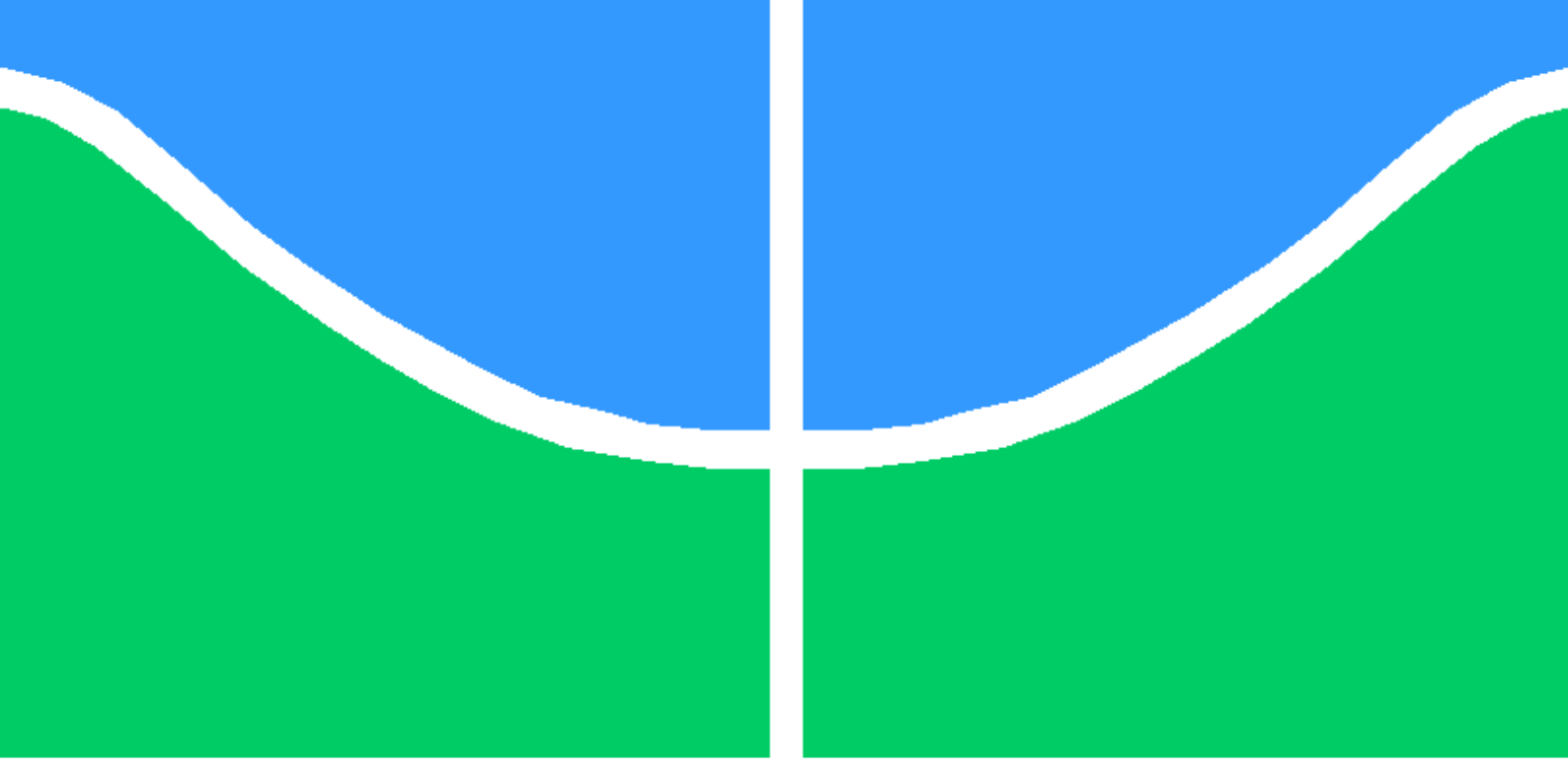

TRABALHO DE GRADUAÇÃO

COMPARAÇÃO DE SEQUÊNCIAS BIOLÓGICAS UTILIZANDO A PLATAFORMA XD2000i DE HARDWARE RECONFIGURÁVEL

Marcelo Ramos Colletti

Brasília, Fevereiro de 2016

UNIVERSIDADE DE BRASÍLIA

DEPARTAMENTO DE CIÊNCIA DA COMPUTAÇÃO

UNIVERSIDADE DE BRASILIA
Departamento de Ciência da Computação

TRABALHO DE GRADUAÇÃO

COMPARAÇÃO DE SEQUÊNCIAS BIOLÓGICAS UTILIZANDO A PLATAFORMA XD2000i DE HARDWARE RECONFIGURÁVEL

Marcelo Ramos Colletti

Relatório submetido ao Departamento de Ciência da Computação como requisito parcial para obtenção do grau de Bacharel em Engenharia de Computação

Banca Examinadora

Prof. Alba Cristina Magalhães Alves de Melo, CIC/UnB _____
Orientador

Prof. Pedro de Azevedo Berger, CIC/UnB _____
Examinador interno

Prof. Ricardo Pezzuol Jacobi, CIC/UnB _____
Examinador interno

UNIVERSIDADE DE BRASÍLIA
DEPARTAMENTO DE CIÊNCIA DA COMPUTAÇÃO

COMPARAÇÃO DE SEQUÊNCIAS BIOLÓGICAS UTILIZANDO A PLATAFORMA XD2000i DE HARDWARE RECONFIGURÁVEL

MARCELO RAMOS COLLETTI

ORIENTADORA: ALBA CRISTINA MAGALHÃES ALVES DE MELO

DISSERTAÇÃO DE GRADUAÇÃO EM
ENGENHARIA DE COMPUTAÇÃO

BRASÍLIA/DF: FEVEREIRO - 2016.

FICHA CATALOGRÁFICA

C698c
COLLETTI, Marcelo Ramos
Comparação de Sequências Biológicas Utilizando
a Plataforma XD2000i de Hardware Reconfigurável / Marcelo Ramos Colletti. - Brasília, 2016.
50f. ; il
Orientação: Prof. Dra. Alba Cristina Magalhães Alves de Melo, 2016)
Monografia (Graduação) - Universidade de Brasília,
Faculdade de Tecnologia, 2016.
Departamento de Ciência da Computação.
1. Bioinformática 2. FPGA
3. Comparação de Sequências Biológicas

CESSÃO DE DIREITOS

NOME DO AUTOR: Marcelo Ramos Colletti.

TÍTULO DA DISSERTAÇÃO DE GRADUAÇÃO: Comparação de Sequências Biológicas Utilizando a Plataforma XD2000i de Hardware Reconfigurável.

GRAU / ANO: Bacharel / 2016

Marcelo Ramos Colletti
Universidade de Brasília
Campus Universitário Darcy Ribeiro - Asa Norte
CEP 70910-900
Brasília-DF - Brasil

RESUMO

Uma das tarefas mais básicas da bioinformática é verificar se duas sequências biológicas são similares. Essa tarefa está se tornando cada vez mais custosa pois as bases de dados e o volume de informações disponíveis possuem crescimento exponencial e os algoritmos de programação dinâmica mais comumente usados possuem complexidade computacional quadrática. Dessa forma, são necessárias técnicas de processamento que obtenham alto desempenho para executar comparações de sequências biológicas. A utilização de FPGA (*Field Programmable Gate Array* para a realização de tais tarefas tem se mostrado atrativa. O presente trabalho propõe, implementa e analisa uma solução eficiente para comparação de sequências biológicas na plataforma XD2000i^{TM}um usando *array* sistólico unidirecional.

Paralvras-chave: Bioinformática, FPGA, Comparação de Sequências Biológicas.

ABSTRACT

One of the most basic tasks in Bioinformatics is to verify if two biological sequences are similar. This task is becoming increasingly harder, as genetics database and the amount of available data grows in an exponencial manner. Besides, the most widely used dynamic programming algorithms has quadratic time complexity. From this reason comes the need to develop techniques that can achieve high performance in biological sequence comparison. The use of FPGAs (Field Programmable Gate Array) for performing such tasks has been shown to be a very interesting approach. This work proposes, implements and evaluates an efficient solution for biological sequence comparison utilizing the XtremeData XD2000i, which is a high performance FPGA-base platform.

Keywords: Bioinformatics, FPGA, Biolgical Sequences Comparison.

SUMÁRIO

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

AFU	Accelerator Functional Unit
AHM	Accelerator Hardware Module
ALUT	Adaptive Lookup Table
API	Application Programming Interface
ASIC	Applcation Specific Integrated Circuit
CPU	Central Processing Unit
CUPS	Cell Updates Per Second
DW	Diagonal Wavefront
FIFO	First-in First-out
FPGA	Field Gate Programmable Array
FSB	Front-side Bus
HDL	Hardware Description Language
LUT	Lookup Table
MCH	Memory Controller Hub
NW	Needleman-Wunsh
PE	Elemento de Processamento
QDR	Quad Data Rate
RAM	Random Access Memory
SRAM	Static Ram
SW	Smith-Waterman
VHDL	VHSIC Hardware Description Language
VWF	Vector Waveform File

Capítulo 1

Introdução

Os avanços da técnologia permitem que biólogos extraiam informações genéticas a taxas cada vez maiores. Isso causou um grande crescimento da quantidade de informação disponível, permitindo a realização em larga escala de análises para descobrir características comuns no mais diversos organismos estudados pela biologia. O enorme volume de dados biológicos disponíveis em bancos de dados genômicos nos coloca face ao problema de conseguir analisar os dados na mesma taxa com que são produzidos. A bioinformática possui papel fundamental nessa tarefa.

Há duas principais formas de comparar um par de sequências biológicas: por meio de (a) um método exato, que é capaz de calcular a solução ótima para o problema de comparação; ou (b) um método heurístico, que utiliza modelos não exatos para determinar boas soluções, com baso nos grandes conjuntos de exemplos conhecidos [12]. Os métodos heurísticos buscam uma aproximação do resultado ótimo, enquanto os métodos exatos sempre produzem o ótimo. Porém, os métodos exatos mais comuns, possuem complexidade quadrática de tempo.

Como os algoritmos de soluções exatas possuem complexidade de tempo $O(n^2)$, foram propostas soluções paralelas para acelerar esses algoritmos. Dentre essas soluções, se encontram implementações de algoritmos de programação dinâmica em CPU [13], GPU [14], FPGA [3], CellBEs [15] (*Cell Broadband Engines*) e Intel Xeon Phi [16]. As soluções em FPGA são capazes de gerar circuitos integrados reconfiguráveis altamente especializados na comparação de sequências.

Umas das dificuldades para que o uso de FPGAs seja mais disseminado é a existência de tarefas altamente dependentes do sistema *Host-FPGA*. Essas tarefas podem demandar mais tempo do que o projeto do circuito que implementa o algoritmo de comparação de sequências, passo obrigatório no desenvolvimento da solução. Por esse motivo, buscam-se soluções que facilitem o desenvolvimento da interface entre o *host* e a FPGA. A plataforma XD2000i permitem que os dados sejam transmitidos sequencialmente entre a memória do host e a unidade funcional customizada pelo usuário gravada em FPGA. Tal transferência é programada através de uma API (*Application Programming Interface* de alto nível, facilitando bastante a programação dessa tarefa.

O objetivo do presente trabalho de graduação é propor, implementar e avaliar uma solução para a comparação de sequências biológicas com o algoritmo de Smith-Waterman [17] para a plataforma XD2000i^{TM}. A proposta foi baseada no projeto de Carvalho [2] e foram propostas duas extensões:

(1) cálculo de escore máximo; e (2) unidade de controle. A unidade de controle realiza o controle do circuito projetado por Carvalho [2] através de uma máquina de estados.

O restante desta monografia está organizado da seguinte maneira. O capítulo 2 apresenta uma introdução sobre a comparação de sequências biológicas e os algoritmos relacionados. O capítulo 3 descreve as FPGAs e apresenta as soluções existentes que utilizam FPGA na comparação de sequências biológicas. O capítulo 4 descreve a plataforma XD2000i^{TM}. No capítulo 5, é apresentada a estratégia de comparação de sequências na plataforma. O capítulo 6 mostra os resultados sintetizados e testes realizados. Por fim, o capítulo 7 apresenta as conclusões do presente trabalho e sugere trabalhos futuros.

Capítulo 2

Comparação de Sequências Biológicas

Uma das tarefas mais básicas da bioinformática é verificar se duas sequências, ou parte delas, são biologicamente relacionadas. Ao se comparar duas sequências, procura-se evidenciar as similaridades e também as divergências através de um processo evolutivo. Esse processo ocorre quando resíduos são alterados, ou inseridos/removidos da sequência original [12].

Entende-se por sequência biológica uma sequência de bases nitrogenadas, que formam RNA ou DNA; ou uma sequência de aminoácidos, que formam proteínas.

Na comparação de duas sequências biológicas, calculam-se a partir de métodos computacionas algumas métricas que ajudam a identificar o seu grau de relacionamento. Uma dessas méticas é o escore, que é atribuído a um alinhamento. Um alinhamento é definido como um pareamento, resíduo a resíduo, das sequências. Um par de resíduos das duas sequências pode ser definido como *match*, quando os pares são iguais, *mismatch*, quando os pares são distintos ou um *gap*, quando o resíduo de uma sequência está alinhado com uma lacuna. Um esquema de escore é definido para cada par, havendo pontuações para *match*, *mismatch* e *gap*, e a soma das pontuações do alinhamento compõem o escore, que quantifica o grau de similaridade do alinhamento [12].

