40
6.3 Codificação da sequência de banco de dados S_{BD} em binário e em hexadecimal. 40
6.4 Resultados das sínteses do circuito de particionamento v.1 implementando diferentes quantidades de PEs. 44
6.5 Resultados das execuções do circuito de particionamento v.1 na plataforma XD2000i com as sequências de consulta S_C e as sequências de banco de dados S_{BD} mostradas na Tabela 6.2. 46

Capítulo 1

Introdução

A Bioinformática é um campo interdisciplinar que engloba a ciência da computação, a biologia e a matemática. Ela consiste na coleta e análise de dados biológicos com o auxílio de ferramentas computacionais e algoritmos. Uma importante operação da Bioinformática é a comparação de sequências biológicas, que tem como resultado o escore ótimo determinando o grau de similaridade entre duas sequências. O escore ótimo é obtido por meio de algoritmos exatos que calculam uma matriz de programação dinâmica, possuindo complexidade quadrática $O(mn)$, onde m e n são os tamanhos das sequências.

Devido ao grande tempo de execução dos algoritmos exatos, soluções paralelas foram propostas na literatura em GPU (*Graphics Processing Units*) [14], Intel Phi [13], FPGA (*Field Programmable Gate Arrays*) [27], entre outros. As soluções em FPGA atingiram altíssimo desempenho, obtendo 3,04 trilhões de células atualizadas por segundo (TCUPS) em 128 FPGAs. No entanto, em FPGA o tamanho das sequências biológicas é restringido, acomodando no máximo até 2048 caracteres de cada sequência em uma FPGA. Desse modo, a execução da operação de comparação de sequências biológicas deve ser feita em partes, comparando partições das sequências por vez. Isso resulta no cálculo de uma parte da matriz de programação dinâmica a cada execução.

O objetivo do presente trabalho de graduação é propor, implementar e avaliar uma solução em FPGA para a comparação exata de sequências biológicas com particionamento, visando a posterior integração com a ferramenta MASA [25]. Foram propostos dois circuitos de particionamento (v.1 e v.2), onde o circuito v.1 recebe a coluna, linha e diagonal intermediárias da matriz de programação dinâmica, calcula uma parte da matriz e produz o escore. O circuito v.2, da mesma maneira, recebe a coluna, linha e diagonal intermediárias da matriz de programação dinâmica, calcula uma parte da matriz e produz o escore, linha, coluna e diagonal finais da partição da matriz que foi calculada.

A plataforma hardware *XtremeData* XD2000i^{TM} foi escolhida para a execução em FPGA do projeto aqui apresentado. Esta ferramenta é importante, pois possibilita a

transferência de dados sequencialmente entre a CPU e a FPGA através de uma API (*Application Programming Interface*) de alto nível, o que simplifica bastante a programação da interface entre o *host* e a FPGA.

O presente documento está organizado da seguinte maneira. O capítulo 2 apresenta algoritmos exatos básicos para a comparação de sequências biológicas. O capítulo 3 descreve a arquitetura MASA, assim como todas as versões anteriores a ela. O capítulo 4 descreve a plataforma XD2000i. O capítulo 5 apresenta as soluções propostas para os circuitos de particionamento v.1 e v.2. O capítulo 6 apresenta os resultados experimentais de simulações e sínteses no software Quartus II 8.1 [1] e execuções na plataforma XD2000i. Finalmente, o capítulo 7 apresenta a conclusão da monografia e sugere trabalhos futuros.

Capítulo 2

Comparação de Sequências Biológicas

Neste capítulo serão apresentados algoritmos exatos de comparação de sequências biológicas (Seções 2.2 a 2.5), além de definir termos comuns utilizados nos capítulos deste trabalho de graduação (Seção 2.1).

2.1 Alinhamento e Escore

A comparação de sequências biológicas é uma operação importante para a determinação de informações funcionais, estruturais e evolutivas em sequências de DNA, RNA ou proteínas. Se duas sequências são semelhantes entre si, é um forte indicativo de que elas possuem um ancestral comum e, caso isso se comprove, as mudanças que ocorreram durante suas evoluções podem ser identificadas, além de existirem funções provavelmente similares.

Na Bioinformática, a comparação produz um alinhamento entre sequências de DNA, RNA ou proteínas. É importante salientar que estas sequências são representadas textualmente através de um conjunto de caracteres que são chamados de alfabeto $\sum$. Para as sequências de DNA, o alfabeto é $\sum = \{A, C, G, T\}$, para as sequências de RNA é $\sum = \{A, C, G, U\}$ e para as sequências de aminoácidos é $\sum = \{A, C, D, E, F, G, H, I, K, L, M, N, P, Q, R, S, T, V, W, Y\}$ [15].

O alinhamento pode ser feito dispondo as sequências uma logo abaixo da outra gerando duas linhas, a fim de buscar uma série de caracteres que sejam similares e possuam a mesma ordem nas duas sequências. Caso em uma coluna haja caracteres iguais, chama-se de *match*. Caso contrário, ou chama-se de *mismatch*, se os caracteres não forem idênticos, ou um espaço é inserido (*gap*) em uma das posições da coluna [9].

Existem dois tipos básicos de alinhamentos: o global e o local. Para produzir um alinhamento global ótimo entre pares de sequências biológicas, utiliza-se o algoritmo Needleman-Wunsh (NW) [17]. Neste alinhamento, as sequências são consideradas por completo, utilizando-se todos os seus caracteres. Neste caso, os *gaps* podem ser bastante utéis para melhorar a qualidade do alinhamento. O algoritmo Smith-Waterman (SW) [26] é usado para produzir um alinhamento local entre pares de sequências. No alinhamento local, o alinhamento termina ao fim de regiões com forte similaridade, gerando o alinhamento em uma substring. Sua prioridade é encontrar essas regiões com alta densidade de *matches* [9].

Cada alinhamento entre pares de sequência biológicas possui um escore. Através da comparação de cada par de caracteres em um alinhamento, *matches*, *mismatches* e *gaps* são gerados e um valor é associado a cada um deles por meio de um determinado sistema de pontuação. O escore do alinhamento é calculado ao somar-se todos estes valores e aquele alinhamento que tiver o maior escore dentre todos os obtidos entre as mesmas sequências é o alinhamento ótimo [15].

A Figura 2.1 ilustra um alinhamento e seu escore. Nessa figura, as pontuações para *matches* (ma), *mismatches* (mi) e *gaps* (G) são, respectivamente, +1, -1 e -2.

$$\underbrace{\begin{array}{ccccccc} A & T & C & A & C & G & A \\ A & - & C & A & T & G & C \\ \hline +1 & -2 & +1 & +1 & -1 & +1 & -1 \end{array}}_{score\,=\,0}$$

Figura 2.1: Alinhamento entre as sequências $S_0 = ATCACGA$ e $S_1 = ACATGC$ e seu escore.

2.2 Needleman-Wunsh (NW)

O algoritmo Needleman-Wunsh (NW) [17] tem por objetivo fornecer o alinhamento global ótimo de sequências biológicas. O algoritmo é dividido em duas fases de processamento: (1) cálculo da matriz de programação dinâmica e (2) obtenção do alinhamento ótimo (*traceback*).

Fase 1 - Cálculo da matriz de programação dinâmica: Na primeira fase, sequências S_0 e S_1 de tamanhos m e n, respectivamente, são fornecidas. É calculada uma matriz de programação dinâmica H da seguinte maneira. Primeiramente, escreve-se horizontalmente uma sequência no topo da matriz e a outra escreve-se verticalmente ao lado esquerdo desta. Na primeira posição, o escore é $H_{0,0} = 0$, que equivale à ocorrência de

nenhum *gap* nas duas sequências. Uma primeira linha e coluna extras são adicionadas para possibilitar que o alinhamento leve em consideração *gaps* no início do alinhamento. Dessa forma, a pontuação em cada posição corresponde a $H_{i,0} = -i \cdot G$ e a $H_{0,j} = -j \cdot G$.

Para alcançar uma posição (i, j), pode-se escolher entre 3 caminhos: (1) um movimento diagonal da posição $(i-1, j-1)$ para a posição (i, j), com nenhuma penalidade de *gap* G, (2) um movimento de $i-1$ na coluna j para (i, j), com uma penalidade de *gap* G, ou (3) um movimento de $j-1$ na coluna i para (i, j), com uma penalidade de *gap* G.

A equação de recorrência do NW é descrita pela Equação 2.1:

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

onde $sbt(S_0[i], S_1[j]) = ma$, se $S_0[i] = S_1[j]$, e $sbt(S_0[i], S_1[j]) = mi$, caso contrário.

Fase 2 - Obtenção do alinhamento ótimo (*traceback*): Cada célula $H_{i,j}$ da matriz guarda uma indicação (ponteiro) para a célula que foi usada na equação de recorrência ($H_{i-1,j-1}$, $H_{i,j-1}$ ou $H_{i-1,j}$). Estes ponteiros são usados na fase 2 para construir um caminho de maiores escores através da matriz, a fim de obter o alinhamento global ótimo. Na fase 2, o processamento inicia-se na posição final $H_{m,n}$, que possui o escore ótimo. A partir dessa célula, os ponteiros são percorridos de trás para frente até que se atinja a célula inicial da matriz. Se existe uma seta na diagonal em $H_{i,j}$, $S_0[i]$ é alinhado com $S_1[j]$; se há uma seta para cima, $S_0[i]$ é alinhado com espaço (-) e se há uma seta da direita para a esquerda, $S_1[j]$ é alinhado com espaço.

A Figura 2.2 apresenta um exemplo de um alinhamento global ótimo com escore 4 entre duas sequências de DNA. O sistema de pontuação utilizado foi: *Match*: +1; *Mismatch*: -1; *Gap*: -2. A Tabela 2.3 apresenta a matriz de programação dinâmica para este alinhamento global ótimo.

$$\underbrace{\begin{array}{cccccccccccccc} A & T & G & A & G & A & T & C & C & A & G & T & A & T \\ A & - & G & A & T & T & T & C & C & A & G & T & - & T \\ \hline +1 & -2 & +1 & +1 & -1 & -1 & +1 & +1 & +1 & +1 & +1 & +1 & -2 & +1 \end{array}}_{score = 4}$$

Figura 2.2: Alinhamento global ótimo entre as sequências $S_0 = ATGAGATCCAGTAT$ e $S_1 = AGATTTCCAGTT$.

	*	A	T	G	A	G	A	T	C	C	A	G	T	A	T
*	**0**	−2	−4	−6	−8	−10	−12	−14	−16	−18	−20	−22	−24	−26	−28
A	−2	**1**	← **-1**	−3	−5	−7	−9	−11	−13	−15	−17	−19	−21	−23	−25
G	−4	−1	0	**0**	−2	−4	−6	−8	−10	−12	−14	−16	−18	−20	−22
A	−6	−3	−2	−1	**1**	−1	−3	−5	−7	−9	−11	−13	−15	−17	−19
T	−8	−5	−2	−3	−1	**0**	−2	−2	−4	−6	−8	−10	−12	−14	−16
T	−10	−7	−4	−3	−3	−2	**-1**	−1	−3	−5	−7	−9	−9	−11	−13
T	−12	−9	−6	−5	−4	−4	−3	**0**	−2	−4	−6	−8	−8	−10	−10
C	−14	−11	−8	−7	−6	−5	−5	−2	**1**	−1	−3	−5	−7	−9	−11
C	−16	−13	−10	−9	−8	−7	−6	−4	−1	**2**	0	−2	−4	−6	−8
A	−18	−15	−12	−11	−8	−9	−6	−6	−3	0	**3**	1	−1	−3	−5
G	−20	−17	−14	−11	−10	−7	−8	−7	−5	−2	1	**4**	2	0	−2
T	−22	−19	−16	−13	−12	−9	−8	−7	−7	−4	−1	2	**5**	← **3**	1
T	−24	−21	−18	−15	−14	−11	−10	−7	−8	−6	−3	0	3	−1	**4**

Figura 2.3: Matriz de programação dinâmica gerada pela comparação entre as sequências $S_0 = ATGAGATCCAGTAT$ e $S_1 = AGATTTCCAGTT$ com o algoritmo NW.

2.3 Smith-Waterman (SW)

O algoritmo Smith-Waterman (SW) [26] produz o alinhamento local ótimo de sequências biológicas. O cálculo da matriz de programação dinâmica possui duas diferenças em comparação com o cálculo do algoritmo Needleman-Wunsch (Seção 2.2): (a) a primeira linha e coluna são preenchidas com zeros e (b) a equação de recorrência de SW é descrita pela Equação 2.2 [26]:

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

Da mesma forma que no algoritmo NW, são mantidos os caminhos dos ponteiros na matriz. Entretanto, ao invés do algoritmo SW iniciar o *traceback* na última posição $H_{m,n}$, ele inicia na posição $H_{i,j}$ que contém o maior escore da matriz e termina quando o valor 0 é encontrado. Dessa forma, o melhor alinhamento local é obtido entre duas *substrings*.

A Figura 2.4 apresenta um exemplo de um alinhamento local ótimo com escore 6 entre

duas sequências de DNA. O sistema de pontuação utilizado foi: *Match*: +1; *Mismatch*: -1; *Gap*: -2. A Tabela 2.5 apresenta a matriz de programação dinâmica para essa comparação.

$$\underbrace{\begin{array}{cccccc} T & C & C & A & G & T \\ T & C & C & A & G & T \\ \hline +1 & +1 & +1 & +1 & +1 & +1 \end{array}}_{score = 6}$$

Figura 2.4: Alinhamento local ótimo entre as sequências $S_0 = ATGAGATCCAGTAT$ e $S_1 = AGATTTCCAGTT$.

	*	A	T	G	A	G	A	T	C	C	A	G	T	A	T
*	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
A	0	1	0	0	1	0	1	0	0	0	1	0	0	1	0
G	0	0	0	1	0	2	0	0	0	0	0	2	0	0	0
A	0	1	0	0	2	0	3	1	0	0	1	0	1	1	0
T	0	0	2	0	0	1	1	4	2	0	0	0	1	0	2
T	0	0	1	1	0	0	**0**	2	3	1	0	0	1	0	1
T	0	0	1	0	0	0	0	↖ **1**	1	2	0	0	1	0	1
C	0	0	0	0	0	0	0	0	↖ **2**	2	1	0	0	0	0
C	0	0	0	0	0	0	0	0	1	↖ **3**	1	0	0	0	0
A	0	1	0	0	1	0	1	0	0	1	↖ **4**	2	0	1	0
G	0	0	0	1	0	2	0	0	0	0	2	↖ **5**	3	1	0
T	0	0	1	0	0	0	1	1	0	0	0	3	↖ **6**	4	2
T	0	0	1	0	0	0	0	2	0	0	0	1	4	5	5

Figura 2.5: Matriz de programação dinâmica gerada pela comparação entre as sequências $S_0 = ATGAGATCCAGTAT$ e $S_1 = AGATTTCCAGTT$ com o algoritmo SW.

2.4 Gotoh

A Equação 2.1 pode ser generalizada ao substituir a constante G pela função de penalidade $\gamma(k)$, onde k é o comprimento de uma sequência consecutiva de gaps. Um caso particular para esta generalização é o modelo *affine gap*, que considera uma penalidade G_{first} pela abertura de um *gap* e outra penalidade G_{ext} para os *gaps* subsequentes [15]. Sua função de penalidade $\gamma(k)$ é dada por $\gamma(k) = -G_{first} - (k-1) \cdot G_{ext}$, onde $-G_{first}$

é a penalidade para o primeiro *gap* de uma sequência consecutiva de *gaps* e G_{ext} é a penalidade para os demais *gaps* da mesma sequência. O algoritmo de Gotoh [10] calcula o alinhamento global ótimo com complexidade $O(n^2)$ para este modelo.

A matriz de programação dinâmica é construída de forma que, em cada posição (i, j), serão mantidas três variáveis resultantes de três situações distintas: (1) $S_0[i]$ alinhado com $S_1[j]$, (2) um *gap* alinhado com $S_1[j]$ e (3) $S_0[i]$ alinhado com um *gap*. Para cada situação, define-se as matrizes H, E e F.

Portanto, a Equação 2.1 é modificada de forma que se obtenha as fórmulas de recorrência descritas pelas Equações 2.3, 2.4 e 2.5. Essas fórmulas podem ser adaptadas para a obtenção de um alinhamento local ótimo ao limitar-se o valor mínimo da matriz de programação dinâmica H para 0.

