UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
PROGRAMA DE PÓS-GRADUAÇÃO EM MICROELETRÔNICA - PGMICRO

RAFAEL MENDES MALLMANN

Arquiteturas em Hardware para o Alinhamento Local de Sequências Biológicas

Dissertação apresentada como requisito parcial
para a obtenção do grau de Mestre em
Microeletrônica

Prof. Dr. Gilson Wirth
Orientador

Porto Alegre, maio de 2010

CIP – CATALOGAÇÃO NA PUBLICAÇÃO

Mallmann, Rafael Mendes

Arquiteturas em Hardware para o Alinhamento Local de Sequências Biológicas / Rafael Mendes Mallmann. – Porto Alegre: PGMICRO da UFRGS, 2010.

70 f.: il.

Dissertação (mestrado) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Microeletrônica - PGMICRO, Porto Alegre, BR–RS, 2010. Orientador: Gilson Wirth.

1. Smith-Waterman. 2. Distância Levenshtein. 3. Distância de edição. 4. Array sistólico. 5. Hardware dedicado. 6. Alinhamento local. 7. Programação dinâmica. 8. FPGA. 9. ASIC. 10. Comparação de genomas. I. Wirth, Gilson. II. Título.

UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
Reitor: Prof. Carlos Alexandre Netto
Vice-Reitor: Prof. Rui Vicente Oppermann
Pró-Reitor de Pós-Graduação: Prof. Aldo Bolten Lucion
Diretor do Instituto de Informática: Prof. Flávio Rech Wagner
Coordenador do PGMICRO: Prof. Ricardo Augusto da Luz Reis
Bibliotecária-chefe do Instituto de Informática: Beatriz Regina Bastos Haro

“All models are wrong, but some are useful.”
— GEORGE E. P. BOX

SUMÁRIO

LISTA DE ABREVIATURAS E SIGLAS 6

LISTA DE FIGURAS 7

LISTA DE TABELAS 8

RESUMO 9

ABSTRACT 10

1 INTRODUÇÃO 11

2 FUNDAMENTOS DE BIOLOGIA MOLECULAR E BIOINFORMÁTICA 14
2.1 Material Genético 14
2.2 Dogma Central da Biologia Molecular 15
2.3 Bancos de Dados Genéticos 16

3 ALGORITMOS PARA ALINHAMENTO DE SEQUÊNCIAS 18
3.1 Distância Levenshtein 19
3.2 Smith-Waterman 21
3.3 Alinhamento Múltiplo 23
3.4 Métodos Heurísticos 24

4 ARQUITETURAS ESTUDADAS 26
4.1 A Run-Time Reconfigurable System for Gene-Sequence Searching . . . 26
4.2 A Smith-Waterman Systolic Cell 27
4.3 An Efficient Digital Circuit for Implementing Sequence Alignment Algorithn in an Extended Processor 27
4.4 Biological Information Signal Processor 28
4.5 SWASAD: An ASIC Design for High Speed DNA Sequence Matching . 28
4.6 Hyper Customized Processors for Bio-Sequence Database Scanning on FPGAs 29
4.7 A Highly Parameterized and Efficient FPGA-Based Skeleton for Pairwise Biological Sequence Alignment 30
4.8 High-Speed Implementation of Smith-Waterman Algorithm for DNA Sequence Scanning in VLSI 30
4.9 Differential Scoring for Systolic Sequence Alignment 31
4.10 Acceleration of Smith-Waterman Using Recursive Variable Expansion . 32

4.11 Arquiteturas em FPGA para Comparação de Sequências Biológicas em Espaço Linear . . . 33
4.12 Families of FPGA-based Accelerators for Approximate String Matching 33
4.13 A Reconfigurable Accelerator for Smith-Waterman Algorithm 33

5 PROJETO EM HARDWARE PARA O ALINHAMENTO LOCAL DE SEQUÊNCIAS . . . 35
5.1 Matriz Dinâmica . . . 35
5.2 Particionamento de Sequências . . . 38
5.3 Protocolo de Comunicação . . . 40
5.4 Unidade de Processamento para a Distância Levenshtein . . . 42
5.5 Unidade de Processamento para o Smith-Waterman . . . 44

6 RESULTADOS E COMPARAÇÕES . . . 48
6.1 Prototipando para FPGA . . . 49
6.1.1 Utilizando o barramento PCI Express . . . 52
6.2 Prototipando para ASIC . . . 54
6.2.1 Front End . . . 54
6.2.2 Back End . . . 57
6.3 Comparações . . . 59

7 CONCLUSÕES . . . 65

REFERÊNCIAS . . . 67

LISTA DE ABREVIATURAS E SIGLAS

API	Application Programming Interface
ASIC	Application-Specific Integrated Circuit
BD	Base de Dados
BLAST	Basic Local Alignment Search Tool
BLOSUM	BLOcks of Amino Acid SUbstitution Matrix
BRAM	Block Random-access memory
CUDA	Compute Unified Device Architecture
CUPS	Cell Updates Per Second
DDBJ	Base de Dados de DNA do Japão
DMA	Acesso Direto a Memória
DNA	Ácido DesoxirriboNucleico
EMBL	Laboratório Europeu de Biologia Molecular
FPGA	Field-Programmable Gate Array
GPU	Graphics Processing Unit
HDL	Hardware Description Language
LUT	Look up Table
NCBI	National Center for Biotechnology Information
PAM	Point Accepted Mutation
PCI	Peripheral Component Interconnect
PCIe	Peripheral Component Interconnect Express
RAM	Random-access memory
RNA	Ácido Ribonucleico
SDC	Synopsys Design Constraints
VHDL	VHSIC hardware description language
VHSIC	Very-High-speed Integrated Circuit
UP	Unidade de Processamento

LISTA DE FIGURAS

Figura 2.1: Crescimento dos bancos de dados genético GenBank/EMBL/DDBJ (1995-2010) 17
Figura 2.2: Exemplo do formato FASTA 17

Figura 3.1: Matriz de substituição Blosum62 23

Figura 5.1: Deslocamento do vetor *wavefront* na matriz de similaridades 36
Figura 5.2: Princípios básicos de uso do array sistólico 36
Figura 5.3: Exemplo de como utilizar array sistólico para algoritmos de programação dinâmica 37
Figura 5.4: Diagrama de blocos da arquitetura para o particionamento de sequências 38
Figura 5.5: Exemplo da arquitetura projetada de particionamento de sequências . 40
Figura 5.6: Máquina de estados que controla a arquitetura projetada 41
Figura 5.7: *Datapath* da unidade de processamento para a distância Levenshtein . 44
Figura 5.8: Máquina de estados que controla o decremento e o incremento do registrador com a distância de edição 45
Figura 5.9: Unidade de processamento para o Smith-Waterman que implementa diretamente as equações do algoritmo, retirado de Oliver et al. (2005a). 45
Figura 5.10: Unidade de processamento projetada para o Smith-Waterman 47

Figura 6.1: Fluxo utilizado no projeto das arquiteturas. 50
Figura 6.2: Top-level do projeto após síntese lógica 55
Figura 6.3: Detalhe mostrando a conexão entre UPs após síntese lógica 56
Figura 6.4: Gerador de memória fornecido pela empresa Artisan 56
Figura 6.5: Anéis e stripes de alimentação configurados 58
Figura 6.6: Visão do posicionamento dos blocos projetados utilizando o Amoeba View 58
Figura 6.7: Versão final do SWAffine em ASIC com árvore de *clock* em destaque 59

LISTA DE TABELAS

Tabela 2.1: O padrão utilizado pelo código genético 16

Tabela 3.1: Exemplo do algoritmo distância Levenshtein 20
Tabela 3.2: Exemplo do algoritmo Smith-Waterman 23

Tabela 5.1: Exemplo do algoritmo distância Levenshtein com otimização proposta por Lipton e Lopresti (1985) 43

Tabela 6.1: Dados da síntese de uma UP para o dispositivo XC5VLX330T-1 . . . 52
Tabela 6.2: Dados da síntese dos projetos para o dispositivo XC5VLX330T-1 . . 53
Tabela 6.3: Desempenho do barramento PCIe usando DMA 53
Tabela 6.4: Relatório de área após síntese lógica 55
Tabela 6.5: Relatório de potência após síntese lógica 56
Tabela 6.6: Caminho crítico determinado após síntese lógica 56
Tabela 6.7: Relatório de área após síntese física 58
Tabela 6.8: Relatório de potência após síntese física 58
Tabela 6.9: Particionamento de sequências em relação ao BD Swiss-Prot 60
Tabela 6.10: Comparação entre o desempenho da arquitetura em hardware dedicado e o software SSearch36 61
Tabela 6.11: Comparação entre as arquiteturas estudadas e arquiteturas projetadas para a distância Levenshtein 62
Tabela 6.12: Comparação entre as arquiteturas estudadas e arquiteturas projetadas para o Smith-Waterman com affine gap 63

RESUMO

Bancos de dados biológicos utilizados para comparação e alinhamento local de sequências tem crescido de forma exponencial. Isso popularizou programas que realizam buscas nesses bancos. As implementações dos algoritmos de alinhamento de sequências Smith-Waterman e distância Levenshtein demonstraram ser computacionalmente intensivas e, portanto, propícias para aceleração em hardware. Este trabalho descreve arquiteturas em hardware dedicado prototipadas para FPGA e ASIC para acelerar os algoritmos Smith-Waterman e distância Levenshtein mantendo os mesmos resultados obtidos por softwares. Descrevemos uma nova e eficiente unidade de processamento para o cálculo do Smith-Waterman utilizando *affine gap*. Também projetamos uma arquitetura que permite particionar as sequências de entrada para a distância Levenshtein em um array sistólico de tamanho fixo. Nossa implementação em FPGA para o Smith-Waterman acelera de 275 a 494 vezes o algoritmo em relação a um computador com processador de propósito geral. Ainda é 52 a 113% mais rápida em relação, segundo nosso conhecimento, as mais rápidas arquiteturas recentemente publicadas.

Palavras-chave: Smith-Waterman, distância Levenshtein, distância de edição, array sistólico, hardware dedicado, alinhamento local, programação dinâmica, FPGA, ASIC, comparação de genomas.

Hardware Architectures for Local Biological Sequence Alignment

ABSTRACT

Bioinformatics databases used for sequence comparison and local sequence alignment are growing exponentially. This has popularized programs that carry out database searches. Current implementations of sequence alignment methods based on Smith-Waterman and Levenshtein distance have proven to be computationally intensive and, hence, amenable for hardware acceleration. This Msc. Thesis describes an FPGA and ASIC based hardware implementation designed to accelerate the Smith-Waterman and Levenshtein distance maintaining the same results yielded by general softwares. We describe an new efficient Smith-Waterman affine gap process element and a new architecture to partitioning and maping the Levenshtein distance into fixed size systolic arrays. Our FPGA Smith-Waterman implementation delivers 275 to 494-fold speed-up over a standard desktop computer and is also about 52 to 113% faster, to the best of our knowledge, than the fastest implementation in a most recent family of accelerators.

Keywords: DNA sequence scanning, Smith-Waterman, edit distanve, Levenshtein distance, dynamic programming, systolic array, FPGA, ASIC, VLSI, protein sequences, query sequence, subject sequence.

1 INTRODUÇÃO

A biologia molecular está repleta de perguntas fascinantes. Como as espécies se adaptam ao seu ambiente? Quais genes são responsáveis pelas principais doenças humanas? Por que é necessário produzir novas vacinas anualmente? A introdução de métodos matemáticos e a grande quantidade de dados genômicos sequenciados nos permitem responder esse tipo de questão. A disponibilidade desses dados e os algoritmos desenvolvidos para processá-los desencadearam uma revolução na biologia, muitas vezes comparada ao mesmo fenômeno ocorrido na física no início do século XX. Os efeitos dessa revolução são sentidos em muitos outros campos da ciência. Aplicações no campo da medicina, desenvolvimento de vacinas, forense, antropologia e epidemiologia prometem aumentar nossa qualidade e expectativa de vida, bem como nosso entendimento do mundo.

Pense no problema de ter que escrever o conjunto de todas as instruções necessárias para construir e operar um ser humano, e pense que essa informação deve ser acessada de forma distribuída em cada uma das trilhões de células que compõe esse corpo. A solução encontrada pela natureza para esse problema é o genoma. Calcula-se que seu tamanho completo é de 3.5 bilhões de caracteres e cada célula do corpo possui duas cópias desse conjunto de instrução (GUSFIELD, 1997).

Atualmente a tecnologia de instrumentos para sequenciar dados biológicos tem avançado drasticamente. As primeiras máquinas utilizadas no fim da década de 80 sequenciavam cerca de 4800 pares de bases por dia. Em 2010, duas importantes empresas do setor (*Illumina* e *Life Technologies*) anunciaram o lançamento de máquinas capazes de gerar, respectivamente, 25 bilhões e 100 bilhões de pares de bases em um dia (VENTER, 2010). Alguns aspectos relativos à precisão dos resultados obtidos por esses equipamentos ainda são discutidos, porém é inegável o avanço obtido em apenas 20 anos.

Todos esses dados biológicos sequenciados, atualmente, são armazenados em repositórios públicos que permitem livre acesso. Porém, com a redução dos custos dessas máquinas de sequenciamento, é relativamente comum empresas trabalharem com sequências privadas que são utilizadas para desenvolver novos fármacos e produtos de biotecnologia. Atualmente há um setor emergente denominado de medicina personalizada, que promete sequenciar um genoma humano personalizado por US$ 6000. Desse modo, é fácil prever um mundo em alguns anos onde a informação genética será tão abundante como hoje são nossas informações financeiras.

Para Lecompte et al. (2001) o momento atual é denominado era pós-genômica, pois existe uma grande quantidade de informação biológica, onde o desafio é processar e extrair significado dessas. Dados sobre futuras doenças que o paciente possa desenvolver, ou a produção de fármacos mais eficientes que sejam elaboradas a partir das informações

de cada genoma pessoal são algumas promessas de aplicações da medicina personalizada. Mesmo hoje, a biotecnologia já utiliza muito o processamento em larga escala de dados biológicos para o desenvolvimento dos seus produtos.

É nesse contexto que nosso trabalho pretende contribuir. É conhecido pela literatura que processar dados biológicos utilizando processadores de propósito geral não é a melhor forma de lidar com esse problema em virtude da dificuldade da indústria de processadores de continuar avançando segundo a lei de Moore, segundo a qual o número de transistores em um circuito integrado dobra aproximadamente a cada dois anos. A computação híbrida, que agrega processadores de propósito geral e dedicados em um mesmo sistema, é o caminho natural para aumentar o desempenho de aplicações dedicadas (como é o caso da biologia molecular). Algumas tecnologias se apresentam como candidatas a esse modelo de computação: ASICs (*Application-Specific Integrated Circuit*), FPGAs (*Field-Programmable Gate Array*) e GPUs (*Graphics Processing Unit*).

Os sistemas projetados utilizando ASICs em geral são os mais rápidos e que consomem a menor potência. Porém, a principal desvantagem desse tipo de solução é em relação a flexibilidade e ao custo para pequena escala. Um sistema em ASIC, após fabricado é fixo - não permite modificação no projeto ou na configuração do algoritmo (exceto se o mesmo tenha sido projetado prevendo essa modificação). Para aplicação em biologia molecular esse tipo de solução não se apresenta como a mais adequada. Os algoritmos, em geral, permitem uma grande gama de parâmetros. Implementar em ASICs todas as configurações e modos de execução de um algoritmo em um mesmo circuito aumentaria a complexidade do projeto e reduziria o desempenho. Assim, essa solução é uma ótima candidata quando se tem um problema específico e com parâmetros bem conhecidos. Se o custo não é uma variável importante, visto que a escala para aplicações em biologia molecular ainda é pequena, essa tecnologia não deve ser desprezada, pois os melhores resultados de desempenho e potência serão encontrados nela.

Os FPGAs permitem que um algoritmo possa ser acelerado de forma semelhante ao utilizado pela tecnologia ASIC - por meio de uma arquitetura dedicada para resolução do problema em questão. Esse tipo de tecnologia utiliza uma matriz de elementos lógicos que permite que qualquer função lógica possa ser implementada nela. Uma grande vantagem dessa estrutura é que ela pode ser configurada diversas vezes, o que flexibiliza o uso e o fluxo de projeto de arquiteturas para essa tecnologia. Porém, em detrimento de permitir a reconfiguração do circuito, o desempenho global do sistema é, em geral, inferior ao encontrado em circuitos ASICs. Contudo, para a biologia molecular, se mostra uma solução bastante competitiva. É possível, por exemplo, projetar diversas arquiteturas para diferentes algoritmos (ou modos de execução de um mesmo algoritmo) e carregá-las em um único dispositivo FPGA conforme as necessidades do usuário. Ainda, analisando o desempenho por custo, o FPGA é a solução mais atraente, quando comparado à ASIC, para aplicações em biologia molecular.

Por fim, as placas gráficas denominadas GPU estão sendo bastante adotadas para a aceleração de algoritmos em diversas áreas do conhecimento - incluindo biologia molecular. Antigamente elas eram apenas utilizadas para acelerar aplicações gráficas, muito utilizadas por jogos e processamento de imagens. Porém a empresa Nvidia, uma das principais fabricantes do setor, lançou no mercado uma API (*Application Programming Interface*) denominada CUDA (*Compute Unified Device Architecture*) que permite que algoritmos desenvolvidos em diferentes linguagens possam ser paralelizados utilizando as diversas unidades de processamento da placa gráfica. Essa solução possui o mais alto

grau de flexibilidade e o menor custo. Adaptar um algoritmo para processar em uma placa gráfica usando uma API é uma tarefa bastante simples quando comparado ao projeto de circuitos dedicados que é necessário em ASIC e FPGA. Percebemos então que essa solução tem o menor tempo de projeto, o que faz que tenha o menor custo (o valor do hardware para as versões com maior desempenho se pode comparar com uma placa FPGA). Entretanto, a flexibilidade é diretamente proporcional a aceleração que esse tipo de solução alcança - em geral bastante inferior que as apresentadas por ASIC e FPGA.

Nesse trabalho nos concentramos em explorar arquiteturas em hardware dedicado utilizando a tecnologia ASIC e FPGA. Não utilizamos GPU, por essa ser uma arquitetura fixa, apenas permite que algoritmos sejam implementados nela - e não o projeto de novas arquiteturas. Trabalhamos com dois tipos de algoritmos para lidar com abordagens distintas. Para uma análise preliminar de similaridade entre duas sequências biológicas, a distância Levenshtein (também denominada distância de edição) é uma boa abordagem. A implementação em hardware é bastante eficiente, o que faz com que rapidamente seja possível analisar a proximidade entre duas sequências. Porém, em muitos casos, é necessário realizar comparações que exijam maior precisão e confiabilidade dos resultados (MCMAHON, 2008). Para esses casos, projetamos uma arquitetura que implementa o algoritmo Smith-Waterman utilizando *affine gap* e matrizes biológicas de substituição - BLOSUM (*Blocks of Amino Acid Substitution Matrix*), PAM (*Point Accepted Mutation*) etc.

Apresentamos como principais contribuições uma nova arquitetura de array sistólico, que enfatiza o uso de um protocolo de comunicação e o particionamento de sequências. Além disso, apresentamos uma nova unidade de processamento para o algoritmo Smith-Waterman com *affine gap* que reduz o número de somadores e comparadores de máximo permitindo maior frequência de operação e menor área quando comparada com outras arquiteturas. Mostramos o fluxo de prototipação desses sistemas utilizando FPGA e ASIC e analisamos o desempenho e a aceleração obtida em relação ao uso de processadores de propósito geral.

2 FUNDAMENTOS DE BIOLOGIA MOLECULAR E BIOINFORMÁTICA

A genética pode ser definida como o estudo da hereditariedade, e num sentido mais restrito, o estudo dos genes (JONES; PEVZNER, 2004). Foi iniciada com a descoberta dos "fatores de hereditariedade"por George Mendel por volta de 1860. Entrou na era molecular a partir da metade do século XX com a definição de estrutura de dupla hélice do DNA (*Ácido Desoxirribonucleico*), proposta por James Watson e Francis Crick, e com o surgimento das primeiras técnicas de sequenciamento, desenvolvidos por Sanger e Coulson em 1975 (CRISTIANINI; HAHN, 2007).

As questões mais fundamentais da biologia molecular tratam do funcionamento das células. Em organismos unicelulares e multicelulares é necessário descobrir como as células reagem em seu ambiente, como os genes afetam essas reações e como os organismos se adaptam a novos ambientes (SACCONE; PESOLE, 2003).

O objetivo desse capítulo é apresentar uma revisão das ideias básicas da biologia molecular, permitindo um conhecimento mínimo para o entendimento dos próximos capítulos. Entendemos que na biologia existem muitas exceções: todas as generalizações e regras que apresentaremos aqui estarão erradas para alguns organismos. Porém, de forma geral, os principais pontos estão corretamente expostos.