Algumas características são definidas na obtenção do alinhamento:

1. **Tipo de alinhamento**. O alinhamento pode ser classificado como *global*, *local* ou *semi-global*. O alinhamento global inclui todos os resíduos das sequências. Já o alinhamento local permite a exclusão de prefixos e/ou sufixos, gerando assim uma subsequência. O alinhamento semi-global permite a exclusão de somente prefixos ou somente sufixos das sequências.

2. **Tipo de sequência**. Uma sequência biológica pode ser formada por sequências de bases nitrogenadas, que formam DNA ou RNA; ou sequências de aminoácidos, que formam proteínas.

3. **Similaridade entre os resíduos**. A partir de um método estatístico, é gerada uma matriz de substituição que indica os valores de similaridade entre um par de resíduos alinhados [18]. Uma matriz $20x20$ armazena os valores de todas as combinações de pares de aminoácidos, no caso de comparação de proteínas. A matriz BLOSUM62 é um exemplo de matriz de

substituição [19].

4. **Penalidades por gap**. *Gaps* são lacunas que representam a inserção ou remoção de um resíduo como resultado de um processo de evolução, portanto, recebem uma penalização. Há dois modelos de penalidade por *gap*. O primeiro, e mais simples, é o modelo *linear gap*. Nesse modelo um *gap* de tamanho k é penalizado por $\gamma(k) = -kG$, onde G é uma constante. O segundo modelo é chamado *affine gap*, no qual o *gap* é penalizado por $\gamma(k) = -G_{first} - (k-1)G_{ext}$, onde G_{first} é o custo de abertura de gap e G_{ext} é o custo de expansão de gap [12].

Na literatura são encontrados diversos algoritmos para encontrar um alinhamento ótimo de sequências biológicas. Esses algoritmos usam programação dinâmica, e são muito importantes para a análise de sequências. Na seção 2.1 é mostrado o algoritmo de Needleman-Wunsch. Na seçao 2.2 é detalhado o algoritmo de Smith-Waterman. A seção 2.3 apresenta o algoritmo de Gotoh e a seção 2.4 descreve o algoritmo de Myers& Miller.

2.1 Needleman-Wunsh (NW)

O algoritmo de Needleman-Wunsch (NW) [20] é utilizado para ser obter o melhor alinhamento global entre duas sequências, permitindo-se uso de gaps para melhorar o alinhamento.

O alinhamento global ótimo é obtido a partir de duas etapas: (1) construção da matriz de programação dinâmica H indexada pelos índices i e j de cada sequência; e (2) *traceback* para construir o alinhamento propriamente dito.

2.1.1 Etapa 1 - Construção da matriz H

A matriz H armazena os escores máximos do alinhamento de proteínas, na qual cada elemento $H_{i,j}$ é o máximo escore do alinhamento entre os segmentos $x_{1...i}$ e $y_{1...j}$. O elemento $H(0,0)$ é igual a zero por definição. Os elementos da primeira coluna e linha da matriz são definidos a partir do custo linear de alinhamento com *gap* d, onde $H(i,0) = -id$ e $H(0,j) = -jd$. O restante da matriz é calculado recursivamente a partir de uma equação de recorrência;

Para a construção da matriz, o elemento $H(i,j)$ pode ser obtido de três maneiras:

1. $H(i,j) = H(i-1,j-1) + s(x_i, y_i)$ caso x_i e y_j estejam alinhados, onde $s(x_i, y_j)$ é um escore na matriz de substituição; ou a pontuação de $match/mismatch$ caso x_i e y_j sejam caracteres iguais/diferentes;
2. $H(i,j) = H(i-1,j) - d$ caso x_i esteja alinhado com um *gap*;
3. $H(i,j) = H(i,j-1) - d$ caso y_j esteja alinhado com um *gap*.

Dessa maneira, a equação de recorrência do algoritmo NW é dada pela equação (2.1).

$$H(i,j) = max \begin{cases} H(i-1,j-1) + s(x_i, y_i), \\ H(i-1,j) - d, \\ H(i,j-1) - d. \end{cases} \tag{2.1}$$

2.1.2 Etapa 2 - Traceback

Para cada elemento $H(i,j)$, guarda-se um ponteiro para a sua origem, indicando-se assim a partir de qual elemento da matriz $H(i,j)$ é derivado. A partir do final da matriz H, segue-se os ponteiros de forma a construir o alinhamento propriamente dito. Para cada elemento $H(i,j)$, determina-se o alinhamento de acordo com a origem daquele termo. Caso o elemento tenha vindo de $H(i-1,j-1)$, alinha-se x_i e y_j. Caso seja derivado de $H(i-1,j)$, alinha-se x_i com um gap. Caso seja derivado de $H(i,j-1)$, alinha-se y_j com um gap. Esse procedimento é realizado até que $i = j = 0$.

No algoritmo NW, são armazenados $(i+1)*(j+1)$ escores, que compõem a matriz H. Portanto, o algoritmo possui complexidade $O(nm)$ para tempo e espaço de memória, onde n e m são os tamanhos das sequências [20]. Como n e m são geralmente muito próximos, dizemos que o algoritmo possui complexidade $O(n^2)$. Quanto maior o valor de n, mais demorada é a comparação de sequências biológicas mais longas.

2.1.3 Exemplo

Para ilustrar o funcionamento do algoritmo NW, foi realizado o alinhamento de duas sequências $S_1 = GGCTGATCGA$ e $S_2 = ATTCGAG$. A penalidade de gap linear foi definida como $d = -1$ e as pontuações de *match* e *mismatch* foram definidas como $ma = +1$ e $mi = -1$, respectivamente. A figura 2.1 mostra a matriz de programação dinâmica H preenchida, onde em cada elemento da matriz observa-se, em vermelho o escore $s(x_i, y_j)$, no centro o valor daquele elemento e as setas que indicam a origem do elemento. As células em destaque indicam o *traceback*, de onde obtem-se o alinhamento global ótimo.

2.2 Smith-Waterman

O algoritmo de Smith-Waterman (SW) [17] é uma adaptação do NW utilizada para se obter o alinhamento local entre sequências, isto é, o alinhamento ótimo entre subsequências. A importância da obtenção do alinhamento de tais subsequências vem do fato de que somente algumas pequenas partes de longas sequências protéicas ou de DNA podem ter preservado similaridade, enquanto o restante das sequências não é mais similar.

Há duas principais diferenças entre os algoritmos NW e SW. A primeira é a equação de recorrência, que introduz o valor *zero*, para evitar valores negativos de $H(i,j)$. O valor de uma célula $H(i,j) = 0$ corresponde ao início de um novo alinhamento. Sendo assim, é preferível iniciar um novo alinhamento a introduzir um escore negativo no alinhamento. Portanto, a equação de

D(i,j)		G	G	C	T	G	A	T	C	G	A
	0	-1	-2	-3	-4	-5	-6	-7	-8	-9	-10
A	-1	-1	-2	-3	-4	-5	-4	-5	-6	-7	-8
T	-2	-2	-2	-3	-2	-3	-4	-3	-4	-5	-6
T	-3	-3	-3	-3	-2	-3	-4	-3	-4	-5	-6
C	-4	-4	-4	-2	-3	-3	-4	-4	-2	-3	-4
G	-5	-3	-3	-3	-3	-2	-3	-4	-3	-1	-2
A	-6	-4	-4	-4	-4	-3	-1	-2	-3	-2	0
G	-7	-5	-3	-4	-5	-3	-2	-2	-3	-2	-1

Alinhamento Global Ótimo

```
--AT--TCGAG
   |  ||||
GGCTGATCGA-
```

Figura 2.1: Exemplo de alinhamento utilizando o algoritmo de Needleman-Wunsch.

recorrência de SW é dada pela equação (2.2). Por causa dessa modificação, a primeira linha e coluna são inicializadas com zero.

$$H(i,j) = max \begin{cases} 0, \\ H(i-1,j-1) + s(x_i, y_i), \\ H(i-1,j) - d, \\ H(i,j-1) - d. \end{cases} \tag{2.2}$$

A segunda diferença é o *traceback*. O *traceback* é iniciado a partir da posição (i,j) onde $H(i,j)$ é o maior valor encontrado na matriz de programação dinâmica. Ao contrário de NW, realiza-se o *traceback* da mesma forma que em NW até que se encontre um valor $H(i,j) = 0$. Esse valor indica que o alinhamento local está terminado e é o melhor alinhamento, desconsiderando-se os demais. A figura 2.2 mostra o melhor alinhamento local entre as mesmas sequências do exemplo da figura 2.1 utilizando o algoritmo de SW.

2.3 Gotoh

Na equação (2.1) utilizamos o modelo de *linear gap*, no qual o custo de *gap* é um múltiplo do comprimento da sequência de *gaps*. Dessa forma, a penalidade para o início de um *gap* é a mesma que a penalidade para a expansão do *gap*. Tal modelo não modela bem o que ocorre na natureza pois a probabilidade de ocorrer um *gap* após a ocorrência de outro *gap* é alta [12]. Então, Gotoh ajusta a equação de recorrência de NW incluindo uma função genérica $\gamma(k)$, que é função do comprimento. Esse ajuste faz com que a complexidade de tempo seja $O(n^3)$ [21].

D(i,j)		G	G	C	T	G	A	T	C	G	A
	0	0	0	0	0	0	0	0	0	0	0
A	0	0	0	0	0	0	1	0	0	0	1
T	0	0	0	0	1	0	0	2	1	0	0
T	0	0	0	0	1	0	0	1	1	0	0
C	0	0	0	1	0	0	0	0	2	1	0
G	0	1	1	0	0	1	0	0	1	3	2
A	0	0	0	0	0	0	2	1	0	2	4
G	0	1	1	0	0	1	1	1	0	1	3

Alinhamento Local Ótimo

```
ATTCGA
|| |||
AT-CGA
```

Figura 2.2: Exemplo de alinhamento utilizando o algoritmo de Smith-Waterman.

$$H(i,j) = max \begin{cases} H(i-1,j-1) + s(x_i, y_i), & \\ H(k,j) + \gamma(i-k), & k = 0, ..., i-1, \\ H(i,k) + \gamma(j-k), & k = 0, ..., j-1. \end{cases} \tag{2.3}$$

Para tornar a programação dinâmica novamente $O(n^2)$ utiliza-se a função genérica $\gamma(k)$ na forma $\gamma(k) = -G_{first} - (k-1)G_{ext}$. Chama-se o modelo de *affine gap*, onde G_{first} é a penalidade de início de *gap* e G_{ext} é a penalidade para extensão do *gap*, e definimos $E(i,j)$ o maior escore dado que x_i está alinhado com um *gap* e $F(i,j)$ o maior escore dado que y_j está alinhado com um *gap*. A equação de recorrência relacionada à equação (2.3) agora pode ser escrita conforme as equações (2.4), (2.5) e (2.6):

$$H(i,j) = max \begin{cases} H(i-1,j-1) + s(x_i, y_i), \\ E(i-1,j-1) + s(x_i, y_i), \\ F(i-1,j-1) + s(x_i, y_i). \end{cases} \tag{2.4}$$

$$E(i,j) = max \begin{cases} H(i-1,j) - G_{first}, \\ E(i-1,j) - G_{ext}. \end{cases} \tag{2.5}$$

$$F(i,j) = max \begin{cases} H(i,j-1) - G_{first}, \\ F(i,j-1) - G_{ext}. \end{cases} \tag{2.6}$$

2.4 Myers & Miller

Os algoritmos NW, SW e Gotoh obtêm o escore em espaço linear, caso executem somente a etapa 1 do algoritmo (cálculo da matriz de programação dinâmica). No entanto, caso se deseje recuperar o alinhamento, a complexidade de espaço é quadrática. Logo, esses algoritmos possuem complexidade de espaço elevada e se tornam inviáveis para sequências biológicas muito longas, fato que se observa principalmente no tamanho da memória necessária para armazenar a matriz de programação dinâmica. Por exemplo, uma matriz para duas sequências com um milhão de bases ocuparia 4 *terabytes* de memória. Com o intuito de tornar a obtenção do alinhamento ótimo de sequências muito longas viável, foram desenvolvidos alguns algoritmos que reduzem a complexidade de espaço para $O(m+n)$, que anteriormente era $O(n^2)$, tornando o algoritmo linear em relação ao espaço.

Hirschberg [22] desenvolveu um algoritmo para solucionar o problema da Maior Subsequência Comum (LCS - *Longest Common Subsequence*) em espaço linear. Myers & Miller [23] adaptaram o algoritmo de Hirschberg para tornar o algoritmo de Gotoh linear em relação ao espaço.

O algoritmo de Myers & Miller [23] é dividido em duas partes, que são executadas de maneira recursiva até obter-se o alinhamento ótimo. O objetivo é encontrar, inicialmente, um ponto médio por onde passa um alinhamento ótimo. Na primeira parte, o alinhamento é feito por colunas até a coluna central $\frac{n}{2}$. Na segunda parte, realiza-se o alinhamento das duas sequências invertidas S'_0 e S'_1 até a mesma coluna $\frac{n}{2}$. São armazenados quatro vetores durante o processamento:

- *CC* — Armazena o escore dos alinhamentos que terminam em *match* ou *mismatch*.
- *DD* — Armazena o escore dos alinhamentos que terminam em *gap*.
- *CC'* — Análogo ao vetor *CC*, armazena o processamento das sequências invertidas.
- *DD'* — Análogo ao vetor *DD*, armazena o processamento das sequências invertidas.