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

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

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

2.5 Myers-Miller (MM)

Os algoritmos vistos nas Seções 2.2, 2.3 e 2.4 possuem uma complexidade de memória $O(n^2)$, que é um fator limitante quando deseja-se encontrar o alinhamento ótimo entre sequências longas. Desse modo, foram implementados algoritmos de complexidade de memória $O(n)$, onde n é o tamanho da menor sequência.

O algoritmo de Hirschberg [11], que foi desenvolvido para resolver o problema da Maior Subsequência Comum (LCS - *Longest Common Subsequence*), foi adaptado por Myers-Miller (MM) [16] para que o algoritmo de Gotoh (Seção 2.4) tivesse uma versão em espaço linear. O algoritmo MM visa encontrar o ponto médio (*crosspoint*) pelo qual passa um alinhamento ótimo e, para isso, a computação deste algoritmo é realizada em duas partes. Na primeira parte, a computação é feita por linhas e termina na linha central $\frac{m}{2}$. Na segunda parte, as sequências são invertidas e é feita novamente uma computação que termina na mesma linha central $\frac{m}{2}$.

Os valores finais da linha $\frac{m}{2}$ resultante da primeira parte são armazenados nos vetores CC, que contém os escores dos alinhamentos terminados em *gap*, e DD, que contém os escores dos alinhamentos terminados em *match* ou *mismatch*. Da mesma forma, a computação realizada nas sequências invertidas tem seus valores finais da linha $\frac{m}{2}$ armazenados nos vetores CC' e DD'.

O escore K_j de um alinhamento que cruza a linha $\frac{m}{2}$ e a coluna j é calculado por $K_j = max\{CC_j + CC'_j, DD_j + DD'_j - G_{open}\}$, onde G_{open} é a penalidade por abrir um *gap*.

O valor máximo K_{j^*}, definido por $K_{j^*} = max_{0 \le j \le n} K_j$, é encontrado dentre todos os valores de K_j, determinando que a célula $(\frac{m}{2}, j^*)$ é o ponto médio pelo qual passa o alinhamento ótimo.

Em seguida, essa computação é realizada recursivamente em mais duas seções da matriz: a primeira é de $(0, 0)$ a $(\frac{m}{2}, j^*)$ e a segunda é de $(\frac{m}{2}, j^*)$ a (m, n). A cada repetição desta computação, as seções da matriz vão ficando cada vez menores até se tornarem triviais. Conclui-se, então, que apenas duas linhas são necessárias para comparar as duas sequências, o que nos dá a complexidade linear de memória. Na Figura 2.6, as áreas mais escuras apresentam as seções da matriz utilizadas para os dois primeiros níveis de recursão.

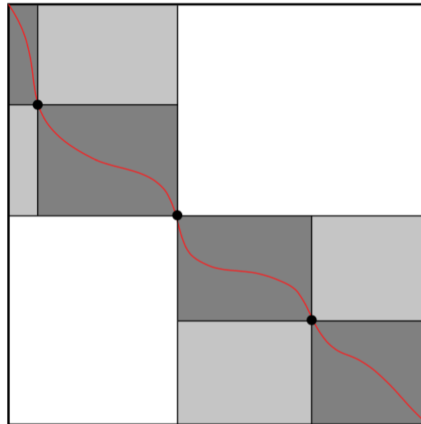


Figura 2.6: O algoritmo Myers-Miller realizado com dois níveis de recursão. Fonte: [22].

Capítulo 3

CUDAlign

O CUDAlign é a ferramenta desenvolvida por Sandes [22] que permite alto grau de paralelismo para a recuperação eficiente do alinhamento ótimo de sequências muito longas de DNA em plataformas de alto desempenho. O CUDAlign foi primeiramente proposto para processar em múltiplas GPUs da NVIDIA, mas, posteriormente, a arquitetura de software chamada de Multi-Platform Architecture for Sequence Aligners (MASA) foi criada para possibilitar a portabilidade do CUDAlign para diferentes plataformas de *hardware* e *software*. Nas seções a seguir, serão descritas mais profundamente estas duas ferramentas propostas.

3.1 Histórico

Nesta seção, serão apresentadas as diversas versões pelas quais evoluiu o software CUDAlign desenvolvido por Sandes [22].

3.1.1 CUDAlign 1.0

O CUDAlign 1.0 foi desenvolvido na plataforma CUDA da NVIDIA, com o propósito de obter o escore da comparação de sequências biológicas longas de DNA utilizando o algoritmo Smith-Waterman com *affine-gap* (Seção 2.4) e memória linear em uma GPU [19]. O cálculo da equação de recorrência do SW com *affine-gap* foi implementado de modo a obter paralelismo utilizando a técnica de *wavefront* em dois níveis: o paralelismo externo e o paralelismo interno.

No primeiro nível de paralelismo, chamado paralelismo externo, as células da matriz de programação são agrupadas em blocos com R linhas e C colunas, resultando em um grid G com B colunas de blocos. Este agrupamento cria anti-diagonais, chamadas *diagonais externas*, onde a técnica de *wavefront* é aplicada para obter o paralelismo.

O outro nível de paralelismo, o paralelismo interno, ocorre dentro de um bloco. Cada bloco possui um número T de *threads* que trabalham em conjunto para processar $R \times C$ células do bloco, onde cada *thread* é responsável por processar α linhas. Analogamente ao que é feito no paralelismo externo, as células dentro do bloco são agrupadas formando as *diagonais internas*, sendo processadas pelas *threads* também utilizando a técnica de *wavefront*.

3.1.2 CUDAlign 2.0

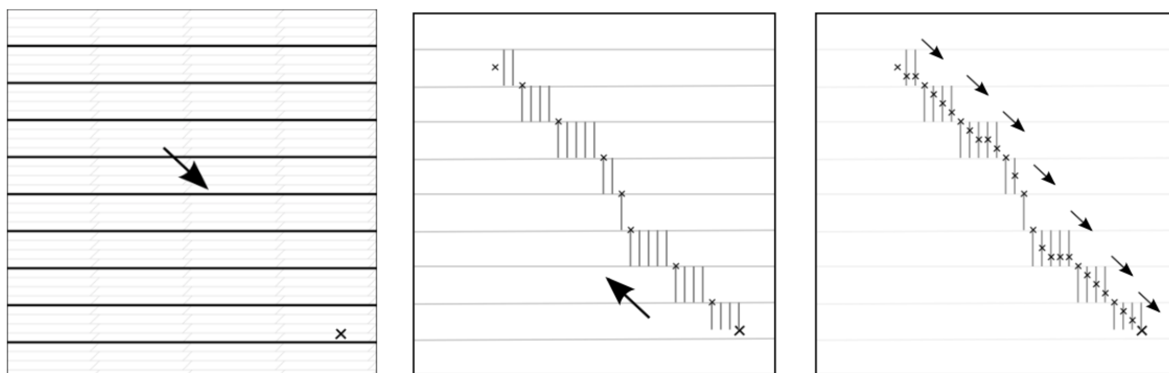
O CUDAlign 2.0 foi baseado nos algoritmo de Myers e Miller (Seção 2.5) e FastLSA [7] e tem como objetivo obter o alinhamento local ótimo de sequências longas de DNA com uso linear de memória e tempo reduzido [20]. Algumas coordenadas do alinhamento ótimo são encontradas e então são incrementadas iterativamente até obter o alinhamento completo utilizando uma quantidade restrita de memória. Foram atingidos 23,1 bilhões de células processadas por segundo (GCUPS) em uma GPU. O CUDAlign 2.0 foi dividido em seis estágios descritos a seguir.

Estágio 1

No Estágio 1, o escore ótimo e a coordenada final do alinhamento são encontrados utilizando o modelo *affine-gap* (Seção 2.4). Neste estágio, o CUDAlign 1.0 (Seção 3.1.1) é executado, além de salvar algumas linhas da matriz em disco, denotadas *linhas especiais*. Elas serão utilizadas posteriormente no Estágio 2 e para efetuar *check-points*, que irão auxiliar na restauração da execução caso o algoritmo seja interrompido. A Figura 3.1(a) apresenta um exemplo de saída do Estágio 1. A cada 4 fileiras de blocos, existem linhas mais escuras representando as linhas especiais. O escore ótimo está posicionado no "$\times$".

Estágio 2

O Estágio 2 tem como objetivo encontrar a coordenada inicial do alinhamento e todas as coordenadas do alinhamento ótimo que cruzam as linhas especias obtidas no Estágio 1. A Figura 3.1(b) ilustra as saídas do Estágio 2. Foram aplicadas duas otimizações neste estágio: procedimento de *matching* baseado em objetivo e execução ortogonal. No procedimento de *matching* baseado em objetivo, diferentemente do algoritmo original de Myers-Miller, ao encontrar-se o escore máximo já conhecido, chamado escore-alvo, o procedimento é encerrado e uma nova iteração na próxima linha especial é iniciada, a fim de buscar a próxima coordenada. O escore-alvo, inicialmente obtido do Estágio 1, é então atualizado com o escore da última coordenada obtida da linha especial. Para ganhar desempenho ao realizar o procedimento de *matching* baseado em objetivo, a execução



(a) O Estágio 1 encontra o escore ótimo e sua posição. Linhas especiais são salvas em disco.

(b) O Estágio 2 encontra as coordenadas nas quais o alinhamento ótimo intercepta as linhas especiais. Colunas especiais são salvas em disco.

(c) O Estágio 3 encontra as coordenadas que interceptam o alinhamento ótimo pelas colunas especiais salvas no Estágio 2.

(d) O Estágio 4 executa o algoritmo de Myers e Miller em cada partição formada por coordenadas sucessivas.

(e) O Estágio 5 obtém o alinhamento completo concatenando o alinhamento de cada partição.

(f) O Estágio 6 permite a visualização textual e gráfica do alinhamento ótimo obtido.

Figura 3.1: CUDAlign 2.0 executado em seis estágios. Fonte: [22].

ortogonal é efetuada. Ao invés de executar as *threads* na direção horizontal, assim como é feito no Estágio 1, as *threads* são executadas verticalmente, em direção ortogonal ao Estágio 1.

Estágio 3

No Estágio 3, partições bem definidas com começo e fim são formadas, diferentemente do Estágio 2, que conhece apenas a coordenada final e tem de encontrar a coordenada inicial. A partir dessas coordenadas bem definidas, o Estágio 3 consegue então aumentar o número de coordenadas conhecidas que cruzam o alinhamento ótimo. Na Figura 3.1(c) pode ser visto um exemplo de saída do Estágio 3.

Estágio 4

Com as partições formadas por coordenadas sucessivas encontradas no Estágio 3, o Estágio 4 tem como objetivo, através da aplicação do algoritmo Myers-Miller em cada partição, aumentar iterativamente o número de coordenadas do alinhamento ótimo até que as partições formadas entre as coordenadas tenham um tamanho menor que a constante *tamanho máximo de partição*. A otimização *divisão balanceada* foi aplicada no algoritmo de Myers-Miller, que define que a matriz pode ser dividida tanto em sua linha central quanto em sua coluna central, e a execução ortogonal foi utilizada para obter os *crosspoints*. A Figura 3.1(d) ilustra uma iteração do Estágio 4.

Estágio 5

No Estágio 5, ocorre a obtenção do alinhamento ótimo completo através da concatenação do alinhamento em CPU de cada partição obtida no Estágio 4. O alinhamento das partições é feito utilizando o algoritmo Needleman-Wunsch. A Figura 3.1(e) mostra um exemplo de alinhamento ótimo resultante do Estágio 5.

Estágio 6

O Estágio 6 é opcional caso se deseje visualizar o arquivo binário, criado no Estágio 5, que representa o alinhamento. A Figura 3.1(f) apresenta uma representação textual obtida a partir de um procedimento que pode ser feito no arquivo.

3.1.3 CUDAlign 2.1

O CUDAlign 2.1 propõe a otimização *Block Pruning* (BP), o que acelera em mais de 50% o cálculo da matriz de programação dinâmica para sequências similares em uma GPU [24]. Esta otimização permite o descarte de blocos de células da matriz que não irão contribuir para a obtenção do alinhamento ótimo, pois eles possuem um escore tão baixo que é comprovado matematicamente que um alinhamento passando por eles não será melhor que um já encontrado até o momento. O *Block Pruning* é aplicado no Estágio 1, onde o escore máximo não é conhecido, acelerando o maior estágio do processamento. Foram atingidos 50,7 GCUPS em uma GPU.

3.1.4 CUDAlign 3.0

A versão CUDAlign 3.0 tem a capacidade de utilizar *clusters* com múltiplas GPUs (homogêneas e heterogêneas) a fim de possibilitar a obtenção do escore ótimo entre sequências longas de DNA maiores que 100 milhões de pares de base (MBP) em um tempo hábil [21].

A arquitetura Multi-GPU foi proposta com o poder de distribuir uma única comparação em várias GPUs simultaneamente, tendo como intuito o aceleramento do Estágio 1, a etapa mais demorada do CUDAlign. Foram atingidos 1,73 TCUPS (trilhões de CUPS) em 64 GPUS.

3.1.5 CUDAlign 4.0

Com a versão CUDAlign 4.0, é possível a recuperação de alinhamentos ótimos longos com múltiplas GPUs. A estratégia *Incremental Speculative Traceback* (IST) foi proposta para paralelizar eficientemente os estágios 2 a 5 do CUDAlign, que é a etapa de recuperação do alinhamento ótimo. Ela especula valores de *crosspoint*, enquanto a GPU está ociosa, para recuperar partes do alinhamento baseando-se nestes valores. Foram atingidos 10,37 TCUPS em 384 GPUS.

3.1.6 Arquitetura MASA

A arquitetura MASA foi proposta para que o CUDAlign pudesse ter sua portabilidade simplificada para diversas outras arquiteturas, tais como CPU, Intel Phi, GPUs de diversos fabricantes e outros [25]. O MASA suporta alinhamentos ótimos locais, globais e semi-globais. A arquitetura MASA é descrita mais detalhadamente na Seção 3.2.

3.2 Multi-platform Architecture for Sequence Aligners (MASA)

A arquitetura do CUDAlign teve o seu código independente (portável) desacoplado do seu código específico (não portável), a fim de que pudesse ser aplicado, não apenas na arquitetura CUDA da NVIDIA, mas também em outras plataformas de *hardware*, tais como GPUs de outros fabricantes, CPUs, FPGAs, e plataformas de *software*, tais como, OpenCL , OpenMP e OmpSs. Com isso, foi gerada uma nova arquitetura de software chamada MASA.

A arquitetura MASA é flexível e customizável, sendo formada por módulos e otimizações que facilitam o desenvolvimento de ferramentas similares ao CUDAlign, obtendo o alinhamento de sequências de qualquer tamanho com métodos exatos em diversas plataformas. Para que o CUDAlign possa migrar para diferentes ambientes, é necessário reescrever o código específico, que ocupa cerca de 10% do código fonte, enquanto que o código independente, que ocupa cerca de 90% do código fonte, pode ser reutilizado. O resultado de cada nova customização dessa união entre o código independente e o espe-

cífico em uma biblioteca estática é chamado de "extensão". A arquitetura MASA e seus componentes (Seção 3.2.1) e o procedimento usado para implementar quatro extensões (Seção 3.2.2) são apresentados nesta seção.

3.2.1 Visão geral

A arquitetura MASA foi dividida em 5 módulos: Gerência de dados, Comunicação, Estatísticas, Gerenciamento dos Estágios, que são reutilizados a cada nova migração para outra plataforma, e o *Aligner*, que, por sua vez, é dividido em outros 3 módulos. Dois destes módulos, *Tipo de Processamento* e *Block Pruning*, são customizáveis dependendo da plataforma a ser utilizada. O outro módulo, que executa o cálculo da matriz de programação dinâmica, realiza a parte onde se concentra o maior tempo de execução do CUDAlign, e, por isso, tem maior probabilidade de sofrer otimizações específicas da plataforma. A divisão da arquitetura MASA nestes módulos pode ser vista na Figura 3.2. Cada um dos módulos são descritos a seguir.