2.1 Material Genético

Toda a vida no planeta depende de três tipos de molécula: DNA, RNA (*Ácido Ribonucleico*) e proteínas. Basicamente, o DNA representa uma grande biblioteca responsável por descrever o funcionamento da célula. Em contrapartida, o RNA é responsável por transferir pequenos pedaços dessa biblioteca para diferentes regiões da célula, onde esses pequenos volumes são utilizados para a síntese de proteínas. Essas, formam enzimas que, por meio de reações bioquímicas, enviam sinais para outras células, formando assim algumas partes do corpo (como a queratina em nossa pele) e fazem todo o trabalho da célula (JONES; PEVZNER, 2004).

O DNA foi descoberto em 1869 por Johann Friedrich Miescher quando ele isolou o núcleo de células sanguíneas brancas (leucócitos). No início do século XX já se sabia que o DNA era uma molécula composta por quatro tipos de bases: adenina (A), timina (T), guanina (G) e citosina (C). Originalmente, os biólogos descobriram um quinto tipo de base, denominada uracila (U), quimicamente semelhante a timina. A partir de 1920, o ácido nucleico foi agrupado em duas classes chamada DNA e RNA, que diferem na

composição das bases: o DNA utiliza o T e o RNA utiliza o U. Formalmente, o conjunto de todo o DNA associado a um organismo é denominado genoma.

As proteínas são formadas por cadeias de aminoácidos. Um proteína típica contém 200-300 aminoácidos, porém algumas são bem menores (20-40) e outras bem maiores (dezenas de milhares de aminoácidos). As proteínas podem ser modeladas como cadeias unidimensionais, porém não tem esse formato na célula. Elas são na realidade estruturas tridimensionais, onde o seu formato é o responsável por determinar a sua função. Existem 20 tipos de aminoácidos, o tipo e a posição nas cadeias são os responsáveis por determinar o formato da estrutura e, consequentemente, a função. O número de possibilidades de proteínas é enorme. Se assumirmos que todas as proteínas possuam tamanho máximo de 400 aminoácidos, se pode criar 20^{400} diferentes tipos de proteínas. Segue abaixo, os alfabetos utilizados para representar DNA (2.1), RNA (2.2) e aminoácidos (2.3):

$$N_{DNA} = \{A, C, G, T\} \tag{2.1}$$

$$N_{RNA} = \{A, C, G, U\} \tag{2.2}$$

$$A = \{A, R, N, D, C, Q, E, G, H, I, L, K, M, F, P, S, T, W, Y, V\} \tag{2.3}$$

2.2 Dogma Central da Biologia Molecular

O modo mais fácil de explicar como as proteínas são feitas (ignorando muitos detalhes e condensando complicadas reações celulares) é por meio do diagrama apresentado em (2.4).

$$DNA \rightarrow RNA \rightarrow Proteina \tag{2.4}$$

Basicamente, o dogma central postula que "DNA produz RNA que produz proteínas"(a qual auxilia que seja produzido DNA novamente). O processo que produz proteínas a partir de DNA é dividido em duas etapas: transcrição ($DNA \rightarrow RNA$) e tradução ($RNA \rightarrow Proteina$) (SACCONE; PESOLE, 2003). É sabido que o dogma central nem sempre está correto, porém para a nossa abordagem ele é suficiente.

A máquina celular responsável por produzir proteínas é denominada ribossomo. O ribossomo utiliza uma sequência de nucleotídeos (no caso, RNA) e traduz essa cadeia de caracteres para uma nova cadeia de caracteres de aminoácidos. Para o ribossomo não trabalhar diretamente com o DNA é necessário que seja utilizado uma cópia do mesmo. Devido a razões físicas da célula é mais fácil o ribossomo traduzir aminoácidos a partir de RNA. Assim, a célula transcreve os genes codificados em DNA para RNA (denominado RNA mensageiro) e os envia para o ribossomo.

O processo de transcrição de DNA para RNA mensageiro possui correspondência um por um entre a sequência de origem (DNA) e a sequência destino (RNA). A diferença básica é o uso do nucleotídeo uracila no lugar da timina. O próximo passo é o processo de tradução de RNA para aminoácidos. Esse processo não é tão simples, tendo em vista que o alfabeto do RNA é composto por 4 símbolos e o alfabeto dos aminoácidos por 20 símbolos. Nesse caso, não é possível fazer uma correspondência um por um entre a sequência de origem e a de destino. Dessa forma, a correspondência possível é utilizar 3 nucleotídeos para representar 1 aminoácido permitindo uma faixa de representação de 64 aminoácidos (4^3).

Assim, todos os organismos na terra utilizam esse método para traduzir nucleotídeos para aminoácidos. É denominado *códon* o conjunto de 3 nucleotídeos que formam um

Tabela 2.1: O padrão utilizado pelo código genético

	A	G	C	T
A	AAA K AAG K AAC N AAT N	AGA R AGG R AGC S AGT S	ACA T ACG T ACC T ACT T	ATA I ATG M ATC I ATT I
G	GAA E GAG E GAC D GAT D	GGA G GGG G GGC G GGT G	GCA A GCG A GCC A GCT A	GTA V GTG V GTC V GTT V
C	CAA Q CAG Q CAC H CAT H	CGA R CGG R CGC R CGT R	CCA P CCG P CCC P CCT P	CTA L CTG L CTC L CTT L
T	TAA * TAG * TAC Y TAT Y	TGA * TGG W TGC C TGT C	TCA S TCG S TCC S TCT S	TTA L TTG L TTC F TTT F

aminoácido. A Tabela 2.1 mostra como é feita essa tradução. É percebido que diferentes *códons* podem representar um mesmo aminoácido. O conjunto de aminoácidos formados a partir da tradução representam proteínas. Como dito anteriormente, as proteínas é que são as responsáveis por grande parte do trabalho realizado na célula - incluindo cópia de DNA, mover materiais para dentro da célula e comunicar com células vizinhas.

2.3 Bancos de Dados Genéticos

O primeiro passo para qualquer análise genética é obter as sequências necessárias para o estudo. Todos os genomas sequenciados que foram objetos de publicações acadêmicas estão disponíveis na internet. As principais revistas e congressos científicos exigem que a sequência objeto do trabalho seja armazenada em alguma base de dados pública.

O principal agente responsável pela distribuição dessas base de dados (BD) é o consórcio formado pelos três maiores bancos de dados genéticos: a Base de Dados de DNA do Japão (DDBJ), o Laboratório Europeu de Biologia Molecular (EMBL) e o *GenBank* (financiado pelo governo dos Estados Unidos). Esses bancos são denominados bancos primários, pois são os responsáveis pelo armazenamento e distribuição de forma gratuita de todas as sequências públicas já mapeadas. O principal problema em relação à confiabilidade desses bancos consiste que as informações não são curadas, ou seja, as sequências armazenadas não são verificadas por um comitê para analisar a sua relevância. Em março de 2010, havia aproximadamente 120 bilhões de bases de nucleotídeos armazenadas nesses três bancos. A Figura 2.1 mostra o crescimento dos mesmos a partir de abril de 1995 até março de 2010. Ainda, segundo o relatório NCBI (2009) desde o seu nascimento em 1982 até abril de 2009 o número de bases tem dobrado a cada 18 meses.

Existem ainda outros bancos de dados públicos, denominados secundários, que deri-

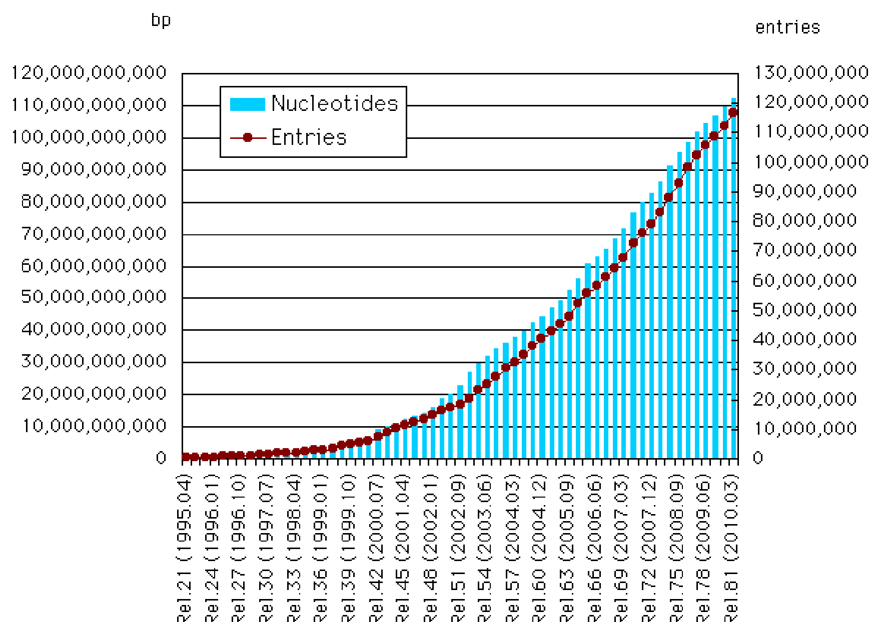


Figura 2.1: Crescimento dos bancos de dados genético GenBank/EMBL/DDBJ (1995-2010)

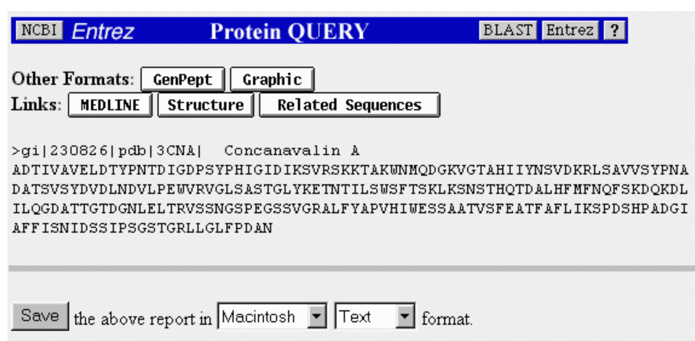


Figura 2.2: Exemplo do formato FASTA

vam dos primários. Basicamente eles se especializam em nichos e geralmente são curados. Como exemplo podemos citar o UniprotKB/Swiss-Prot, especializado em sequências de proteínas, possuindo todos os dados curados. Outro exemplo é o Prosite especializado em apontar domínios funcionais em uma proteína utilizando consensos armazenado em seu banco de dados. Por fim, o KEGG é um banco de dados secundário responsável por representações e mapas de vias metabólicas.

Existem muitos formatos utilizados para armazenar as sequências nesses bancos. Nenhum padrão foi criado, porém observa-se que praticamente todos os bancos públicos utilizam (além de seus próprios formatos) o formato FASTA. Esse é um formato muito simples e compacto para armazenar a informação biológica. A Figura 2.2 mostra um exemplo de uma sequência no formato FASTA. Observamos na primeira linha o caractere de maior (>) que simboliza o início da sequência seguido por algumas informações sobre a mesma. Logo após, a segunda linha, já inicia com a sequência propriamente dita. O caractere de fim de sequência é representado pelo fim do arquivo ou pelo início de uma nova sequência.

3 ALGORITMOS PARA ALINHAMENTO DE SEQUÊNCIAS

Nesse capítulo apresentamos alguns dos principais algoritmos utilizados para alinhamento de sequências biológicas. O desenvolvimento desses algoritmos foram obtidos em saltos lentos. Os principais resultados vieram com espaçamento, em geral, de uma década: distância Levenshtein na década de 60, alinhamento global Needleman-Wunsch em 70, alinhamento local Smith-Waterman em 80 e heurísticos no início dos anos 90.

Medir a similaridade entre duas sequências (DNA, RNA ou proteínas) é considerado o mais importante cálculo na genômica computacional e tem se tornado uma tarefa diária para biólogos (CRISTIANINI; HAHN, 2007). Existem diferentes tipos de alinhamentos, que serão discutidos nesse capítulo, mas o objetivo principal é determinar o quão semelhantes são as sequências envolvidas no alinhamento.

Consideramos dois tipos básicos de algoritmos de alinhamento: global e local. Podemos considerar que, se duas sequências possuem o mesmo ancestral, se espera que elas possuam muitos símbolos em comum. Assim, o alinhamento busca aproximar os símbolos das sequências analisadas. Permite-se inserir lacunas entre as duas sequências, denominadas ao longo desse trabalho de *gap*, como forma de representar resultados evolutivos de inserção ou remoção de genes - denominado *indel*. Assim em (3.1) mostramos um exemplo de alinhamento global e em (3.2) um exemplo de alinhamento local retirados de Ticona (2003).

$$\begin{array}{l} \text{Dadas as sequências:} \\ A = \{G, A, A, G, G, A, T, T, A, G\} \\ B = \{G, A, T, C, G, G, A, G\} \\ \text{Tem-se o seguinte alinhamento global:} \\ \begin{array}{cccccccccc} G & A & A & _ & G & G & A & T & T & A & G \\ | & | & & & | & | & | & & & | & | \\ G & A & T & C & G & G & A & _ & _ & A & G \end{array} \end{array} \tag{3.1}$$

$$\begin{array}{l} \text{Dadas as sequências:} \\ A = \{A, A, G, A, C, G, G\} \\ B = \{T, C, G, A, A, G\} \\ \text{Tem-se o seguinte alinhamento local:} \\ \begin{array}{ccccccccccc} _ & _ & _ & _ & A & A & G & A & C & G & G \\ & & & & | & | & | & & & & \\ T & C & G & G & A & A & G & & & & \end{array} \end{array} \tag{3.2}$$

Percebe-se que no alinhamento global é buscado o melhor alinhamento das sequên-

cias por todo o seu comprimento. Já o alinhamento local busca o melhor alinhamento em uma determinada parte da sequência. Definimos alinhamento ótimo como o alinhamento que resulta o maior escore final. O algoritmo que encontra o alinhamento global ótimo é o Needleman-Wunsch, apresentado em Wunsch (1970) e foi utilizado como inspiração para o desenvolvimento do algoritmo de alinhamento local ótimo Smith-Waterman. Esse, apresentado em Smith e Waterman (1981). No capítulo 3.2 detalhamos o funcionamento do algoritmo Smith-Waterman e apresentamos um exemplo de uso.

Existem muitas aplicações para o alinhamento de sequências. Por exemplo, partindo do princípio que é conhecida a função de uma proteína em um organismo se utiliza o alinhamento para buscar proteínas semelhantes e fazer inferências sobre o funcionamento dessas proteínas que ainda não se tem evidências experimentais. Denomina-se esse processo de predição de proteínas. Outra aplicação é a construção de árvores evolutivas (ou árvores filogenéticas) responsáveis por determinar a história evolutiva do gene, caracterizar ancestrais e caracterizar famílias gênicas e proteicas (JONES; PEVZNER, 2004). Nesse contexto, o alinhamento entre sequências é uma etapa importante no fluxo utilizado para a criação dessas árvores. Outro uso do alinhamento de sequências é na própria montagem da sequência. Uma das etapas do fluxo, exige que seja realizada a comparação entre duas sequências para fazer uma máscara e descartar parte da sequência que não é necessária.

3.1 Distância Levenshtein

Em 1966, Vladimir Levenshtein introduziu o conceito de distância de edição entre duas cadeias de caracteres como o valor mínimo de operações necessárias para transformar uma das cadeias na outra (JONES; PEVZNER, 2004). Utiliza-se três operações básicas para se calcular a distância de edição: inserir, apagar e substituir. Usualmente o custo de substituir um caractere em outro é fixo em *2* e o custo de apagar e inserir é a metade deste. A maneira mais comum de implementar esse algoritmo é utilizando programação dinâmica, técnica que consiste em dividir uma tarefa complexa em pequenas tarefas menores (SACCONE; PESOLE, 2003). Podemos dividir o algoritmo em duas etapas. A primeira é responsável por buscar o valor da distância de edição e a segunda pelo alinhamento das duas sequências. Em (3.3) mostramos a distância de edição entre duas cadeias de caracteres e o alinhamento. A seguir explicaremos melhor como é calculada cada etapa do algoritmo.

$$
\begin{array}{l}
\text{Dadas as sequências:} \\
A = \{A, T, A, G, C\} \\
B = \{C, A, T, A, G\} \\
\text{Tem-se o seguinte alinhamento:} \\
\begin{array}{ccccc}
_ & A & T & A & G \; C \\
 & | & | & | & | \\
C & A & T & A & G
\end{array} \\
\text{Distância Levenshtein = 2:}
\end{array}
\tag{3.3}
$$

Consideramos as duas sequências de entrada como vetores de caracteres. Assim $A = \{a_1, a_2, a_3, a_m\}$ e $B = \{b_1, b_2, b_3, b_n\}$, denominados, respectivamente, de alvo e de busca. Inicia-se o cálculo da distância de edição inicializando uma matriz *h(i,j)* de tamanho $(m+$

Tabela 3.1: Exemplo do algoritmo distância Levenshtein

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

$1) * (n+1)$. A primeira linha e a primeira coluna da matriz é inicializada utilizando a condição inicial $h(0,1) = 1, h(0,2) = 2$ até $h(0,j) = m$ e $h(1,0) = 1, h(2,0) = 2$ até $h(1,j) = i$. Observa-se que é inserido um elemento nulo em $h(0,0) = 0$ que será fixo durante todo o cálculo, juntamente com a primeira coluna e a primeira linha da matriz. Assim, para cada elemento da matriz que não sejam os elementos fixos é calculado o valor de *d* conforme (3.4). Utiliza-se sempre as posições noroeste (*a*), norte (*b*) e oeste (*c*) de cada elemento da matriz para efetuar esse cálculo, isto é *h(i-1, j-1)*, *h(i-1, j)* e *h(i, j-1)*. O campo *sub* é igual a *2* e os campos *ins* e *del* iguais a 1.

Por exemplo, vamos considerar o cálculo do primeiro elemento *h(1,1)* da matriz mostrada na Tabela 3.1. Observamos que os valores utilizados no cálculo para o elemento noroeste é *h(i-1,0)=0*, para o norte é *h(i-1, j)=1* e para o oeste é *h(i, j-1)=1*. Analisando a equação mostrada em (3.4), percebemos que o primeiro termo para esse exemplo é igual a *2* (já que o caractere de busca é diferente do caractere de alvo). Analogamente, o segundo e o terceiro termo também são iguais a *2*. Assim, o mínimo entre os três termos da equação calculados será o valor final *d* da matriz *h(i,j)*.

Por fim, um apontador registra de qual dos termos da matriz (noroeste, norte ou oeste) que resultou o valor mínimo e o próximo elemento da matriz é calculado. O deslocamento da matriz *h(i,j)* pode ser tanto na horizontal quanto na vertical. Discutimos no capítulo 5.1 o uso da técnica *wavefront* que permite paralelizar o cálculo dos elementos da matriz. O último termo da matriz, *h(m,n)*, é o valor da distância de edição final. Para realizar o alinhamento, se percorre o caminho de cada apontador a partir do último até o primeiro termo. O caminho do alinhamento é mostrado pelos elementos achurados na Tabela 3.1.

$$d = min \begin{cases} \begin{cases} a & if\ (A(i) = B(j)) \\ a + sub & if\ (A(i) \neq B(j)) \end{cases} \\ b + ins \\ c + del \end{cases} \tag{3.4}$$

Essa técnica é bastante utilizada para buscar similaridades entre sequências biológicas. Ainda, o algoritmo Smith-Waterman que será descrito no capítulo 3.2, pode ser considerado um caso geral para a distância de edição, pois possibilita adicionar custos nas operações dependendo de sua localização utilizando as matrizes de substituição.

Existem muitas aplicações para o algoritmo Levenshtein, além de buscar similaridades entre sequências de DNA e proteínas. Grande parte das ferramentas de busca *web* utilizam a distância Levenshtein para sugerir palavras para a busca, quando acreditam que essa tenha sido digitada incorretamente. Outros usos para esse algoritmo são reco-

nhecimento automático de fala, filtros contra *spam* e ferramentas para tradução de textos (WOREK, 2002).

3.2 Smith-Waterman

O algoritmo Smith-Waterman foi desenvolvido por T.F. Smith e M.S Waterman, apresentado no trabalho Smith e Waterman (1981), para calcular o alinhamento ótimo entre duas sequências de caracteres. O algoritmo utiliza como entradas uma sequência denominada alvo e outra denominada busca e encontra o alinhamento ótimo entre elas. É considerada a técnica mais popular utilizada para alinhar sequências, pois permite buscar similaridades em partes da sequência - diferente do alinhamento global que busca sempre alinhar a sequência como um todo. Em relação a métodos heurísticos como o Blast (Basic Local Alignment Search Tool) e o FASTA, possui como vantagens retornar sempre o alinhamento ótimo. Em contrapartida, possui maior complexidade computacional - na ordem de *O(m*n)*.