O escore $K_i = min\{CC_i + CC'_i, DD_i + DD'_i - G_{open}\}$ é atribuído ao alinhamento que atravessa a coluna $\frac{n}{2}$ através da linha i, onde G_{open} é a penalidade de abertura de *gap* ($G_{open} = G_{first} - G_{ext}$). Dessa forma, o escore K_i* é definido como $K_i* = max\{K_i\}, \forall\ 0 \leq i < m$. De forma recursiva, realiza-se o processamento novamente dividindo o tamanho da seção em tamanhos menores. A figura 2.3 ilustra o processamento recursivo do algoritmo de Myers & Miller.

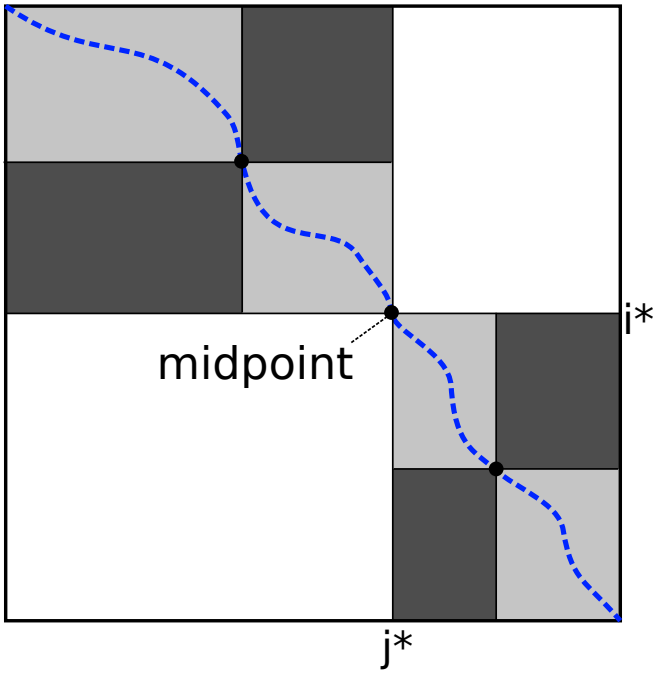

Figura 2.3: Processamento recursivo do algoritmo de Myers & Miller [7].

Capítulo 3

Comparação Paralela de Sequências Biológicas em FPGA

3.1 FPGA

As FPGAs (*Field Programmable Gate Arrays* são circuitos integrados em larga escala que, diferentemente dos ASICs (*Application Specific Integrated Circuits*), podem ser reconfiguráveis após sua fabricação [24]. Essa é uma grande vantagem em relação aos hardwares específicos que após terem seus projetos modificados são necessariamente descartados. Desse modo, após o projeto do novo circuito com modificações necessárias, é necessário fabricá-lo e testá-lo.

As FPGAs são compostos por elementos de computação que contém elementos lógicos configuráveis, interconexões e registradores. Esses blocos geralmente são dispostos na configuração de um *grid*, dessa forma permitindo que interconexões sejam realizadas para transmitir informação de um bloco para um ou mais blocos. Os elementos lógicos realizam operações lógicas básicas que são necessárias no projeto de um *hardware* específico. Algumas FPGAs possuem elementos lógicos mais complexos que realizam operações de multiplicação, normalmente usadas em aplicações de processamento de sinais, ou até mesmo estruturas responsáveis por operações de entrada e saída. Outros elementos também são comuns às FPGAs:

- (Tabelas de *Lookup*): As LUTs (*Lookup-Tables*) são tabelas que implementam funções lógicas predefinidas a partir de uma combinação de entradas. Elas provêm uma maneira rápida de se obter o *output* de uma função lógica, já que operações de acesso a memória são mais rápidas que o cálculo dessa operação lógica. As LUTs geralmente são associadas a registradores do tipo *flip-flop* e sua implementação varia de fabricante para fabricante.
- (Elementos de computação): Os elementos de computação são estruturas que implementam funções lógicas simples. São geralmente compostos por registradores e elementos lógicos de baixo nível, como *lookup-tables*.
- (Memória): Os dispositivos atuais possuem blocos de memória internos, seja do tipo SRAM

ou de outro tipo. Os blocos de memória podem ser utilizados por todos os elementos de computação ou pode-se definir grupos de elementos que têm acesso à memória.

- (Elementos de Interconexão): Os elementos de interconexão possibilitam a interligação entre os elementos de processamento, memória e outros elementos da FPGA, como os elementos de entrada e saída. Esses elementos podem se organizar de forma a interligar tanto elementos de computação próximos como interligar blocos distantes e a memória da FPGA.
- (Elementos de E/S): Os elementos de entrada e saída (E/S) normalmente são elementos de borda que permitem a compatibilidade entre os blocos internos da FPGA e a interface da placa de FPGA. Os blocos de entrada e saída gerenciam a comunicação entre a placa de FPGA e um computador.

A figura 3.1 [8] mostra a organização comum de uma FPGA, onde os elementos de computação estão organizados em forma de *grid* e são separados pelos elementos de interconexão. A figura 3.2 [8] mostra um elemento de computação típico de uma FPGA, composto por uma *Lookup-Table* de 4 entradas e um registrador do tipo *flip-flop*.

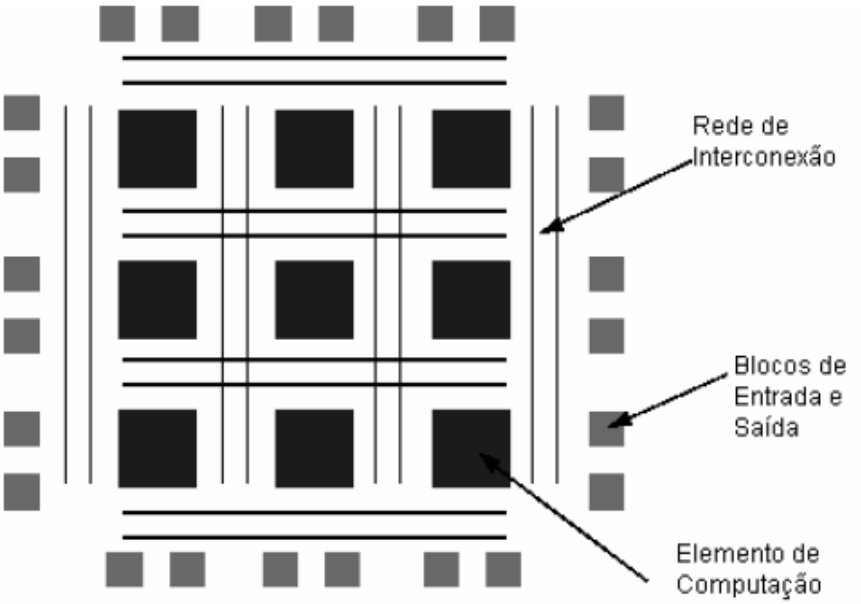

Figura 3.1: Organização típica de uma FPGA [8].

Umas das maneiras utilizadas para realizar a síntese de hardware reconfigurável é a utilização de linguagens de descrição de hardware (HDL). Essas linguagens permitem que um circuito seja descrito na forma de módulos, cada um contendo uma descrição do seu comportamento ou estrutura. Diferentemente do fluxo sequencial de execução de um software, o fluxo de processamento de um hardware descrito em linguagem HDL é determinado pela reação dos módulos aos sinais e estímulos gerados pelos demais módulos. Um exemplo seria um módulo que realiza a soma de dois sinais de entrada e gera um sinal de saída. O sinal de saída reage conforme os sinais de entrada mudam. Uma ferramenta utiliza a descrição de hardware para gerar um arquivo de configuração

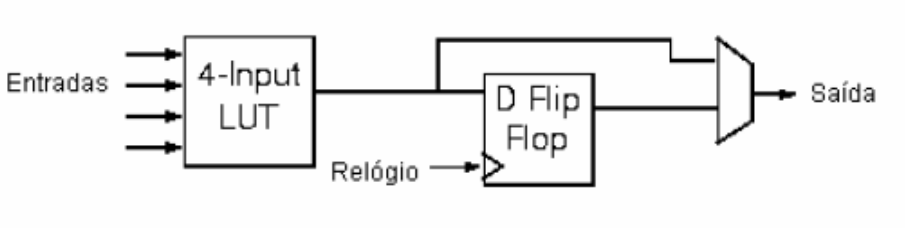

Figura 3.2: Elemento de computação [8].

da FPGA correspondente. Dessa forma, uma vez configurada a FPGA, essa passa a se comportar como o hardware descrito inicialmente. As linguagens de descrição de hardware mais populares são VHDL e Verilog.

As linguagens HDL disponibilizam diversos recursos para a modelagem de hardware. Esses recursos podem levar a abordagens diferentes. Existem basicamente duas abordagens distintas para a especificação do circuito [25]: (a) Modelagem comportamental e (b) Modelagem estrutural.

Na modelagem comportamental o projetista define o comportamento desejado do circuito utilizando um nível maior de abstração e uma ferramenta de síntese decompõe aquele comportamento em um circuito físico. Nesse modelo podem ser utilizadas estruturas de decisão e repetição para modelar o comportamento, assim como em linguagens de programação de alto nível. Os circuitos descritos de forma comportamental nem sempre são sintetizáveis, isto é, nem sempre é possível gerar um circuito a partir de uma descrição comportamental somente.

Na modelagem estrutural o circuito é definido em termos dos seus módulos e das conexões entre os módulos. Cada módulo, ou componente básico, é definido e pode ser reutilizado várias vezes, assim novos módulos podem ser definidos utilizando componentes previamente definidos. Nessa modelagem é possível definir hierarquias que simplificam o projeto do *hardware* desejado.

3.2 Abordagens em FPGA para comparação de sequências

Na literatura existem diversas propostas baseadas em FPGA para a aceleração da comparação de pares de sequências utilizando o método *diagonal wavefront* DW [9]. Nos algoritmos de NW e SW cada célula (i, j) depende somente das células $(i-1, j)$, $(i-1, j-1)$ e $(i, j-1)$, portanto pode-se observar que a anti-diagonal da matriz pode ser calculada paralelamente. Primeiramente, a célula $(1, 1)$ é calculada. Após, as células $(1, 2)$ e $(2, 1)$ podem ser calculadas em paralelo. O processamento segue dessa forma até o término do cálculo da matriz . Esse método é chamado de *diagonal wavefront*. A figura 3.3 [9] mostra o fluxo de computação do método DW.

Para quantificar o desempenho de cada implementação utiliza-se a unidade CUPS, que significa atualizações de células por segundo. Uma atualização significa o cálculo de uma célula da matriz. A maioria das implementações atuais atingem a ordem de milhões de atualizações de células por segundo, portanto é incluído na unidade o prefixo *Giga*. Assim a unidade utilizada se torna GCUPS. A performance é calculada pelo tamanho da matriz dividido pelo tempo necessário para

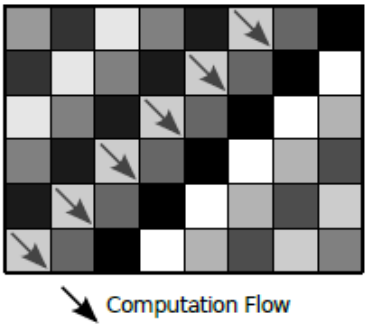

Figura 3.3: Método *diagonal wavefront* [9].

calculá-la $(mxn)/(tempo)$.

As propostas encontradas na literatura que implementam o método de DW em FPGAs utilizam um *array* sistólico [26] com N elementos de processamento (PE). Cada PE é capaz de realizar a comparação de uma célula da matriz por turno. Sendo assim, quando todos os PEs estão realizando processamento, o array é capaz de atualizar N células da matriz a cada turno. A figura 3.4 mostra o exemplo de um *array* sistólico de PEs. No Tempo $T0$ os elementos de processamento já possuem os elementos de uma sequência armazenados. A cada espaço de tempo T os elementos da outra sequência são deslocados e a comparação é feita com o elemento presente no PE de acordo com as relações de recorrência do algoritmo escolhido. Nas subseções 3.2.1 a 3.2.6 serão mostradas algumas propostas que utilizam *arrays* sistólicos para implementar o método DW.

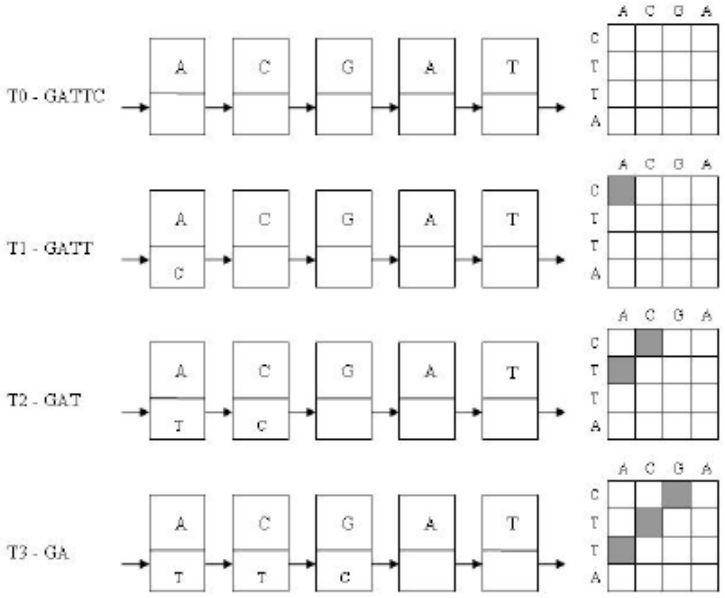

Figura 3.4: Exemplo de *Array* sistólico de 5 elementos [2].