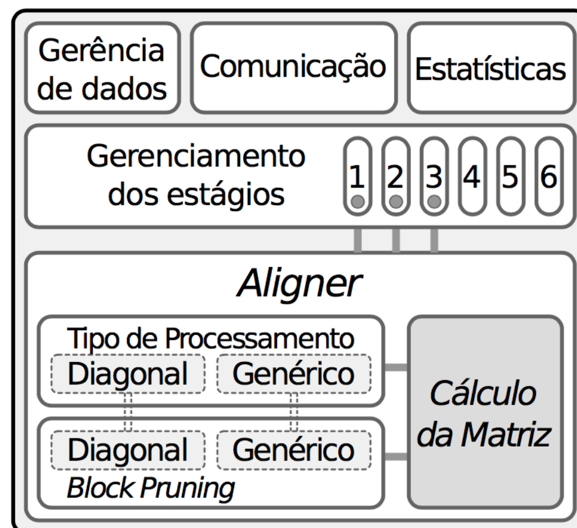


Figura 3.2: Arquitetura MASA. Fonte: [22].

Aligner: Este módulo comporta o código específico (não portável) de cada plataforma, que é responsável por executar a equação de recorrência de NW (Seção 2.2) ou SW (Seção 2.3). Apesar de ser necessário otimizar este código para cada extensão, a arquitetura MASA permite que alguns recursos não dependentes da plataforma possam ser reutilizados pelo *Aligner*, tais como as técnicas de *Block Pruning* e uma versão básica portável do algoritmo NW/SW, que depende do ambiente para poder ser reutilizada.

Gerência de dados: Este módulo é responsável pelo gerenciamento de dados de entrada e saída do MASA. Ele executa tarefas como a leitura das sequências de entrada,

processamento de parâmetros que são passados por linha de comando, armazenamento de linhas e colunas especiais, geração de arquivos binários e textuais, restauração de *checkpoints*, armazenamento de *crosspoints*, entre outras.

Estatísticas: Este módulo fornece informações como o tempo que um estágio levou para ser executado, a quantidade usada de memória e disco, entre outras.

Gerenciamento dos Estágios: Este módulo coordena a execução dos estágios primeiramente propostos no CUDAling 2.0 (Seção 3.1.2) e aprimorados nas versões 2.1 (Seção 3.1.3), 3.0 (Seção 3.1.4) e 4.0 (Seção 3.1.5). Ele utiliza o *Aligner* para coordenar os estágios 1 a 3 e executa os estágio 4 a 6 em CPU. Nos estágios 1 a 3, a matriz de programação dinâmica é particionada, onde cada partição, contendo a sua primeira linha e coluna, é enviada para o *Aligner*. Em seguida, o *Aligner* fornece como saída a última linha e coluna, o escore máximo da partição e as linhas e colunas especiais. No estágio 1, a partição enviada possui o tamanho total da matriz de programação dinâmica, mas pode sofrer subparticionamento caso possua um tamanho maior que o máximo suportado pelo *Aligner*. No estágios 2 e 3, as partições serão menores, tendo o tamanho delimitado de acordo com as linhas especiais salvas nos estágios anteriores. Quando a última linha e coluna da partição são recebidas, o *Aligner* procura pelo *crosspoint* utilizando o procedimento de *matching* por objetivo, otimização do Estágio 2 (Seção 3.1.2). Assim que o *crosspoint* é encontrado, encerra-se o cálculo desta partição e uma nova partição é enviada para o *Aligner*, repentindo-se este procedimento até que se encontre as extremidades do alinhamento em todos estes estágios. Na Figura 3.3 pode ser visto como é realizado este processamento em partições. Os estágios 4 a 6 não utilizam o código específico do *Aligner*, podendo ser executados em CPU através do código que é reutilizável em todas as plataformas.

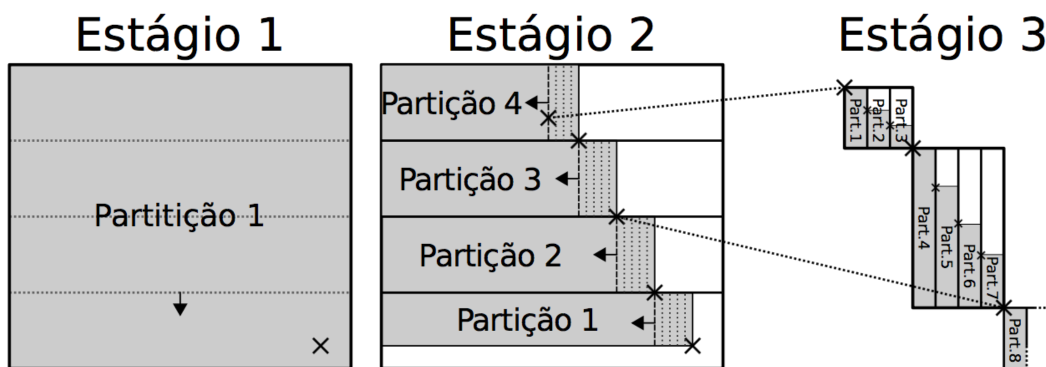


Figura 3.3: Processamento em partições coordenado pelo módulo Gerenciamento dos Estágios. Fonte: [22].

Comunicação: Este módulo permite a execução de múltiplas GPUs, ou múltiplos dispositivos, em um *wavefront* distribuído.

Block Pruning (BP): Como o Block Pruning (BP) é a otimização mais importante do CUDAling, ele foi implementado para alinhamentos locais de forma independente da plataforma. As otimizações desenvolvidas foram o *Diagonal BP*, para paralelizações feita por diagonais, e o *Generic BP*, para paralelizações genéricas.

Tipo de processamento: Como foi observado que o cálculo da célula $H_{i,j}$, segundo as equações de recorrência de SW, NW e Gotoh (Seção 2.4), depende das células $H_{i-1,j}$, $H_{i,j-1}$ e $H_{i-1,j-1}$, determinou-se que a arquitetura MASA forneceria duas formas portáveis de se calcular a matriz de programação: por diagonal, que está associada ao *Diagonal BP*, ou de maneria genérica, que pode, a depender do código específico da plataforma, processar os dados em qualquer ordem, mas sempre respeitando a dependência de dados.

3.2.2 Como integrar novas implementações

Foram implementadas quatro extensões utilizando diferentes ferramentas e modelos de programação (OpenMP [6], OmpSs [3, 8] e CUDA [18]), para diferentes plataformas de *hardware* (multicore, GPUs e Intel Phi). As extensões resultantes são denotadas por MASA-OpenMP/CPU, MASA-OpenMP/Phi, MASA-OmpSs/CPU e MASA-CUDAlign, e podem ser vistas na Figura 3.4 com os componentes MASA que foram utilizados em cada uma. A criação destas extensões foi possibilitada através de diferentes modificações aplicadas no Algoritmo 1, enquanto que o modelo *affine gap* (Seção 2.4) é utilizado por todas.

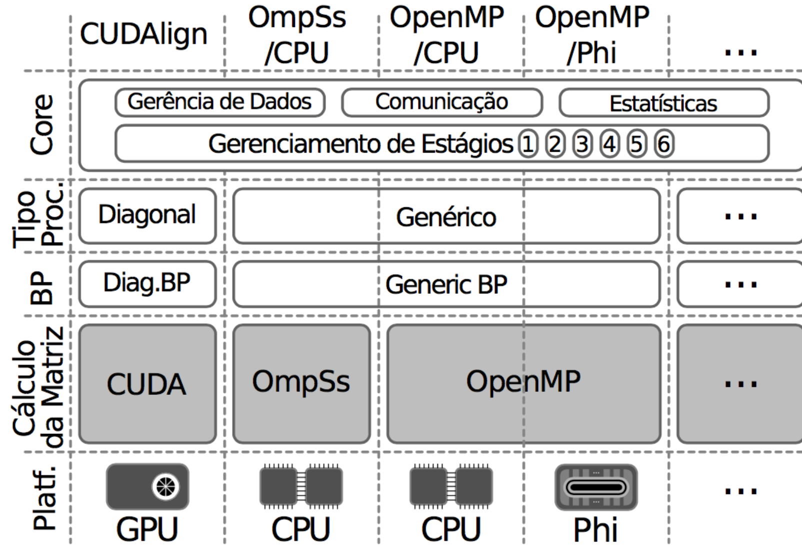


Figura 3.4: Extensões MASA com os componentes MASA utilizados em cada uma. Fonte: [22].

Algorithm 1 Pseudocódigo do *Aligner* baseado em bloco. Fonte: [22]

```
 1: procedure Aligner::scheduleBlocks
 2:     for each diagonal do
 3:         for each block in diagonal do
 4:             alignBlock(block)
 5:         end for
 6:     end for
 7: end procedure
 8:
 9: procedure Aligner::alignBlock(block)
10:     if block está na 1ª linha then block.row ← receiveFirstRow
11:     if block está na 1ª coluna then block.col ← receiveFirstColumn
12:     if  Not isBlockPruned(block) then
13:         block.score := processBlock(block)
14:         dispatchScore(block.score)
15:     end if
16:     if isSpecialRow(block.row) then dispatchRow(block.row)
17:     if block está na última coluna then dispatchColumn(block.col)
18: end procedure
19:
20: procedure main(args)
21:     processor = new CPUBlockProcessor()
22:     MASA::entryPoint(args, new Aligner(processor))
23: end procedure
```

Capítulo 4

Plataforma XD2000i^{TM}

Devido à sua natureza altamente paralela, o algoritmo Smith-Waterman (Seção 2.3) pode ser portado de maneira eficiente para FPGAs (Field Programmable Gate Arrays), obtendo desempenho geralmente superiores às GPUs. No presente trabalho de graduação, escolheu-se a plataforma *XtremeData* XD2000i^{TM} [12] para implementar em FPGA o projeto aqui apresentado. As Seções 4.1 a 4.6 descrevem detalhadamente o funcionamento desta plataforma e as configurações necessárias para que ela rode adequadamente.

4.1 Componentes da plataforma XD2000i

A plataforma XD2000i consiste de *hardware* e *software* fornecidos pela *XtremeData* e pelo usuário, cujas partes podem ser vistas na Figura 4.1.

O PC de desenvolvimento é o *hardware* da plataforma fornecido pela *XtremeData*, contendo os seguintes componentes [12]:

- Placa mãe *Dual XEON* com um processador *Intel XEON*.
- Módulo XD2000i 1067M FSB, chamado *Accelerator Hardware Module* (AHM), contendo uma FPGA *Bridge* conectada à *Northbridge* (MCH) e duas FPGAs de aplicação Altera Stratix III EP3SE260F1152C3 (*AppA* e *AppB*) que contêm 255.000 elementos lógicos e 203.000 registradores.
- Sistema Operacional Linux CentOS.

O projeto do *design* de referência, também fornecido pela *XtremeData*, é composto de *software* e arquivos de *hardware*, que são distribuídos como arquivos zip compactados. O *software* do *design* de referência contém códigos fonte e executáveis para diversos programas de exemplo. Já os arquivos de *hardware* contêm códigos fonte em HDL para o

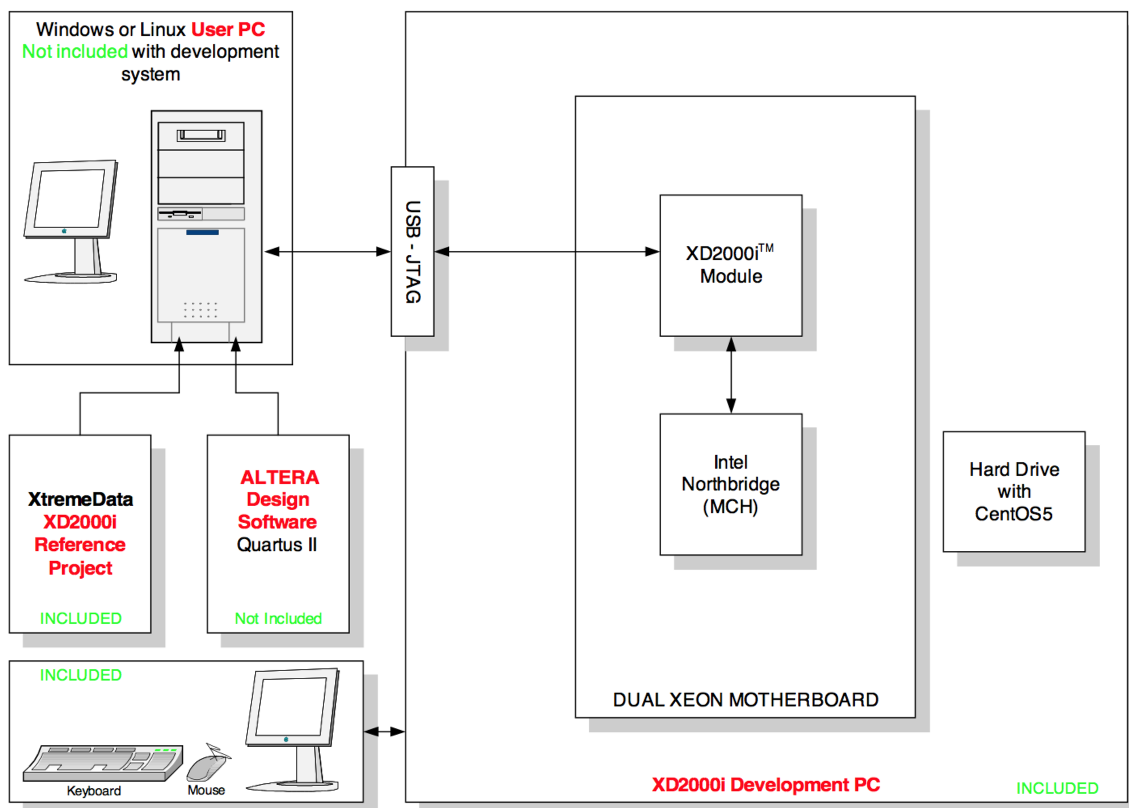


Figura 4.1: Componentes da plataforma XD2000i. Fonte: [12].

design de referência da FPGA de aplicação e arquivos de configuração para os diversos dispositivos no módulo XD2000i.

A *XtremeData* disponibiliza juntamente com os outros componentes um teclado, *mouse* e monitor.

O item fornecido pelo usuário é um PC rodando o *software* Quartus II, que pode conectar-se à XD2000i via cabo JTAG.

4.2 Projeto do *design* de referência

O *design* de referência consiste no *design* da FPGA da XD2000i, nas aplicações de teste de *software* e nos *drivers* de dispositivo. O design de referência implementa a *Accelerator Functional Unit* (AFU), que é a porção da plataforma XD2000i definida pelo usuário residindo em uma ou ambas FPGAs de aplicação. Uma visão de alto nível do design de referência da FPGA de aplicação pode ser vista na Figura 4.2.

Todo dado é sequencialmente transferido do sistema de memória para a AFU através da FPGA *Bridge*, que é acessada por meio de uma simples interface de software. O fluxo

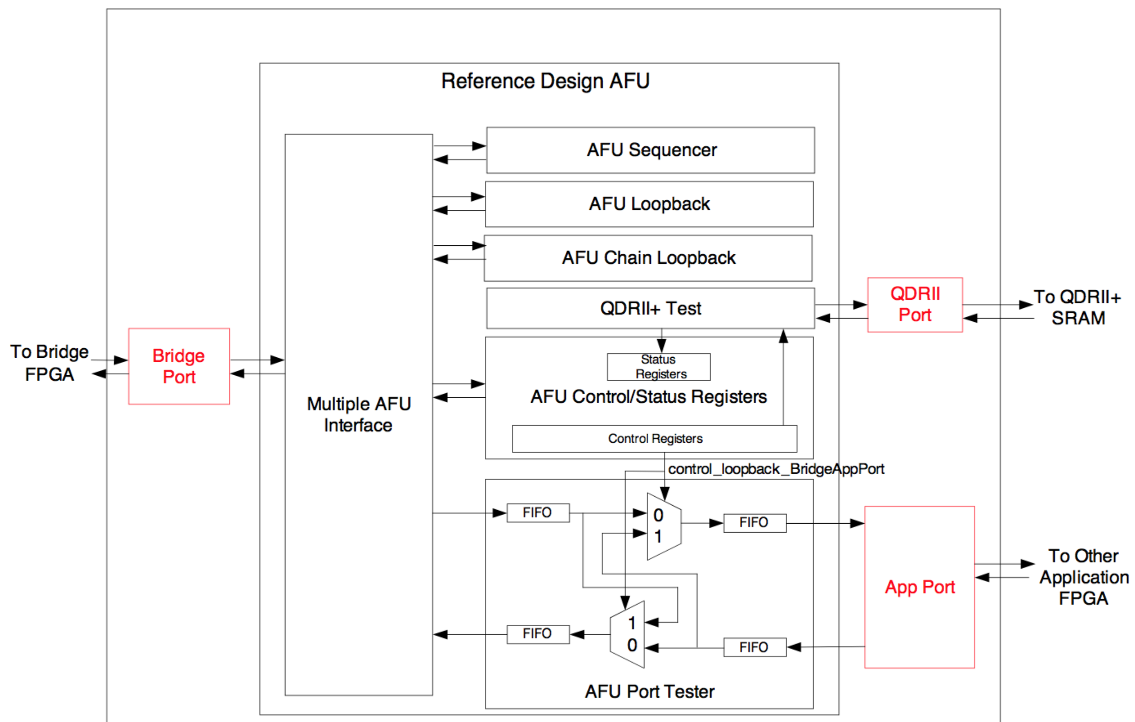


Figura 4.2: Design de referência da FPGA de aplicação. Fonte: [12].

de dados do sistema como um todo pode ser visto na Figura 4.3.