Consideramos as duas sequências de entrada como vetores de caracteres. Assim $A = \{a_1, a_2, a_3, a_m\}$ e $B = \{b_1, b_2, b_3, b_n\}$, denominados, respectivamente, alvo e busca. Essas sequências podem representar DNA ou RNA (alfabeto com 4 caracteres) ou proteínas (alfabeto com 20 caracteres). O algoritmo possui as condições iniciais mostradas em (3.5) e (3.6) e constrói uma matriz de similaridades *h(i,j)* utilizando as equações mostradas em (3.7), (3.8) e (3.9).

$$h(i,0) = e(i,0) = f(i,0) = 0 \quad 0 \leq i \leq m \tag{3.5}$$

Na equação (3.5) observamos que são inicializadas três matrizes *m*n*, onde *m* é o número de elementos no vetor A e *n* o número de elementos no vetor B. Observa-se que a primeira linha das matrizes é preenchida com 0.

$$h(0,j) = e(0,j) = f(0,j) = 0 \quad 0 \leq j \leq n \tag{3.6}$$

Analogamente, a equação 3.6 inicializa a primeira coluna das três matrizes com 0.

A equação (3.7) busca calcular o escore máximo entre as duas sequências de caracteres. O primeiro termo da equação garante que a matriz somente terá valores positivos. O segundo termo soma ao elemento noroeste da posição *h(i,j)* o valor de uma matriz de substituição - representada por *s(i,j)*. A matriz de substituição receberá como entrada os elementos dos vetores *A* e *B* que estão sendo comparados. Caso esses elementos sejam iguais, o resultado da matriz será um valor positivo e consideramos que houve um *match*. Caso sejam distintos, o resultado da matriz será um valor negativo e consideramos que houve um *mismatch*. Essas matrizes são utilizadas para melhorar a qualidade dos alinhamentos com base em fatores biológicos e probabilísticos (HENIKOFF; HENIKOFF, 1992). A Figura 3.1 mostra a matriz de substituição Blosum62. Observamos na figura que a matriz retorna um valor para cada possibilidade de comparação, sempre retornando positivo quando ocorre um *match* e negativo quando ocorre um *mismatch*.

$$h(i,j) = max \begin{cases} 0 \\ h(i-1,j-1) + s(i,j) \\ e(i,j) \\ f(i,j) \end{cases} \quad 1 \leq i \leq n \; and \; 1 \leq j \leq m \tag{3.7}$$

O terceiro termo da equação (3.7) é calculado na equação (3.8). Observamos que nessa equação, no primeiro termo, é o elemento norte da posição *h(i,j)* que é subtraído por uma penalidade denominada *gap open*. Nesse caso, se considera que houve um *mismatch* entre as sequências e se insere um *gap* em uma das sequências para melhor alinhá-la. O segundo termo calcula outra penalidade denominada *gap extend*, caso já tenha sido aberto um *gap* anteriormente - utiliza a matriz *e(i,j)* que é responsável por armazenar os *gaps* que foram abertos e os que foram prolongados. A equação (3.9) é análoga, porém os termos analisados estão na posição oeste de *h(i,j)*, representam o quarto termo da equação (3.7), e utilizam a matriz de *gap* relacionada com esse termo, denominada *f(i,j)*.

$$e(i,j) = max \begin{cases} h(i-1,j) - g_o \\ e(i-1,j) - g_e \end{cases} \quad 1 \leq i \leq n \; and \; 1 \leq j \leq m \tag{3.8}$$

$$f(i,j) = max \begin{cases} h(i,j-1) - g_o \\ f(i,j-1) - g_e \end{cases} \quad 1 \leq i \leq n \; and \; 1 \leq j \leq m \tag{3.9}$$

Dentre todos os termos calculados na equação (3.7) apenas o que possuir maior valor será armazenado na matriz de similaridades *h(i,j)*. Analogamente, na equação (3.8) o que possuir maior valor será armazenado na matriz de gap *e(i,j)* e na equação 3.9 o que possuir maior valor será armazenado na matriz *f(i,j)*. O maior valor da matriz *h(i,j)* será o escore máximo (ou o escore de similaridade) entre as duas sequências. Quanto maior esse valor mais semelhantes são as sequências.

A próxima operação a ser realizada após se obter o escore máximo é denominada *trace-back*. A mesma busca, a partir do escore máximo, o caminho de volta, ou seja, de quais elementos aquele valor se originou. Assim, se verifica qual elemento (noroeste, norte ou oeste) originou o elemento atual e se retorna para esse. Realiza-se esse procedimento até se encontrar um valor nulo. A posição das sequências nos vetores *A* e *B* que fazem parte desse caminho formam o alinhamento ótimo entre as duas sequências.

Gotoh (1982) introduziu dois conceitos distintos para análise do algoritmo Smith-Waterman. O primeiro, demonstrado nas equações (3.7), (3.8) e (3.9) é o caso geral, denominado *affine gap*, utilizado quando as penalidades de *gap open* e *gap extend* são diferentes entre si. Esse modo é considerado biologicamente mais preciso, porém observando as equações percebemos que utiliza mais recursos computacionais que no caso onde os valores de *gap open* e *gap extend* são iguais - denominado *linear gap*. Para esse caso, podemos simplificar as equações (3.7), (3.8) e (3.9) para a equação (3.10) sem mais necessitarmos de matrizes de *gap* para registrar se um *gap* foi aberto ou se foi prolongado.

$$h(i,j) = max \begin{cases} 0 \\ h(i-1,j-1) + s(i,j) \\ h(i-1,j) + g_o \\ f(i,j-1) + g_o \end{cases} \quad 1 \leq i \leq n \; and \; 1 \leq j \leq m \tag{3.10}$$

	Ala	Arg	Asn	Asp	Cys	Gln	Glu	Gly	His	Ile	Leu	Lys	Met	Phe	Pro	Ser	Thr	Trp	Tyr	Val
Ala	4																			
Arg	-1	5																		
Asn	-2	0	6																	
Asp	-2	-2	1	6																
Cys	0	-3	-3	-3	9															
Gln	-1	1	0	0	-3	5														
Glu	-1	0	0	2	-4	2	5													
Gly	0	-2	0	-1	-3	-2	-2	6												
His	-2	0	1	-1	-3	0	0	-2	8											
Ile	-1	-3	-3	-3	-1	-3	-3	-4	-3	4										
Leu	-1	-2	-3	-4	-1	-2	-3	-4	-3	2	4									
Lys	-1	2	0	-1	-3	1	1	-2	-1	-3	-2	5								
Met	-1	-1	-2	-3	-1	0	-2	-3	-2	1	2	-1	5							
Phe	-2	-3	-3	-3	-2	-3	-3	-3	-1	0	0	-3	0	6						
Pro	-1	-2	-2	-1	-3	-1	-1	-2	-2	-3	-3	-1	-2	-4	7					
Ser	1	-1	1	0	-1	0	0	0	-1	-2	-2	0	-1	-2	-1	4				
Thr	0	-1	0	-1	-1	-1	-1	-2	-2	-1	-1	-1	-1	-2	-1	1	5			
Trp	-3	-3	-4	-4	-2	-2	-3	-2	-2	-3	-2	-3	-1	1	-4	-3	-2	11		
Tyr	-2	-2	-2	-3	-2	-1	-2	-3	2	-1	-1	-2	-1	3	-3	-2	-2	2	7	
Val	0	-3	-3	-3	-1	-2	-2	-3	-3	3	1	-2	1	-1	-2	-2	0	-3	-1	4

Figura 3.1: Matriz de substituição Blosum62

Tabela 3.2: Exemplo do algoritmo Smith-Waterman

*	*	*A*	*T*	*C*	*G*	*T*	*G*
*	0	0	0	0	0	0	0
G	0	0	0	0	10	5	10
T	0	0	10	5	5	20	5
C	0	0	5	20	15	15	15
T	0	0	10	15	15	25	20
G	0	0	5	14	25	20	35

A Tabela 3.2 mostra um exemplo de matriz de similaridades *h(i,j)*. Para esse caso se utilizou as sequências de nucleotídeos $A = \{A, T, C, G, T, G\}$ e $B = \{G, T, C, T, G\}$. Utilizamos o algoritmo com *affine gap*, onde o valor $g_o = -5$ e $g_e = -1$. Para simplificar o exemplo criamos uma matriz de substituição para nucleotídeos onde um *match* retorna o valor *10* e um *mismatch* o valor *-5*. O caminho em destaque representa o *trace-back*.

3.3 Alinhamento Múltiplo

O alinhamento par a par, descrito nos capítulos 3.1 e 3.2 é geralmente o primeiro passo para determinar a função biológica das sequências. Entretanto, alinhar simultaneamente múltiplas sequências pode revelar uma grande riqueza biológica de informação e é utilizado em muitas análises avançadas em biologia (GUSFIELD, 1997). Em (3.11) podemos visualizar um exemplo com o alinhamento (global) múltiplo entre três sequências hipotéticas.

$$
\begin{array}{l}
\text{Dadas as sequências:} \\
A = \{V, I, V, A, L, A, S, V, E, G, A, S\} \\
B = \{V, I, V, A, D, A, V, I, S\} \\
C = \{V, I, V, A, D, A, L, L, A, S\} \\
\text{Tem-se o seguinte alinhamento múltiplo:} \\
\begin{array}{cccccccccccc}
V & I & V & A & L & A & S & V & E & G & A & S \\
V & I & V & A & D & A & _ & V & I & _ & _ & S \\
V & I & V & A & D & A & L & L & A & _ & _ & S
\end{array}
\end{array}
\tag{3.11}
$$

Um algoritmo muito usado para alinhamento múltiplo é o Clustal, atualmente denominado de ClustalW. A ideia básica do Clustal é quebrar o problema de alinhamento múltiplo em múltiplos problemas de alinhamento par a par (SACCONE; PESOLE, 2003). Na primeira etapa do algoritmo são lidas todas as sequências de entrada e agrupadas par a par cobrindo todas as possibilidades de alinhamento (excluindo os idênticos). Após os pares terem sido gerados é calculado o alinhamento global usando o algoritmo de Needleman-Wunsch para cada par. As últimas etapas do algoritmo constroem uma árvore de similaridade entre os escores calculado pelo Needleman-Wunsch de cada par e utiliza essa árvore para gerar o alinhamento global.

Alguns trabalhos, como Oliver et al. (2005b), Lin et al. (2005), tem explorado acelerar o clustal utilizando uma arquitetura dedicada em hardware que implementa o Needleman-Wunsch ou o Smith-Waterman. Esses trabalhos reportam que o alinhamento par a par usando o Needleman-Wunsch no algoritmo Clustal representa 90% do tempo de processamento. Assim, uma abordagem explorada para acelerar esse algoritmo é remover do software do ClustalW a etapa que realiza os alinhamentos par a par e calculá-los utilizando uma arquitetura dedicada.

3.4 Métodos Heurísticos

Os métodos heurísticos para analisar sequências biológicas surgiram no fim da década de 80. Como o número de sequências mapeadas estava aumentando nos bancos de dados públicos, o uso de técnicas de programação dinâmica (como os algoritmos citados nos capítulos 3.1 e 3.2) tornava-se impraticável devido a complexidade na ordem de *O(n*m)*. A solução encontrada foi o uso de métodos heurísticos - métodos rápidos que não garantem calcular a solução ótima. O mais popular desses métodos já desenvolvidos é o Blast.

O Blast é uma família de algoritmos desenvolvidos pelo NCBI (*National Center for Biotechnology Information*), que são os mesmos administradores do *GenBank*. Assim, uma configuração típica do Blast é buscar similaridades entre uma sequência específica e um banco de dados. Usar o Smith-Waterman para esse tipo de busca utilizando bancos de dados muito grandes é impraticável. No capítulo 6.3 mostramos alguns teste desse tipo utilizando o banco de dados *Swiss-Prot* e reportamos os tempos.

Com o crescente aumento no tamanho dos bancos de dados biológicos, mesmo o algoritmo heurístico do blast tem se tornado bastante lento para algumas aplicações. Assim, algumas arquiteturas em hardware dedicado exploram a aceleração do algoritmo. Uma dessas arquiteturas, apresentadas por Park et al. (2009) sugere manter o algoritmo blast em software filtrando as entradas do algoritmo para diminuir o tamanho do banco de dados. O filtro é projetado em hardware e utiliza o algoritmo Smith-Waterman como

unidade central do sistema. Os autores relatam uma aceleração de 6 vezes na execução do Blast comparado com a versão sem o filtro.

4 ARQUITETURAS ESTUDADAS

Este capítulo tem o objetivo de descrever algumas arquiteturas em hardware encontradas na literatura para o cálculo do Smith-Waterman. Encontramos diferentes abordagens para o cálculo da matriz dinâmica utilizada pelo algoritmo. A maioria das arquiteturas estudadas partem da premissa de utilizar múltiplas unidades de processamento para calcular as células da matriz de similaridade. Apesar de termos estudado também as arquiteturas recentemente publicadas Lloyd e Snell (2008), Boukerche et al. (2010), para algoritmos que também calculam alinhamento de sequências buscamos analisar mais detalhadamente as que implementam o Smith-Waterman com *affine gap* e a distância Levenshtein.

Uma observação importante, que muitas vezes não é considerada nas arquiteturas apresentadas é em relação ao tipo de dados de entrada que podem ser processados. Especificadamente, o DNA possui quatro caracteres para codificar a timina, guanina, citosina e adenina (T, G, C e A). Porém, as proteínas são produzidas por 20 diferentes aminoácidos. Assim, é necessário apenas dois bits para representar os 4 caracteres que codificam o DNA e 5 bits para os aminoácidos que codificam as proteínas. O valor de saída das arquiteturas é o escore máximo encontrado e, em alguns casos, informação sobre em qual elemento da matriz de similaridade ocorreu o valor máximo (para posterior alinhamento em software).

Adota-se como unidade de medida de desempenho o CUPS (*Cell Updates Per Second*). Para calcular o CUPS se multiplica a frequência de operação pelo número de UPs (Unidades de Processamento) do circuito. Em (4.1) é mostrada a fórmula para obter o CUPS. Outros critérios adotados além do desempenho são o número de unidades de processamento em paralelo e o tamanho máximo de caracteres de uma sequência possível de ser calculado.

$$CUPS = freq_{clock} * UP \tag{4.1}$$

4.1 A Run-Time Reconfigurable System for Gene-Sequence Searching

Nesse artigo Puttegowda et al. (2003) apresentam uma arquitetura para calcular a distância Levenshtein utilizando sequências de nucleotídeos. Os autores exploram a reconfiguração em tempo de execução do dispositivo FPGA para reduzir o tamanho da unidade de processamento. Assim, é mantido fixo (*hardcode*) o valor de *match* e *mismatch* e também a própria sequência de alvo. Dessa forma é possível integrar 7000 UPs em um dispositivo FPGA XC2V6000, onde uma UP ocupa 4 slices. O desempenho relatado é de

1260 GCUPS e frequência de operação de 180 MHz.

Uma característica interessante do trabalho é a divisão do array sistólico em 4 *cores* distintos de 1750 UPs. Assim, é possível comparar até 4 sequências em paralelo que possuam esse tamanho. O autor não implementa o particionamento de sequências, apenas cita poder conectar várias placas com a arquitetura proposta quando for necessário comparar sequências grandes. O sistema é implementado em uma placa com barramento PCI (*Peripheral Component Interconnect*) e acesso a memória RAM (*Random-access memory*) externa. Em Guccione e Keller (2002) é mostrada uma arquitetura bastante semelhante a esta, utilizando a ferramenta JBITS da empresa Xilinx para fazer a reconfiguração do dispositivo.

4.2 A Smith-Waterman Systolic Cell

Nesse artigo Yu et al. (2003) apresentam uma arquitetura que implementa a distância Levenshtein para comparar nucleotídeos. O autor utiliza um sistema com memória de entrada (8 bytes) e memória de saída (2 bytes) e 4032 unidades de processamento em um array sistólico. A frequência obtida para um FPGA XCV1000E-6 é de 202 MHz e o desempenho de 814 GCUPS. A principal contribuição dos autores nesse trabalho foi alocar uma UP em apenas 3 slices. Para isso utilizaram macros do FPGA da Xilinx que permite que os recursos dos slices sejam configurados manualmente no código HDL (Descrição em Linguagem de Hardware). Eles também compartilharam alguns recursos entre duas UPs, fazendo com que a unidade mínima da arquitetura seja duas UPs que ocupam 6 slices. Não foi utilizada reconfiguração do dispositivo em tempo de execução. Os autores não projetaram o particionamento da sequência.

4.3 An Efficient Digital Circuit for Implementing Sequence Alignment Algorithn in an Extended Processor

Nesse trabalho Kundeti et al. (2008) apresentam uma nova arquitetura para calcular a distância Levenshtein. O objetivo dos autores é projetar um co-processador, que possa ser embarcado a um processador de propósito geral e acessado utilizando o conjunto de instrução (com modificações) do mesmo. Os autores relatam alguns dos trabalhos que implementam a distância de edição utilizando arrays sistólicos, tais como Lipton e Lopresti (1985). Porém, consideram que esse tipo de solução possui muitas desvantagens em relação ao tamanho máximo de sequências que podem ser comparadas e também, no caso de sequências menores, o consumo de potência do array quando este é sub-utilizado. Assim, mesmo oferecendo um desempenho inferior, os autores apresentam uma arquitetura serial (com a computação baseada em um registrador de *shift* e alguns somadores) capaz de calcular um bloco de tamanho fixo (denominado t) durante cada chamada da instrução projetada para o cálculo da distância de edição. Desse modo, a complexidade do cálculo do algoritmo que em um processador de propósito geral é de $O(m*n)$, passa para $O(m/t * n/t)$.

A arquitetura foi prototipada utilizando a tecnologia ASIC (TSMC 0.13 um), em Verilog e utilizando os softwares Encounter da Cadence para síntese lógica e Synopsys VCS para verificação do *netlist* gerado. O trabalho não relata se a síntese física foi realizada e, desse modo, se conclui que os resultados apresentados são referentes às estimativas de

desempenho do Encounter após a síntese lógica. A frequência máxima de operação (com o *slack* igual a 0) é de 1 GHz utilizando 460 *gates*. Os autores compararam o desempenho dessa arquitetura com um processador de propósito geral Pentium 2.4 GHz. Para diferentes valores de *t* (*t=8, 16, 32*) o desempenho foi de, respectivamente, 3.77, 4.33 e 5.50 vezes maior. Por meio da tabela de resultados apresentada pelos autores, inferimos que no melhor caso da arquitetura é possível comparar até 15 caracteres por ciclo, resultando em um desempenho de 15 GCPUS.

Apesar dos resultados serem significativos quando comparados com uma arquitetura de propósito geral, apresenta um desempenho bastante inferior que as que utilizam array sistólico. Isso é devido ao grau de paralelismo alcançado, que nesse caso altera a complexidade do algoritmo para *O(m+n-1)*.

4.4 Biological Information Signal Processor

Nesse trabalho Chow et al. (1991) relatam um dos primeiros esforços para projetar um sistema em hardware dedicado para o cálculo do Smith-Waterman. A arquitetura projetada é bastante flexível e permite que várias opções sejam configuradas pelo usuário. Assim, é possível escolher entre alinhamento local ou global, diferentes valores de *gap* e tipos de matrizes de substituição. Ainda, permite calcular o Smith-Waterman para proteínas e nucleotídeos para sequências de até 4194304 caracteres.

O projeto integra 16 unidades de processamento com precisão máxima na pontuação do escore de 16 bits. O autor projetou um ASIC utilizando tecnologia de 1 mm. As diversas opções de configuração do chip aumentaram significativamente a área da unidade de processamento e a complexidade do controle. Os autores relatam uma aceleração de 445 vezes em relação a um supercomputador Cray 2 de US$ 20M (consideram como US$ 50K o custo do seu sistema). A área do circuito é de 100 mm2 e a frequência de operação de 12.5 MHz. Os autores apresentam como resultado de desempenho o valor de 0.2 GCUPS.

4.5 SWASAD: An ASIC Design for High Speed DNA Sequence Matching

Em Han e Parameswaran (2002) é mostrado o projeto de um circuito ASIC para calcular o Smith-Waterman. Obteve-se uma frequência de 50 MHz, porém o gargalo do sistema foi considerado o barramento PCI. Assim, o projeto seria capaz de operar em uma frequência muito mais alta caso não houvesse as interconexões com esse barramento. O fluxo de projeto adotado utilizou a ferramenta Leonardo Spectrum para a síntese lógica com tecnologia AMI 0.5 um. A síntese física foi realizada utilizando a ferramenta ICStation da empresa Mentor Graphics.

O barramento interno da unidade de processamento é de 8 bits que pode ser expandido para 16 bits utilizando uma lógica de *overflow* que reduz as UPs do circuito. Sem *overflow* o escore máximo que pode ser obtido é de apenas 256. Percebemos que o escore máximo é determinado pelo tipo de pontuação aplicada. Porém, consideramos baixo o valor de 256 para um uso prático do chip. O autor não analisa o tempo gasto para tratar o *overflow*.

Os autores mostram que, ao invés de utilizar comparadores na unidade de processa-

mento (são necessários 3), o uso de subtratores é mais eficiente. Eles realizaram comparações do impacto de projetar esse dispositivo com comparadores e com subtratores e, segundo eles, os subtratores ocupam menor área e possuem menor atraso.