3.2.1 Hoang and Lopresti (1992) [1]

Hoang and Lopresti [1] propuseram uma implementação bidirecional do *array* sistólico para calcular o escore e o alinhamento usando o algoritmo de distância de edição [27]. Essa foi uma das primeiras implementações em FPGA para o cálculo de matrizes de programação dinâmica. Foram obtidos dez mil alinhamentos de sequências de DNA utilizando sequências de tamanho $n = 100$ e 32 FPGAs *Xilinx XC3090*. Utilizando 248 PEs obteve-se o desempenho de 0.27 GCUPS.

3.2.2 Carvalho (2003) [2]

Essa proposta implementa um *array* sistólico unidirecional com elementos de processamento (PEs) compactos, de modo que um grande número de PEs sejam colocados na FPGA. A proposta assume a existência de uma memória externa, onde as células $H(i, j)$ da matriz de programação dinâmica são colocadas. Os PEs usam quantidades de bits diferentes para armazenar o escore, ou seja, PEs de menor ordem usam menos *bits*. Essa proposta passou somente por uma simulação funcional, não havendo dados a respeito do desempenho em FPGA.

3.2.3 Oliver et al. (2005) [3]

Oliver [3] propôs um *array* sistólico unidirecional de PEs para executar uma comparação local utilizando *gap linear* e *affine gap*, obtendo somente o escore ótimo. Para alternar entre o modelo de *gap linear* e *affine gap* é necessário reconfigurar a FPGA. Os PEs foram posicionados na FPGA de forma "zig-zag" para otimizar a espaço ocupado. A arquitetura foi escrita em Verilog e sintetizada para uma FPGA *Virtex II XC2V6000*. Os desempenhos obtidos para sequências de tamanho $n = 2016$ foram de 10.6 GCUPS e 5.8 GCUPS com *gap linear* e *affine gap*, respectivamente.

3.2.4 Zhang et al. (2007) [4]

Zhang [4] propôs um *array* sistólico unidirecional para executar o algoritmo Smith-Waterman para sequências de proteínas e DNA utilizando *gap linear* e *affine gap*. Nessa implementação obtêm-se somente o escore. A arquitetura proposta foi sintetizada na plataforma XD1000, que contém uma FPGA Altera Stratix II. Foram usados 384 elementos de processamento ligados a um *clock* de 66.7 MHz. Foi obtido um desempenho de 25.6 GCUPS comparando duas sequências de $65,526$ aminoácidos.

3.2.5 Caffarena et al. (2007) [5]

Caffarena [5] também propôs um *array* sistólico unidirecional que usa aritmética de 1 bit para executar o algoritmo de distância de edição com *gap linear* para sequências de DNA. Nessa comparação, foram comparadas sequências de 11000 nucleotídeos em uma FPGA *Xilinx XC2V6000-5* operando a 100 MHz. Foi obtido um desempenho de 156 GCUPS

Sec	Ano	Problema					Algoritmo		Solução HPC		
		Seq	Tamanho	Tipo	Gap	Out	Algo	Complex Mem	#PU	Paral	Perf GCUPS
3.2.1	1992	DNA	10^3	global	linear	score align	edit	quad	32	fine (DW)	0.27
3.2.2	2003	DNA	10^1	local	linear	matriz	SW quad	linear	1	fine (DW)	só síntese
3.2.3	2005	Prot	10^4	local	linear affine	score	SW Gotoh	linear	1	fine (DW)	10.26
3.2.4	2007	DNA prot	10^5	local	linear affine	score	SW Gotoh	linear	1	fine (DW)	25.60
3.2.5	2007	DNA	10^4	global	linear	score	edit	linear	1	fine (DW)	156.00
3.2.6	2013	DNA prot	10^3	local	linear	score	SW	linear	128	coarse fine (DW)	3040.00

Tabela 3.1: Comparação entre implementações em FPGA

3.2.6 L. Wienbrandt (2013) [6]

Wienbrandt [6] utilizou a plataforma *Rivyera S3-5000* para acelerar a execução do algoritmo SW com *gap linear*. A plataforma é composta por 128 FPGAs *Xilinx Spartan3-5000* que são conectadas por um barramento de alta performance organizado em um *array* sistólico. A máquina hospedeira, responsável por conectar as FPGAs por meio de um barramento PCIe, possui um processador *Intel i7* e $12GB$ de memória RAM.

O autor propôs um *array* sistólico unidirecional para comparar sequências de DNA ou de proteínas. Para as sequências de proteínas foi utilizada a matriz de substituição BLOSUM62 e para as sequências de DNA foi utilizada a matriz de substituição NUC44. Nessa implementação obtêm-se somente o escore. Cada FPGA é capaz de realizar quatro comparações de DNA simultaneamente. Portanto o sistema é capaz de realizar 512 comparações simultaneamente. Foi obtido um desempenho de 3.04 TCUPS.

3.3 Quadro comparativo

Na tabela 3.1 é apresentado um comparativo entre as diversas abordagens em FPGA apresentadas na seção 3.2. Percebe-se que as abordagens permitem a comparação de sequências de DNA e de proteínas. O tamanho máximo das sequências aumentou de 10^1 em [2] para 10^5 em 2007 [4] e depois foi reduzido para 10^3 em 2013 [6].

Nota-se também o aumento significativo da performance, em GCUPS, saltando de 0.27 GCUPS em [1] para 3040.00 em [6]. Tal aumenta se deve principalmente à evolução da tecnologia das FPGAs ao longo de 21 anos e de projetos extremamente sofisticados e que permitem o uso de várias FPGAs em paralelo.

Capítulo 4

Plataforma XD2000i^{TM}

A plataforma *XtremeData* XD2000i$^{\text{TM}}$[10] foi a plataforma escolhida para a implementação do algoritmo *Swith-Waterman* (seção 2.2). Algumas das características da plataforma são:

- Placa mãe *Dual XEON*;
- Módulo XD2000i^{TM} 1067M FSB contendo uma FPGA *Bridge* e duas FPGAs de aplicação Altera Stratix III EP3SE260F1152C3 (*appA* e *appB*), com 255,000 elementos lógicos e 203,000 registradores;
- Sistema Operacional Linux CentOS

A figura 4.1 mostra a arquitetura básica da plataforma. Como pode ser visto, está incluso um módulo programável XD2000i, que possui comunicação com o processador por meio do *Memory Controller Hub* (MCH). Estão inclusos, também um *mouse*, teclado e monitor. O sistema usado para desenvolvimento roda a ferramenta Quartus II e não está incluso.

Nas seções 4.1 a 4.5 são apresentados os recursos fornecidos pela plataforma XD2000i^{TM} e os passos para configurar a plataforma e enviar e receber dados de um *hardware* descrito em linguagem HDL.

4.1 *Design* de referência

Juntamente à plataforma é disponibilizada uma arquitetura de referência para o teste das funcionalidades da máquina. Na arquitetura de referência se encontra um projeto na ferramenta ALTERA Quartus II contendo os arquivos fonte em VHDL e arquivos de configuração dos módulos da plataforma XD2000i^{TM}. Esse projeto é usado como base para o desenvolvimento de módulos dos usuários nas FPGAs de aplicação.

Na figura 4.2 pode-se ver o fluxo de dados do sistema de uma forma geral. O bloco denominado AFU, dentro de cada uma das FPGAs de aplicação, denota *Accelerator Functional Unit*. Esse módulo é a porção de sistema definida pelo usuário e especifica a funcionalidade da FPGA de

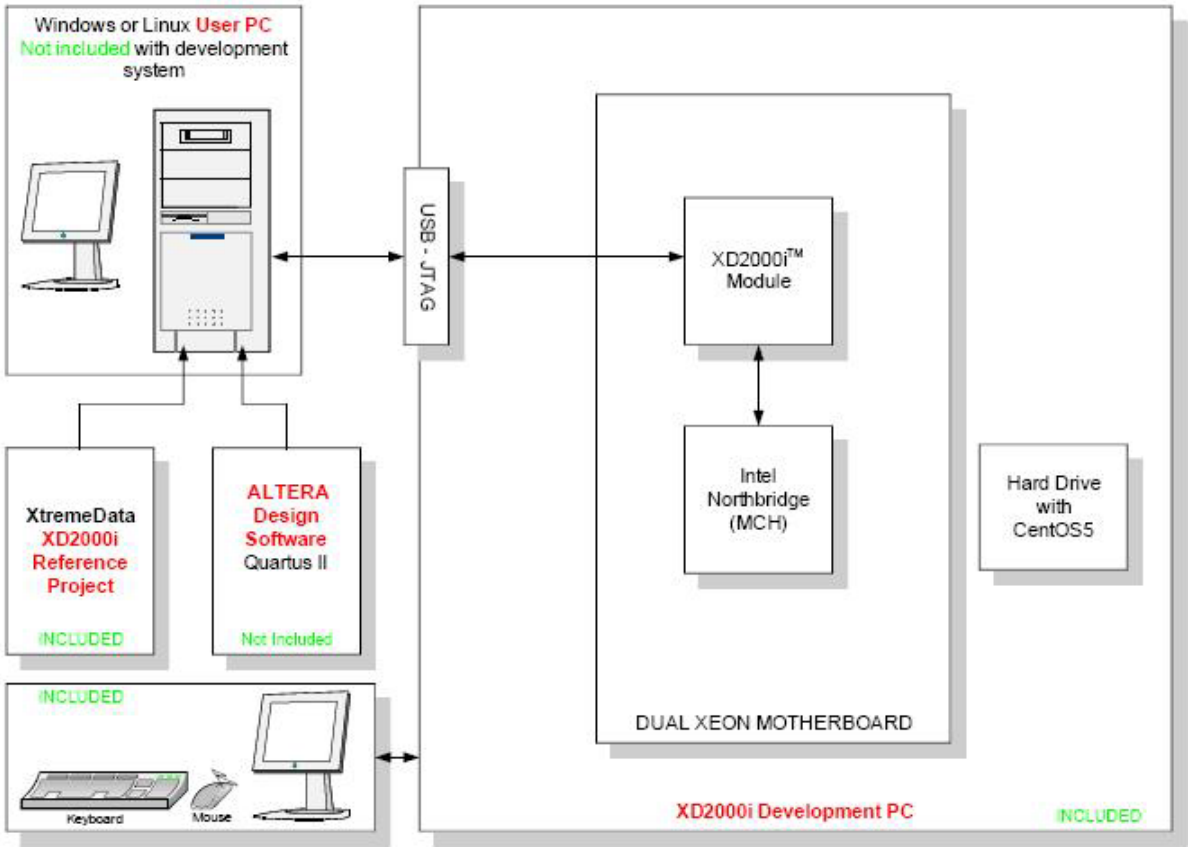

Figura 4.1: Arquitetura básica da plataforma XD2000i^{TM}[10].

aplicação, ou seja, a implementação explícita do algoritmo desenvolvido pelo usuário. O fluxo de dados se dá em um modelo mestre-escravo. Todos os dados são transferidos entre a memória RAM e a AFU através da interface entre a FPGA de aplicação e a FPGA *Bridge* e da interface entre a FPGA *bridge* e o controlador de memória (MCH), denominada *Front Side Bus* (FSB).

A figura 4.3 mostra a AFU de múltiplas interfaces implementada pelo *design* de referência.

4.2 *Workspace*

Uma *workspace* é uma porção da memória principal do sistema (RAM) que pode ser usada como fonte ou destino de transferências ao módulo em hardware. O endereçamento físico é realizado pelo FSB, enquanto o módulo do usuário somente enxerga o fluxo de dados de entrada e saída. Assim, o módulo é visivel a nível de sistema operacional como um dispositivo no diretório /dev/fap. Desse modo, o acesso à FPGA para transferência de dados se dá com a obtenção do descritor de arquivo do dispositivo e alocação do espaço de memória a ser utilizado. Comandos do tipo *workspace* são utilizados para enviar e receber dados do módulo e devem especificar o endereço de memória da fonte, tamanho da fonte, endereço de memória do destino e tamanho do destino. Na implementação utilizada, baseada no *design* de referência, uma chamada é feita e permanece bloqueada até que os dados da resposta sejam recebidos.