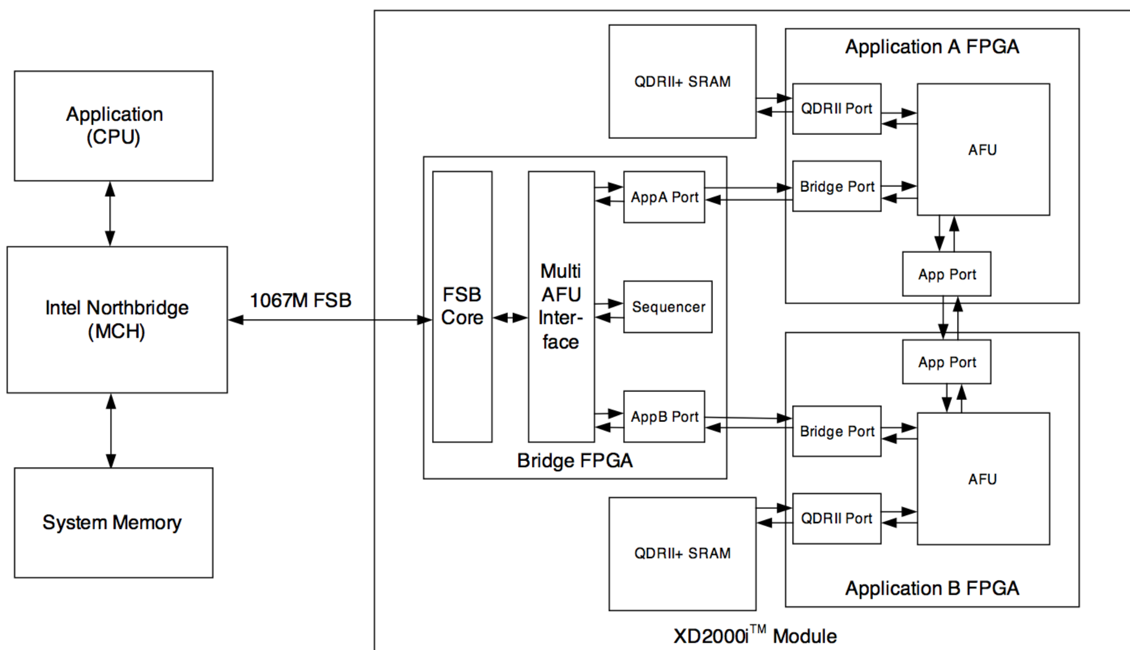


Figura 4.3: Fluxo de dados do sistema. Fonte: [12].

Os dados são transferidos do sistema de memória para o *Memory Controller Hub* (MCH), prosseguindo para a FPGA *Bridge* através de uma interface chamada *Front Side Bus* (FSB). Em seguida, eles são transferidos por uma interface para a FPGA de aplicação de destino. Cada FPGA implementa um conjunto de AFUs que estão conectadas a um multiplexador, chamado *Multiple AFU Interface* (MAFU).

4.3 Aplicações do módulo XD2000i

No módulo XD2000i, é implantada uma interface de transferência de dados da seguinte maneira: primeiramente, um *buffer* de dados de tamanho conhecido é preparado pela CPU no sistema de memória, que, através de um comando *Send/Receive*, o envia para o módulo. Em seguida, o sistema de memória recebe do módulo um novo *buffer* de dados, cujo tamanho é também conhecido, mas podendo ser diferente do tamanho do *buffer* enviado. Com a transação completa, a aplicação de usuário é avisada e ela então sabe que existem dados válidos disponíveis. As memórias de origem e de destino podem ser de um mesmo ou de diferentes *Workspaces*, que são porções do sistema de memória usadas para as ações de transferência do módulo.

O módulo utiliza endereços físicos gerenciados pelo FSB para endereçar os dados do *Workspace*, que estão contiguamente estabelecidos na memória física. Eles são visualizados apenas como uma *stream* de dados pela a aplicação de usuário, enquanto que o módulo aparece como um dispositivo no diretório /dev/fap. Dessa forma, a manipulação de um *Workspace* é feita por um *driver* de dispositivo no espaço de usuário.

Para alocar um *Workspace* de tamanho previamente definido, o usuário primeiramente “abre” a FPGA de dentro do módulo para obter o descritor de arquivo. Com o endereço do *Workspace* retornado, o usuário preenche o *buffer* de dados de origem e submete um comando de *Workspace*, especificando o endereço e tamanho da memória de origem e de destino. Esta chamada fica bloqueada até que os dados de resposta sejam recebidos. Quando o *Workspace* não é mais necessário, ele é liberado e o módulo é “fechado” ao término da transação.

As aplicações do módulo possuem as seguintes restrições:

- O tamanho dos dados do comando de *Workspace* não pode ser nulo e deve ser um múltiplo positivo de 64 Bytes.
- Os *buffers* de origem e de destino devem ser especificados e ter pelo menos 64 Bytes.
- Os *buffers* de origem e de destino podem ter tamanhos iguais ou diferentes entre si.
- O tamanho máximo de um *Workspace* é de 4MB.

- O número máximo de *Workspaces* alocados é 4096.
- Apenas uma operação de envio pode estar ativa por vez. A porção de recebimento do comando *Workspace* deve estar completa antes do próximo envio ser submetido. Caso seja desejado, a requisição de envio pode retornar imediatamente, possibilitando trabalho adicional da CPU ser executado antes de submeter a requisição de recebimento.
- Os comandos de *Workspace* do *design* de referência são processados sincronizadamente, ou seja, um envio é imediatamente seguido por um recebimento.

4.4 FPGA *Bridge*

A FPGA *Bridge* realiza a interface entre o FSB e o módulo XD2000i e não é configurável pelo usuário. Como dito na Seção 4.1, a *XtremeData* fornece arquivos de *hardware* que contêm a *stream* de dados para a configuração da *Bridge*. Este arquivo de configuração .pof é armazenado em uma memória *flash* e é carregado automaticamente dentro da FPGA *Bridge* quando o sistema é inicializado.

Como pode ser observado na Figura 4.3, a FPGA *Bridge* contém o FSB, a MAFU, as portas de interface para as FPGAs de aplicação e o sequenciador de teste. O FSB se conecta ao MCH (*Northbridge*), transmitindo os dados para a MAFU. Baseando-se no cabeçalho embutido na *stream* de dados, a MAFU direciona os dados para uma das AFUs da *Bridge*, podendo ser a FPGA *AppA*, a FPGA *AppB* ou o sequenciador de teste.

4.5 Protocolo FIFO da AFU

A AFU se conecta à interface da Bridge usando um protocolo *Data/Valid/Ready* (DVR), que é um *handshake* ponto-a-ponto do tipo FIFO. No *design* de referência, o arquivo fifo_afudrv.vhd é instanciado diversas vezes para para implementar FIFOs com interfaces *write-side* e *read-side*, podendo ser do tipo AFU ou DRV. Desse modo, quatro diferentes tipos de FIFOs podem ser criadas: uma que pode ter na escrita uma interface do tipo AFU e na leitura uma interface do tipo DRV ou vice-versa; ou a FIFO pode ter ou uma interface do tipo AFU ou uma do tipo DRV em ambas portas de escrita e leitura. Como estas portas são assíncronas, elas podem ter diferentes larguras de dados, mas o tamanho deve ser de no máximo 256 bits. A seguir, será descrita a diferença entre estes dois tipos de interface.

4.5.1 Interface FIFO tipo AFU

A interface do tipo AFU é empregada pelas interfaces da AFU do usuário. Esta interface requer que o *target* aceite dados sempre que o <initiator>_data_valid é setado. O controle de fluxo é efetuado ao requirir que o *initiator* transmita dados apenas quando ele sente que o <target>_data_ready é setado. A seguir, são apresentadas as regras do Protocolo FIFO do tipo AFU:

1. *Initiator* pode setar <initiator>_data_valid sempre que o dado é válido e quando ele sente que <target>_data_ready é setado.

2. *Initiator* deve ressetar <initiator>_data_valid sempre que ele sente que <target>_data_ready é ressetado.

3. *Target* aceita dados sempre que ele sente que <initiator>_data_valid é setado.

4. *Target* deve aceitar um mínimo de 16 transferências após ressetar <target>_data_ready para permitir que transferências ainda em processamento sejam completadas.

4.5.2 Interface FIFO tipo DRV

A interface do tipo DRV permite a transferência de dados quando <initiator>_data_valid e <target>_data_ready estão setados concorrentemente. Pode ser conveniente para a porta de entrada da AFU converter sua interface do tipo AFU para uma do tipo DRV pelo motivo de que esta interface permite a supressão de dados em chegada sem precisar providenciar armazenamento de buffer adicional. A regras do protocolo FIFO do tipo DRV são apresentadas a seguir:

1. *Initiator* seta <initiator>_data_valid sempre que o dado é válido.

    - <initiator>_data_valid pode ser setado enquanto <target>_data_ready está ressetado.

2. *Target* seta <target>_data_ready sempre que estiver pronto para aceitar dados.

    - <target>_data_ready pode setado enquanto <initiator>_data_valid está ressetado.

3. A transferência de dados ocorre quando <initiator>_data_valid e <target>_data_ready estão setados concorrentemente.

4.6 Configuração da plataforma XD2000i

Nesta etapa, o PC de desenvolvimento já foi configurado, o código de software já está pronto para ser rodado e, desse modo, é necessário então estabelecer a comunicação com a FPGA *Bridge*. Em seguida, é possível finalmente realizar a programação da FPGA de aplicação. As seções a seguir descrevem o passo-a-passo de como esta etapa é efetuada.

4.6.1 Estabelecimento da comunicação com a FPGA *Bridge*

Para abrir uma comunicação direta com a *Bridge*, os seguintes procedimentos de carregar os *drivers* e verificar o sistema toda vez que o módulo XD2000i é inicializado são aplicados.

Habilitação do segmento AHM FSB

O módulo XD2000i ocupa o *socket* secundário da CPU e em alguns sistemas o segmento FSB secundário é desabilitado caso a CPU não seja detectada no *socket* 1 durante a inicialização do sistema. Apenas em sistemas com a placa mãe SuperMicro Rack Mount DP X7DGT-SG007 que o *script* tcl apresentado a seguir deve ser executado para habilitar a interface AHM *socket* 1 FSB. Este *script* deve ser sempre rodado após a inicialização do sistema.

```
$ cd <target_dir>/software_<ver>/bin
$ sudo chmod 777 fsb_bus.tcl
$ ./fsb_bus.tcl 1 on
```

Carregamento do *driver* de dispositivo da XD2000i

O *driver* de dispositivo da AFU XD2000i possui dois componentes: fappip_afu_07.ko, que implementa a interface de *hardware* de baixo nível; e o fapdrv_afu_07.ko, que implementa a interface de aplicação. É com o carregamento deste *driver* que o AHM, ou módulo XD2000i, pode ser acessado por todos os usuários através do arquivo /dev/fap. O procedimento do carregamento mostrado a seguir deve ser realizado toda vez que o sistema for inicializado.

```
$ cd <target_dir>/software_<ver>/bin
$ sudo /sbin/insmod fappip_afu_07.ko
$ sudo /sbin/insmod fapdrv_afu_07.ko
$ sudo chmod 666 /dev/fap
```

Pode ser verificado se o *driver* de dispositivo foi devidamente carregado através do comando de leitura do log do sistema:

```
$ dmesg | grep fap
```

4.6.2 Programação da FPGA de aplicação

O programa executável ahmAppDownload configura a FPGA de aplicação desejada com o arquivo de configuração .rbf (*Raw Binary File*), que pode ser obtido a partir da síntese lógica do *design* de referência no *software* Quartus II. O comando a seguir configura a FPGA *AppA* com o arquivo appA_test.rbf:

```
$ ./ahmAppDownload -A -a appA_test.rbf
```

Caso a programação da FPGA obtenha sucesso, os seguintes resultados serão impressos no *prompt* de comando:

```
downloadAppAStatusIsOk = true
All runningStatusIsOk == true
```

Capítulo 5

O Projeto do Circuito de Particionamento

Este capítulo apresenta o projeto da solução encontrada para a comparação de sequências biológicas baseada no algoritmo de Smith-Waterman (Seção 2.3) utilizando particionamento na plataforma XD2000i (Capítulo 4). Ele foi baseado em um projeto já existente implementado por Colletti [5], obtendo duas versões: (1) o circuito original recebe adicionalmente a coluna, a linha e a diagonal iniciais da matriz de programação dinâmica; (2) o circuito da versão 1 tem como saída, além do escore, a coluna, a linha e a diagonal finais da matriz de programação dinâmica.

5.1 Solução de Colletti (2016)

O circuito implementado por Colletti [5] é baseado na solução proposta por Carvalho [4], que mapeou o algoritmo de SW (Seção 2.3) em uma estrutura sistólica linear unidirecional, aplicando paralelismo proporcionado pelo método *diagonal wavefront* DW [23]. A seguir, serão descritas detalhadamente estas técnicas utilizadas.

5.1.1 Método *diagonal wavefront* DW

O método *diagonal wavefront* DW [23] define que as células de cada anti-diagonal da matriz de programação dinâmica podem ser calculadas de forma paralela devido à computação da célula (i,j) que, segundo a relação de recorrência de SW (Seção 2.3), pode ser feita se as células (i, j-1), (i-1, j-1) e (i-1, j) tiverem sido computadas. Uma vez que essa dependência de dados existe, é possível calcular anti-diagonal por anti-diagonal. A Figura 5.1 mostra a paralelização do cálculo de cada célula por este método.

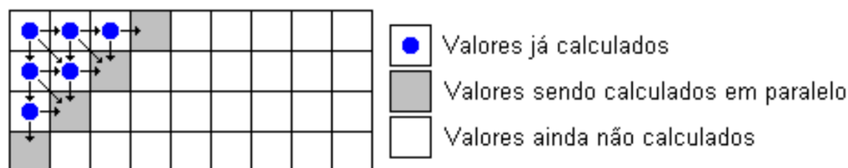


Figura 5.1: O método *diagonal wavefront*. Fonte: [4].

5.1.2 Estrutura sistólica

Para obter o paralelismo fornecido pelo método DW (Seção 5.1.1), o algoritmo SW (Seção 2.3) foi mapeado em um estrutura sistólica linear unidirecional com N elementos de processamento (PE), que correspondem a inúmeras instâncias de uma unidade menor responsáveis por todo processamento da estrutura. A quantidade de PEs vai depender apenas da sequência de consulta, sendo que a sequência do banco de dados pode ter qualquer tamanho, respeitada a capacidade de memória do sistema. Cada PE é responsável pelo cálculo das células de uma coluna da matriz de programação dinâmica. Esta estrutura sistólica possibilita a redução da complexidade do algoritmo SW de $O(n \times m)$ para $O(n + m)$, ou seja, o tempo que era quadrático é reduzido para linear.