O SWASAD possui 64 unidades de processamento por chip. O esforço dos projetistas foi de integrar o maior número de unidades possíveis. O projeto permite diferentes pontuações para *match* e *mismatch* (mas não utiliza matrizes de substituição) e para as pontuações de *gap open* e *gap extend*. Os autores mostram como resultado o desempenho de 3.2 GCUPS. Na conclusão do artigo, os autores relatam que poderiam utilizar 1024 unidades de processamento se o projeto utilizasse uma tecnologia de 100 nm. O circuito utiliza como entradas apenas sequências de nucleotídeos.

4.6 Hyper Customized Processors for Bio-Sequence Database Scanning on FPGAs

Nesse artigo Oliver et al. (2005a) apresentam algumas arquiteturas para calcular o Smith-Waterman para comparação de proteínas (não é citado o uso da arquitetura para DNA/RNA). Os autores utilizam uma arquitetura básica de array sistólico e projetam duas unidades de processamentos distintas: uma para usar o Smith-Waterman com *affine gap* e outra com *linear gap*. Utilizam 16 bits para representar os dados na matriz.

Os autores demonstram preocupação com a questão do particionamento de sequências. Afirmam que, na prática, dificilmente as sequências cabem na estrutura do array sistólico. Assim, a metodologia utilizada foi quebrar a sequência em um conjunto finito de pedaços. O autores implementaram a quebra de sequências utilizando projetos distintos para cada tipo de uso do algoritmo. Os autores armazenam nas unidades de processamento apenas a coluna da matriz de substituição correspondente à sequência alvo armazenada na UP. Assim, é consumido alguns ciclos toda vez que uma sequência alvo é armazenada para carregar as matrizes de substituição em cada UP. Na quebra das sequências esses ciclos são novamente consumidos, pois com novas sequências novas matrizes de substituição devem ser carregadas.

A primeira arquitetura projetada calcula *linear gap* e permite particionar a sequência até 3 vezes. Esse projeto utilizou 252 unidades de processamento operando a 55 MHz e pode comparar sequências com tamanho máximo de 756 (com desempenho de 13.9 GCUPS). A segunda arquitetura, análoga a primeira, permite particionar a sequência em até 12 vezes e comparar sequências com tamanho máximo de 2016 caracteres (desempenho de 9.2 GCUPS).

A terceira arquitetura projetada calcula *affine gap* (o que torna a unidade de processamento mais complexa). Na primeira abordagem, permite quebrar a sequência em até 3 vezes e 168 unidades de processamento (operando a 45 MHz) podem comparar sequência com no máximo 504 caracteres (7.6 GCUPS de desempenho). A quarta arquitetura, permite quebrar em 12 vezes a sequência - 119 unidades de processamento para comparar sequências com até 1428 caracteres (5.2 GCUPS).

O dispositivo alvo foi o FPGA Virtex-2 XC2V6000 da empresa Xilinx. Não são mostrados detalhes da síntese, porém é comparado o desempenho da arquitetura com o desempenho de um Pentium IV para a mesma tarefa (é obtida uma aceleração de 170 vezes para *linear gap* e 125 para *affine gap*). O autor relata que mesmo na arquitetura

de *affine gap* a estratégia utilizada foi criar diversos projetos para os diferentes valores de *gap* (tornando esses valores constantes em cada arquitetura).

4.7 A Highly Parameterized and Efficient FPGA-Based Skeleton for Pairwise Biological Sequence Alignment

Nesse trabalho Benkrid et al. (2009) apresentam uma arquitetura que permite alto grau de parametrização em tempo de projeto. Isso significa que o código, desenvolvido em Handel-C, permite que diversos fatores relacionados a arquitetura possam ser facilmente configurados. Os autores defendem a ideia que o uso de linguagem de síntese de alto nível (Handel-C, System C, System Verilog etc..) diminuem o abismo que existe entre os profissionais de bioinformática e os projetistas de hardware. Assim, como diretiva de projeto os autores optaram por desenvolver o hardware como se estivem desenvolvendo software, mesmo sabendo que perderiam em desempenho. O resultado é uma arquitetura que permite ser implementada em qualquer dispositivo FPGA e (segundo autor) ASIC.

Entre os parâmetros que podem ser alterados em tempo de projeto destacamos o tipo de sequência (proteína, RNA ou DNA), o tipo de penalidade (affine ou linear gap), o tamanho máximo da sequência comparada e o tipo de algoritmo (Smith-Waterman ou Needleman-Wunsch). A ideia é ter um banco de dados com diversas dessas configurações sintetizadas e carregar no dispositivo FPGA o projeto que melhor se adeque com a necessidade do usuário.

Os autores também destacam a importância do particionamento de sequências no array sistólico. Os autores apresentam uma arquitetura que utiliza uma FIFO (*First In First Out*) na saída da última unidade de processamento, conforme a apresentada em Moldovan e Fortes (1986). Porém nenhum detalhe sobre essa implementação é apresentada, nem em relação ao tamanho máximo de sequências que é possível comparar.

Por fim, os autores apresentam resultados da implementação da arquitetura em uma placa Alpha Data com FPGA Virtex-2 XC2VP100-6. Utilizaram a matriz de substituição Blosum50 e compararam 288 sequências de proteína com tamanho médio de 288 caracteres. O número de unidades de processamento desse experimento foi de 135 (operando a 40 MHz) e o cálculo foi finalizada em 88 ms com um tempo inicial de configuração do dispositivo de 244 us.

Os autores ainda apresentam o resultado de desempenho de diversas configurações de sua arquitetura básica. A performance em GCUPS varia de 4 a 12, o número de unidades de processamento de 100 a 250 e a frequência de 40 a 60 MHz. Basicamente as diferenças são em relação ao tipo de pontuação *affine* ou *linear gap* e número de bits para representar os dados na matriz (o melhor resultado, 252 UP com GCUPS 12.02 utiliza 10 bits). Os demais resultados utilizam 16 bits.

4.8 High-Speed Implementation of Smith-Waterman Algorithm for DNA Sequence Scanning in VLSI

A ideia principal do trabalho de Cheng e Parhi (2008) é diminuir o tempo de computação do array sistólico. Não é mostrado no trabalho o projeto de uma arquitetura completa, capaz de receber sequências em uma memória de entrada, compará-las e enviá-las para

uma memória de saída. Os autores se dedicaram a projetar unidades de processamento que pudessem reduzir o processamento do array sistólico. Para isso eles utilizam como premissa aumentar o *throughput* do sistema, o que significa fazer com que as unidades de processamento calculem mais de uma entrada por ciclo de *clock*. Considerando o número de entradas por unidade de processamento sendo *J*, os autores pretendem diminuir a complexidade do algoritmo de *O(m+n-1)* para *O(m+(n-1/J))*. Onde *m* é o número de caracteres da sequência alvo e *n* o número de caracteres da sequência de busca. Evidentemente que essa abordagem é válida somente para comparações onde a sequência de busca são bem maiores que as sequências de alvo.

No trabalho os autores mostram diagramas de blocos das unidades de processamento para calcular *affine* e *linear gap*. Projetaram também unidades de processamento recebendo 2 sequências de busca por ciclo de *clock* e 3 sequências de busca por ciclo de *clock*. Os autores utilizaram técnicas de *pipeline*, para não aumentar o caminho crítico da unidade de processamento e também otimizar os recursos de hardware (reduzir a área). Dessa forma os autores comparam o resultado de sua unidade de processamento com a projetada por Oliver et al. (2005a), utilizando o mesmo dispositivo Virtex II VC2V6000. Eles alcançaram a mesma frequência de operação (55 MHz), porém compararam 3 sequências de busca no mesmo instante de *clock* (o outro trabalho compara 1 sequência por instante de *clock*). Eles não mostram nenhum detalhe de sua arquitetura: número de unidades de processamento, número de bits para representar a matriz e tamanho máximo de sequências comparadas. Relatam, porém, na conclusão do paper que a principal desvantagem de sua arquitetura é que o aumento, em área, da unidade de processamento é maior que *J* vezes em relação à área da unidade de processamento de Oliver et al. (2005a). Isso demonstra que para processar 3 sequências de busca no mesmo instante de *clock* eles utilizam uma área de unidade de processamento mais de 3 vezes maior que a utilizada por Oliver et al. (2005a). Percebemos, com isso, que a principal contribuição apresentada por essa arquitetura é quando se compara sequências alvo pequenas e fixas com um banco de dados de sequências grandes.

4.9 Differential Scoring for Systolic Sequence Alignment

Nesse trabalho Serna (2007) apresentam uma arquitetura de array sistólico que modifica o modo como as pontuações de *match* e *gap* são realizadas. Os autores afirmam que o tamanho em bits utilizado para representar a matriz de similaridade e o valor de escore máximo estão diretamente relacionados com a complexidade do array sistólico. Assim, é mostrado uma nova maneira de se calcular as penalidades do algoritmo Smith-Watreman. Os autores utilizam o conceito de *affine gap* e *linear gap* porém modificam a forma como é realizada a pontuação usando para isso o conceito de agrupar a pontuação de células vizinhas. Dessa forma, eles exemplificma que para alinhar sequências de tamanho (10^6), com penalidades menores que 4 bits são necessários 22 bits para representar os escores na matriz de similaridades. Porém, utilizando a técnica apresentada por eles são necessários 4 bits - o que representa uma redução de 82%. Isso reduz bastante todo o sistema, tamanho de memória, unidades de processamento e frequência de operação.

A principal desvantagem da arquitetura apresentada é não calcular o algoritmo Smith-Waterman em sua concepção original, conforme detalhado em Smith e Waterman (1981), e amplamente utilizado por biólogos. Dessa forma, mesmo com a significativa redução do hardware, é difícil comparar essa arquitetura com as demais por não calcular o mesmo

algoritmo (e sim uma variação do mesmo). Os autores relataram que prototiparam o circuito utilizando o dispositivo FPGA Spartan-3 XC3S200 da empresa Xilinx e obtiveram frequência de 100 Mhz. Porém não é relatado mais nenhum detalhe do protótipo - tamanho máximo de sequências, número de unidades de processamento.

4.10 Acceleration of Smith-Waterman Using Recursive Variable Expansion

Nesse artigo não é apresentada nenhuma implementação, porém uma nova abordagem em hardware para aumentar o desempenho do algoritmo Smith-Waterman. A ideia principal desse trabalho Nawaz et al. (2008) é reduzir o tempo de computação do array sistólico - semelhante ao de Cheng e Parhi (2008). Porém ao invés de utilizar *pipeline* na unidade de processamento os autores optaram por utilizar uma técnica denominada de *recursive variable expansion*. Basicamente, transformam as 3 equações básicas do algoritmo Smith-Waterman em 8 equações para que possa haver um maior paralelismo no array sistólico. Utilizando essa técnica, demonstraram que 2 unidades de processamento vizinhas podem receber 2 sequências de busca para serem comparadas. Como as dependências foram removidas, o vetor *wavefront* se desloca em blocos de *2X2* nas unidades de processamento, com exceção das bordas onde é realizado um tratamento especial.

Assim, consideraram em seu estudo que a unidade de processamento serial necessita de 4 ciclos de *clock* para finalizar seu cálculo (eles identificaram quatro operações dependentes dentro da unidade de processamento, e as dividiram de forma que cada operação seja executada em um ciclo). Eles definem o tempo de computação geral do sistema de array sistólico serial como $4(m+(n-1))$. Utilizando esse método, o tempo é reduzido para $5(m/2 + ((n/2)-1))$. Evidentemente que o custo dessa arquitetura, como no trabalho apresentado por Cheng e Parhi (2008) recai no tamanho da unidade de processamento. Como dito no início desse capítulo, os autores apresentam nesse trabalho apenas a ideia de otimização de hardware (sem nenhuma implementação).

Podemos fazer uma estimativa do aumento na área que essa arquitetura pode ocasionar comparando a sua unidade de processamento com uma unidade de processamento serial típica. A primeira, conforme dados do trabalho, utiliza 14 somadores, 17 comparadores e 4 LUTs (*Look up Table*) para armazenar as matrizes de substituição. Uma unidade de processamento típica do Smith-Waterman possui (no mínimo) 3 somadores e 3 comparadores e as 4 LUTs com a matriz de substituição. Assim, percebemos que o deslocamento do vetor *wavefront* em blocos 2X2 vai gerar um aumento na área de cada unidade de processamento em mais de 4 vezes. Esse tipo de solução é ideal para utilizar em sequências pequenas prototipadas em famílias de FPGA que possuam grande capacidade. Nesse caso não faria sentido muitas unidades de processamento em paralelo (visto as sequências serem pequenas) e haveria uma grande quantidade de área disponível no dispositivo para ser utilizada.

4.11 Arquiteturas em FPGA para Comparação de Sequências Biológicas em Espaço Linear

Corrêa (2008) apresenta em sua tese de doutorado uma arquitetura genérica de array sistólico com o intuito de implementar UPs para os algoritmo Smith-Waterman e o DIALIGN. O projeto é todo desenvolvido utilizando SystemC, transformado para Verilog utilizando a ferramenta Forte e sintetizado para FPGA Virtex 2 XC2VP70.

A arquitetura apresentada utiliza um protocolo de comunicação para carregar as sequências de DNA nas diversas UPs e, após o processamento, obter o maior escore e a localização. Para o Smith-Waterman foram projetadas duas arquiteturas utilizando 20 e 100 UPs com a versão de *linear gap* do algoritmo. A frequência obtida foi de 174,7 MHz (100 UPs) e 180 MHz (20 UPs). O autor utiliza como comparação uma implementação em software do algoritmo executada em um processador de propósito geral. A aceleração obtida é de 246,9 vezes.

4.12 Families of FPGA-based Accelerators for Approximate String Matching

Nesse trabalho Court e Herbordt (2007) apresentam um *framework* capaz de gerar diversas arquiteturas otimizadas para os algoritmos de alinhamento local (Smith-Waterman) e global (Needleman-Wunsch). A premissa dos autores para o desenvolvimento dessa ferramenta é que, em geral, as arquiteturas previamente desenvolvidas permitem a aceleração em determinada condição específica do algoritmo. Eles defendem que uma arquitetura em hardware suficientemente genérica acarretaria muitos atrasos e não possibilitaria a aceleração que uma arquitetura específica permite. Assim, eles apresentam uma família de arquiteturas que podem ser facilmente geradas utilizando as especificidades dos algoritmos.

Os autores mapearam 3 blocos básicos que são os responsáveis por definir a forma de uso dos algoritmos Smith-Waterman e Needleman-Wunsch. O primeiro denominaram *CharRule*, que busca definir o tipo de dados que serão processados (nucleotídeos ou aminoácidos) e o tipo de matriz de substituição que será utilizado. O segundo componente, *MatchCell*, visa fornecer informações sobre as fórmulas que implementam o algoritmo. O último elemento é denominado *Sequencer* e permite configurar como será a saída do algoritmo (apenas o escore ou se será necessário o *traceback*).

Os autores não revelam se utilizaram *affine* ou *linear gap* para implementar o Smith-Waterman. Utilizando proteínas como entrada e matriz de substituição os autores alocaram 138 UPs em um dispositivo FPGA XC2VP70. A frequência alcançada é de 39 MHz com desempenho de 5,4 GCUPS. Relatam também uma aceleração em relação ao algoritmo executado em software de 186 vezes.

4.13 A Reconfigurable Accelerator for Smith-Waterman Algorithm

Nesse paper Jiang et al. (2007) apresentam uma nova unidade de processamento e uma nova técnica de *floorplaning* para melhorar o desempenho dos algoritmos Smith-Waterman e Needleman-Wunsch. A unidade de processamento apresentada modifica as

equações acrescentando um quarto termo (aos três termos utilizados tradicionalmente). A ideia básica desse novo conceito é, após ter analisado o caminho crítico da unidade de processamento, reduzi-lo adicionando mais lógica (antecipando alguns cálculos dentro da unidade). Utilizando essa técnica eles afirmam ter reduzido o atraso do caminho crítico em 25%.

Outra melhoria no projeto citada pelos autores é a forma como foi realizado o *floorplanning* do array sistólico. Basicamente, o *floorplaning* é uma etapa no projeto de sistemas em chip que define onde os diversos blocos projetados serão alocados (o que impacta no roteamento do sistema). Assim, um bom *floorplaning* pode permitir que o sistema utilize menor área e opere em frequências mais altas. A técnica geralmente utilizada pelas arquiteturas estudadas ou não citam essa etapa (o que nos permite concluir que ela foi realizada de forma automática pela ferramenta de síntese) ou utilizam a técnica de zig-zag no array sistólico em uma coluna. Os autores empregam essa mesma técnica de zig-zag, porém utilizam duas colunas (deixando um espaço livre entre elas). Segundo os autores esse método reduz o caminho crítico do roteamento do array sistólico com os componentes de interface (FIFOs de entrada e saída e controlador para o barramento PCI).

Em relação à quebra de sequências os autores citam que sua arquitetura é capaz de fazer esse particionamento, mas não fornecem nenhum detalhe de como é realizado e qual o tamanho máximo de sequências possíveis de serem comparadas. A tecnologia alvo de implementação foi FPGA da empresa Altera (dispositivo EP1S30). O desempenho alcançado da arquitetura é de 6.6 GCUPS (porém, não informam o número de unidades de processamento nem a frequência). Eles relatam uma aceleração de 330 vezes no algoritmo comparando com uma versão correspondente do mesmo executado em um processador Xeon 2.8 GHz.

Para comparar sua arquitetura com a proposta por Oliver et al. (2005a) os autores repetiram o processo de síntese, dessa vez para o mesmo dispositivo utilizado por Oliver et al. (2005a) - XC2V6000. Nesse capítulo do trabalho eles informam que sua implementação utiliza 80 unidades de processamento (em parte esse valor mais baixo reflete o aumento de lógica dentro da unidade) e frequência de 88 MHz - aumento significativo, comparado com os 55 MHz alcançados por Oliver et al. (2005a).

5 PROJETO EM HARDWARE PARA O ALINHAMENTO LOCAL DE SEQUÊNCIAS

Como explicado no capítulo 3.2, o algoritmo Smith-Waterman possui complexidade computacional *O(n*m)* tornando seu cálculo impraticável considerando o atual tamanho dos bancos de dados genéticos. Assim, muitos esforços, detalhados no Capítulo 4, foram empreendidos no intuito de acelerar o cálculo desse importante algoritmo. Nesse capítulo apresentamos uma arquitetura de array sistólico, largamente utilizada para implementar em hardware algoritmos de programação dinâmica. Em seguida, detalhamos o protocolo projetado para a comunicação entre hardware e software e mostramos como mapear as equações do Smith-Waterman e da distância Levenshtein em unidades de processamento de forma eficiente.

5.1 Matriz Dinâmica

O uso de programação dinâmica para o alinhamento de sequências, introduzido por Wunsch (1970) e Smith e Waterman (1981), permite a atualização de muitas células da tabela de programação dinâmica em paralelo. Esse paralelismo é obtido com uma técnica denominada de *wavefront*, porque as células iniciam seu processamento em paralelo na forma de uma onda em relação ao tempo. Observamos na Figura 5.1 que, para respeitar a dependência entre os elementos (noroeste, norte e oeste) a única maneira de processarmos os dados em paralelo é utilizando a anti-diagonal da matriz de similaridades mostrada.

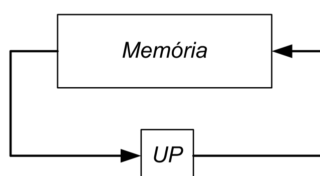
Assim, no instante de tempo *t=1* processa-se apenas a célula 1 - exemplificada na Figura 5.1 como o elemento *(1,1)*. No instante de tempo *t=2*, processa-se a célula *(1,2)* e *(2,1)* em paralelo. Observa-se que as dependências necessárias para o cálculo, nesse caso, foram previamente calculadas no instante de tempo *t=1*. No instante de tempo *t=6*, de forma análoga, calcula-se as células *(4,3)*, *(3,4)*, *(2,5)* e *(1,6)*. A Figura 5.1 mostra essas células processadas em paralelo no instante de tempo *t=6* em hachurado. Nesse caso, observa-se que o maior grau de paralelismo é alcançado nos instantes de tempo *t=3*, *t=4*, *t=5* e *t=6*, quando 4 células estão sendo calculadas em paralelo.

Nesse exemplo também observamos que o tempo de computação do algoritmo é reduzido de *O(m*n)* para *O(m+(n-1))*. Novamente recorrendo ao exemplo da Figura 5.1, com *m=7* e *n=4*, são necessários *10* ciclos de tempo para finalizar o cálculo utilizando o vetor *wavefront*. Caso não utilizássemos essa técnica seriam necessários *28* ciclos. O crescimento quadrático da solução sem utilizar o vetor *wavefront* é que torna proibitivo o uso do algoritmo para bancos de dados muito grandes.