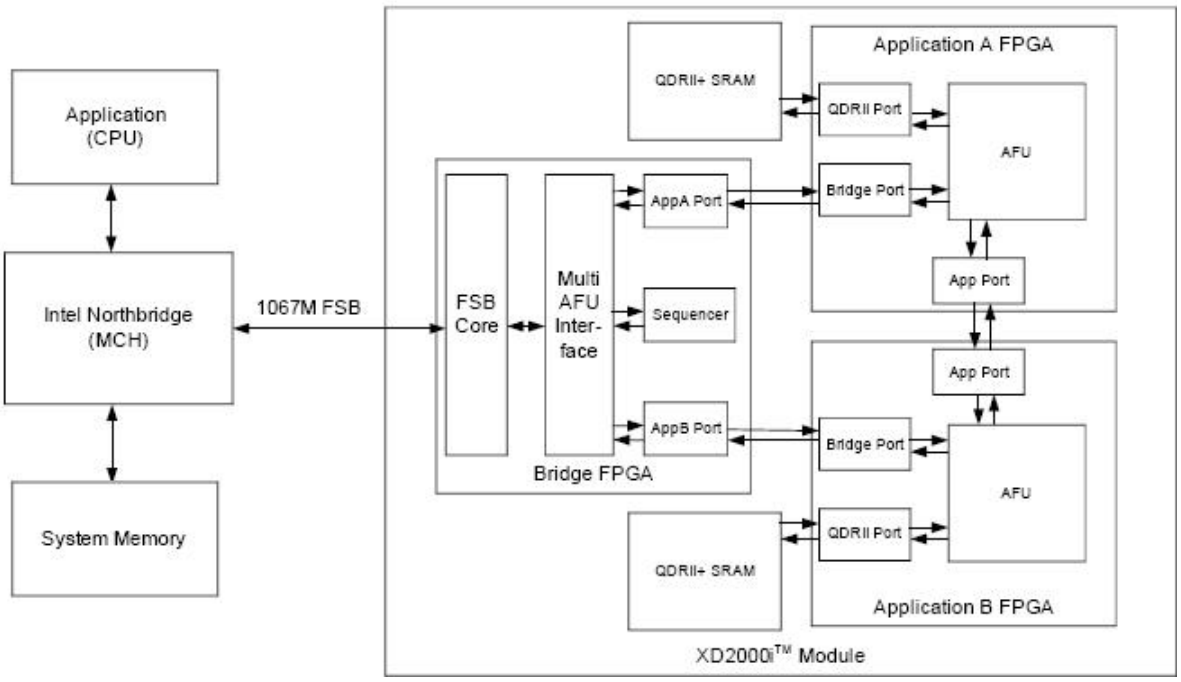

Figura 4.2: Fluxo de dados na plataforma.

Algumas restrições são aplicadas ao módulo XD2000i^{TM}[10]:

- Os dados enviados e recebidos através de um comando do tipo *workspace* deve possuir tamnho múltiplo de 64 bytes;
- Os *buffers* de entrada e saída devem ter pelo menos 64 bytes;
- Os *buffers* de entrada e saída podem ter tamanhos distintos;
- Um *workspace* pode ter no máximo 4MB;
- Pode-se alocar no máximo 4096 *workspaces*;
- Apenas uma operação de envio de dados está ativa ao mesmo tempo. O recebimento deve ser completado antes da próxima chamada de envio;
- Os comandos são processados de forma síncrona (envio seguido imediatamente de um recebimento).

4.3 FPGA *Bridge*

A FPGA *Bridge* é a interface entre o barramento FSB e o módulo AFU. A sua arquitetura é fornecida pela plataforma e não é customizável pelo usuário.

Um arquivo de extensão .pof possui a configuração da FPGA *Bridge* e é carregado automaticamente em uma memória *flash* durante a inicialização do sistema. A FPGA *Bridge* é mostrada na figura 4.2. Os principais componentes da FPGA *bridge* são:

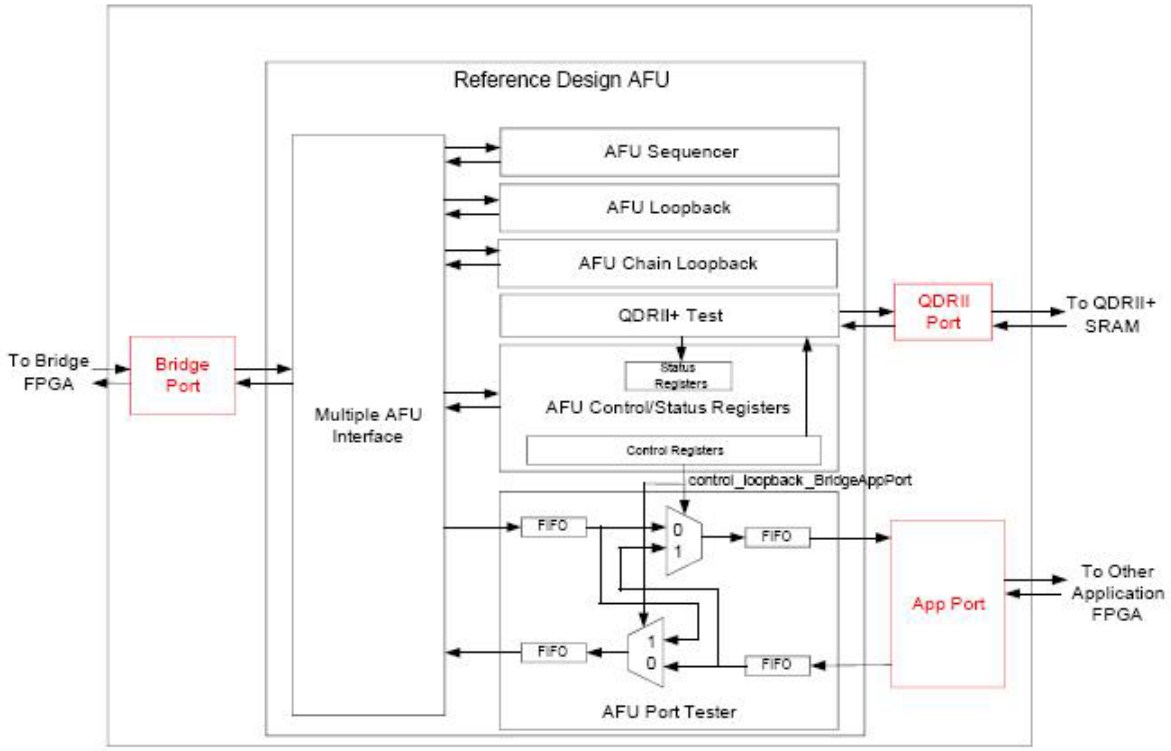

Figura 4.3: AFU de múltiplas interfaces.

- Núcleo FSB
- Interface para múltiplas AFUs
- Portas de interface para as FPGAs de aplicação appA e appB

O núcleo FSB, ou *northbridge*, é conectado ao controlador de memória e transfere o fluxo de dados às múltiplas AFUs. De acordo com o cabeçalho recebido, os dados são direcionados à interface correta, sendo elas possivelmente as FPGAs de aplicação *appA* e *appB* ou o sequenciador de testes.

4.4 Protocolo *FIFO* da AFU

As FPGAs de aplicação se comunicam com a FPGA *bridge* através um protocolo *Data/Valid/Ready* (DVR). Esse protocolo introduz um *handshake* ponto-a-ponto entre as FPGAs para a transferência de dados. Portanto, são estabelecidas algumas regras para que a transferência aconteça. São transferidos 256 bits a cada leitura ou escrita.

No *design* de referência, o arquivo *fifo_ afudrv.vhd* define as possíveis interfaces FIFOs. Há dois tipos de interfaces FIFO: *AFU-Type* e *DRV-Type*.

4.4.1 Interface FIFO *AFU-Type*

Nessa interface, o alvo deve aceitar dados sempre que <initiator>_data_valid está em nível lógico alto. O controle do fluxo de dados ocorre requerindo que o *initiator* só transmita dados quando percebe que <target>_data_ready está em nível lógico alto. De maneira simplificada: sempre que uma unidade anuncia que seus dados a serem transmitidos são válidos, a outra unidade deve aceitar.

As regras do protocolo *AFU-Type* são:

1. *Initiator* pode setar <initiator>_data_valid quando seus dados são válidos e percebe que <target>_data_ready está setado;
2. *Initiator* deve setar em nível lógico baixo <initiator>_data_valid quando percebe que <targer>_data_ready está baixo;
3. *Target* deve aceitar dados sempre que percebe que <initiator>_data_valid está setado;
4. *Target* deve aceitar no mínimo 16 transferências após setar em nível baixo <target>_data_ready para permitir a conclusão de possíveis transferências.

4.4.2 Interface FIFO *DRV-Type*

Nessa interface, os dados são transferidos quando <initiator>_data_valid e <target>_data_ready estão em nível alto concorrentemente. Assim, a AFU pode controlar o fluxo de dados de entrada eliminando a necessidade de um *buffer* adicional para armazenar os dados de entrada.

As regras do protocolo *DRV-Type* são:

1. *Initiator* seta <initiator>_data_valid quando seus dados estão válidos;
2. *Target* seta <target>_data_ready quando está pronto para receber dados;
3. A transferência ocorre somente quando <initiator>_data_valid e <target>_data_ready estão setados.

4.5 Configuração da plataforma

4.5.1 Estabelecimento de comunicação com a FPGA *bridge*

A cada inicialização do sistema uma sequência de passos deve ser executada para abrir a comunicação com a *bridge* e, posteriormente, programar a FPGA de aplicação com o *hardware* do usuário:

4.5.1.1 Habilitar o segmento AHM FSB

O módulo XD2000i^{TM} ocupa o *socket* secundário de CPU. Alguns sistemas desabilitam o segmento FSB secundário se uma CPU não é detectada no *socket* durante a inicialização do sistema. Para sistemas com placa mãe SuperMicro Rack Mount DP X7DGT-SG007, um *script* tcl deve ser executado para habilitar a interface FSB do módulo. Tais sistemas são identificados pelo campo "'IN02"' no número serial de fabricação.

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

4.5.1.2 Carregar o *driver* do dispositivo

O *driver* do dispositivo do módulo AFU possui dois componentes: **fappip_afu_07.ko**, que implementa a interface de baixo nível com o *hardware* e **fapdrv_afu_07.ko**, que implementa a interface de aplicação. Após o carregamento dos *drivers*, o módulo é acessado através do arquivo /dev/fap. Os seguintes comandos são usados para carregar os *drivers*:

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

Para verificar que os *drivers* foram carregados com sucesso deve-se olhar o log do sistema.

```
$ dmesg | grep fap
```

A figura 4.4 mostra o *log* após carregar os *drivers* com sucesso.

```
fappip: status = 0x00000000
fappip: FSB Accelerator Hardware Module - PIP Module, v0.75
fappip: Copyright(c) 2007-2008 Intel Corporation
fapdrv: FSB Accelerator Hardware Module - Device Driver, v0.75
fapdrv: Copyright(c) 2007-2008 Intel Corporation
```

Figura 4.4: *Log* após carregar *drivers*.

4.5.1.3 Programar a FPGA de aplicação

O programa executável ahmAppDownload programa uma ou mais FPGAs de aplicação com um arquivo de extensão .rbf. Esse arquivo é denominado *Raw Binary File* e é gerado pelo *software* Quartus II após a síntese da descrição de *hardware* em linguagem HDL.

```
ahmAppDownload usage:
-h [ -help ] Print this help message
-version Print program version
-a [ -rbfPathA ] arg AppA rbf path to download
```

```
-b [ -rbfPathB ] arg AppB rbf path to download
-A [ -loadAppA ] arg Load AppA fpga
-B [ -loadAppB ] arg Load AppB fpga
-t [ -debugPrintThreshold ] arg (=0) Debug print level
```

O comando a seguir programa a FPGA de aplicação appA com o arquivo swlc20_appA.rbf:

```
$ ./ahmAppDownload -A -a swlc20_appA.rbf
```

A figura 4.5 mostra os resultados após carregar da FPGA com sucesso.

```
ahmFsbAppDownload.cpp - Line 235 <> LogInfo <> "App Download Complete" = App Download Complete <>    = 2ba52fa46de2h
ahmFsbAppDownload.cpp - Line 239 <> LogInfo <> elapsedTimeUs = 721631 <>    = 00000b02dfh
cancelTimeout()
ahmFsbAppDownload.cpp - Line 254 <> LogTestStatus <> thisTestStatusIsOk = true <>    = 0000000001h
ahmLogger.h - Line 282 <> LogDelete <> Delete Logger =  <>    = 2ba52fa4680ch
downloadAppAStatusIsOk = true
All runningStatusIsOk == true
```

Figura 4.5: Resultados após carregar FPGA appA.

Capítulo 5

Projeto do algoritmo de comparação de sequências

Esse capítulo apresenta a estratégia de comparação de sequências com o algoritmo SW na plataforma XD2000i^{TM}. Optou-se por utilizar uma solução já existente em VHDL [2] e adaptá-la para retornar o escore ótimo como resultado. Além disso, foi projetada a integração com a plataforma XD2000i$^{\mathrm{TM}}$com componentes de *hardware* e *software*. Na seção 5.1 são apresentados os passos da metodologia empregada no projeto. Os passos da metodologia são detalhados nas seções 5.2 a 5.4.