A implementação do vetor sistólico é feita de forma que cada uma das bases da sequência de consulta é armazenada em um PE ao qual correspondem. A cada ciclo de *clock*, a sequência de banco de dados é deslocada uma posição para dentro do vetor, permitindo que as bases passem por todos os PEs. Como o deslocamento é realizado da esquerda para a direita, a sequência de banco de dados deve entrar na estrutura sistólica de forma invertida, a fim de que a sua primeira base seja comparada com a primeira base da sequência de consulta. Desse modo, uma anti-diagonal da matriz de programação dinâmica é calculada a cada ciclo de *clock*. A Figura 5.2 mostra um exemplo de como a estrutura sistólica se comporta.

5.1.3 Elemento de processamento

O elemento de processamento (PE) projetado por Carvalho aplica a equação de recorrência de SW (Equação 2.2), que determina que cada célula da matriz de programação dinâmica depende apenas da célula da linha e coluna anteriores (célula da diagonal), da célula da mesma linha e coluna anterior (célula à esquerda) e da célula da linha anterior e mesma coluna (célula da linha superior). Foi assumido, portanto, que estes valores devem estar presentes em cada PE. Como o valor da célula da coluna à esquerda calculado no ciclo *clock* anterior viria do PE adjacente, é necessário armazenar apenas os valores

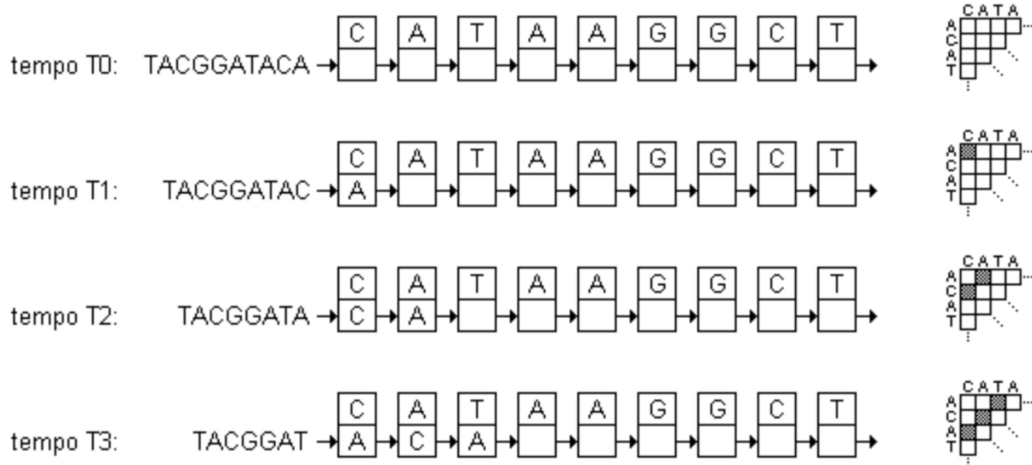


Figura 5.2: A estrutura sistólica linear unidirecional. Fonte: [4].

das células da diagonal e da linha superior. A estrutura inicial do PE é mostrada na Figura 5.3.

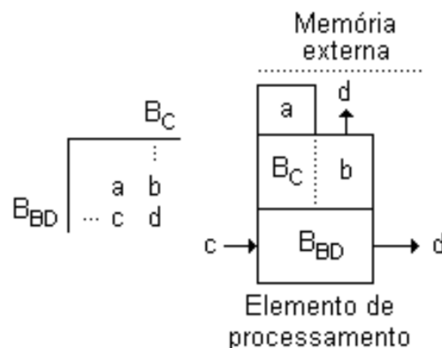


Figura 5.3: Estrutura interna inicial do elemento de processamento. Fonte: [4].

Em cada PE do vetor sistólico é encontrada uma base da sequência de consulta (B_C) armazenada. Após um ciclo de *clock*, uma nova base da sequência de banco de dados (B_{BD}) é deslocada para dentro do PE, proveniente do PE adjacente. Complementarmente, o valor calculado no tempo de *clock* anterior dentro deste PE à esquerda também será fornecido ao PE, passando a ser o valor na coluna à esquerda (c). Em paralelo a isso, na transição para este *clock* em que são recebidos estes dados, são armazenados o valor calculado internamente no PE e o valor na sua célula à esquerda, obtidos no ciclo de *clock* anterior, passando a serem os valores na linha superior (b) e na diagonal (a), respectivamente. Dessa forma, o PE possui o necessário para que a equação de recorrência

de SW seja calculada, onde o novo valor resultante da célula da matriz de programação dinâmica será passado para o PE adjacente à direita para que mais uma célula possa ser computada.

5.1.4 Cálculo da equação de recorrência

Para o cálculo deste novo valor da célula da matriz de programação dinâmica, foi desenvolvida uma lógica combinacional para implementar a equação de recorrência. Pela equação de recorrência, notou-se que valores fixos (1, -1 e -2) são somados aos valores armazenados no PE e que essas operações, $H_{i-1,j-1}+sbt(S_0[i], S_1[j])$, $H_{i-1,j}-G$ e $H_{i,j-1}-G$, devem ser realizadas em paralelo.

Dessa forma, o valor na diagonal é somado com 1 e -1, valores de *match* e *mismatch*, respectivamente. Os resultados dessas duas somas serão as entradas de um multiplexador, que terá como seletor o resultado da comparação entre uma base da sequência de consulta e uma da sequência de banco de dados. O valor selecionado será armazenado temporariamente em $RES1$. A Figura 5.4 mostra como essa operação é implementada.

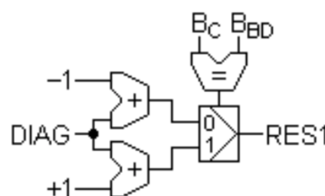


Figura 5.4: Circuito combinacional para o cálculo do valor na diagonal com as pontuações de *match* e *mismatch*. Fonte: [4].

Paralelamente, os valores na linha superior e na coluna à esquerda são somados com -2, que é a pontuação de *gap*. Similarmente, os resultados das somas serão as entradas de um multiplexador, cujo seletor será a comparação entre estes valores que estão na linha superior e na coluna à esquerda. O resultado encontrado será armazenado temporariamente em $RES2$. A Figura 5.5 mostra a implementação deste cálculo.

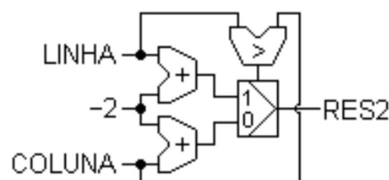


Figura 5.5: Circuito combinacional para o cálculo dos valores na linha superior e coluna à esquerda com a pontuação de *gap*. Fonte: [4].

Por fim, após a obtenção dos valores em $RES1$ e $RES2$, o maior dentre eles será a saída calculada naquela célula utilizando a equação de recorrência.

A Figura 5.6 representa o circuito implementado por Carvalho com cinco PEs, que tem como entrada os sinas *clk*, *rst*, *flag_in* e *base_in*. O sinal *flag_in* indica que o sinal *base_in*, que representa uma base da sequência de banco de dados, é válido naquele instante para um determinado PE. A Figura 5.7 mostra a encapsulação deste circuito.

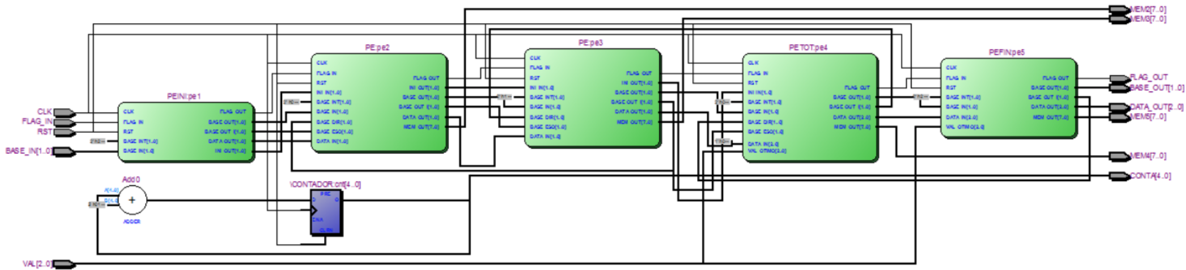


Figura 5.6: Circuito implementado por Carvalho com cinco elementos de processamento. Fonte: [5].

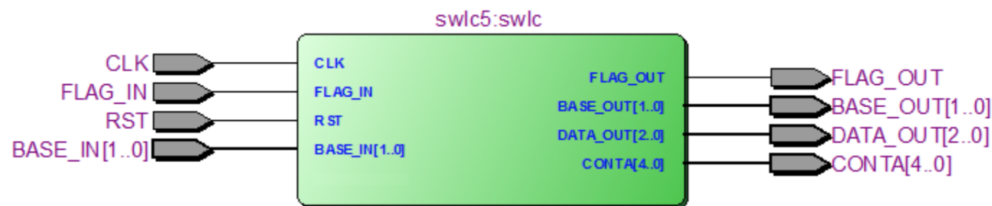


Figura 5.7: Circuito da Figura 5.6 encapsulado. Fonte: [5].

5.1.5 Cálculo do escore máximo local

Como extensão ao circuito proposto por Carvalho, Colletti adicionou um circuito em cada PE para que fosse realizado o cálculo do escore máximo local. O PE calcula o seu novo escore e compara este valor com o escore calculado no instante de tempo anterior. O maior valor resultante será comparado com o escore calculado até aquele instante pelo PE à esquerda, que é recebido a cada ciclo de *clock*. No ciclo de *clock* seguinte, o novo escore máximo encontrado é passado para o PE vizinho à direita, para que ele também possa computar seu novo escore. A estrutura sequencial para este cálculo do escore máximo pode ser vista na Figura 5.8.

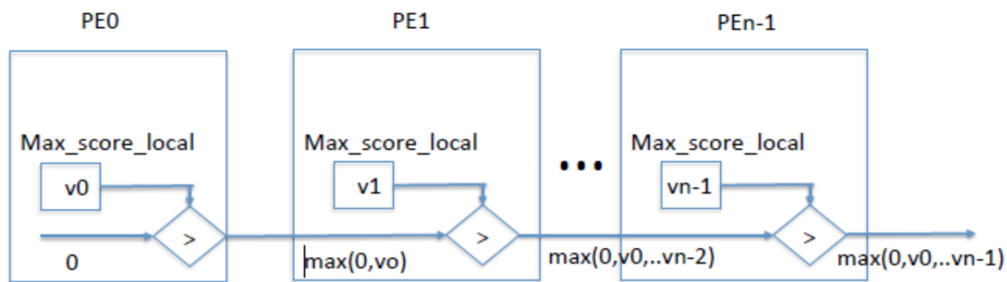


Figura 5.8: Estrutura sequencial para o cálculo do escore máximo local. Fonte: [5].

5.1.6 Unidade de controle

Uma outra extensão realizada por Colletti foi a implementação de uma unidade de controle. A unidade de controle foi necessária para que a passagem das bases da sequência de banco de dados *base_in* para os PEs fosse possível e para que os sinais de controle fossem setados corretamente. Desse modo, uma máquina de estados finita foi desenvolvida, cujos estados são *Read_data*, *Execute* e *Write_data*. O circuito permanece no estado inicial *Read_data* até quando o sinal *data_in_valid* seja setado, indicando que o dado presente na entrada *data_in* é válido e que pode ser armazenado no vetor *data_in_r*, cujo conteúdo será a sequência de banco de dados. Uma vez que isso ocorre, a máquina de estados passa para o estado *Execute*, que é onde ocorre a passagem das bases para o primeiro PE. A cada ciclo de *clock*, os dois *bits* menos significativos do vetor *data_in_r*, representando uma base, são passados para o PE. O vetor sofre a cada *clock* um *shift right* lógico para que a próxima base esteja nos dois *bits* menos significativos. O circuito permanece neste estado até que as duas sequências sejam completamente comparadas na estrutura sistólica, gerando o escore máximo local. Este momento é determinado quando o sinal *flag_out* retorna ao nível lógico baixo, transicionando a máquina para o estado *Write_data*. Neste estado, o escore máximo é armazenado na saída do circuito *data_out* (Figura 5.11(b)) e o sinal *data_out_valid* é setado para indicar que o dado de saída é válido. Por fim, a máquina retorna ao estado *Read_data* para esperar a chegada de novos dados de entrada. A Figura 5.9 descreve esta máquina de estados.

5.2 Circuito de Particionamento proposto v.1

A solução proposta no presente trabalho de graduação tem como intuito a comparação de sequências biológicas dividida em etapas, possibilitando um tamanho ilimitado para as sequências. O circuito original desenvolvido por Colletti (Seção 5.1) exige que a comparação seja feita com as sequências completas, cujo tamanho máximo deve ser de

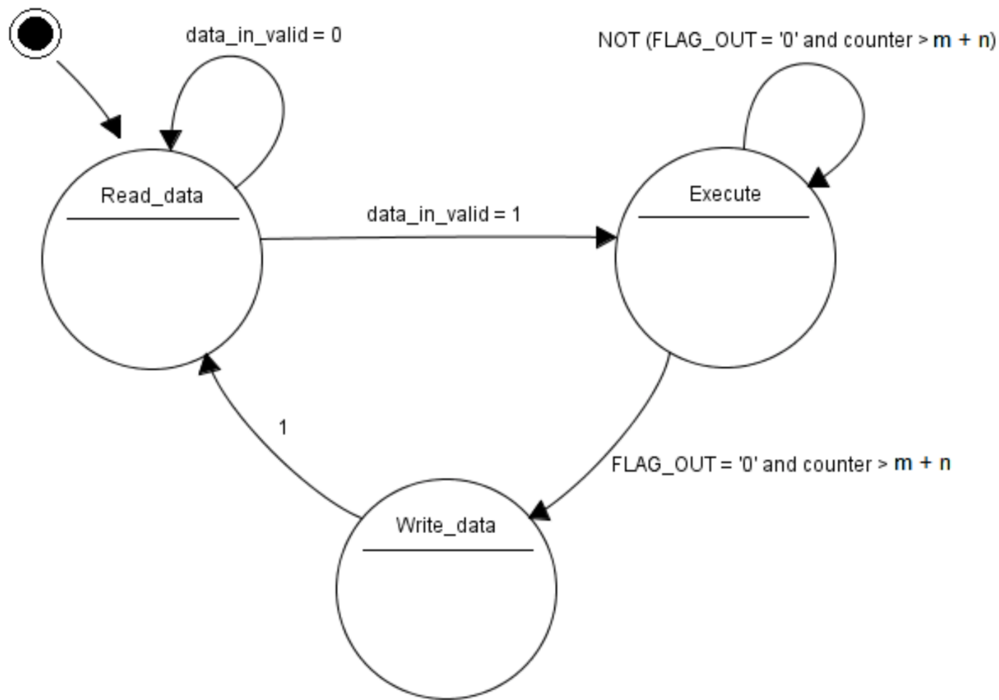


Figura 5.9: Máquina de estados implementada na unidade de controle. Fonte: [5].

20 bases. Em contrapartida, o novo circuito implementado permite a comparação entre partições de sequências de qualquer tamanho. A única restrição é que as partições devem ter um tamanho máximo de 20 bases. Com isso, o presente trabalho irá considerar daqui em diante que a matriz de programação dinâmica, toda vez que for citada, será aquela gerada pela comparação entre as partições das sequências biológicas, ou seja, será apenas uma parte menor da matriz inteira resultante da comparação das sequências completas (Figura 5.10).

O circuito de particionamento v.1 tem como entradas adicionais, além das subsequências a serem comparadas, a coluna, a linha e a diagonal iniciais da matriz de programação dinâmica. Desse modo, a entrada *data_in* será composta pela sequência de banco de dados e pela coluna inicial da matriz de programação dinâmica (Figura 5.11(a)). Quando a máquina de estados está no estado inicial *Read_data* e o sinal *data_in_valid* é setado, os conjuntos de bits referentes à sequência de banco de dados são armazenados no vetor *base_in_r* e os conjuntos de bits referentes à coluna inicial são armazenados no vetor *column_in*. A linha e a diagonal iniciais são setadas de forma *hard-wired*, tal como a sequência de consulta.