	C	A	T	G	C	C	T
C							
T							
T							
A							

Figura 5.1: Deslocamento do vetor *wavefront* na matriz de similaridades

Modo de processamento de dados tradicional:



Utilizando array sistólico:

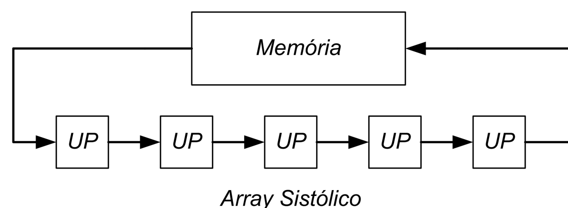


Figura 5.2: Princípios básicos de uso do array sistólico

A técnica mais utilizada para implementar em hardware algoritmos de programação dinâmica, aproveitando o paralelismo do vetor *wavefront*, é utilizando array sistólico. Esse tipo de arquitetura foi introduzido por Kung (1982), onde é apresentada diversas geometrias de arrays sistólicos. Ainda, os autores descrevem como principal vantagem do uso desse tipo de estrutura o alto grau de paralelismo alcançado e a simplicidade e regularidade do projeto. O cálculo da transformada rápida de Fourier é utilizado como exemplo de uso para essa arquitetura. Na Figura 5.2 observamos um esquemático do array sistólico. Basicamente, os dados da memória são carregados em uma fila com diversas unidades de processamento. No exemplo mostrado na Figura 5.2 (geometria unidimensional), cada UP recebe dados da UP anterior (com exceção da primeira UP que recebe dados da memória), processa esses dados e os envia para a próxima UP (com exceção da última UP que envia dados para a memória). É mostrado também na Figura 5.2 a implementação tradicional de um algoritmo em um processador de propósito geral, utilizando a arquitetura de Von Neumann, onde apenas uma UP é utilizada para efetuar cálculos.

O uso da arquitetura apresentada por Kung (1982) para resolver problemas de comparação de sequências biológicas foi introduzido por Lipton e Lopresti (1985). Nesse trabalho é apresentada uma arquitetura de array sistólico implementada em ASIC utilizando a distância Levenshtein para comparação de nucleotídeos. A Figura 5.3 mostra um exemplo de como funciona essa abordagem. As sequências de alvo e de busca são mostradas em (a) na Figura 5.3 em duas memórias e as UPs estão vazias. No instante de

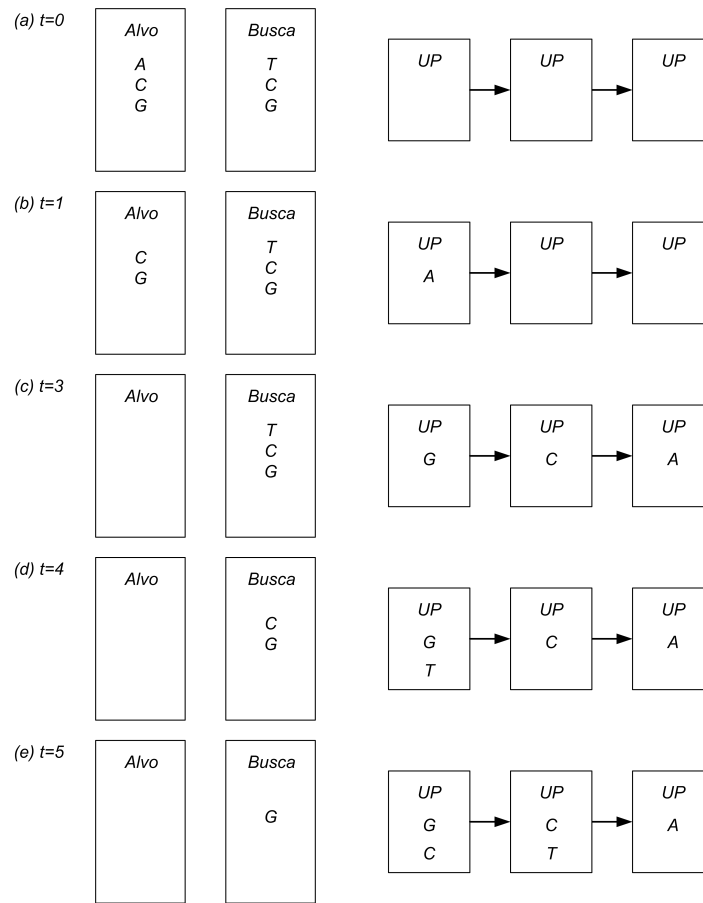


Figura 5.3: Exemplo de como utilizar array sistólico para algoritmos de programação dinâmica

t=1 em (b), inicia-se o carregamento das unidades de processamento com os caracteres da sequência alvo. Em (c) todos os caracteres da sequência alvo foram carregadas nas UPs, e cada uma delas contém uma base da sequência alvo. Observa-se que a ordem de entrada das sequência alvo no array sistólico deve ser invertida para que ocorra o correto processamento. Em seguida, em (d), o primeiro caractere da sequência de busca será carregado na primeira UP. Em (e), o caractere da primeira UP move-se para a segunda UP e um novo caractere da sequência de busca é carregado na primeira UP. Nos próximos instantes de tempo o comportamento será análogo, até que toda a sequência de busca tenha percorrido todas as UPs. A sequência de busca entra no array sistólico sem a necessidade de ter a sua ordem invertida.

Assim, a arquitetura sistólica pode ser utilizada para calcular uma matriz dinâmica em paralelo utilizando a técnica *wavefront*. Basicamente, o array sistólico é composto por n unidades de processamento - onde n é o número máximo de caracteres nas sequências do tipo alvo. Cada UP é responsável por calcular valores em uma coluna fixa da matriz dinâmica. As dependências (noroeste, norte e oeste) são respeitadas no cálculo conforme a técnica *wavefront* mostrada na Figura 5.2. A última etapa é verificar o maior escore obtido na matriz dinâmica. Essa etapa é distinta para diferentes implementações e serão descritas nos capítulos 5.4 e 5.5.

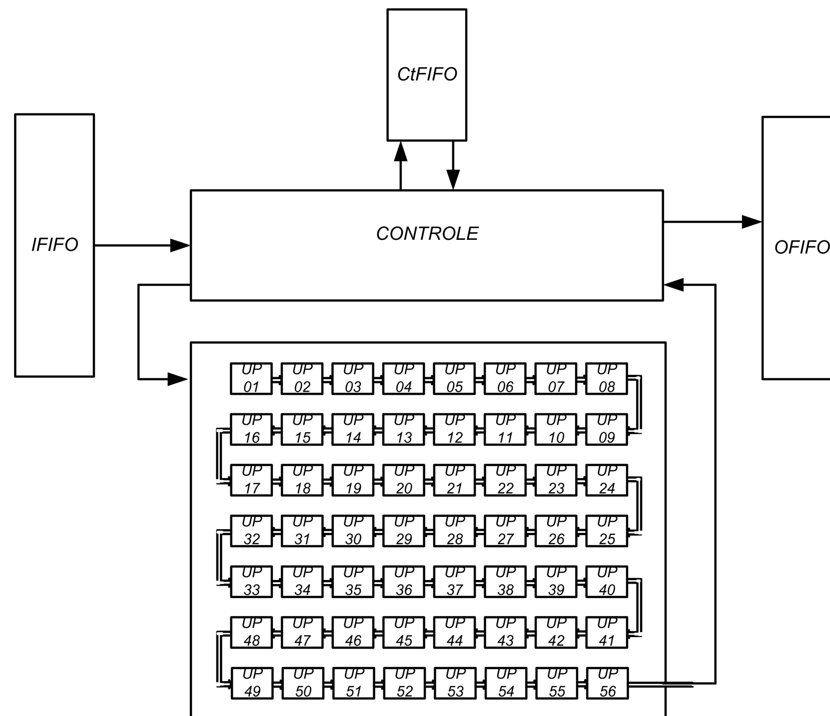


Figura 5.4: Diagrama de blocos da arquitetura para o particionamento de sequências

5.2 Particionamento de Sequências

Um problema crítico no cálculo da distância de edição é em relação ao tamanho máximo das sequências que um projeto em hardware dedicado pode calcular. Não só o número de sequências estão crescendo nos bancos de dados públicos, mas também o tamanho máximo das sequências. Em 2004 o GenBank, um dos principais repositórios de informações biológicas, removeu o limite máximo antes imposto de 350 Kilo bases por sequência (NCBI, 2008). Assim, é cada vez mais importante que as novas arquiteturas permitam que sequências grandes possam ser processadas. O grande limitante disso para sistemas que utilizam array sistólico é que cada caractere da sequência alvo deve ser armazenado em uma unidade de processamento. Isso faz com que sejam necessárias, por exemplo, 2,8 milhões de unidades de processamento se a sequência alvo for o *Staphylococcus aureus* (conforme sequência disponibilizada no GenBank). Mesmo utilizando a distância Levenshtein apresentada no Capítulo 3.1, não é factível projetar um processador dedicado com esse número de unidades de processamento. Nesse contexto, apresentamos na Figura 5.4 uma arquitetura que utiliza como base a proposta por Lipton e Lopresti (1985), porém permite que sequências grandes possam ser particionadas em um número factível de unidades de processamento.

Analisando a arquitetura mostrada na Figura 5.4, observamos o uso de 3 *FIFOs*. A denominada *IFIFO* é responsável por armazenar os dados que estão entrando no chip e fornecê-los para as demais unidades. Para otimizar o desempenho da entrada de dados, optamos por uma arquitetura de *FIFO* de porta dupla, que permite que em um mesmo ciclo de *clock* um dado possa ser escrito e lido. Essa escolha também permitiu que utilizássemos frequências de operação distintas para a escrita de dados na FIFO e para a leitura. Basicamente, é vantajoso utilizarmos frequências distintas visto que a frequência de entrada dos dados é determinada pela frequência máxima do barramento de entrada/saída

utilizado - o que pode fazer com que o circuito perda em desempenho se nos limitarmos a essa frequência.

Observamos que, após os dados serem escritos na *FIFO* de entrada, eles são decodificados pela unidade de controle (o protocolo de comunicação utilizado será explicado em detalhes no capítulo 5.3). Essa unidade é a responsável por transferir os dados já decodificados para o array sistólico. Inicialmente são transferidas os caracteres da sequência alvo (armazenadas em cada unidade de processamento). Em seguida, os caracteres da sequência de busca fluem entre cada unidade de processamento seguindo o processamento *wavefront* explicado no capítulo 5.1. A última unidade de processamento é conectada novamente ao controle. Nesse estágio, a unidade de controle transfere o escore máximo para a OFIFO. Essa, também é de porta dupla e opera com distintas frequências de *clock* pois também se comunica com o barramento de entrada e saída.

Caso a unidade de controle verifique que a sequência que está sendo comparada necessita de particionamento, alguns procedimentos são realizados. Primeiramente, conforme explicado anteriormente, cada unidade de processamento armazena em um registrador interno o valor que será o elemento norte para o cálculo (que é o valor calculado por ela mesma no ciclo de *clock* anterior, exceto no primeiro ciclo que é um valor constante). Assim, para realizar o cálculo de um elemento da matriz a UP necessita receber os valores noroeste e oeste. Percebemos que, para permitir o particionamento de sequências, os únicos valores que precisamos armazenar são os calculados pela última unidade de processamento (que serão os valores noroeste e oeste da primeira UP após o particionamento). Os valores norte já são armazenados por cada UP e não necessitam ser armazenados pela *CtFIFO*. Dessa forma, após a nova sequência alvo ter sido armazenada nas UPs, os valores da *CtFIFO* devem ser carregados na primeira UP como valores noroeste e oeste. Em seguida o cálculo da matriz prosegue normalmente, conforme explicado no Capítulo 5.1. A única diferença é que a última UP, sempre que uma sequência é particionada, armazena os valores calculados por ela na *CtFIFO*. Por esse motivo, para não comprometermos o paralelismo, projetamos a *CtFIFO* com porta dupla para que possa ser lida pela primeira UP e escrita pela última UP em um mesmo ciclo de *clock*.

Na Figura 5.5 observamos um exemplo de como é realizado o particionamento de uma sequência alvo que possui um número de caracteres maior que o número de UPs. Nesse exemplo a sequência de alvo possui 5 caracteres e o array sistólico possui 2 UPs. Em (a) observamos que os dois primeiros caracteres da sequência alvo já foram decodificadas pela unidade de controle e estão sendo carregados nas UPs. Observa-se que eles devem ser armazenadas em ordem inversa as desejadas para o cálculo, devido as características de deslocamento do array sistólico. Em (b) já observamos os caracteres da sequência de busca fluindo pelo array sistólico, nesse estágio observamos também que a última unidade de processamento armazena na *CtFIFO* todos os valores por ela calculados. Em (c) se observa que o último caractere da sequência de busca foi calculado e um novo segmento da sequência alvo começa a ser armazenado nas UPs. Em (d) o novo segmento já está armazenado e a sequência de busca novamente flui pelo array sistólico, só que nesse estágio os elementos noroeste e oeste necessários para o cálculo são lidos da *CtFIFO*. A última UP novamente grava na *CtFIFO* os novos valores calculados. Esse processo se repete até o comando de fim de processamento ser decodificado pela unidade de processamento.

.

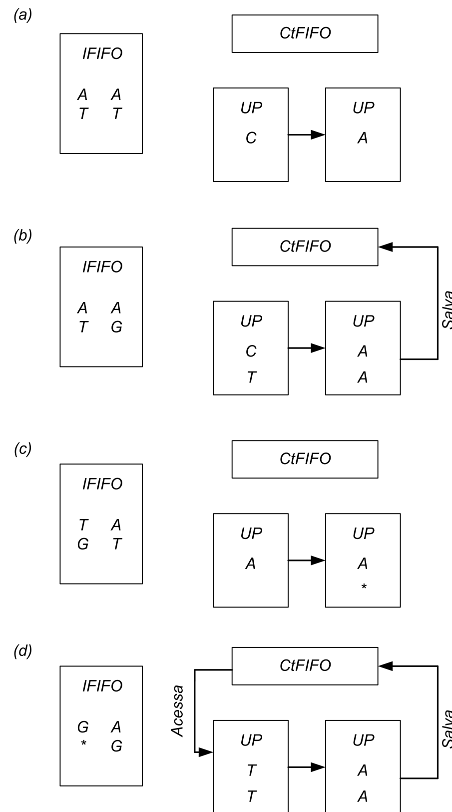


Figura 5.5: Exemplo da arquitetura projetada de particionamento de sequências

5.3 Protocolo de Comunicação

Para que possamos processar dados na arquitetura apresentada no Capítulo 5.2 necessitamos estabelecer como será o formato desses dados. Assim, optamos por projetar um protocolo de comunicação bastante simples. Esse protocolo possui três comandos básicos: (i) início de alvo, (ii) início de busca e (iii) fim do processamento. Dessa forma, são necessários 2 bits para representar esses 3 comandos. Os dados que processamos são sequências biológicas. Portanto, se desejamos trabalhar apenas com nucleotídeos (C, A, T, G), necessitamos de 2 bits para representa-los. Já se vamos processar aminoácidos (20 caracteres) são necessários 5 bits (o que permite que essa arquitetura processe nucleotídeos e aminoácidos). Como concatenamos os bits de comandos e de dados no protocolo, compactamos o tamanho total de bits para 6 bits, no caso de aminoácidos, e 3 bits no caso nucleotídeos usando o bit menos significativo para indicar se o dado é comando ou base.

O particionamento das sequências alvo, explicado no Capítulo 5.2, deve ser explicitado no protocolo. Assim, os dados já são previamente particionados de forma que o controle do hardware possa decodificá-los. Para armazenar os dados no formato do protocolo é necessário saber quantas unidades de processamento o hardware utiliza. Os 3 bits menos significativos são os responsáveis pelos comandos do protocolo. Assim, o valor *001* representa o início da sequência alvo, *101* o início da sequência de busca e *111* representa o fim do processamento.

Após definirmos o protocolo de comunicação, projetamos a máquina de estados mostrada na Figura 5.6 responsável por decodificar esse protocolo e controlar todo o processamento do hardware. Observamos 4 estados nessa máquina: *IDLE*, *LTARGET*, *COM-*

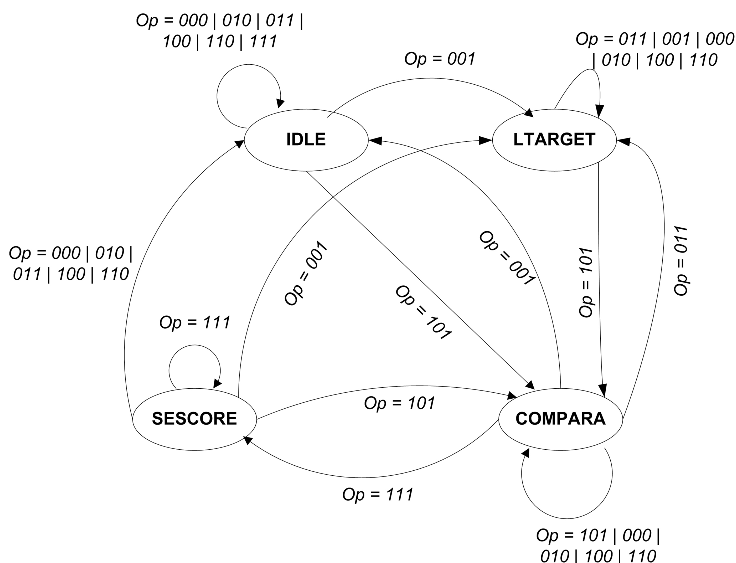


Figura 5.6: Máquina de estados que controla a arquitetura projetada

PARA, *SESCORE*. Inicialmente a máquina começa no estado de *IDLE*, e permanece nele até que seja verificada duas situações: (i) um comando de início de alvo ou (ii) um comando de início de busca. O primeiro caso indica que um novo cálculo de alinhamento está começando (carregando primeiro os caracteres da sequência de alvo e em seguida os caracteres da sequência de busca). Já o segundo caso, possibilita que uma sequência fixa de alvo possa ser comparada com um conjunto de sequências de busca (sem que a sequência de alvo seja carregada a cada comparação). Essa característica é muito útil no uso prático do algoritmo, visto que os usuários usualmente buscam a similaridade de uma sequência alvo contra um extenso bancos de dados de sequências já conhecidas. O estado *LTARGET* considera que todos os caracteres recebidos são de alvo e os armazena nas unidades de processamento. Isso ocorre até a máquina de estados receber o comando de início de busca. A partir desse comando, a máquina entra no estado *COMPARA*. Nesse estado considera que todos os caracteres recebidos são de busca e que as bases da sequência de alvo já foram previamente carregadas nas unidades de processamento. A máquina somente trocará de estado com: (i) um comando de nova sequência ou (ii) um comando de fim do processamento.

No primeira caso supramencionado, indica que ocorreu o particionamento de sequência. Então, os dados da unidade de processamento que foram salvos na *CtFIFO* durante o estado *COMPARA*, serão utilizados pela primeira UP no próximo estado *COMPARA* e a máquina retorna para o estado de *LTARGET*. Essa variação de estados entre *LTARGET* e *COMPARA* pode ocorrer repetidas vezes, até que toda a sequência de alvo tenha sido processada. O segundo caso supramencionado é quando o estado *COMPARA* identifica o fim do processamento. Então a máquina troca para o estado *SESCORE* que é o responsável por escrever o valor final na *OFIFO*. Após esse estado a máquina tem duas opções: (i) retornar ao estado *IDLE* ou (ii) retornar para o estado *COMPARA*. No primeiro caso aguarda um novo processamento. Já no segundo caso, recebe um comando de nova sequência de busca. Isso significa que se deseja comparar um conjunto de sequências com uma sequência alvo.

A principal vantagem do uso da máquina de estados e protocolo de comunicação projetados é que permitem particionar sequências grandes em um número menor de unidades

de processamento. Ainda, o desempenho do sistema não é afetado quando a sequência alvo não necessita ser particionada. Evidentemente que, caso seja necessário o particionamento, existe um aumento no número de ciclos para que as novas sequências de alvo sejam carregadas nas unidades de processamento que é detalhado no Capítulo 6.3. Outro custo desse sistema é que para cada particionamento da sequência alvo é necessário que toda a sequência de busca flua nas unidades de processamento - o que aumenta significativamente o fluxo de dados processados. Sabemos que a solução ótima consistem em ter o mesmo número de unidades de processamento que sequências de alvo. Porém, sabemos que isso não é factível com a tecnologia atual e com o aumento do tamanho das sequências. Assim, uma das diretrizes do nosso projeto foi não afetar o desempenho das sequências que são possíveis de ser calculadas sem ser particionadas e, caso seja necessário o particionamento, tornar o cálculo possível - acrescentando um aumento no número de ciclos e área.

5.4 Unidade de Processamento para a Distância Levenshtein

Conforme explicado no Capítulo 3.1 a distância Levenshtein é definida como o valor mínimo de transformações (apagar, substituir e inserir) necessárias para transformar a cadeia de caracteres de busca na cadeia alvo. Utiliza-se valores de penalidade fixos para cada tipo de transformação que são somados para cada caracteres da sequência. Assim, o valor final é considerado a distância entre as duas sequências. Dessa forma, sequências iguais possuem distância zero e quanto maior o valor da distância mais diferentes são as sequências.