5.1 Metodologia

A implementação do algoritmo de Smith-Waterman (SW) [17] na plataforma XD2000i^{TM} foi dividida em cinco etapas:

1. Teste da plataforma: nessa etapa, foi efetuado um teste simples de *loopback*, o qual envia um dado e o recebe em seguida;
2. Estudo da solução existente: a solução de Carvalho [2] foi estudada em detalhes;
3. Adaptação da solução existente: foi projetado um circuito e incluído na solução de [2];
4. Simulação funcional: foi feita a simulação na ferramenta Quartus II Simulator [28];
5. Integração com a plataforma: nessa etapa foram projetados os componentes de *hardware* e *software* responsáveis pela interface com o *host*;

5.2 Teste da plataforma

Após realizar a configuração da plataforma, descrita na seção 4.5, foi realizado um teste simples para verificação do funcionamento da mesma. Esse teste simples consistiu em gerar um *hardware* no

Quartus II 8.1 [11] que recebe dados de entrada e repassa esses dados para a saída. Tal *hardware* implementa um teste de *loopback* para verificar o funcionamento da plataforma. O *hardware* sintetizado foi gravado na FPGA appA (figura 4.2). Um programa em linguagem C++ foi então escrito utilizando as classes do *software* de referência descrito na seção 5.7 . Na execução desse programa, um conjunto de *bits* foi enviado e os mesmos *bits* foram recebidos no vetor de destino. Confirmou-se, então, o funcionamento correto do *hardware* de *loopback* configurado na plataforma.

5.3 Solução existente

A solução proposta por Carvalho [2] implementa o algoritmo proposto por Smith-Waterman [17] utilizando a estrutura sistólica linear unidirecional mostrada na figura 3.4. A estrutura, composta de inúmeros elementos de processamento (PEs), permite a redução da complexidade de tempo para $O(n+m)$, deixando de ser quadrática. Isso é possível por meio de operações simultâneas paralelas.

A cada instante de tempo, um determinado PE deve possuir três informações para a atualização da matriz de similaridades em uma posição $H(i,j)$: os valores $H(i-1,j)$, $H(i,j-1)$ e $H(i-1,j-1)$. O valor da coluna à esquerda, $H(i,j-1)$, é proveniente do PE à esquerda. Dessa maneira, o PE precisa armazenar somente os valores da linha superior, $(i-1,j)$, e da diagonal, $H(i-1,j-1)$. A figura 5.1 [2] mostra a estrutura de um elemento de processamento. A cada ciclo de *clock*, o PE recebe uma nova base da sequência de banco de dados e o valor da célula à esquerda (c). Os valores da diagonal (a) e da linha superior (b) foram armazenados na transição de *clock* que deslocou a nova base de banco de dados e o valor (c). Nesse momento o PE possui todas as informações necessárias para o cálculo da equação de recorrência. Para fins de implementação, as bases foram codificadas utilizando 2 bits. As bases A, T, C e G são codificadas como 00, 01, 10 e 11, respectivamente.

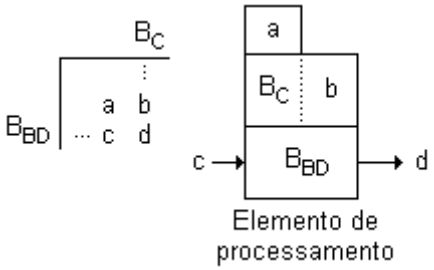

Figura 5.1: Estrutura de um elemento de processamento.

Relembrando a equação de recorrência 2.2 temos a equação 5.1:

$$H(i,j) = max \begin{cases} 0, \\ H(i-1,j-1) + s(x_i, y_i), \\ H(i-1,j) - d, \\ H(i,j-1) - d. \end{cases} \tag{5.1}$$

A equação 5.1 foi dividida em três etapas para o cálculo do valor de saída: cálculo do valor

$H(i-1, j-1) + s(x_i, y_i)$, cálculos dos valores $H(i-1, j) - d$ e $H(i, j-1) - d$ e comparação com zero. As duas primeiras etapas são realizadas simultaneamente dentro do PE.

Na primeira etapa, o valor da diagonal é somado com as pontuações de *match* e *mismatch*, 1 e -1, respectivamente. Os resultados entram em um multiplexador que tem como seletor o resultado da comparação entre as bases das sequências de consulta e bando de dados. O valor calculado é armazenado temporariamente em $RES1$. A figura 5.2 [2] mostra o circuito combinacional responsável pelo cálculo.

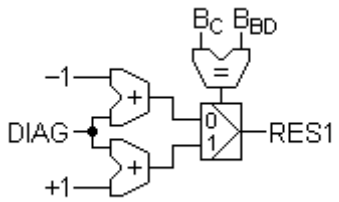

Figura 5.2: Circuito combinacional para cálculo de *match* ou *mismatch* [2].

Na segunda etapa, realizada simultaneamente com primeira, os valores da linha superior e coluna à esquerda são somados à pontuação de *gap*, -2, e entram em um multiplexador. Esse multiplexador tem como seletor a comparação entre os valores da linha superior e coluna à esquerda. Dessa maneira, a saída $RES2$ recebe o maior valor entre $LINHA - 2$ e $COLUNA - 2$. A figura 5.3 [2] mostra o circuito combinacional resultante da implementação da segunda etapa.

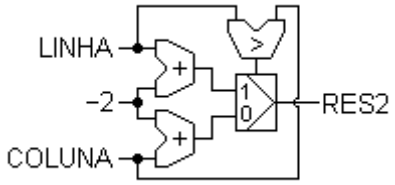

Figura 5.3: Circuito combinacional para cálculo de *gap* [2].

Na terceira etapa basta comparar os valores $RES1$ e $RES2$ com zero, e selecionar o maior deles. O valor resultante é o valor da equação de recorrência naquela posição.

A figura 5.4 mostra o circuito com cinco elementos de processamento após o processo de síntese no Quartus II. O circuito gerado apresenta como entrada os sinais *clk*, *rst*, *flag_ in* e *base_ in*. O sinal *flag_ in* indica que o sinal de entrada *base_ in* é válido naquele instante. O circuito implementado por [CARVALHO] [2] pode ser encapsulado, como mostrado na figura 5.5. Observa-se que não foi implementada uma unidade de controle que, a cada ciclo de *clock*, passa as bases das sequências de banco de dados, seta corretamente o sinal *flag_ in* e aguarda o final do processamento. O circuito original somente aceita uma base de entrada e uma *flag*. Sem uma unidade de controle, o circuito original não tem funcionalidade. Essa unidade de controle é proposta na seção 5.4.

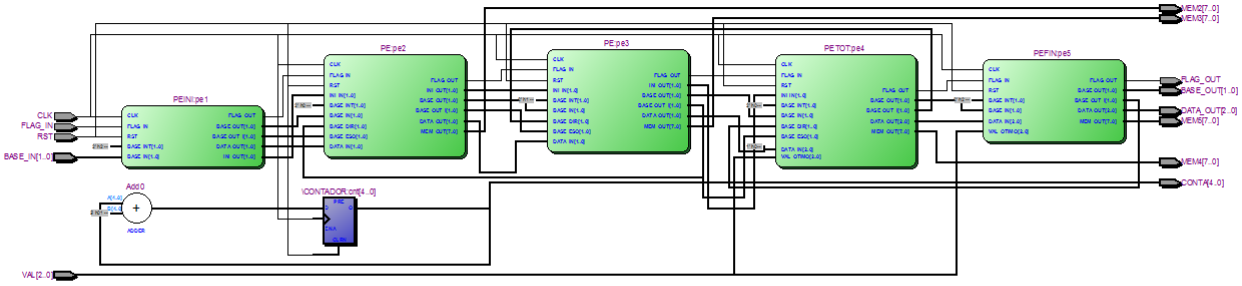

Figura 5.4: Circuito com cinco PEs.

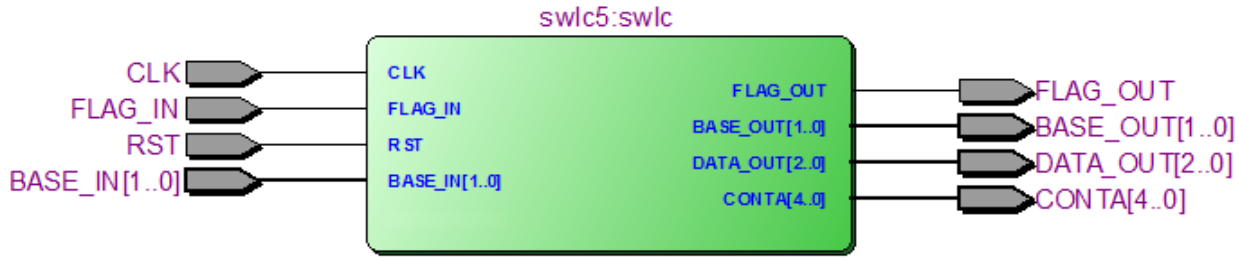

Figura 5.5: Circuito da figura 5.4 encapsulado.

5.4 Adaptação da solução existente

Com o objetivo de calcular o escore máximo local, foi proposta uma implementação que estendesse alguns aspectos da implementação apresentada na seção 5.3. Foram feitas duas extensões: (1) adição de um circuito no elemento de processamento para calcular o escore máximo local em cada PE; (2) implementação de uma unidade de controle que encapsula a solução proposta por [CARVALHO] [2].

5.4.1 Circuito adicional para cálculo de escore máximo

Um circuito adicional foi acrescentado na arquitetura da entidade que descreve o PE. A cada ciclo de *clock*, o PE recebe o escore máximo calculado pelo PE à esquerda até aquele momento. Após calcular seu novo escore, o PE compara esse valor com o escore cálculado no instante de tempo anterior. O maior desses valores é comparado com o valor que foi recebido do PE à esquerda. O novo escore máximo é passado ao PE à direita. A figura 5.6 mostra a estrutura sequencial para o cálculo do escore máximo.

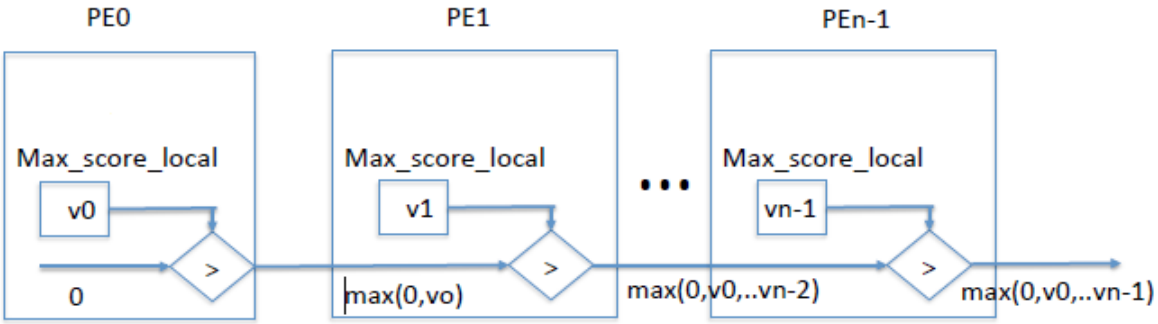

Figura 5.6: Estrutura sequencial proposta para cálculo do escore máximo.

5.4.2 Unidade de controle

A unidade de controle implementa uma máquina de estados finita que controla o recebimento de dados, o repasse de bases e sinais de controle para o circuito proposto por [CARVALHO] [2] e o envio do escore máximo. Para que haja comunicação com a FPGA *bridge*, as entradas e saídas da unidade de controle se dão aos moldes do protocolo FIFO descrito na seção 4.4. O sinal *data_in_valid* indica que o dado está presente na entrada *data_in*. Os dados são lidos e armazenados em um vetor e a máquina de estados vai do estado inicial *Read_data* para o estado de *Execute*. Nesse estado, a cada ciclo de *clock*, os dois *bits* menos significativos do vetor, que representam uma base, entram no circuito proposto. Uma operação de *shift right* lógico é feita no vetor para que a próxima base esteja nos dois *bits* menos significativos. Em um *array* sistólico unidirecional com n PEs, são geralmente necessários, no melhor caso, $m + n$ ciclos de *clock* para comparar uma sequência de tamanho m com uma sequência de tamanho n. Portanto, no momento em que o primeiro PE recebe a primeira base da sequência de banco de dados, são necessários $m + n$ ciclos de *clock* até que o último PE produza em sua saída o escore máximo local resultante da comparação das duas sequências. Dessa forma, o sinal *flag_in*, que indica a entrada de uma base válida, deve ser setado durante m ciclos de *clock*, para que todas as bases da sequência de banco de dados entrem nos PEs, e a máquina de estados permaneça no estado *Execute* por $m + n$ ciclos de *clock*. Quando o sinal *flag_out* retorna ao nível lógico baixo, todos os sinais *flag_in* setados percorreram os PEs e há a indicação de que o escore máximo local para as sequências está cálculado. A máquina de estados transiciona para o estado *Write_data* e o sinal *data_out* recebe o escore máximo local. O sinal *data_out_valid* é setado para indicar à FPGA *bridge* que o dado de saída é válido. A máquina de estado retorna imediatamente para o estado *Read_data* para aguardar o recebimento de novos dados. A máquina de estados descrita pode ser visualizada na figura 5.7.