No estado *Execute*, os valores das células da coluna inicial entram no primeiro PE analogamente à entrada das bases da sequência de banco de dados. Como a coluna

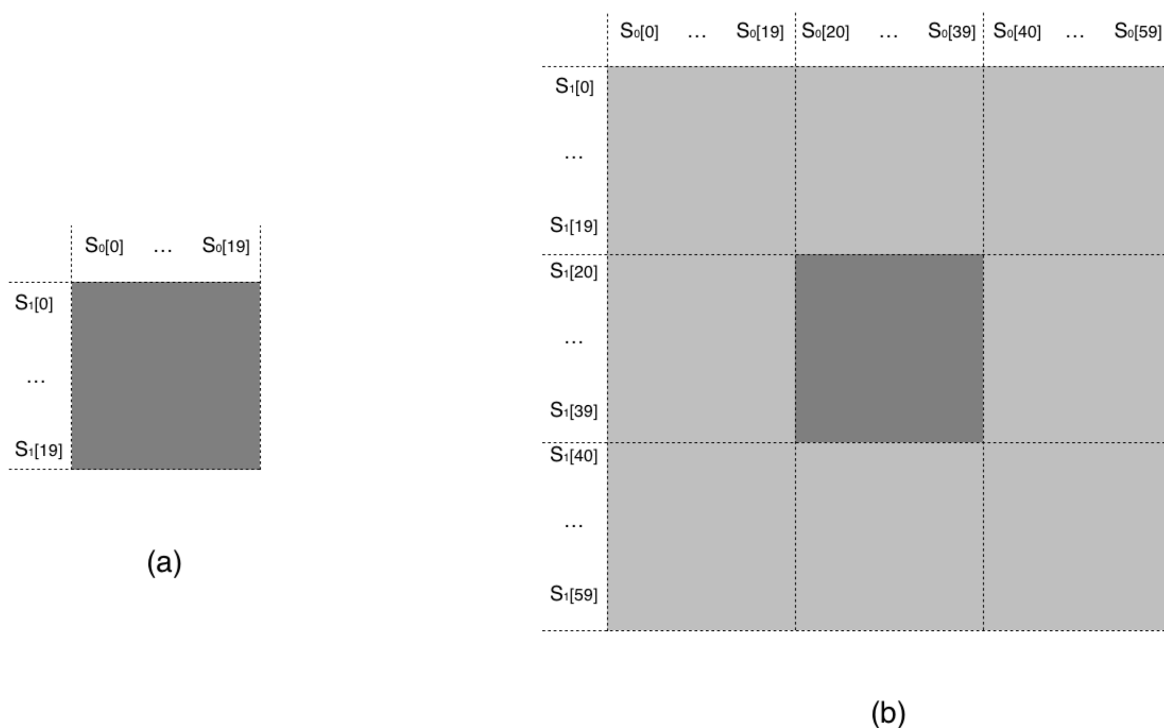


Figura 5.10: (a) Cálculo sem particionamento. (b) Cálculo com particionamento. As áreas com o cinza mais escuro representam a parte da matriz de programação dinâmica sendo calculada no momento. Com isso, nota-se que a matriz da Figura 5.10(a), de tamanho 20×20, deve ser calculada por inteiro de uma vez. Já na Figura 5.10(b), pode-se observar que a matriz está dividida em várias partes delimitadas pelas partições das sequências biológicas sendo comparadas. Cada uma das partes pode ser calculada por vez, o que determina um tamanho ilimitado para as sequências.

inicial também é codificada de forma invertida, são obtidos, neste caso, os 9 bits menos significativos do vetor *column_in*, que representam o valor de uma célula da coluna inicial. Estes bits são passados para o PE a cada ciclo de *clock* e, para que o valor da próxima célula esteja nos bits menos significativos, este vetor também sofre um *shift right* lógico.

Dessa forma, diferentemente da implementação feita no circuito de Colletti, no primeiro ciclo de *clock* de cada PE em que o sinal *flag_in* é setado pela primeira vez, o valor na linha superior será o valor de uma célula da linha inicial da matriz de programação dinâmica. Já o valor na diagonal, caso esteja-se tratando dos PEs de 2 a n, também será o valor de uma célula da linha inicial, só que este valor será o que está na coluna à esquerda da linha superior. Caso trata-se do primeiro PE, o valor na diagonal será o valor da diagonal inicial da matriz de programação dinâmica.

Apenas nos ciclos de *clock* seguintes é que a implementação da linha superior e da diagonal será como a original. Ou seja, na transição do clock, o valor da célula calculado

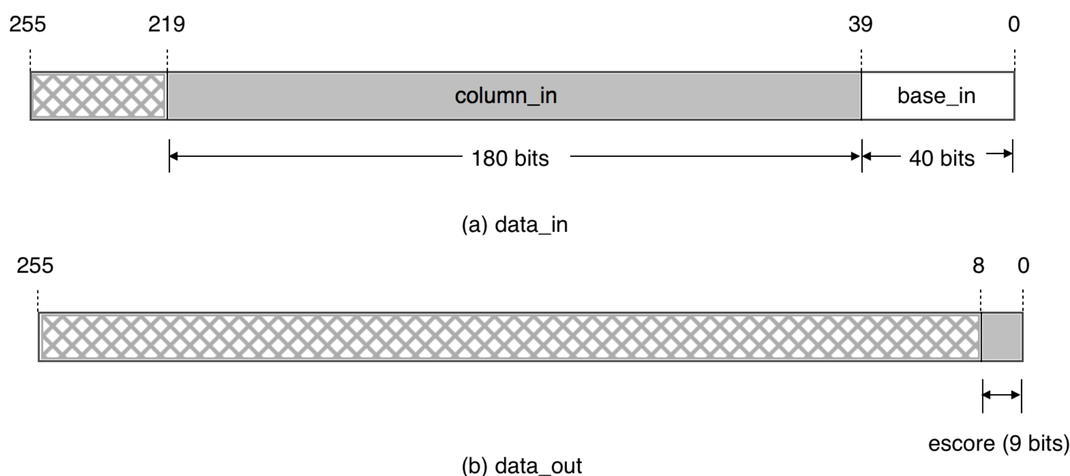


Figura 5.11: Formato das entradas e saídas do circuito de particionamento v1. As áreas hachuradas representam bits ignorados.

dentro do PE no ciclo de clock anterior será armazenado e se tornará o valor na linha superior no ciclo de clock corrente; e o valor que estava na coluna à esquerda do PE no ciclo de clock anterior também será armazenado e se tornará o valor na diagonal no ciclo de clock corrente.

Para que um PE possa reconhecer qual valor a linha superior e a diagonal devem assumir, foi proposto o uso de um contador dentro cada PE. Este contador começa a ser incrementado no momento em que o sinal *flag_in* de um PE é setado. Quando o sinal *flag_in* retorna ao nível lógico baixo, o contador é ressetado. Este contador se comporta como o seletor de dois multiplexadores usados para selecionar os valores da linha superior e diagonal de um PE. A Figura 5.12 mostra o circuito combinacional resultante.

5.3 Circuito de Particionamento proposto v.2

O circuito de particionamento v.2 originou-se do implementado na v.1 (Seção 5.2), só que ele fornecerá como *output* não apenas o escore máximo resultante, mas também a última coluna, linha e diagonal da matriz de programação dinâmica (Figura 5.13(b)), para que possíveis partições existentes, provenientes das sequências biológicas, possam ser comparadas em seguida. Alterou-se também o circuito v.1 para que o conteúdo do dado passado em *data_in* contivesse, além da sequência de banco de dados e da coluna inicial da matriz de programação dinâmica, a linha e a diagonal iniciais da matriz, não sendo mais então setadas de forma *hard-wired* (Figura 5.13(a)).

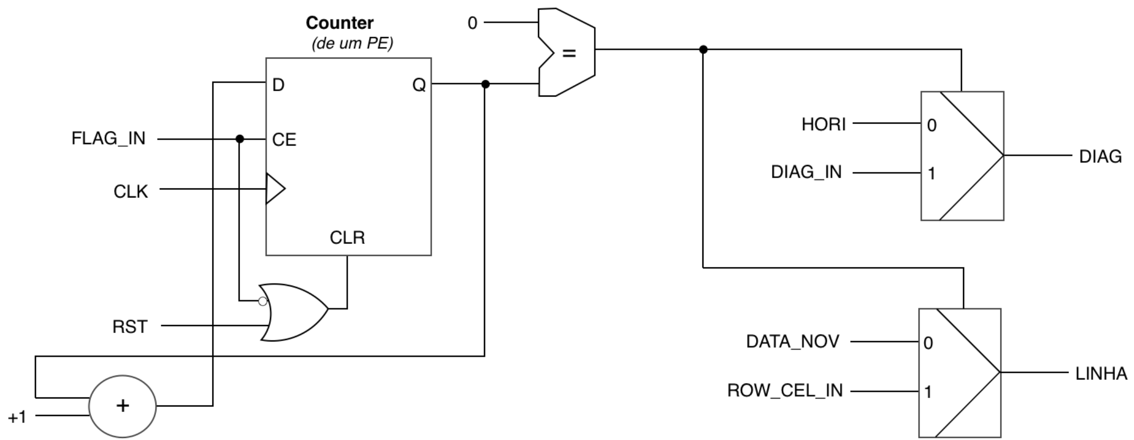


Figura 5.12: Circuito combinacional para determinar os valores da linha superior e diagonal de acordo com o valor registrado pelo contador dentro de um PE. O contador começa a ser incrementado apenas quando FLAG_IN é setado. HORI representa o valor na coluna à esquerda do PE no ciclo de *clock* anterior. DATA_NOV representa o valor calculado no interior do PE no ciclo de *clock* anterior. DIAG_IN e ROW_CEL_IN representam, caso trata-se dos PEs 2 a n, o valor de uma célula da linha inicial da matriz de programação dinâmica. Caso trata-se do PE1, DIAG_CEL é a diagonal inicial da matriz de programação dinâmica. LINHA e DIAG serão utilizados posteriormente nos circuitos combinacionais apresentados nas Figuras 5.4 e 5.5.

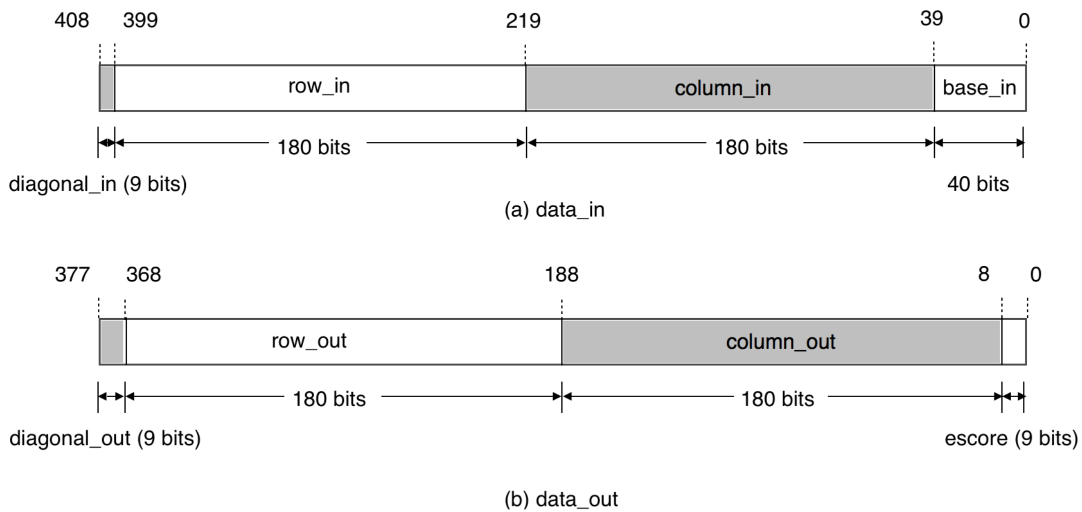


Figura 5.13: Formato das entradas e saídas do circuito de particionamento v2.

Para a obtenção da última coluna da matriz de programação dinâmica, a cada ciclo *clock* a saída do último PE será armazenada em um elemento do *array column_out* correspondente. Em uma estrutura sistólica com n PEs, para identificar o instante em

que o último PE começa a gerar os valores das suas células, observa-se quando o contador da unidade de controle fica maior que o tamanho n da sequência de consulta, sendo incrementado a cada ciclo de *clock* no momento em que a máquina de estados vai para o estado *Execute*.

Para a obtenção da última linha, o último valor calculado dentro de cada PE será armazenado no *array row_out*. Notou-se que o primeiro PE gera o valor da sua última célula quando o contador da unidade de controle fica maior que o tamanho m da sequência de banco de dados e que, a cada ciclo *clock* seguinte, o PE adjacente à direita também gerará seu último valor e assim por diante.

A última diagonal da matriz de programação dinâmica é obtida quando o valor do contador da unidade de controle é igual a soma $m + n$ dos tamanhos das sequências, sendo armazenada em *diagonal_out*.

A Figura 5.14 ilustra os ciclos de *clock*, registrados pelo contador, correspondentes ao momento do cálculo dos valores das células da última coluna, linha e diagonal da matriz de programação dinâmica. O resultado de cada valor é obtido apenas no ciclo de *clock* seguinte.

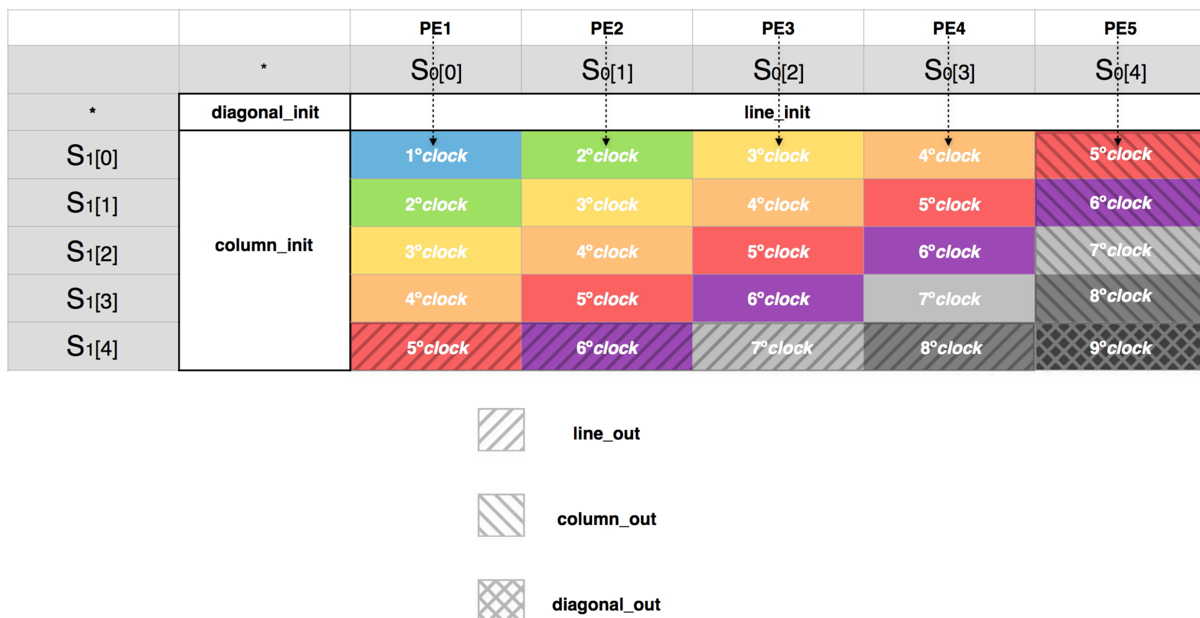


Figura 5.14: Obtenção da última coluna, linha e diagonal da matriz de programação dinâmica resultante da comparação entre as sequências S_0, de tamanho $n = 5$, e S_1, de tamanho $m = 5$. A última coluna e linha serão obtidas a partir do momento em que contador da unidade de controle registrar que o ciclo de *clock* corrente é maior que n e m, respectivamente. A última diagonal é obtida quando o contador registrar que o ciclo de *clock* corrente é igual a $m + n = 10$.