Lipton e Lopresti (1985) observaram que, ao utilizar valores fixos e predefinidos para as transformações, é possível simplificar a equação da distância Levenshtein permitindo representar os elementos da matriz de similaridades com apenas 2 bits. Essa simplificação ocorre quando se utiliza como penalidade o valor *1* para inserir ou apagar e *2* para substituir. Observou-se que utilizando essa penalidade os valores de *b* e *c* na equação (3.4) podem ter apenas dois valores $a \pm 1$. Assim, a equação (3.2) pode ser simplificada e obtêm-se a equação (5.1).

$$\begin{cases} a & if\ ((b\ or\ c) = a - 1)\ or\ (S_i = T_j) \\ a + 2 & if\ ((b\ and\ c) = a + 1)\ and\ (S_i \neq T_j) \end{cases} \tag{5.1}$$

Utilizando a equação (5.1) se percebe que *b*, *c* podem ter apenas dois possíveis valores, *a+1* ou *a-1*. Analogamente *d* também é representado com dois valores *a* e *a+2*. Assim, com 2 bits podemos representar as 4 possibilidades de valores para o cálculo de d na matriz dinâmica. Para calcular a distância de edição entre as duas sequências um registrador é conectado a saída da última unidade de processamento do array sistólico. Esse registrador opera como um contador e é inicializado com o tamanho máximo da sequência de busca. Utiliza-se uma máquina de estados que verifica, com base nos valores de d da última UP, quando o contador deve ser incrementado ou decrementado. Após toda a sequência de busca ter fluido pelo array sistólico, o contador possui a distância de edição final.

A Tabela 5.1 mostra um exemplo do funcionamento da distância Levenshtein com a otimização proposta por Lipton e Lopresti (1985). Na Tabela 3.1 mostramos o mesmo exemplo porém sem utilizar a otimização. Analisando esses exemplos observamos que,

Tabela 5.1: Exemplo do algoritmo distância Levenshtein com otimização proposta por Lipton e Lopresti (1985)

*	*	*C*	*A*	*T*	*A*	*G*
*	0	1	2	3	0	1
A	1	2	1	2	3	0
T	2	3	2	1	2	3
A	3	0	3	2	1	2
G	0	1	0	3	2	1
C	1	0	1	0	3	2

independente do tamanho das sequências, utilizando a otimização do algoritmo é sempre possível representar todos os elementos da matriz de substituição com apenas 2 bits. Essa características é um dos grandes benefícios do uso dessa arquitetura, pois permite reduzir a área da UP e processar sequências de tamanho arbitrário. Observamos que utilizar 2 bits para representar os elementos da matriz só é possível utilizando a codificação para valor de inserir, substituir e remover fixa supramencionada, se modificarmos a pontuação teremos que aumentar a faixa de representação de valores (e consequentemente aumentar o número de bits).

Na Figura 5.7 apresentamos o *datapath* da unidade de processamento responsável por calcular a distância Levenshtein. Observamos que a unidade recebe como entrada os valores noroeste, norte e oeste e envia para a próxima UP o valor norte e o valor calculado (que serão reconhecidos como noroeste e oeste, respectivamente). O valor norte é sempre o valor calculado pela UP no ciclo de *clock* anterior (assim, já está armazenado na UP no início do cálculo). Utilizamos como estratégia de implementação comparar *b* com *c*, calcular o valor de *b+1* e compará-lo com *a+2*. Se *b* for igual a *c* e *a+2* for igual a *b+1* (e consequentemente igual a *c+1*) a porta lógica *AND* acionará o valor de *a+2* para a saída do *mux1*. Caso contrário, a saída será o valor de *a* devido ao fato do valor de busca e de alvo possuírem prioridade no cálculo efetuado pela unidade de processamento. Se alvo e busca são iguais é acionado o *mux2* com o valor de *a*. Se são diferentes é passado o valor da saída do *mux1*. Economizamos um comparador no projeto da UP calculando *b+1* e inferindo que se *b=c*, *b+1* é igual a *c+1*. Essa estratégia também permitiu reduzir o *fanout* do registrador que armazena o valor de a.

A última unidade de processamento tem sua saída conectada em uma unidade de controle que comanda um registrador, inicializado com o tamanho da sequência de alvo. A unidade de controle é a implementação da máquina de estados mostrada na Figura 5.8, responsável por traduzir os valores compactados da matriz no valor da distância de edição. O primeiro estado da máquina é denominado *IDLE* e espera a inicialização do registrador com o tamanho da sequência de alvo. Quando esse valor é inicializado, a máquina é direcionada para o estado referente ao valor codificado do registrador. Ou seja, se, por exemplo, o valor inicializado for 3 a máquina é direcionado para o estado *3*. Nesse estado espera o primeiro valor calculado (no próximo ciclo de *clock*). Se esse valor for menor que o estado atual, é decrementado o contador de saída e alterado para o estado menor. Caso o valor seja maior, é incrementado o contador e alterado para o estado maior. Após todos os elementos da última unidade de processamento serem calculados a máquina retorna para o estado *IDLE* e o registrador de saída é passado para unidade de

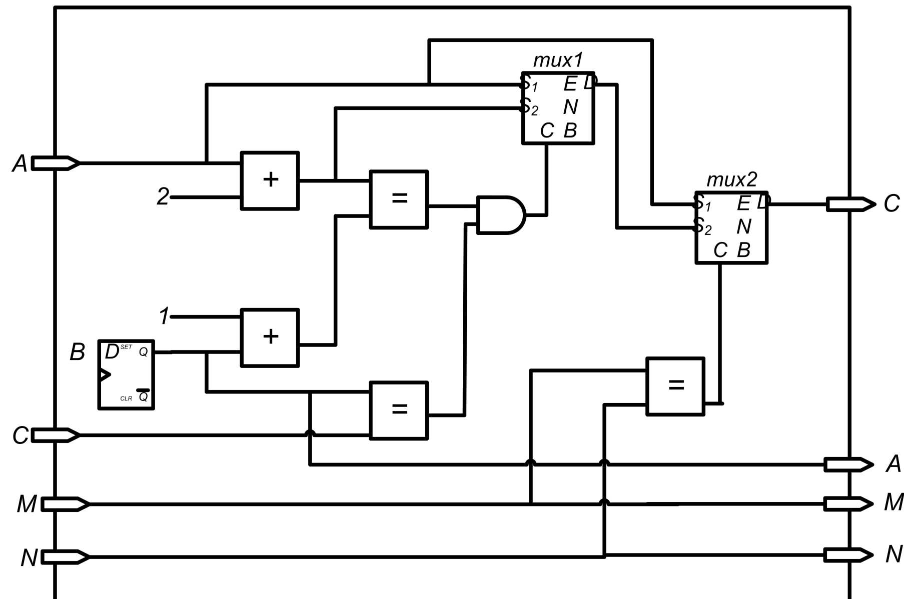


Figura 5.7: *Datapath* da unidade de processamento para a distância Levenshtein

controle mostrado no diagrama de blocos da Figura 5.4 para ser armazenado na *OFIFO*. Na Figura 5.8, *i* representa o valor codificado, *s=0* representa que o registrador deve ser decrementado e *s=1* que o registrador deve ser incrementado.

5.5 Unidade de Processamento para o Smith-Waterman

Com o intuito de reduzir o tamanho do circuito, optamos por não implementar as equações (3.7), (3.8) e (3.9) do algoritmo Smith-Waterman com *affine gap* diretamente em hardware. Observamos que, implementá-la diretamente, implica em utilizar 5 somadores. O primeiro somador é mostrado na equação (3.7), e soma o valor noroeste com a penalidade de match ou mismatch da matriz de similaridade. Os outros 4 somadores são mostrados nas equações (3.8) e (3.9), e são usados para abrir um *gap* ou prolongá-lo.

Nossa estratégia para reduzir o número de somadores se concentrou nos 4 somadores apresentados nas equações (3.8) e (3.9). Assim, observamos que a implementação da equação diretamente em hardware utilizaria dois somadores para o valor de *gap open* e outros dois somadores para o valor de *gap extend*, onde cada somador recebe além das penalidades de *gap* os elementos norte e oeste. Após as somas, é necessários dois comparadores que resultam o valor máximo entre cada uma das somas para o elemento norte e elemento oeste. A Figura 5.9 mostra o circuito que implementa diretamente as equações 3.8 e 3.9 em hardware retirado de Oliver et al. (2005a).

Para reduzir esse número de somadores, percebemos ser factível armazenar um histórico de *gaps* previamente abertos e, dessa forma, calcularmos somente a soma necessária. Para isso utilizamos um registrador de 2 bits que armazena 0, caso tenha ocorrido um *match*, 1 caso tenha ocorrido um *gap* na sequência alvo e 2 caso tenha ocorrido um *gap* na sequência de busca. Analisando o comportamento da matriz de similaridades constatamos que após a abertura de um *gap*, caso um novo *gap* seja aberto na próxima comparação, este será prolongado. O objetivo disso é penalizar de forma mais rigorosa a abertura de um *gap* e de forma menos rigorosa o seu prolongamento. Assim, muitos *gaps* abertos

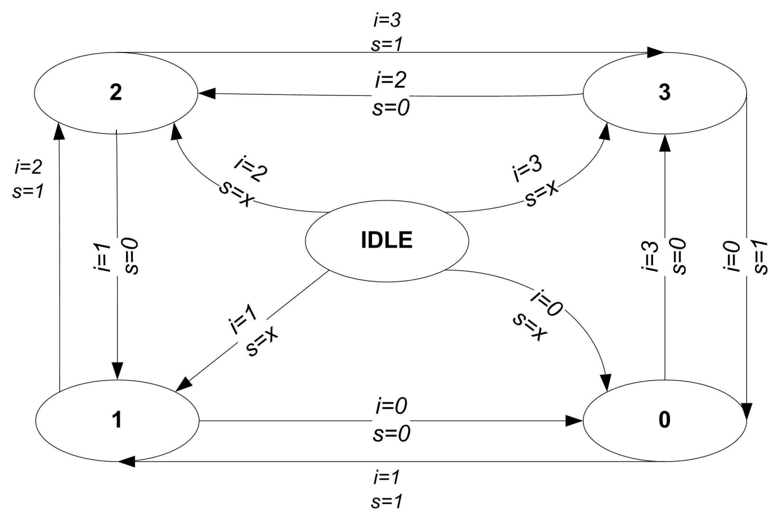


Figura 5.8: Máquina de estados que controla o decremento e o incremento do registrador com a distância de edição

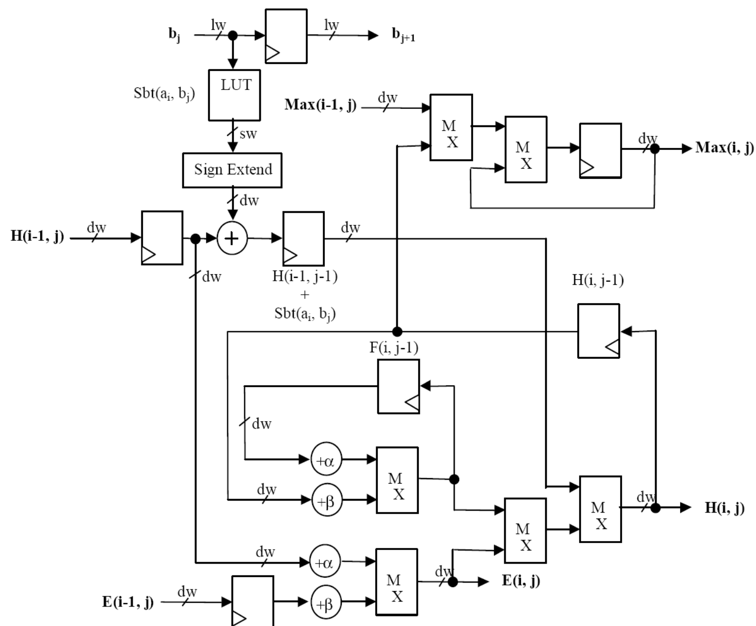


Figura 5.9: Unidade de processamento para o Smith-Waterman que implementa diretamente as equações do algoritmo, retirado de Oliver et al. (2005a).

em diferentes partes da sequência seriam penalizados de forma mais rigorosa que um *gap* aberto com diversos prolongamentos.

Por exemplo, considerando a matriz *H(i,j)* com os elementos noroeste *A*, norte *B*, oeste *C* e o elemento calculado *D*. Caso ocorra um *gap* na sequência de alvo e não tenha ocorrido esse *gap* no cálculo anterior (*i-1*): $D = B - g_o$. Isso corresponde ao *e(i,j)* ser o valor máximo na equação 3.7. Se no próximo cálculo da matriz *H(i,j)*, com *i+1* e considerando *C* o elemento noroeste, *D* o elemento norte, *E* o elemento oeste e *F* o elemento calculado, ocorrer novamente um *gap* na sequência de alvo, *F* será o máximo entre $B - 2.g_o$ e $B - g_o - g_e$. Assim, como $g_o > g_e$, $F = B - g_o - g_e$. Isso permite prever, caso tenha ocorrido um *gap* no cálculo anterior e o mesmo *gap* se repita, que a penalidade será de prolongamento. Ou, caso não tenha ocorrido um *gap* a penalidade será de abertura.

Percebe-se na UP proposta por Oliver et al. (2005a), mostrada na Figura 5.9, os 4 somadores para as penalidades de *gap open* e *gap extend* conectados a um circuito que verifica qual dos valores é o maior (ambos utilizando 16 bits). Nossa estratégia, além de remover 2 somadores de 16 bits, também remove do circuito esses dois comparadores de máximo entre as somas, pois somente é calculada a soma que será utilizada no restante do algoritmo. Assim, utilizamos um registrador de 2 bits que armazena se um *gap* foi aberto, prolongado ou se houve um *match* (nenhum gap foi aberto). Com isso, substituímos os 4 somadores de 16 bits e dois comparadores de máximo também de 16 bits por um registrador de 2 bits, 2 somadores de 16 bits, 2 comparadores de 2 bits e dois multiplexadores de duas entradas.

A Figura 5.10 mostra o diagrama de blocos da unidade de processamento projetada. O valor de *M* e *N* são carregados na UP e utilizados como endereço para acessar o valor de penalidade para *match* ou *mismatch* da matriz de substituição. Esse valor é carregado em um somador de 16 bits junto com o valor de entrada que representa o elemento noroeste (*A*). O valor de entrada *Y* representa o escore obtido pela unidade de processamento anterior, é comparado com o escore calculado nessa unidade e o maior entre eles é passado para a próxima UP. Isso garante que no fim dos cálculos o maior escore estará na última UP (para então ser armazenado na OFIFO). A *LUT S(i,j)* é uma *lookup table* que armazena a coluna da matriz de substituição que corresponde ao caractere M armazenado no registrador. O bloco em destaque mostra a otimização que realizamos nas equações para implementá-las em hardware, conforme descrito no parágrafo anterior. O *Mux01* recebe como controle o valor de histórico de *gap* referente ao ciclo de *clock* anterior e habilita a penalidade correspondente. Esse valor é somado com o valor calculado na UP no ciclo de *clock* anterior (que representa o elemento norte da matriz de similaridade). Analogamente, o *Mux02* realiza a mesma operação porém recebe o valor de *gap* da UP anterior e realiza a soma com o valor de entrada referente ao elemento oeste. O bloco *Gmax* verifica qual dos três elementos previamente calculados é o maior e armazena nos registradores esse valor e o *status* do valor de *gap* - aberto, prolongado ou se ocorreu um *match*. Ao fim do processamento ocorre um *shift* entre os registradores encadeados para atualizar os dados na próxima UP.

Figura 5.10: Unidade de processamento projetada para o Smith-Waterman

6 RESULTADOS E COMPARAÇÕES

Nesse capítulo apresentamos a metodologia utilizada para prototipar as arquiteturas descritas no capítulo 5. Analisamos o desempenho da arquitetura que implementa o Smith-Waterman com *affine gap* com uma versão idêntica do algoritmo em software. Ainda, mostramos um comparativo entre os resultados que encontramos e os principais resultados apresentados pelas arquiteturas estudadas no Capítulo 4. Exploramos duas tecnologias distintas para prototipar as arquiteturas projetadas: FPGA e ASIC.

A tecnologia FPGA consiste de um circuito integrado configurado pelo usuário após a fabricação. O dispositivo contém diversos elementos lógicos e alguns *cores* para uso específico - memórias, controlador PCI Express, DSP etc. A configuração do dispositivo é realizada utilizando uma linguagem de descrição de hardware (HDL), que após as etapas de síntese, posicionamento e roteamento é transformada em um arquivo de configuração para o dispositivo. Analogamente, utilizamos um fluxo *standard cell* para a tecnologia ASIC. A entrada do circuito foi uma descrição em HDL e um *design kit* com a biblioteca de células e parâmetros da tecnologia. Após as etapas de projeto obtemos uma descrição em nível de transistores, já posicionada e roteada estando pronta para ser fabricada.

Algumas diferenças importantes entre as duas tecnologias. Utilizando tecnologia ASIC, em geral, obtêm-se menor potência, menor área e maior desempenho do que utilizando FPGAs. Isso ocorre devido ao fato do dispositivo FPGA já estar fabricado: os elementos lógicos do FPGA permitem que qualquer função lógica possa ser implementada neles - o que os torna maiores e mais lentos que uma porta lógica equivalente em ASIC. Porém, se compararmos FPGAs que utilizam processos de fabricação mais modernos que um processo utilizado em ASIC essas diferenças tendem a diminuir bastante - e em alguns casos até obtêm-se melhores resultados com FPGAs. Isso é bastante comum pois é relativamente barato utilizar um FPGA que utilize um processo de fabricação estado da arte. Porém, fabricar um ASIC nesse processo, exigiria um mercado com grande escala (processador de propósito geral, por exemplo).

Nesse trabalho utilizamos um processo de fabricação ASIC de 180 nm e nosso dispositivo FPGA alvo foi fabricado utilizando um processo de 65 nm. Para comparar as diferenças entre esses processos de fabricação ilustramos com o exemplo dos processadores de propósito geral da Intel. O processo de 180 nm foi utilizado na fabricação do Pentium III lançado em Março de 2000. Já o processo 65nm foi utilizado na fabricação do processador Core 2 Duo lançado em Agosto de 2006.

Em relação ao custo de fabricação, observa-se que quando existe pouca demanda o uso de dispositivos FPGAs é recomendado. Isso se deve, principalmente, ao alto custo de elaboração das máscaras no projeto de fabricação ASIC. Esse é um custo fixo, que

somente é amortizado com uma grande produção - que, para esses casos, faz com que circuitos ASICs possuam menor custo em relação a FPGAs.

Uma vantagem importante do FPGA em relação ao ASIC é a reconfiguração. Devido as características do dispositivo (já ter sido fabricado) podemos carregar diversos projetos nele. Assim, ele permite que implementemos circuitos menos genéricos e mais específicos para determinada tarefa, já que sabemos que podemos reconfigurá-lo, inclusive, em tempo de execução. Para as arquiteturas projetadas nesse trabalho essa característica é bastante interessante. Podemos criar arquiteturas distintas para comparar nucleotídeos ou proteínas, ou para sequências de até 500, 1000, 2000, 5000 caracteres. Isso permite que o circuito seja otimizado para determinado tipo de dados. Se soubermos que uma sequência tem, por exemplo, no máximo 500 caracteres de nucleotídeos utilizamos menos memória e registradores menores. Assim, podemos integrar mais unidades de processamento em paralelo e melhorar o desempenho do sistema. Em ASIC é preciso sempre pensar no pior caso e projetar o circuito visando o mesmo.

A prototipação da arquitetura inicia com a sua descrição em alguma linguagem de descrição de hardware. Optamos por descrever o hardware utilizando VHDL. Todo o projeto em VHDL das arquiteturas foi pensado para ser customizado, em tempo de projeto, utilizando estruturas da linguagem como o *generic*, *for generate* e *constant*. Assim, podemos definir antes da prototipação as características gerais do circuito: número de unidades de processamento em paralelo, tamanho do dados processados (nucleotídeo ou proteína) e tamanho das memórias. Isso permite que adequemos o circuito para determinado tipo de comparação que desejamos realizar. A descrição em VHDL das arquiteturas é utilizada como entrada nos fluxos de projeto FPGA e ASIC. A principal adequação para cada um dos casos é em relação as memórias. Adotamos políticas diferentes para cada um dos fluxos descritas nos capítulos 6.1 e 6.2.

Após descrever o circuito em VHDL partimos para a verificação da funcionalidade do mesmo. Inicialmente realizamos a verificação da arquitetura para a distância Levenshtein com a otimização proposta no Capítulo 5.4. Assim, projetamos um software que realiza o cálculo com a otimização da mesma forma que o hardware. Validamos logicamente esse software projetado com implementações do algoritmo apresentadas em Kleiweg (2009) e Gilleland (2009) usando diversos casos de teste. Em seguida utilizamos esses casos de teste para comparar logicamente com os valores de saída do hardware. Para extrairmos esses valores do hardware utilizamos a ferramenta de simulação lógica ModelSim. Para o Smith-Waterman o fluxo de verificação foi idêntico, com a exceção que utilizamos uma versão em Matlab do Smith-Waterman (do pacote *bioinformatics*) para realizar a verificação. A Figura 6.1 mostra um esquemático do fluxo de projeto utilizado no desenvolvimento das arquiteturas apresentadas neste trabalho.