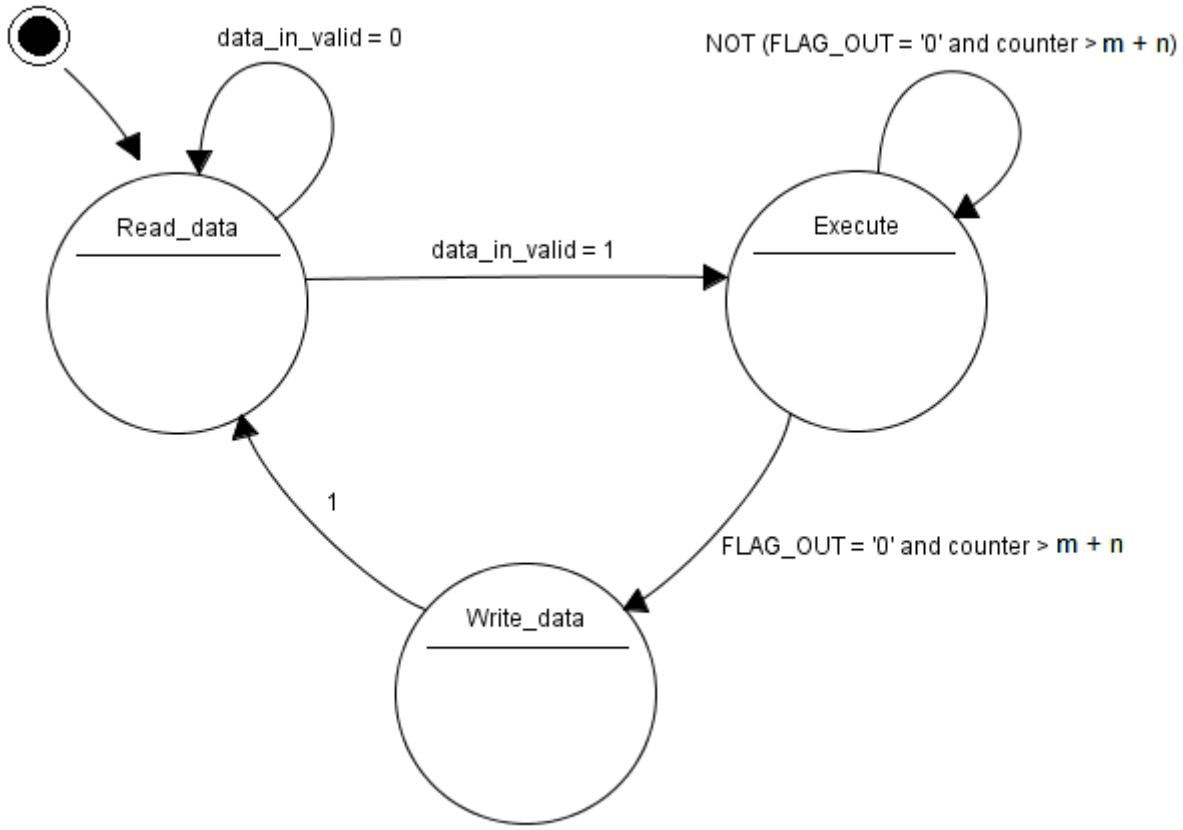

Figura 5.7: Máquina de estados da unidade de controle.

5.5 Simulação funcional

Devido à simplicidade do modelo de entrada e saída de dados da arquitetura desenvolvida, optou-se por realizar a simulação funcional no próprio Quartus II 8.1 [11] com o Quartus II 64-Bit Simulator [28], ao invés de gerar um *testbench* em VHDL e simular com o *Modelsim* [29], padrão geralmente adotado em simulações funcionais. Um arquivo de extensão .vwf, *Vector Waveform File*, é criado para gerar os sinais de entrada que estimularão o módulo em teste. Ao executar a simulação, os sinais de entrada estimulam o módulo em teste e esse produz os sinais de saída. A figura 5.8 mostra o gráfico com os sinais de entrada e saída. Os sinais *clk, reset, data_in, data_in_valid* e *data_out_ready* são sinais de entrada produzidos pelo usuário ao criar a simulação. Os sinais *data_in_ready, data_out* e *data_out_valid* são sinais de saída produzidos pelo módulo.

Na simulação mostrada na figura 5.8, o sinal *data_in*, de tamanho 256 bits, foi setado como $0x42712$. Esse valor convertido em binário equivale a 0000000000000000000001000010011100010010 e representa a sequência arbitrária de bases $CATAGTCAATAAAAAAAAAA$, observando a codificação apresentada na seção 5.3. Vale notar que embora sejam recebidos 256 bits, a arquitetura com 20 PEs tem como entrada sequências de tamanho 20, portanto os demais bits são ignorados.

Sequências com 20 bases utilizam 40 bits em sua codificação. É importante perceber também que a sequência de banco de dados é codificada de forma invertida, para que a primeira base a entrar no circuito esteja nos bits menos significativos e assim sucessivamente.

A sequência de consulta foi setada de forma *hard-wired* nos PEs. Para essa simulação, os PEs continham a sequência de consulta $CATAGTCAATCAGGTTAAGC$. É fácil ver que o algoritmo SW aplicado às duas sequências resulta no escore máximo igual a 10. No momento que o sinal de saída *data_out_valid* é setado, o sinal de saída *data_out* possui o escore máximo calculado pelo circuito. O escore máximo resultante da simulação também foi igual a 10.

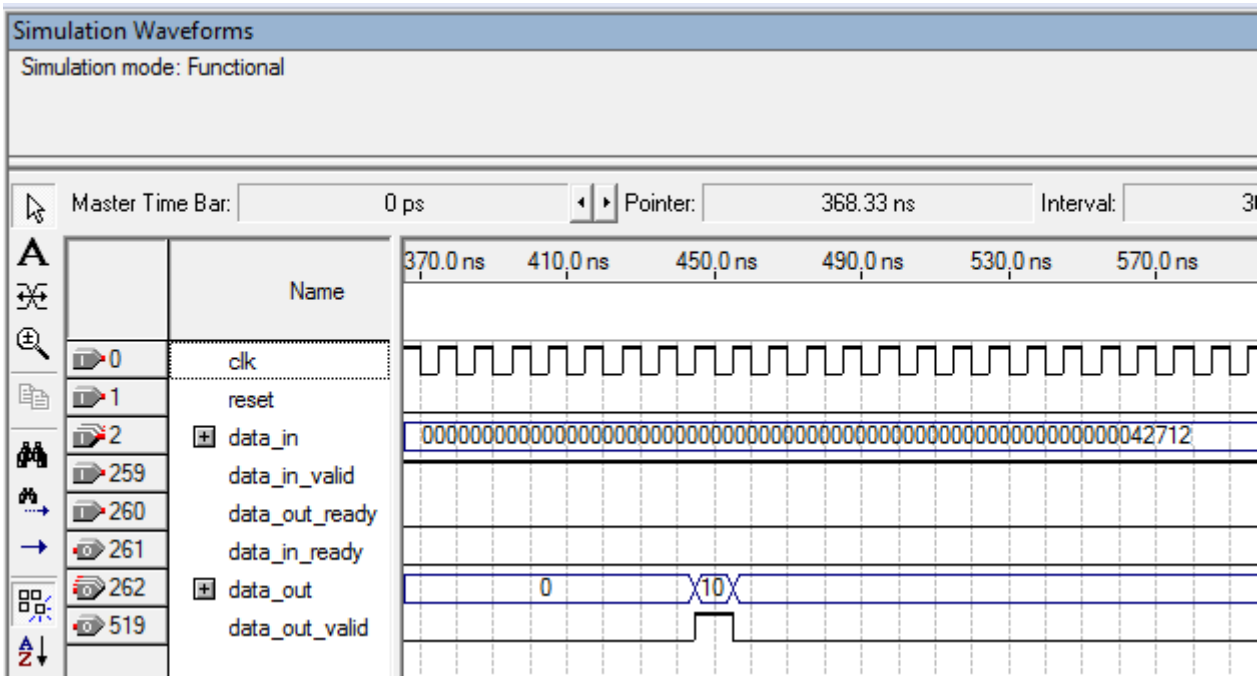

Figura 5.8: Gráfico *waveform* da simulação funcional.

5.6 Projeto de integração com a plataforma

A partir da arquitetura projetada na seção 5.4 e simulada na seção 5.5, é necessário que se integre o módulo gerado com o *design* de referência para que haja a comunicação entre a AFU e a FPGA *bridge* e, logo, com a CPU. O módulo desenvolvido possui entradas e saídas definidas de acordo com o protocolo FIFO mostrado na seção 4.4. A figura 5.9 mostra as entradas e saídas do módulo integrado. É essencial que o módulo respeite as regras de comunicação através do protocolo FIFO, isto é, sempre que quiser ler dados deve setar a saída *data_in_ready* e aguardar até que *data_in_valid* esteja setado para receber dados. De maneira similar, quando desejar escrever dados, deve setar *data_out_valid*. A figura 5.10 mostra o circuito gerado em nível RTL após a integração com as FIFOs de entrada e saída.

O módulo da figura 5.10 substitui a AFU de múltiplas interfaces presente no *design* de referência (figura 4.3). Esse módulo utiliza duas FIFOs para a comunicação com a FPGA *bridge*: uma do

tipo *AFU-Type* (seção 4.4.1, para receber dados, e outra do tipo *DRV-Type* (seção 4.4.2, para enviar dados;

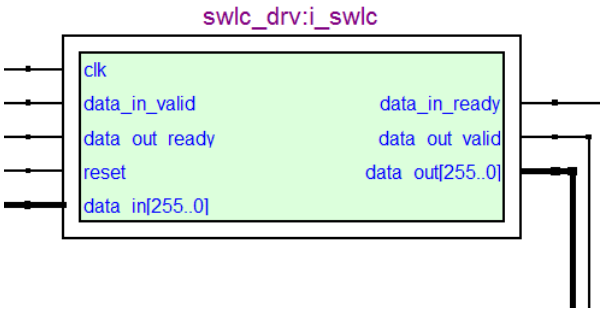

Figura 5.9: Entradas e saídas do módulo.

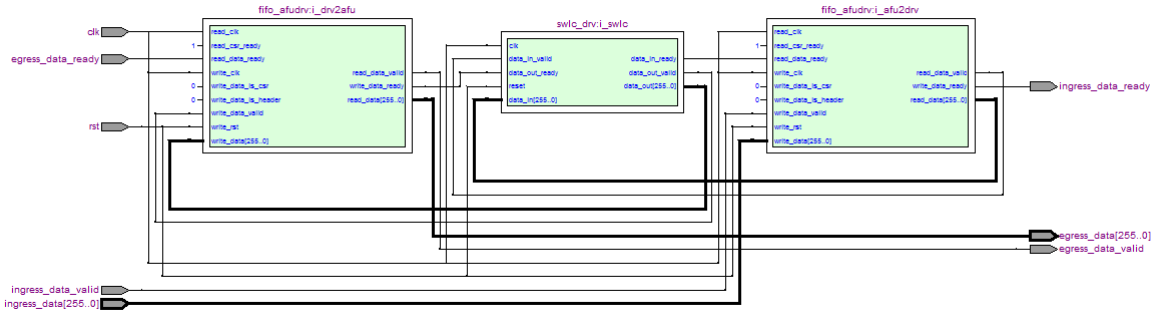

Figura 5.10: AFU customizada.

5.7 *Software* de envio/recepção de dados

A unidade de controle apresentada na seção 5.4.2 implementa uma máquina de estados. Após carregar a FPGA de aplicação com o arquivo .rbf (seção 4.5.1.3), a máquina de estados permanece no estado *Read_ data* enquanto não receber dados. É necessário informar à FPGA *bridge* o destino dos dados, no caso, a FPGA appA. A função setAfuSelector() seleciona a FPGA alvo e a quantidade de *bytes* enviados e recebidos, que devem ser múltiplos de 64 *bytes*.

Assim como o *design* de referência, são disponibilizados *softwares* de referência, em C++, que possibilitam a configuração das FPGAs e comunicação com o módulo *Accelerator Hardware Module* (AHM). O AHM é composto pela FPGA *bridge*, pelas duas FPGAs de aplicação e pelas memórias QDRII+ SRAM.

A classe AhmInstance permite a criação de *workspaces*, mencionado na seção 4.2, necessários para reservar áreas de memória que guardam os dados enviados e recebidos pelo aplicação. Vale notar que os parâmetros *src_ size* e *dst_ size*, mostrados a seguir, indicam o tamanho dos vetores de origem e destino em quantidades de 256 bits. Por exemplo, se o usuário quiser enviar um vetor de $uint64\_t$ contendo 8 posições, o tamanho desse vetor é de 512 bits, portanto $src\_size = 2$.

Para enviar/receber dados da FPGA de aplicação, uma instância *AhmInstance* é criada para abrir o dispositivo /dev/fap. A partir do descritor de arquivo obtido, são criados dois *workspaces* para definir áreas de memória de origem e destino.

```
AhmInstance ahm; // Open the ahm
int ahm_fd = ahm.get_fd(); // fd of the underlying ahm instance
AHM_Workspace srcWS(ahm_fd, src_size_in_bytes);
AHM_Workspace dstWS(ahm_fd, dst_size_in_bytes);
```

Para enviar e receber dados da FPGA, é necessário definir vetores de origem e destino relacionados às *workspaces* previamente declaradas. Esses vetores são passados como parâmetro para a função **start_sync** que envia os dados armazenados em *srcBuf* por meio de uma chamada de sistema *ioctl* e escreve o resultado em *dstBuf*. Esse função possui comportamento blocante, portanto o *software* de envio/recepção de dados permanece parado até que a função retorne e escreva os dados em *dstBuf*.

```
uint64_t* srcBuf = (uint64_t*)(srcWS.get_buf());
uint64_t* dstBuf = (uint64_t*)(dstWS.get_buf());
/* carregar srcBuf, definir srcCountBits256 e dstCountBits256 */
ahm.start_sync(srcBuf, srcCountBits256,
dstBuf, dstCountBits256);
```

Capítulo 6

Resultados

Tendo feito a adaptação do *design* de referência, substituindo as múltiplas AFUs pelo módulo AFU customizado pelo usuário, o circuito descrito em VHDL está pronto para o processo de síntese. O modelo de *timing* é atualizado a cada versão do Quartus II, portanto, por recomendação do fabricante da plataforma, foi utilizado o Quartus II 8.1 para se obter resultados corretos de *timing*.