5.4 Integração com a plataforma XD2000i

Devido à falta de suporte fornecido pela plataforma XD2000i, não foi possível integrar o projeto do circuito de particionamento v.2 (Seção 5.3), que implementa dados de entrada e de saída com tamanhos que ultrapassam os estabelecidos pelas regras da plataforma. Em contrapartida, o projeto do design de referência em VHDL, disponibilizado junto à plataforma XD, foi customizado para que a FPGA appA fosse configurada de tal forma que se comportasse como o circuito de particionamento v.1, descrito na Seção 5.2. Como dito na Seção 4.2, este projeto do design de referência implementa a *Accelerator Functional Unit* (AFU), cujos dados de entrada são sequencialmente transferidos para a AFU por um software através da FPGA *Bridge*. Para haver essa comunicação entre a AFU e a FPGA *Bridge*, o protocolo FIFO deve ser estabelecido adicionando a entrada *data_in_ready*, que é setada quando se deseja ler dados, e a saída *data_out_ready*, que é setada quando se deseja escrever um dado, ao módulo que customiza o projeto do design de referência, cuja arquitetura é a que descreve o circuito de particionamento v.1. Com isso, a integração deste módulo com o projeto do design de referência pode ser efetuada. Este módulo resultante pode ser visto na Figura 5.15. Apesar das entradas e saídas serem iguais às presentes no circuito de Colletti, o conteúdo de *data_in* não é o mesmo, tendo o formato correspondente ao do circuito de particionamento v.1 (Seção 5.2), apresentado na Figura 5.11.

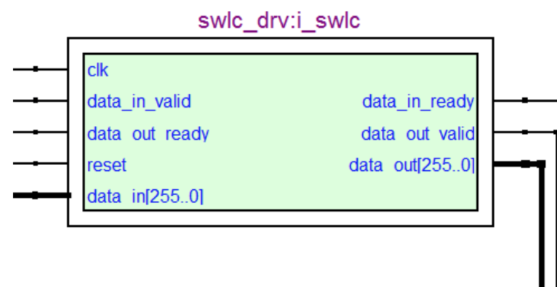


Figura 5.15: O módulo que customiza o projeto de design de referência.

Capítulo 6

Resultados Experimentais

Para verificar o funcionamento dos circuitos de particionamento v.1 (Seção 5.2) e v.2 (Seção 5.3), simulações funcionais no Quartus II 8.1[1] e execuções na plataforma XD2000i foram realizadas utilizando três diferentes pares de sequências biológicas (Seção 6.1).

As seções seguintes descrevem os resultados das simulações dos circuitos de particionamento v.1 e v.2 (Seção 6.2), dos processos de síntese do circuito v.1 (Seção 6.3) e das execuções na plataforma XD2000i do circuito v.1 sintetizado (Seção 6.4).

6.1 Sequências utilizadas

A Tabela 6.1 mostra três pares de sequências biológicas, com 40 bases cada, gerados de forma sintética. As últimas 20 bases de cada uma destas sequências foram escolhidas para representarem as partições, mostradas na Tabela 6.2, que serão utilizadas como as sequências de consulta S_C e as sequências de banco de dados S_{BD} nas simulações no Quartus II 8.1 [1] e nas execuções na plataforma XD2000i (Seção 6.4). A Tabela 6.3 apresenta as sequências de banco de dados S_{BD} codificadas de forma invertida em binário e em hexadecimal, sendo as sequências de consulta setadas de forma *hard-wired*. Por esta tabela é possível observar que as bases A, T, C e G são codificadas como 00, 01, 10 e 11, respectivamente, gerando partições de tamanho 40 bits.

Tabela 6.1: Sequências biológicas geradas de forma sintética.

	S_0	S_1
(a)	CATAGTCAATCAGGTTAAGCCATAGTCAATCAGGTTAAGC	CATAGTCAATAAAAAAAAAACATAGTCAATAAAAAAAAAA
(b)	ATGAGATCCAGTATTCTCAAATGAGATCCAGTATTCTCAA	AGATTTCCAGTTTCGGCTCAAGATTTCCAGTTTCGGCTCA
(c)	TAAAGATTGGCAGTGGGCATATCAAAGTCACACCGGCCTT	AAAGATTTCCAGTTTGGGCATGGCAAAGGCTAAGGCCCCT

Tabela 6.2: Partições das sequências biológicas mostradas na Tabela 6.1.

	S_C (Partição de S_0)	S_{BD} (Partição de S_1)
(a)	CATAGTCAATCAGGTTAAGC	CATAGTCAATAAAAAAAAAA
(b)	ATGAGATCCAGTATTCTCAA	AGATTTCCAGTTTCGGCTCA
(c)	ATCAAAGTCACACCGGCCTT	TGGCAAAGGCTAAGGCCCCT

Tabela 6.3: Codificação da sequência de banco de dados S_{BD} em binário e em hexadecimal.

	S_{BD}	S_{BD} em binário	S_{BD} em hexadecimal
(a)	CATAGTCAATAAAAAAAAAA	0000000000000000000001000010011100010010	$0x42712$
(b)	AGATTTCCAGTTTCGGCTCA	0010011011111001010111001010010101001100	$0x26F95CA54C$
(c)	TGGCAAAGGCTAAGGCCCCT	0110101010111100000110111100000010111101	$0x6ABC1BC0BD$

6.2 Simulação

Simulações funcionais dos módulos que implementam os circuitos de particionamento v.1 e v.2 foram realizadas no Quartus II 8.1 [1] pela ferramenta de simulação Quartus II 64-Bit Simulator [2]. Arquivos .vfw (*Vector Waveform File*) foram criados para conter os valores que os sinais de entrada *clk*, *reset*, *data_in*, *data_in_valid* e *data_out_ready* deverão assumir quando cada simulação for executada. Ao serem estimulados por estes sinais de entrada, os módulos produzirão os sinais de saída *data_in_ready*, *data_out* e *data_out_valid*. As Figuras 6.1 a 6.3 e as Figuras 6.4 a 6.7 mostram os resultados das simulações dos circuitos v.1 e v.2, respectivamente, para cada par de partições de sequências biológicas mostrado na Tabela 6.2. Os sinais *data_in* e *data_out* dos circuito v.1 e v.2 possuem os formatos apresentados nas Figuras 5.11 e 5.13, respectivamente.

A Figura 6.1 apresenta o final da execução da simulação do circuito de particionamento v.1. Como pode ser visto, entre os instantes 440 ns e 460 ns, o sinal *data_out_valid* é setado para 1 e o escore 16 é produzido. As Figuras 6.2 e 6.3 apresentam o mesmo comportamento.

A Figura 6.4 apresenta o final da execução da simulação do circuito de particionamento v.2. Como pode ser visto, entre os instantes 440 ns e 460 ns, quando o sinal *data_out_valid* é setado para 1, o sinal *data_out* é setado com o valor do escore, coluna, linha e diagonal finais da matriz de programação dinâmica. A Figura 6.5 apresenta o mesmo instante da simulação, mas com o valor completo de *data_out*, assim como as Figuras 6.6 e 6.7.

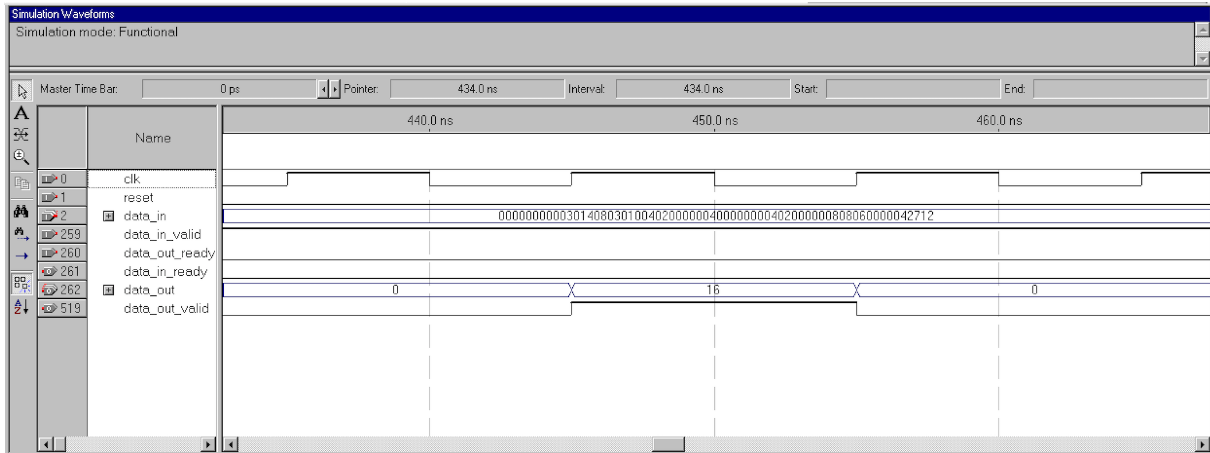


Figura 6.1: Resultado da simulação do circuito de particionamento v.1 (instantes 430 ns a 470 ns) realizada com a sequência de consulta S_C(a) e a sequência de banco de dado S_{BD}(a) mostradas na Tabela 6.2.

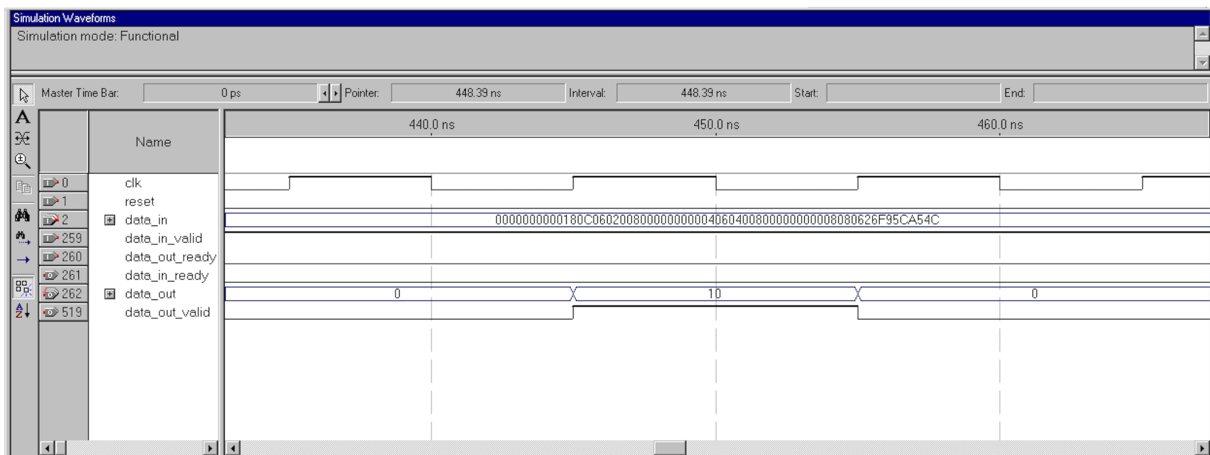


Figura 6.2: Resultado da simulação do circuito de particionamento v.1 realizada com a sequência de consulta S_C(b) e a sequência de banco de dado S_{BD}(b) mostradas na Tabela 6.2.

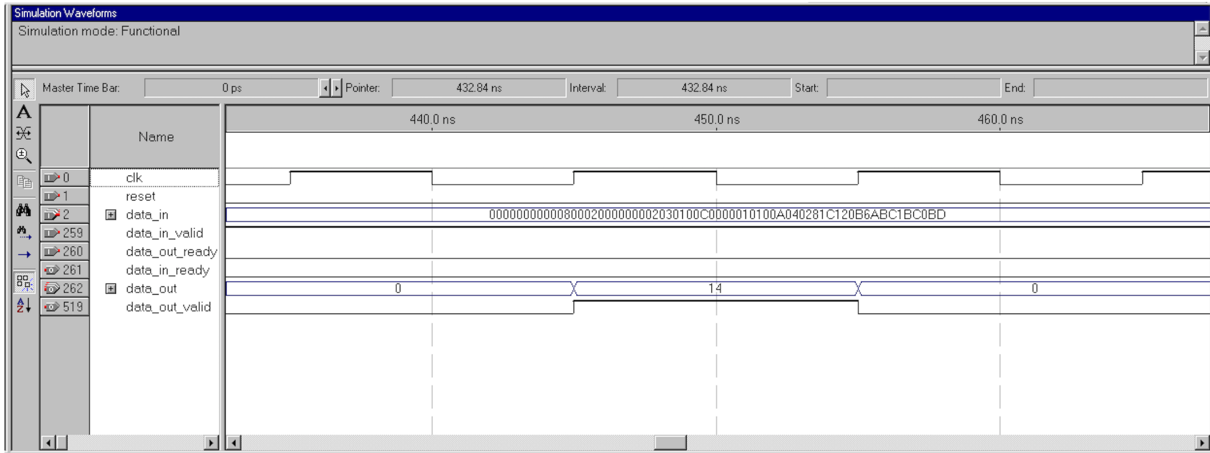


Figura 6.3: Resultado da simulação do circuito de particionamento v.1 realizada com a sequência de consulta S_C(c) e a sequência de banco de dado S_{BD}(c) mostradas na Tabela 6.2.

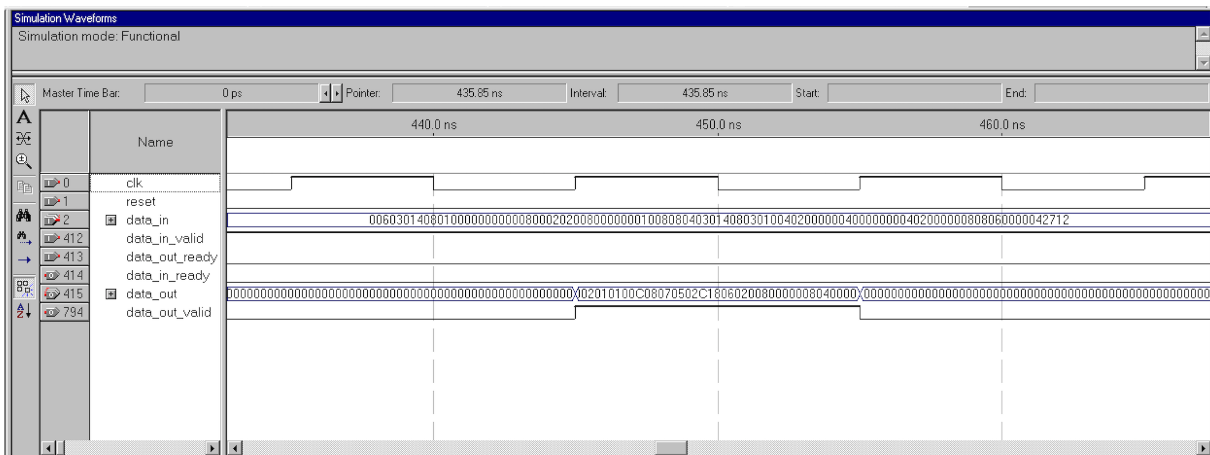


Figura 6.4: Resultado da simulação do circuito de particionamento v.2 (instantes 430 ns a 470 ns) realizada com a sequência de consulta S_C(a) e a sequência de banco de dado S_{BD}(a) mostradas na Tabela 6.2.

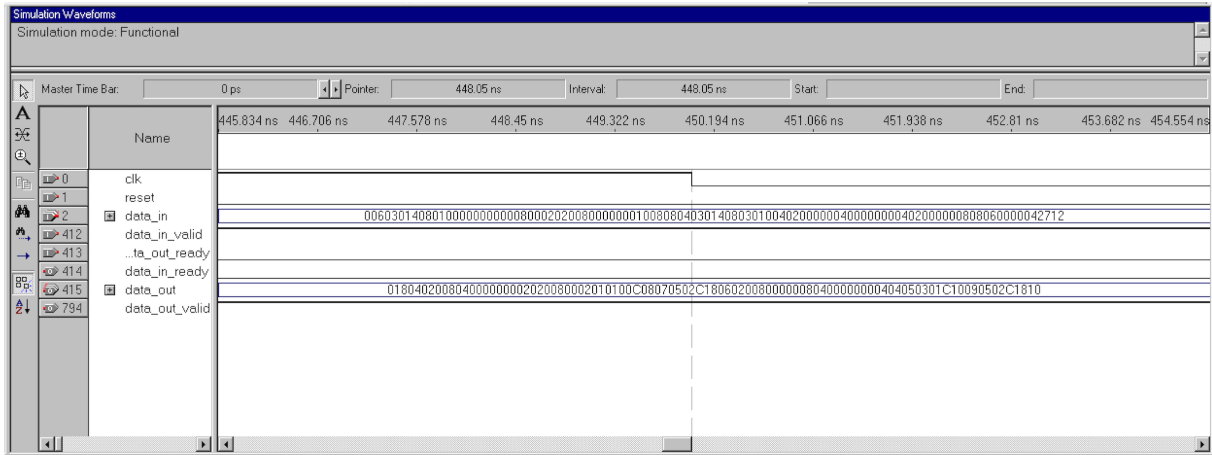


Figura 6.5: Resultado da simulação do circuito de particionamento v.2 realizada com a sequência de consulta S_C(a) e a sequência de banco de dado S_{BD}(a) mostradas na Tabela 6.2.