6.1 Prototipando para FPGA

A tecnologia FPGA permite que rapidamente um circuito projetado em alguma linguagem de descrição de hardware possa ser implementado e testado no silício. O fluxo que adotamos nesse trabalho utiliza as ferramentas da empresa Xilinx. Existe uma grande diferença entre apenas realizar todos os processos de síntese para determinada arquitetura e testar o dispositivo na prática. Ao testar o dispositivo após a síntese é muito comum encontrar problemas que não são detectados nas etapas de simulação. Isso exige um pro-

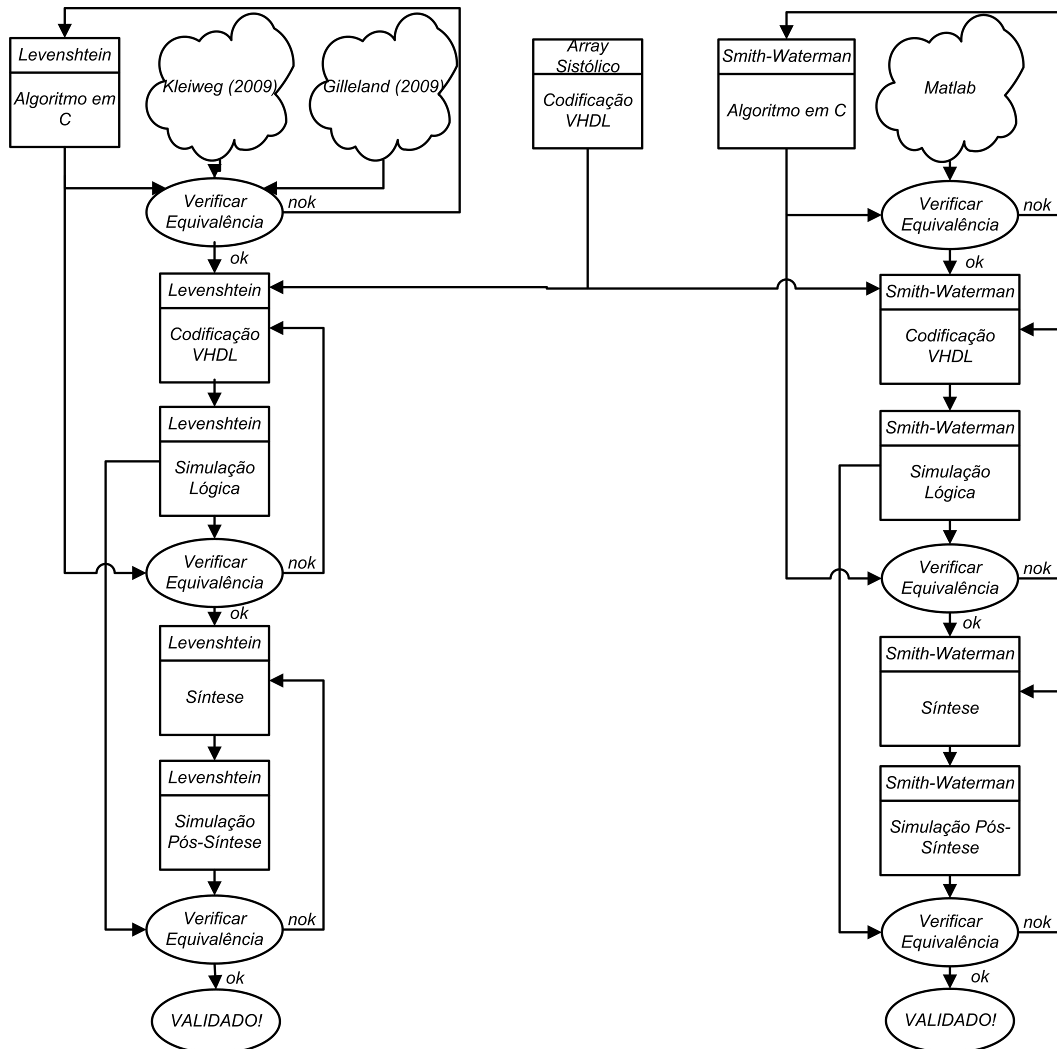


Figura 6.1: Fluxo utilizado no projeto das arquiteturas.

cesso longo, onde a cada possível solução para um problema encontrado todo o processo de síntese seja refeito. Assim, para o escopo desse trabalho, definimos que as duas arquiteturas seriam sintetizadas porém apenas a arquitetura do Smith-Waterman com *affine gap* seria testada em silício (visto que essa será posteriormente implementada em ASIC).

Utilizamos como placa de prototipação a HTG-V5-PCIE-330-1 da empresa Hightech Global. Essa placa possui, entre outras características, um dispositivo FPGA Virtex-5 modelo XC5VLX330T-1, interface PCI Express com 8 lanes, cristal de *clock* e interface para memória RAM. O dispositivo FPGA embarcado nessa placa é de uma família que tem como características privilegiar ao máximo o uso de lógica dedicada. Isso faz com que ele não possua, por exemplo, processadores dedicados no FPGA. Assim, da família Virtex-5 é um dos dispositivos que permite maior integração. Esse dispositivo possui cerca de 11,5 Mbits de memória interna denominada BRAM. Ainda, possui um *core* dedicado para comunicação do FPGA com o meio externo utilizando o barramento PCI Express.

Para a arquitetura da distância Levenshtein, utilizamos os seguintes tamanhos para as memórias do circuito: IFIFO(1.048.576 posições de 6 bits), OFIFO (32.768 posições de 32 bits) e CtFIFO (524.388 posições de 4 bits). Utilizamos a ferramenta Core Generator para gerar essas três memórias. A ferramenta permite configurar os parâmetros das memórias e gera um arquivo *.NGC* e um *.VHD*. O primeiro possui as informações sobre o *core* físico do FPGA que será utilizado (esse arquivo deve ser carregado na síntese) e o segundo é utilizado apenas nas simulações em nível lógico.

Implementamos dois projetos distintos denominados LProt e LNucleo. O primeiro com 7000 unidades de processamento para calcular sequências de proteínas e o segundo com 8000 unidades de processamento para calcular nucleotídeos. Como dito anteriormente, é possível integrar mais unidades de processamento no projeto LNucleo devido a sequências de nucleotídeos necessitarem de apenas dois bits para serem representadas; ao contrários dos 5 bits necessários para sequências de proteínas. Em ambos os projetos podemos calcular sequências com até 524.388 caracteres, resultando em 75 e 66 quebras da sequência de alvo para proteínas e nucleotídeos respectivamente.

Já para a versão do Smith-Waterman com *affine gap* utilizamos os seguintes tamanhos para as memórias do circuito: IFIFO(1.048.576 posições de 6 bits), OFIFO (32.768 posições de 32 bits) e CtFIFO (65536 posições de 32 bits). Implementamos um projeto denominado SWAffine com 650 unidades de processamento que permitem calcular sequências de proteínas com tamanho máximo de 65536 caracteres, resultando em 101 quebras da sequência de alvo. Apesar dessa arquitetura ter como objetivo principal comparar proteínas mapeamos também nucleotídeos nos 5 bits utilizados para representação dos dados; devido ao fato de sobrarem valores nessa faixa de representação. Isso faz com que essa arquitetura também possibilite a comparação de nucleotídeos.

As ferramentas de prototipação da Xilinx permitem que todo o fluxo possa ser realizado no ambiente ISE. Porém, optamos por desenvolver um script próprio para realizar a etapa de prototipação. Dessa forma, podemos facilmente controlar o que está ocorrendo em cada etapa do fluxo e também configurar as opções individuais para cada ferramenta integrante do pacote ISE - utilizamos a versão 11.1.

Inicialmente executamos a ferramenta Xst que a responsável por transformar a descrição em VHDL em uma descrição de *gates* equivalentes - nesse caso um arquivo do tipo *.NGD*. A próxima ferramenta utilizada é o Ngbuild que utiliza a entrada *.NGD* e o

Tabela 6.1: Dados da síntese de uma UP para o dispositivo XC5VLX330T-1

Projeto	*Slices*	*Alvo/Busca (bits)*	*MS (bits)*
LNucleo	6	2	2
LProt	7	2	2
SWAffine	76	5	16

arquivo *.UCF* (com informações sobre a localização dos pinos na placa de prototipação, sinal de *clock* e as *constraints* do projeto). A ferramenta executa diversas análises de DRC (*Design Rule Check*) no *netlist* e gera outro arquivo *.NGD*. Em seguida executamos a ferramenta Map, que é responsável por mapear o projeto para o dispositivo FPGA selecionado utilizando as estruturas de cada dispositivo - Slices, BRAMs etc. Na ferramenta Map também é realizado o posicionamento do circuito no dispositivo FPGA, outra etapa de DRC e gerado um arquivo *.NCD*.

A ferramenta seguinte, denominada Par, realiza o roteamento do circuito até que que as *constraints* descritas no arquivo de entrada no formato *.UCF* sejam alcançadas. Após essa etapa utilizamos a ferramenta Trace para fazer a análise de *timing* e a ferramenta Netgen para gerar um *netlist* do tipo *.SDF* (com as *constraints* de síntese) que possa ser simulado novamente no ModelSim para que seja verificada a equivalência lógica (agora com uma descrição do circuito mais próxima da realidade). A última etapa do fluxo executa a ferramenta Bitgen que gera o *bitstream* que será carregado no FPGA utilizando o software Impact por meio do cabo USB/JTAG conectado no PC e na placa FPGA.

Na Tabela 6.1 mostramos o tamanho da unidade de processamento de cada projeto. Observa-se que existe uma diferença de 1 slice entre as UPs do LProt e LNucleo devido a diferença do tamanho dos dados de alvo e busca de cada um (mesmo possuindo o mesmo tamanho na matriz de similaridade). Já a versão SWAffine ocupa mais de dez vezes o tamanho da versão LProt devido ao fato de trabalhar com uma matriz de similaridade de 16 bits e realizar um processamento mais complexo descrito no Capítulo 5.

Após estimarmos o tamanho de cada unidade de processamento implementamos o circuito completo com os dados mostrados na Tabela 6.2. Ambos os projetos ocupam cerca de 80% do dispositivo utilizando quase 90% das memórias BRAM disponíveis. O uso do dispositivo foi planejado por meio da síntese de cada UP separadamente (mostrado na Tabela 6.1) e as estimativas de uso da memória fornecido pela ferramenta Core Generator. Para os projetos LProt e LNucleo obteve-se a mesma frequência de operação por possuírem exatamente o mesmo caminho crítico. O projeto SWAffine possui um caminho crítico bem maior por ter uma unidade de processamento bem mais complexa que a utilizada nos projetos LProt e LNucleo - por isso a frequência de operação é mais de 2,5 vezes menor.

6.1.1 Utilizando o barramento PCI Express

O padrão PCI Express (PCIe) é a evolução dos antigos barramentos PCI e PCI-X. Possui alta performance, arquitetura de interconexão de propósito geral e foi projetado para uma grande variedade de plataformas computacionais e de comunicação. Sua arquitetura é baseada em pacotes, com uma interface serial ponto a ponto que é retro-compatível com

Tabela 6.2: Dados da síntese dos projetos para o dispositivo XC5VLX330T-1

Projeto	*UPs*	*Slices - %*	*BRAM - %*	*Freq. (MHz)*
LNucleo	8000	41309 - 79%	286 - 88%	376
LProt	7000	42900 - 82%	286 - 88%	376
SWAffine	650	42514 - 82%	279 - 86%	140

Tabela 6.3: Desempenho do barramento PCIe usando DMA

Transação	*TLPs*	*Throughput (MB/s)*	*Tempo (ms)*
Escrita	8192	204	5,13
Leitura	2048	183	1,43

os padrões PCI e PCI-X. O PCIe pode utilizar mais de um *lane*, o que faz com que a velocidade de transmissão seja diretamente proporcional ao seu número. Assim, um *lane* do barramento pode transmitir dados em até 2 Gb por segundo - 8 *lanes* transmitem dados em até 16 Gb por segundo.

Alguns FPGAs da família Virtex-5, como é o caso do XC5VLX330T-1, possuem um *core* denominado de *Endpoint Block* dedicado no FPGA para a comunicação utilizando o barramento PCIe. Esse bloco possui a implementação das principais camadas do barramento PCIe e interface para os *lanes*. O *core* utiliza dois sinais de *clock*, um denominado de *core clock* que é de 200 Mhz e outro que é o *clock* do usuário de 62.5 MHz. A comunicação externa com esse bloco ocorre por meio da aplicação do usuário (em software executado em um computador) e as BRAMs para recebimento e transmissão de dados.

Assim, utilizando um *drive* apropriado é possível se comunicar com o FPGA por meio de DMA (Acesso Direto a Memória), realizando escrita e leitura nas BRAMs. É necessário projetar um *wrapper* em HDL para conectar o driver em software aos sinais do módulo PCIe. Adotamos a ferramenta Windriver, fornecida pela empresa Jungo, para gerar o driver de comunicação entre um computador e o dispositivo FPGA. Essa ferramenta permite gerar drives PCIe para o sistema operacional linux ou windows. O driver gerado para a arquitetura alvo permite que, por meio de algumas APIs, seja possível enviar e receber dados por DMA utilizando o padrão do barramento PCIe.

A unidade mínima do barramento é denominada DWORD e é composta por 4 bytes. As DWORDS são enviadas em conjuntos denominados TLP. O tamanho de um TLP é de 32 DWORD. A IFIFO do projeto Smith-Waterman com *affine gap* tem tamanho 1048576X6. Assim enviamos para preencher a IFIFO 8192 TLPs e recebemos 2048 TLPs da OFIFO. Testamos o desempenho dessa arquitetura de envio e recebimento de dados por DMA através da PCIe utilizando o Windriver e obtivemos a Tabela 6.3. Apesar de nossa placa possuir 8 *lanes* conseguimos habilitar apenas 1 *lane*. Os testes foram realizados em um computador com processador Intel Core 2 Duo de 2.93 GHz, memória RAM de 2 GB e sistema operacional linux Ubuntu 9.04.

6.2 Prototipando para ASIC

Utilizamos um fluxo de projeto denominado *standard cell* para prototipar a arquitetura projetada utilizando a tecnologia ASIC. Por se tratar de um processo mais complexo que a síntese utilizando FPGAs optamos por prototipar apenas a arquitetura para o Smith-Waterman com *affine gap* nessa tecnologia.

Para todo o fluxo de projeto ASIC utilizamos as ferramentas de projeto da empresa Cadence. Ainda, um projeto *standard cell* necessita de um *design kit* que contém todas as informações sobre as bibliotecas de células utilizadas e a tecnologia do processo. Utilizamos o *design kit* SAGE da empresa TSMC com tecnologia de 180 nm.

Dividimos as etapas do projeto de ASIC em duas: *front end* e *back end*. Na primeira etapa transformamos uma descrição em VHDL comportamental para um *netlist* de *gates* equivalentes. Assim, o final dessa etapa resulta em um circuito logicamente equivalente a descrição VHDL projetada.

Na etapa seguinte, utiliza-se como entrada o circuito gerado pela síntese lógica e aplica-se a biblioteca de células equivalente a cada elemento lógico do circuito. Assim é criado o leiaute físico do circuito com a caracterização elétrica de cada bloco. Em seguida, esses blocos são posicionados na área de silício de forma que possa se obter o melhor roteamento entre eles. Essa etapa é realimentada constantemente, ou seja, altera-se o posicionamento até se obter o melhor roteamento.

Em cada uma das etapas podemos obter relatórios sobre o desempenho esperado do circuito em relação a área, tempo e potência. Na prática esses relatórios são sempre estimativas sobre o desempenho do circuito e, em geral, se aproximam da realidade física quanto mais avançamos nas etapas. Por exemplo, o relatório de tempo que informa o caminho crítico do circuito (o maior atraso). Esse caminho crítico define qual será o período do *clock* do circuito, já que o período do *clock* não pode ser menor que o caminho crítico do circuito. Assim, no primeiro relatório de tempo, na síntese lógica, o caminho crítico é apenas uma estimativa, pois as etapas de posicionamento e roteamento ainda não foram realizadas e não existe informações sobre o impacto do caminho crítico após o roteamento (apenas estimativas).

Nas próximas seções descrevemos como foi realizada cada uma dessas etapas para prototipar o circuito mostrado no Capítulo 5, com a unidade de processamento apresentada no Capítulo 5.5. Assim, utilizamos os seguintes tamanhos para as memórias do circuito: IFIFO(32 posições de 3 bits), OFIFO (32 posições de 32 bits). Implementamos 64 unidades de processamento para calcular o Smith-Waterman com *affine gap* para sequências de nucleotídeos. Não implementamos a quebra de sequência nem matrizes de substituição nessa arquitetura, fazendo com que seja calculado no máximo sequências com 64 caracteres.

6.2.1 Front End

A primeira etapa da síntese consiste em elaborar o projeto usando a ferramenta RTL Compiler da Cadence. As duas próximas etapas no fluxo consistem em mapear o projeto, inicialmente para um conjunto de portas genéricas e posteriormente para a biblioteca alvo. Nessa etapa a ferramenta necessita de *constraints* do projeto, como frequência de operação e atraso máximo dos pinos de entrada e saída em relação ao sinal de *clock*. Definimos essas *constraints* em um arquivo do tipo *.SDC* (Synopsys Design Constraints) que

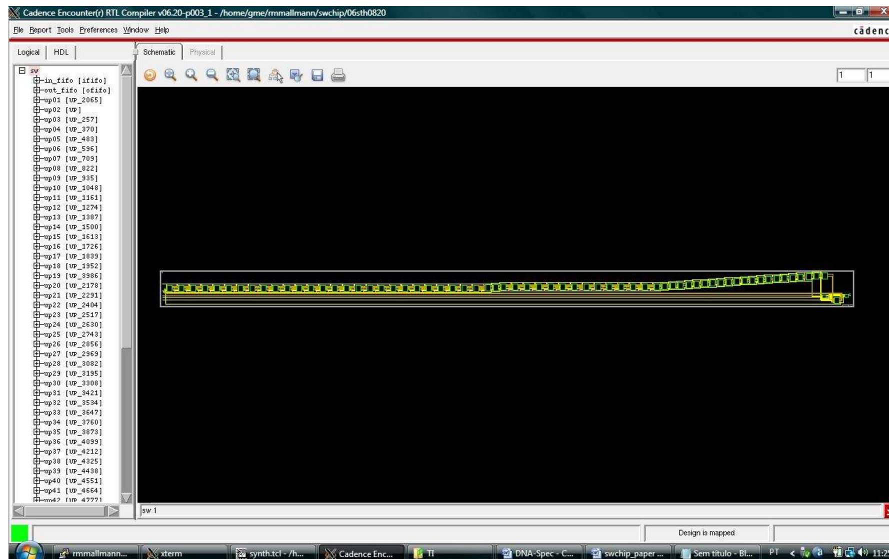


Figura 6.2: Top-level do projeto após síntese lógica

Tabela 6.4: Relatório de área após síntese lógica

Módulo	*Células*	*Área da Célula* (mm^2)	*Área da Rede* (mm^2)
Controle	81	0,041	0,003
IFIFO	200	0,048	0,012
OFIFO	202	0,071	0,014
UP	≈ 1419	$\approx 0,336$	$\approx 0,129$
Total	89899	21,660	15,590

é chamado pelo script de síntese. Utilizamos um sinal de *clock* de 100 MHz, resultado esse obtido após diversas tentativas de síntese que obtiveram *slack* negativo na análise de *timing* do caminho crítico. Definimos o atraso máximo dos pinos de entrada e saída como 20% do período de *clock*, baseado na literatura (WESTE et al., 2005). A Figura 6.2 mostra o *top-level* do projeto mapeado e a Figura 6.3 a conexão entre as UP.

As Tabelas 6.4, 6.5 e 6.6 mostram os resultados de área, potência e atraso obtidos após a síntese lógica. Entendemos que esses resultados ainda não são os definitivos, visto que as etapas de posicionamento e roteamento do circuito realizadas na síntese física modificam esses valores. Porém, é uma estimativa para verificarmos se os resultados estão de acordo com o esperado para uso após a fabricação. Em relação à área utilizada pelo circuito percebemos, como esperado, que as UP's são os elementos que mais ocupam área. O valor apresentado na Tabela 6.4 para área ocupada pela UP é referente a uma UP. Esse valor é aproximado porque, durante a síntese, as 64 UPs apresentaram número de células ligeiramente diferentes. A Tabela 6.5 mostra as estimativas de potência calculadas pela ferramenta. Esse resultado é o mais variável, comparado com o atraso e a área, devido ao fato de os *pads* de I/O, inseridos na síntese física, modificarem esse valor. Na Tabela 6.6 é mostrado o caminho crítico, encontrado pela ferramenta, entre o registrador que armazena a sequência de busca da UP 52 e o registrador que armazena o valor entre a comparação de máximos. O *slack* positivo mostrado significa que não houve violações de tempo.