Foram sintetizadas implementações com 20, 40, 80, 128 e 1024 elementos de processamento. Todas tiverem como alvo a FPGA appA, que é uma Stratix III EP3SE260F1152C3. Essa FPGA possui 255,000 elementos lógicos, 203,000 registradores e 14,688Kbits de memória interna. Foi utilizado um *clock* de 100MHz. A tabela 6.1 mostra os resultados da síntese para cada implementação com número de PEs variável. Observa-se o aumento da quantidade de recursos gastos com o aumento do número de PEs. A figura 6.1 mostra a síntese realizada para a implementação com 20 PEs. A implementação com 1024 PEs foi sintetizada com a mesma unidade de controle das demais, porém, é necessário a criação de mais estágios de controle para receber mais do que 256 *bits*, padrão nas outras implementações. Por esse motivo, a implementação com 1024 PEs não está funcional.

Para testar as implementações com número variável de PEs, foi utilizado o programa em C++ descrito na seção 5.7. Para todos os casos de teste, o vetor que contém as bases da sequência de banco de dados recebeu a sequêcia $CATAGTCAATCAAAAAAAAA$ para ser enviada ao módulo. A sequência de consulta, armazenada diretamente na instanciação dos PEs no código VHDL, foi definida como $CATAGTCAATCAGGTTAAGC$. A sequência de banco de dados possui as doze

PEs	ALUTs	Registradores dedicados	% de utilização	Frequência sintetizada	Pinos de I/O
20	1120	1452	<1%	77MHz	328
40	2185	2056	2%	77MHz	328
80	4990	3461	3%	77MHz	328
128	8423	4564	5%	77MHz	518
1024	96289	49507	61%	77MHz	518

Tabela 6.1: Resultados da síntese utilizando número variável de elementos de processamento

Flow Status	Successful - Sun Jan 24 09:43:13 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	< 1 %
Combinational ALUTs	1,120 / 203,520 (< 1 %)
Memory ALUTs	0 / 101,760 (0 %)
Dedicated logic registers	1,451 / 203,520 (< 1 %)
Total registers	1878
Total pins	328 / 744 (44 %)
Total virtual pins	0
Total block memory bits	229,376 / 15,040,512 (2 %)
DSP block 18-bit elements	0 / 768 (0 %)
Total PLLs	3 / 8 (38 %)
Total DLLs	0 / 4 (0 %)

Figura 6.1: Síntese no Quartus II [11] para 20 PEs.

primeiras bases iguais à sequência de consulta. Dessa forma, o escore resultante é igual a 12. As figuras 6.2 e 6.3 mostram um exemplo de resultados obtidos em um teste feito com uma sequência de doze bases iguais à sequência de consulta, e um exemplo com dez bases iguais.

```
**********************************************************************
*  Sequencia de consulta:                CATAGTCAATCAGGTTAAGC     *
*  Sequencia de banco de dados:          CATAGTCAATCAAAAAAAAA     *
*  Total de dados (bytes): 64                                     *
*  Tempo de execucao (us): 691                                    *
*  Taxa de transferencia (kbps): 94.842258                        *
*  Escore: 12                                                     *
*  Ciclos para execucao: 41                                       *
**********************************************************************
```

Figura 6.2: Exemplo de teste com sequência de 12 bases iguais à sequência de consulta.

Para cada caso de teste, foram computadas as métricas: tempo para a execução da função que envia/recebe dados (*start_ sync*), tempo para execução completa do programa (*wall-clock time*) e contagem de ciclos de *clock* até que seja produzido o escore máximo local válido pelo último PE. Para o tempo de execução da função, foram realizados cinco testes com entradas iguais e anotados os valores mínimo, médio e máximo. A tabela 6.2 apresenta os resultados obtidos. Como mostrado na seção 5.4.2, no melhor caso, são necessários $m + n$ ciclos de *clock* para comparar uma sequência de tamanho m com uma sequência de tamanho n. Em todos os testes, a sequência

```
**********************************************************************
*  Sequencia de consulta:                CATAGTCAATCAGGTTAAGC     *
*  Sequencia de banco de dados:          CATAGTCAATAAAAAAAAAA     *
*  Total de dados (bytes): 64                                     *
*  Tempo de execucao (us): 304                                    *
*  Taxa de transferencia (kbps): 215.578947                       *
*  Escore: 10                                                     *
*  Ciclos para execucao: 41                                       *
**********************************************************************
```

Figura 6.3: Exemplo de teste com sequência de 10 bases iguais à sequência de consulta.

PEs	Tempo (us)			Wall-clock (us)	Ciclos de clock
	Menor	Média	Maior		
20	76	311,4	634	3000	41
40	159	403	544	4000	61
80	80	215,2	488	4000	101
128	64	189	399	4000	149

Tabela 6.2: Resultados dos testes com 20, 40, 80 e 128 PEs.

de banco de dados possuiu tamanho igual a 20. A contagem de *clocks* apresentada na tabela é retornada pela implementação com um ciclo a mais do que o esperado porque a máquina de estados grava a contagem no estado de *Write_data*, após o escore ter sido cálculado. Como pode ser visto na tabela 6.2, houve uma variação muito grande no tempo de execução das comparações. Por exemplo, com 128 PEs, obtivemos 64us como menor tempo e 399us como maior tempo, em 5 comparações. Como a medida do tempo nesse caso envolveu tanto *software* como *hardware*, acreditamos que algum fator de *software* está causando esse efeito. Como pode ser visto na última coluna da tabela 6.2, os ciclos de clock de todas as execuções se mantiveram constantes. Para a execução da aplicação completa (*Wall-clock time*), o tempo variou entre 3ms e 4ms.

Capítulo 7

Conclusões e Trabalhos Futuros

O presente trabalho de graduação projetou, implementou e avaliou uma estratégia para o cálculo do escore máximo da comparação de sequências biológicas utilizando o algoritmo de Smith-Waterman. Foram propostas duas extensões do código proposto por Carvalho [2]: (1) circuito para cálculo de escore máximo; e (2) unidade de controle que encapsula o circuito de Carvalho [2].

A estratégia projetada foi descrita em VHDL e sintetizada utilizando a ferramenta Quartus II [11] para a FPGA de aplicação Stratix III EP3SE260F1152C3, que integra a plataforma XD2000i™. Após a síntese, o módulo gerado foi carregado na FPGA alvo (appA). A comunicação entre a AFU e o host foi programada em C++ utilizando a API fornecida pela plataforma.

Os resultados coletados utilizando 20, 40, 80 e 128 PEs mostram que a solução executa em tempo bastante reduzido, atendendo os requisitos do projeto.

Como trabalhos futuros, sugerimos:

- Implementação, simulação e síntese de uma unidade de controle com mais estágios de leitura de dados, juntamente com o programa para comunicação com o *host*, para aceitar sequências com tamanho maior que 128 bases, possivelmente podendo ter qualquer tamanho.
- Adaptação do módulo para receber uma linha e uma coluna de uma matriz de programação dinâmica, executar o algoritmo de Smith-Waterman e ser capaz de fornecer a última linha e coluna da matriz. Esse adaptação visa a integração com plataformas heterogêneas para que o trabalho do cálculo seja paralelizado entre as mais distintas plataformas.
- Produção de um manual completo para desenvolvimento na plataforma XD2000i, desde os requisitos de implementações de *hardware* aos detalhes da programação usando a API da plataforma, visando incentivar o uso da plataforma pelos alunos do departamento de ciência da computação.

REFERÊNCIAS BIBLIOGRÁFICAS

[1] HOANG, D. T.; LOPRESTI, D. P. Fpl implementation of systolic sequence alignment. p. 183–191, 1992.

[2] CARVALHO, L. G. A. *Uma Abordagem em Hardware para Algoritmos de Comparação de Sequências Baseados em Programação Dinâmica*. Departamento de Ciência da Computação, Universidade de Brasília, Brasília, DF: [s.n.], Dezembro 2003. Dissertação de Mestrado.

[3] OLIVER, T.; SCHMIDT, B.; MASKELL, D. L. Reconfigurable architectures for bio-sequence database scanning on fpgas. *IEEE Trans on Circuits and Systems*, v. 52, n. 12, p. 851–855, 2005.

[4] ZHANG, P.; G, T.; GAO, G. R. Implementation of the smith-waterman algorithm on a reconfigurable supercomputing platform. In: *ACM HPRCTA*. [S.l.]: ACM, 2007. p. 39–48.

[5] CAFFARENA, G. et al. Fpga acceleration for dna sequence alignment. *Journal of Systems and Software*, v. 16, n. 2, p. 245–266, 2007.

[6] WIENBRANDT, L. Bioinformatics applications on the fpga-based high-performance computer rivyera. In: *High-Performance Computing Using FPGAs*. [S.l.]: Springer, 2013. p. 81–103.

[7] SANDES, E. F. O. *Comparação Paralela de Sequências Biológicas Longas utilizando Unidades de Processamento Gráfico (GPUs)*. 2011. 94 p.

[8] CORRÊA, J. M. *Arquiteturas em FPGA para Comparação de Sequências Biológicas em Espaço Linear*. 138 p. Tese (Doutorado) — Departamento de Engenharia Elétrica, Universidade de Brasília, Brasília, DF, 2008.

[9] SANDES, E. F. O.; BOUKERCHE, A.; MELO, A. C. M. Parallel optimal pairwise biological sequence comparison: Algorithms, platforms and classification. *ACM Compution Surveys*, 2016. Accepted.

[10] XTREMEDATA, I. *XD2000iTM Development System User Handbook*. Junho 2009.

[11] ALTERA. *Quartus II Handbook Version 81*. Acesso em: 23 jan. 2016. Disponível em: <https://www.altera.com/en_US/pdfs/literature/hb/qts/archives/quartusii_handbook_8.1.pdf>.

[12] DURBIN, R. *Biological Sequence Analysis: Probabilistic Models of Proteins and Nucleic Acids*. Cambridge University Press, 1998. (Biological Sequence Analysis: Probabalistic Models of Proteins and Nucleic Acids). ISBN 9780521620413. Disponível em: <https://books.google.com.br/books?id=R5P2GlJvigQC>.

[13] FARRAR, M. Striped Smith-Waterman speeds database searches six times over other SIMD implementations. *Bioinformatics*, v. 23, n. 2, p. 156–161, January 2007.

[14] INO, F.; MUNEKAWA, Y.; HAGIHARA, K. Sequence homology search using fine grained cycle sharing of idle GPUs. *IEEE Trans. Parallel Distrib. Syst*, v. 23, n. 4, p. 751–759, 2012. Disponível em: <http://doi.ieeecomputersociety.org/10.1109/TPDS.2011.239>.

[15] WIRAWAN, A. et al. High performance protein sequence database scanning on the cell broadband engine. *Scientific Computing*, v. 17, p. 97–111, 2009.

[16] WANG, L. et al. Xsw: Accelerating biological database search on xeon phi. In: *IEEE IPDPSW - AsHES*. [S.l.: s.n.], 2014. p. 950–957.

[17] SMITH, T. F.; WATERMAN, M. S. Identification of common molecular subsequences. *J Mol Biol*, p. 195–197, March 1981.

[18] MOUNT, D. W. *Bioinformatics: sequence and genome analysis*. [S.l.]: Cold Spring Harbor Laboratory Press, 2004. ISBN 9780879697129.

[19] EDDY, S. R. Where did the blosum62 alignment score matrix come from? *Nature Biotechnology*, v. 22, p. 1035–6.

[20] NEEDLEMAN, S. B.; WUNSCH, C. D. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *J Mol Biol*, p. 443–453, March 1970.

[21] GOTOH, O. An improved algorithm for matching biological sequences. *J Mol Biol*, p. 705–708, December 1982.

[22] HIRSCHBERG, D. S. A linear space algorithm for computing maximal common subsequences. *Commun*, p. 341–343, 1975.

[23] MYERS, E. W.; MILLER, W. Optimal alignments in linear space. *Commun*, v. 4, n. 1, p. 11–17, 1988.

[24] GOMES, V. C. F.; CHARAO, A. S.; VELHO, H. F. C. *Field Programmable Gate Array - FPGA*. [S.l.: s.n.], 2014.

[25] PEDRONI, V. A. *Circuit design and simulation with VHDL*. [S.l.]: The MIT Press, 2010.

[26] KUNG, H. T. Why systolic architectures? *IEEE Computer*, v. 15, n. 1, p. 37–42, 1982.

[27] LEVENSHTEIN, V. I. Binary codes capable of correcting deletions, insertions and reversals. *Soviet Physics Doklady.*, v. 10, n. 8, p. 707–710, fev. 1966.

[28] ALTERA. *Quartus II Simulation Using VHDL Designs*. Acesso em: 25 jan. 2016. Disponível em: <ftp://ftp.altera.com/up/pub/Altera_Material/9.1/Tutorials/VHDL/Quartus_II_Simulation.pdf>.

[29] GRAPHICS, M. *ModelSim User Manual*. Acesso em: 23 jan. 2016. Disponível em: <http://portal.model.com/modelsim/resources/references/modelsim_user.pdf.>.