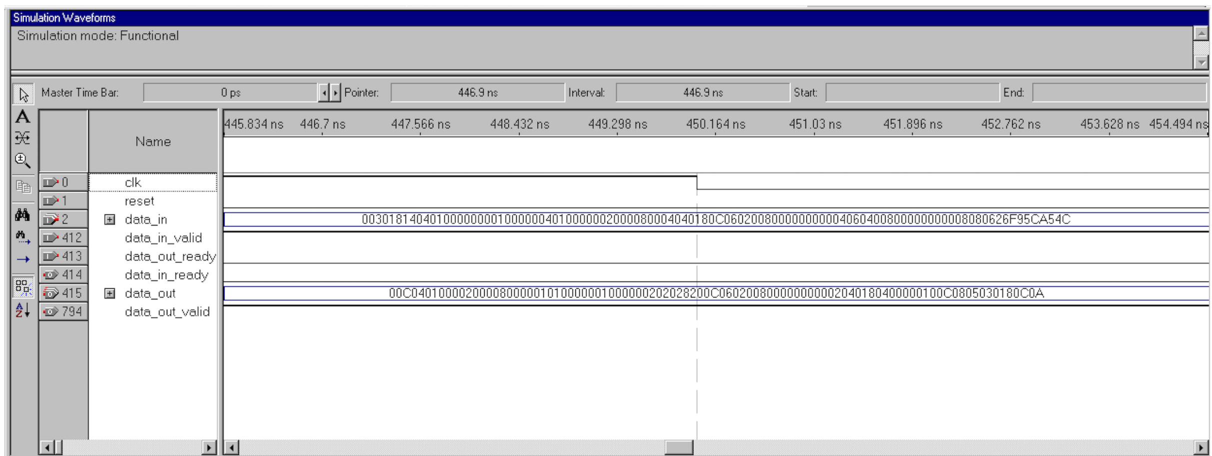


Figura 6.6: Resultado da simulação do circuito de particionamento v.2 realizada com a sequência de consulta S_C(b) e a sequência de banco de dado S_{BD}(b) mostradas na Tabela 6.2.

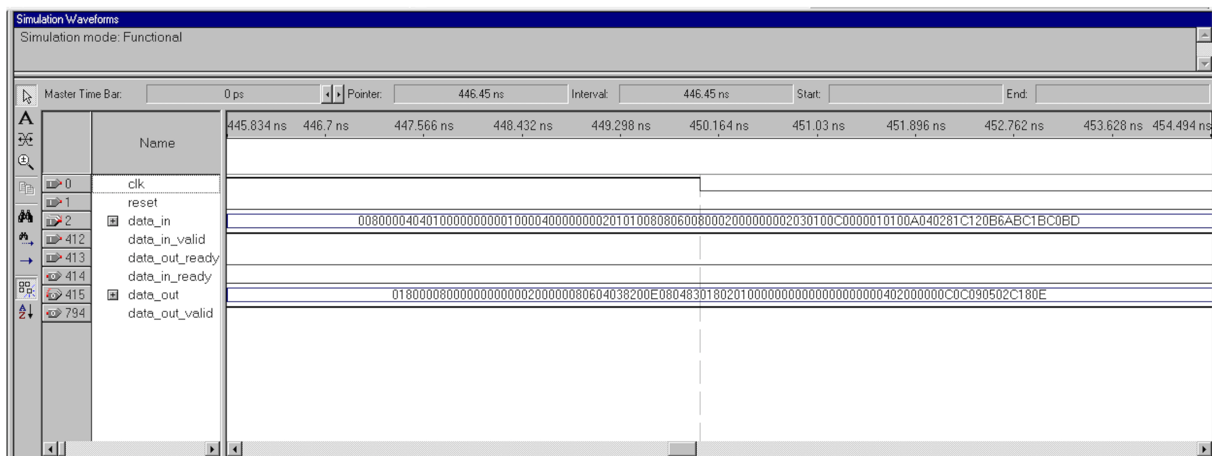


Figura 6.7: Resultado da simulação do circuito de particionamento v.2 realizada com a sequência de consulta S_C(c) e a sequência de banco de dado S_{BD}(c) mostradas na Tabela 6.2.

6.3 Síntese do circuito de particionamento v.1

Implementações do projeto do design de referência customizado pelo módulo que implementa o circuito de particionamento v.1 com 20, 40, 80, 100, 512 e 1024 elementos de processamento (PE) (Seção 5.1.3) foram sintetizadas no Quartus II 8.1 [1]. A FPGA XD2000i de destino foi a FPGA AppA, uma Stratix III EP3SE260F1152C3 com 255.000 elementos lógicos, 203.000 registradores e 14.688Kbits de memória interna. A Tabela 6.4 mostra os resultados dos processos de síntese do circuito v.1 com essas diferentes quantidades de PEs.

Tabela 6.4: Resultados das sínteses do circuito de particionamento v.1 implementando diferentes quantidades de PEs.

PEs	ALUTs	Registradores dedicados	% de utilização	Frequência sintetizada	Pinos de I/O
20	3.416	2.613	2%	77MHz	328
40	6.310	4.213	4%	77MHz	328
80	12.071	7.413	8%	77MHz	328
100	14.951	9.013	9%	77MHz	328
512	74.280	41.973	45%	77MHz	328
1024	138.800	82.933	87%	77MHz	328

Flow Summary

Flow Status	Successful - Sat Oct 22 22:31:54 2016
Quartus II 64-Bit Version	8.1 Build 163 10/28/2008 SJ Full Version
Revision Name	appA_top
Top-level Entity Name	app_top
Family	Stratix III
Device	EP3SE260F1152C3
Timing Models	Preliminary
Met timing requirements	N/A
Logic utilization	87 %
Combinational ALUTs	138,800 / 203,520 (68 %)
Memory ALUTs	0 / 101,760 (0 %)
Dedicated logic registers	82,933 / 203,520 (41 %)
Total registers	83360
Total pins	328 / 744 (44 %)
Total virtual pins	0
Total block memory bits	217,856 / 15,040,512 (1 %)
DSP block 18-bit elements	0 / 768 (0 %)
Total PLLs	3 / 8 (38 %)
Total DLLs	0 / 4 (0 %)

Figura 6.8: Resultado da síntese no Quartus II 8.1 [1] do circuito de particionamento v.1 implementado com 1024 PEs.

6.4 Execução do circuito de particionamento v.1 na plataforma XD2000i

O circuito de particionamento v.1 foi executado na plataforma XD2000i (Capítulo 4) utilizando cada um dos três pares de sequências biológicas mostrados na Tabela 6.2. Para cada par de sequências, foram realizadas três execuções na plataforma. As Figuras 6.9 a 6.11 mostram os resultados de uma das execuções de cada par. O tempo de execução especificado nestas figuras equivale ao tempo gasto para realizar as operações de envio e recebimento de dados, descritas na Seção 4.3, feitas através da API disponibilizada junto à plataforma XD2000i, bem como o tempo de execução da comparação. É possível observar pelas figuras que o tempo de execução sofre uma variação discrepante entre os testes realizados. Acredita-se que esse efeito seja devido à um fator de *software* e ao tamanho dos dados sendo transferidos. A Tabela 6.5 mostra que o tempo de execução variou entre até mesmo as três execuções realizadas para cada par de sequências.

```
****************************************************************
*  Total de dados (bytes): 64                                  *
*  Tempo de execucao (us): 115                                 *
*  Taxa de transferencia (kbps): 569.878261                    *
*  Escore: 16                                                  *
****************************************************************
```

Figura 6.9: Resultado da execução do circuito de particionamento v.1 na plataforma XD2000i utilizando a sequência de consulta S_C(a) e a sequência de banco de dado S_{BD}(a) mostradas na Tabela 6.2.

```
****************************************************************
*  Total de dados (bytes): 64                                  *
*  Tempo de execucao (us): 76                                  *
*  Taxa de transferencia (kbps): 862.315789                    *
*  Escore: 10                                                  *
****************************************************************
```

Figura 6.10: Resultado da execução do circuito de particionamento v.1 na plataforma XD2000i utilizando a sequência de consulta S_C(b) e a sequência de banco de dado S_{BD}(b) mostradas na Tabela 6.2.

```
****************************************************************
*  Total de dados (bytes): 64                                  *
*  Tempo de execucao (us): 948                                 *
*  Taxa de transferencia (kbps): 69.130802                     *
*  Escore: 14                                                  *
****************************************************************
```

Figura 6.11: Resultado da execução do circuito de particionamento v.1 na plataforma XD2000i utilizando a sequência de consulta S_C(c) e a sequência de banco de dado S_{BD}(c) mostradas na Tabela 6.2.

Tabela 6.5: Resultados das execuções do circuito de particionamento v.1 na plataforma XD2000i com as sequências de consulta S_C e as sequências de banco de dados S_{BD} mostradas na Tabela 6.2.

S_C e S_{BD} da Tabela 6.2 utilizadas	Tempo de execução (μs)	
	Menor	Maior
(a)	115	829
(b)	76	867
(c)	125	948

Capítulo 7

Conclusão

O presente trabalho de graduação propôs, implementou e avaliou uma solução em FPGA para a comparação de sequências biológicas longas com particionamento. Foram propostos dois circuitos de particionamento, v.1 e v.2, que foram descritos em VHDL e simulados pela ferramenta de simulação Quartus II 64-Bit Simulator [2]. O circuito de particionamento v.1 foi sintetizado no Quartus II 8.1 [1], tendo como alvo a FPGA de aplicação AppA, uma Stratix III EP3SE260F1152C3, e executado na plataforma XD2000i.

Os resultados experimentais obtidos com 3 pares diferentes de sequência biológicas foram bem satisfatórios. Na síntese do circuito v.1, conseguiu-se colocar até 1024 PEs em uma FPGA, obtendo 87% de utilização. Apesar de não ter sido possível a integração do circuito de particionamento v.2 com a plataforma XD2000i, os resultados das simulações foram suficientes para mostrar uma adequada execução de ambos os circuitos, conforme apresentado nas Figuras 6.1 a 6.7. Os tempos de execução para enviar, processar e receber os dados na plataforma XD2000i com 20 PEs, mesmo que sofrendo grandes variações, foram bastante aceitáveis (até $948\mu s$).

Como trabalhos futuros, sugerimos:

- Integração do circuito de particionamento v.2 com a ferramenta MASA e sua execução na plataforma XD2000i.
- Estender a solução para várias FPGAs trabalhando de maneira cooperativa.
- Aumentar o número de PEs, alterando o projeto, de maneira a, por exemplo, incluir um *pipeline* dentro de cada PE.

Referências

[1] ALTERA. Quartus II Handbook Version 8.1. https://www.altera.com/en_US/pdfs/literature/hb/qts/archives/quartusii_handbook_8.1.pdf. xi, 2, 39, 40, 44, 45, 47

[2] ALTERA. Quartus II Simulation using VHDL designs. ftp://ftp.altera.com/up/pub/Altera_Material/9.1/Tutorials/VHDL/Quartus_II_Simulation.pdf. 40, 47

[3] Javier Bueno, Luis Martinell, Alejandro Duran, Montse Farreras, Xavier Martorell, Rosa M. Badia, Eduard Ayguade, e Jesús Labarta. Productive cluster programming with ompss. In *Proceedings of the 17th International Conference on Parallel Processing - Volume Part I*, Euro-Par'11, pages 555–566, Berlin, Heidelberg, 2011. Springer-Verlag. 17

[4] L. G. A. Carvalho. Uma abordagem em hardware para algoritmos de comparação de sequências baseados em programação dinâmica. Dissertação (Mestrado), Departamento de Ciência da Computação, Universidade de Brasília, Brasília, DF, Dezembro 2003. ix, x, 27, 28, 29, 30

[5] M. Colletti. Comparação de sequências biológicas utilizando a plataforma XD2000i de hardware reconfigurável. Trabalho de Conclusão de Curso (Graduação), Departamento de Ciência da Computação, Universidade de Brasília, Brasília, DF, Fevereiro 2016. x, 27, 31, 32, 33

[6] Leonardo Dagum e Ramesh Menon. Openmp: an industry standard api for shared-memory programming. *Computational Science & Engineering, IEEE*, 5(1):46–55, 1998. 17

[7] A. Driga, P. Lu, J. Schaeffer, D. Szafron, K. Charter, e I. Parsons. FastLSA: A Fast, Linear-Space, Parallel and Sequential Algorithm for Sequence Alignment. *Algorithmica*, 45(3):337–375, 2006. 11

[8] A. Duran, E. Ayguade, R. M. Badia, J. Labarta, L. Martinell, X. Martorell, e J. Planas. OmpSs: a Proposal for Programming Heterogeneous Multicore Architectures. *Parallel Processing Letters*, 21(2):173–193, 2011. 17

[9] R. Durbin, S. Eddy, A. Krogh, e G. Mitchison. *Biological sequence analysis*. Cambridge University Press, 2002. 3, 4

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

[11] D. S. Hirschberg. A linear space algorithm for computing maximal common subsequences. *Commun. ACM*, 18(6):341–343, 1975. 8

[12] XtremeData Inc. XD2000iTM development system user handbook, Junho 2009. ix, 19, 20, 21

[13] James Jeffers e James Reinders. *Intel Xeon Phi Coprocessor High Performance Programming*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2013. 1

[14] David B. Kirk e Wen mei W. Hwu. *Programming Massively Parallel Processors: A Hands-on Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2010. 1

[15] D. W. Mount. *Bioinformatics: sequence and genome analysis*. CSHL Press, 2004. 3, 4, 7

[16] E. W. Myers e W. Miller. Optimal alignments in linear space. *Computer Applications in the Biosciences*, 4(1):11–17, 1988. 8

[17] S. B. Needleman e C. D. Wunsch. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *Journal of Molecular Biology*, 48(3):443–453, March 1970. 4

[18] NVIDIA. NVIDIA CUDA Programming Guide 3.2, 2010. 17

[19] E. F. de O. Sandes e A. C. M. A. de Melo. CUDAlign: using GPU to accelerate the comparison of megabase genomic sequences. In *PPOPP*, pages 137–146. ACM, 2010. 10

[20] E. F. de O. Sandes e A. C. M. A. Melo. Smith-waterman alignment of huge sequences with gpu in linear space. In *International Parallel and Distributed Processing Symposium (IPDPS)*, pages 1199–1211, 2011. 11

[21] E. F. de O. Sandes, G. Miranda, A. C. M. A. Melo, X. Martorell, e E. Ayguade. CUDAlign 3.0: Parallel Biological Sequence Comparison in Large GPU Clusters. In *IEEE/ACM Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, pages 160–169, 2014. 13

[22] E. F. O. Sandes. *Algoritmos Paralelos Exatos e Otimizações para Alinhamento de Sequências Biológicas Longas em Plataformas de Alto Desempenho*. Tese (Doutorado), Departamento de Ciência da Computação, Universidade de Brasília, Brasília, DF, 2015. ix, 9, 10, 12, 15, 16, 18

[23] E. F. O. Sandes, A. Boukerche, e A. C. M. Melo. Parallel optimal pairwise biological sequence comparison: Algorithms, platforms, and classification. *ACM Compution Surveys*, 48(63), Março 2016. 27

[24] E. F. O. Sandes e A. C. M. A Melo. Retrieving smith-waterman alignments with optimizations for megabase biological sequences using gpu. *IEEE Transactions on Parallel and Distributed Systems*, 24(5):1009–1021, 2013. 13

[25] Edans F. O. Sandes, GUILLERMO MIRANDA, E. Ayguade, XAVIER MARTORELL, G. Teodoro, e Alba C. M. A. De Melo. Masa: A multiplatform architecture for sequence aligners with block pruning. *ACM Transactions on Parallel Computing*, 2:1–31, 2016. 1, 14

[26] T. F. Smith e M. S. Waterman. Identification of common molecular subsequences. *Journal of Molecular Biology*, 147(1):195–197, March 1981. 4, 6

[27] Lars Wienbrandt. Bioinformatics applications on the fpga-based high-performance computer rivyera. In Wim Vanderbauwhede e Khaled Benkrid, editors, *High- Performance Computing Using FPGAs*, pages 81–103. Springer New York, 2013. 1