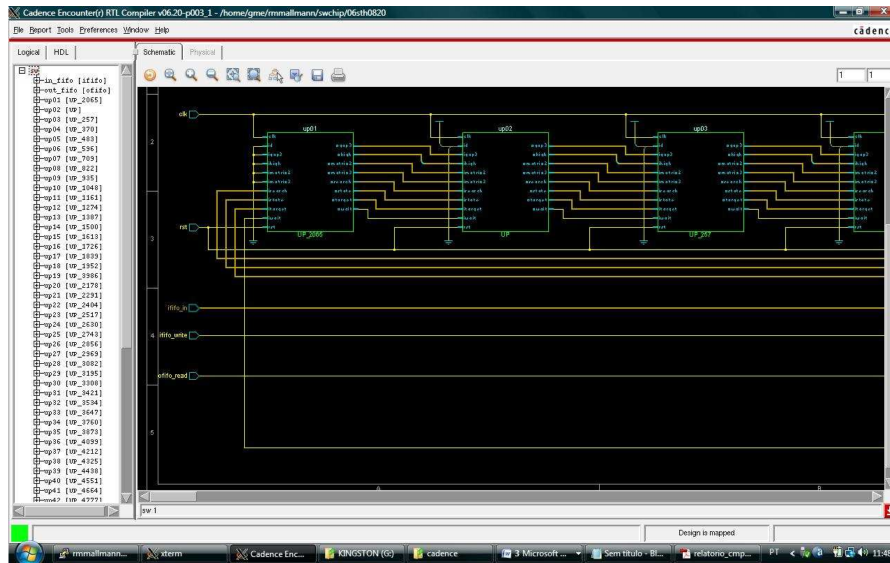


Figura 6.3: Detalhe mostrando a conexão entre UPs após síntese lógica

Tabela 6.5: Relatório de potência após síntese lógica

Leakage (W)	*Internal Power (W)*	*Switching Power (W)*
9,36 u	0,18	0,26

Tabela 6.6: Caminho crítico determinado após síntese lógica

Ponto Inicial	*Ponto Final*	*Slack (ps)*
Up52/Busca	Up53/Máximo[5]	885

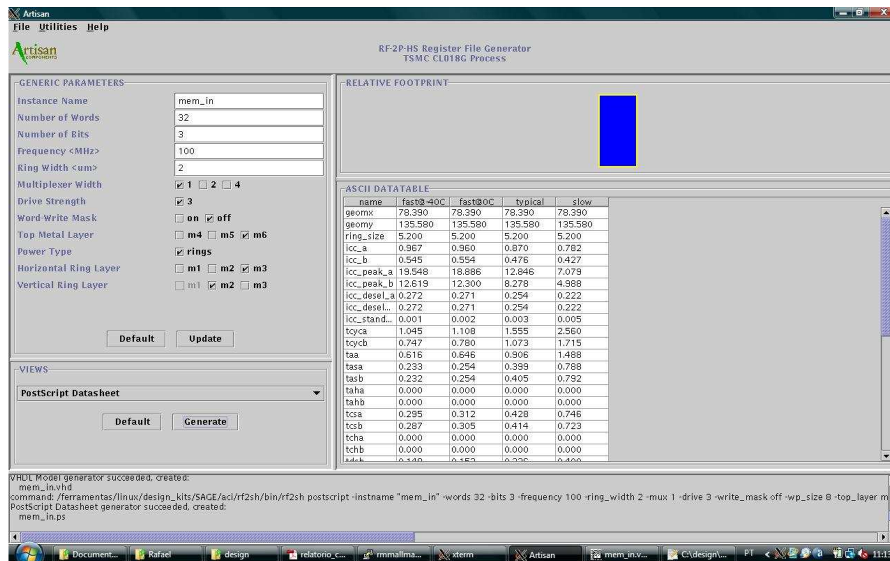


Figura 6.4: Gerador de memória fornecido pela empresa Artisan

Utilizamos a ferramenta LEC para realizar a verificação lógica. Essa verificação consiste em comparar os arquivos em VHDL do projeto (pré-síntese) com o *netlist* gerado pelo RTL Compiler pós-síntese lógica. Para realizar essa verificação, é necessário adicionar alguns comandos ao script que executa o RTL Compiler.

Ao simularmos o dispositivo no NCLaunch verificamos que a FIFO que havíamos sintetizada (descrita em VHDL, utilizando uma estrutura de vetor bidimensional) não operava corretamente. Percebemos, então, que o RTL Compiler não sintetiza corretamente vetores bidimensionais. Assim, utilizamos um gerador de memórias disponibilizado pela empresa Artisan mostrado na Figura 6.4 (para o design kit que estamos utilizando). Com esse gerador geramos dois *Register Files* (RTL, arquivos *.lib* e *.lef*) e projetamos uma lógica de entrada e saída para que esse dispositivo operasse como uma FIFO.

Assim, experimentamos duas abordagens. A primeira consistiu em sintetizar junto com o projeto o arquivo RTL gerado pelo gerador de memória. Após sintetizar o projeto verificamos que a ferramenta não sintetizava corretamente o *netlist* da memória. Por fim, consideramos a memória como uma *black box* do projeto. Tentamos gerar o leiaute do *black box* utilizando o gerador de memória, mas essa opção não está habilitada pelo *design kit*. Por fim, executamos novamente a verificação lógica com o LEC e simulamos o resultado da síntese lógica no NClaunch considerando a memória como um *black box*.

6.2.2 Back End

Utilizamos a ferramenta SoC para realizar a etapa de síntese física. Inicialmente carregamos o *netlist* do projeto, as *constraints* no formato *.SDC*, os arquivos *.lef* (com seis níveis de metal) e os arquivos *.lib*. Após carregarmos o projeto, verificamos que todos os módulos são representados por caixas mostrado na Figura 6.5. A próxima etapa é o *floorplaning*, onde definimos a localização de cada uma dessas caixas na área do chip. Definimos também nessa etapa um tamanho para as bordas de entrada e saída.

Em seguida, definimos os pinos de gnd e vcc e configuramos os anéis e os *stripes* de alimentação (Figura 4.5.1). Nesse estágio o *die* está pronto para ser roteado. Utilizamos o comando *Sroute* para rotear os sinais de *vdd* e *gnd*.

Por fim, chegamos à etapa de geração da árvore de *clock*, posicionamento das células e roteamento. Essas duas últimas etapas foram as mais custosas da síntese física em relação ao tempo de processamento (cerca de 1 hora e 30 minutos em uma workstation Sun). O primeiro relatório de *timing*, após o posicionamento e roteamento, mostra algumas violações. Após executar um posicionamento incremental (otimizações no posicionamento) e repetir o roteamento eliminamos todas as essas violações. A Figura 6.6 mostra a visão do posicionamento das UP's projetadas utilizando a ferramenta Amoeba View. Percebemos que toda a regularidade espacial apresentada anteriormente na Figura 6.3 é perdida no intuito de obter um melhor roteamento. Na Figura 6.7 é mostrada a versão final do chip posicionado e roteado (com a árvore de *clock* em destaque).

A Tabela 6.7 mostra o resultado de área final obtido. Percebemos um aumento de 74,7% em relação à estimativa apresentada nos relatórios de síntese lógica. A Tabela 6.8 mostra os resultados de potência após a síntese física. Percebemos uma variação menor nesses valores em relação à grande variação de área apresentada. Entretanto, esse valor ainda não apresenta o real consumo de potência visto não termos inseridos os pads de entrada e saída (o design kit escolhido não permitia essa opção).

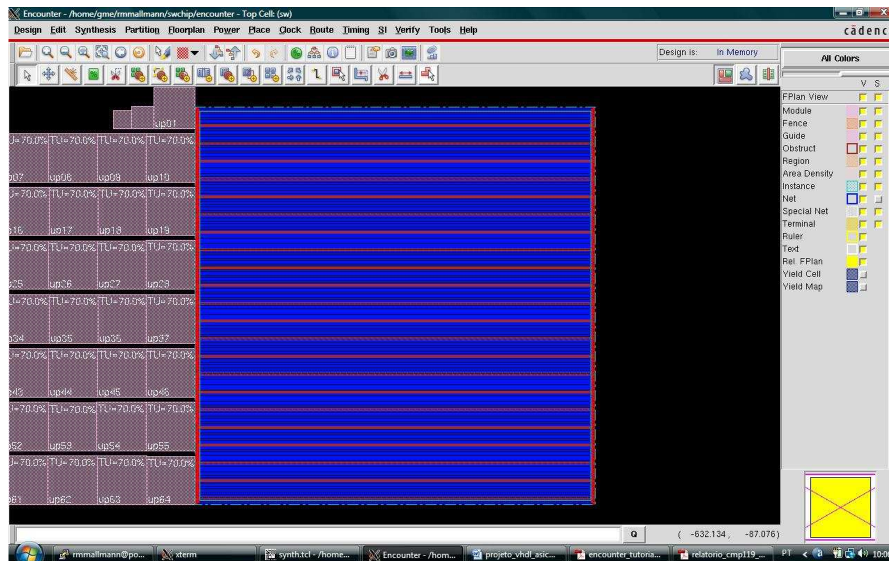


Figura 6.5: Anéis e stripes de alimentação configurados

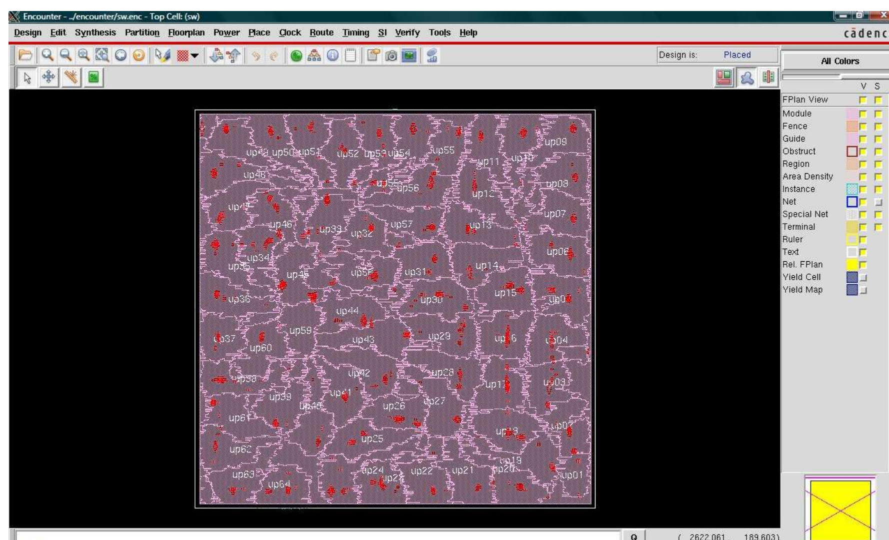


Figura 6.6: Visão do posicionamento dos blocos projetados utilizando o Amoeba View

Tabela 6.7: Relatório de área após síntese física

Módulo	*Área* (mm^2)
Total	65,08

Tabela 6.8: Relatório de potência após síntese física

Leakage (W)	*Internal Power (W)*	*Switching Power (W)*	*Total Power (W)*
9,30 u	0,23	0,08	0,31

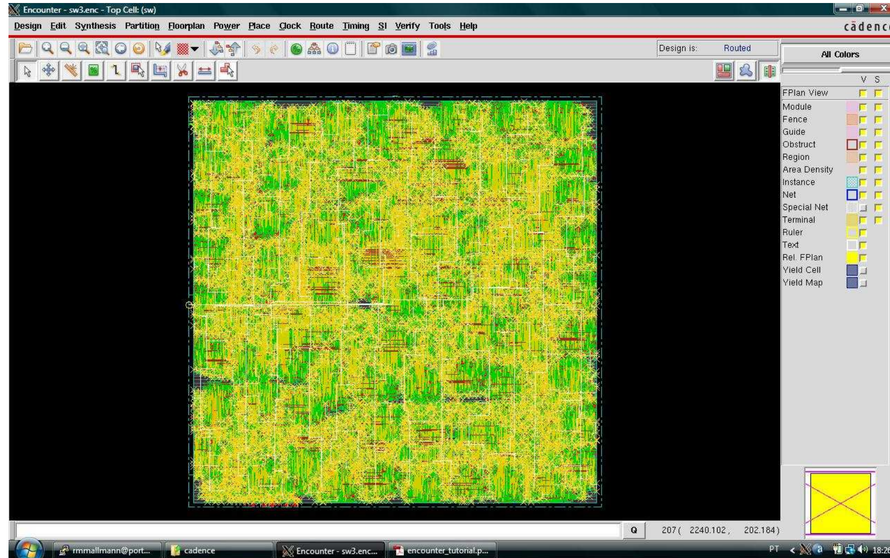


Figura 6.7: Versão final do SWAffine em ASIC com árvore de *clock* em destaque

6.3 Comparações

O primeiro tipo de comparação que realizamos foi em relação ao projeto SWAffine prototipada para FPGA e o software SSearch36 do pacote FASTA distribuido com licença GPL (EMBL-EBI, 2009). Para esse experimento utilizamos o banco de dados de proteínas UniprotKB/Swiss-Prot, com matriz de substituição Blosum62 e valores de *gap open* igual a *-4* e valor de *gap extend* igual a *-1*. O experimento consiste em deixar fixa uma sequência de alvo e compará-la com todo o banco de dados. Esse é um caso típico de uso do algoritmo Smith-Waterman, denominado busca de homologia, onde o usuário busca semelhanças entre sequências conhecidas de outros organismos (presente no BD) e uma sequência de interesse com comportamento desconhecido. Calculamos o tempo necessário para fazer essa tarefa na placa de prototipação HTG-V5-PCIE-330-1 conectada em um computador com processador Intel Core 2 Duo de 2,93 GHz, memória RAM de 2 GB e sistema operacional linux Ubuntu 9.04 executando todo o processamento na placa FPGA. Analogamente, calculamos o tempo para o mesmo conjunto de dados em um computador com processador AMD Turion X2 com 2,2 GHz, 2 GB de memória RAM e sistema operacional linux Ubuntu 9.04. O software SSearch36 foi compilado utilizando o opção *-O2* do compilador GCC.

Em (6.1) e (6.2) apresentamos as fórmulas que modelam o número total de dados de entrada e saída do SWAffine para esse tipo de teste. Nas fórmulas, BD_{seq} é o número total de sequências do banco de dados, BD_{amino} o número total de aminoácidos (caracteres) que o banco possui, UP_n o número de UPs e n o número de quebras necessárias.

$$Data_{in} = n * [(UP_n + 1) + BD_{seq} + BD_{amino}] + 1 \tag{6.1}$$

$$Data_{out} = BD_{seq} * 4 \tag{6.2}$$

O processamento do SWAffine é modelado utilizando a fórmula (6.3), onde UP_{setup} são os ciclos de *clock* extras para a interpretação do protocolo.

$$Proc = n * (UP_n + UP_{setup} + Sys_{calc}) \tag{6.3}$$

Tabela 6.9: Particionamento de sequências em relação ao BD Swiss-Prot

Quebras no Alvo	*Cobertura do Swiss-Prot*
0	88,97%
1	98,58%
2	99,59%
3	99,80%
55	100%

$$Sys_{calc} = UP_n + (BD_{amino} - 1) \qquad (6.4)$$

O processamento total é obtido transformando em tempo o fluxo de dados de $Data_{in}$ e $Data_{out}$ usando os valores de *throughput* da PCIe mostrados na Tabela 6.3. Para transformar em tempo o valor de $Proc$ basta multiplicar pelo período do *clock*. Assim, o tempo total é a soma desses 3 tempos.

$$Tempo_{hw} = tData_{in} + tProc + tData_{out} \qquad (6.5)$$

Para esse experimento segue os valores de cada variável: $BDseq = 516603$, $BD_{amino} = 181919312$, $UP_n = 650$ e $UP_{setup} = 5$.

A Tabela 6.9 mostra a relação entre o particionamento da sequência e o percentual do banco possível de analisar. O tamanho médio das sequências do Swiss-Prot é 356 e com as 650 UPs do SWAffine podemos comparar 88,97% das sequências do banco sem que ocorra nenhum particionamento. Com até 3 quebras comparamos 99,80% do banco. Esse caso considera que a sequência de alvo também está no banco. Porém, no caso de sequências inéditas, serve para ter uma previsão do tamanho médio da sequência. A importância de dividir o banco em relação as quebras de sequência se deve ao fato que uma quebra causa um considerável aumento no tamanho dos dados de entrada e no número de ciclos do processamento como mostrado nas equações (6.1) e (6.3).

Com base nessa análise, elaboramos os casos de teste apresentados na Tabela 6.10. O primeiro caso, com uma sequência de alvo fixa de tamanho 352, exemplifica o caso médio de comparação do Swiss-Prot. Em seguida, exploramos os tamanhos máximos para as comparações com 0, 1, 2 e 3 particionamentos. Para o hardware, as sequências com tamanho entre os intervalos tendem a ter o mesmo desempenho. Por exemplo, a sequência de tamanho 352 e a sequência de tamanho 650. Isso se deve a arquitetura do array sistólico, que devido ao seu paralelismo aproxima os valores máximos e mínimos de comparação. Já no SSearch36 percebemos claramente que o aumento no tempo de processamento é linear ao aumento no tamanho da sequência. Para o caso médio, obtivemos aceleração de 275 vezes. Sequência de tamanho menor ao médio acarretará aceleração inferior. Com o tamanho máximo de sequência, sem que ocorra quebra, temos uma aceleração de 483 vezes. Esse valor se mantém relativamente constante com sequências de até 2500 caracteres.

Consideramos relevantes os resultados de aceleração obtidos em relação ao mesmo algoritmo executado em um processador de propósito geral. O intuito desse experimento é mostrar uma estimativa da aceleração alcançada com o hardware. Evidentemente, os resultados seriam um pouco inferiores caso o software fosse executado em um servidor

Tabela 6.10: Comparação entre o desempenho da arquitetura em hardware dedicado e o software SSearch36

Tam. da Seq. de Alvo	*Tempo em SW (s)*	*Tempo em HW (s)*	*Aceleração*
352	601,12	2,18	275X
650	1055,04	2,18	483X
1300	2146,49	4,34	494X
1900	3125,63	6,50	480X
2500	4109,81	8,67	474X

com um processador estado da arte. Porém, também seriam mais favoráveis caso utilizássemos os novos dispositivos FPGA da família Virtex-6 fabricados em 40 e 45 nm. Em relação ao banco de dados escolhido para o teste (Swiss-Prot) é um dos mais confiáveis e utilizados para comparação de proteínas. Ainda, o teste realiza a comparação de uma sequência contra o banco inteiro. Na prática, busca-se várias sequências contra um ou mais bancos. Utiliza-se também bancos maiores, como o UniprotKB/TrEMBL (mais de 20 vezes maior que o Swiss-Prot). Em nosso experimento, a maior sequência necessitou de um pouco mais de 68 minutos para ser calculada em um processador de propósito geral. Porém, facilmente esse valor seria significativamente maior se aumentássemos o número de sequências e também utilizássemos bancos de dados maiores (o que ocorre muitas vezes na prática).

Para posicionar as arquiteturas que projetamos com o estado da arte montamos duas tabelas distintas. A primeira (Tabela 6.11) mostra o desempenho de arquiteturas que implementam a distância Levenshtein e a segunda (Tabela 6.12) o Smith-Waterman com *affine gap*. É bastante comum encontrar misturadas em uma mesma tabela esses dois algoritmos, porém como mostra a Tabela 6.1 uma UP que implementa a distância Levenshtein somente para nucleotídeos é 12,67 vezes menor que uma UP que implementa o Smith-Waterman com *affine gap*. No trabalho de Chow et al. (1991), onde um dos autores é M.S Waterman (que junto com T.F Smith desenvolveu o algoritmo Smith-Waterman), é relatada a arquitetura de Lipton e Lopresti (1985) que originou os demais trabalhos com array sistólico para a distância Levenshtein. Chow et al. (1991) consideram que as duas arquiteturas são completamente diferentes e, por isso, não podem ser diretamente comparadas. Consideramos que a distância Levenshtein atua em um caso bastante específico do algoritmo, com valores de pontuação fixos e sem utilizar matrizes de substituição. Assim, adotamos como critério no desenvolvimento das nossas arquiteturas separarmos os dois projetos.

Colocamos na mesma tabela arquiteturas em ASIC e FPGA, pois conforme já explicamos anteriormente, em alguns casos (FPGA estado da arte e ASIC com tecnologia antiga) as tecnologias se assemelham em desempenho. Esse é o nosso caso, que implementamos o SWAffine em FPGA (Virtex-5 em 65 nm) e ASIC (180 nm) e obtivemos desempenho superior em relação a frequência no FPGA. As arquiteturas ASIC estudadas também foram implementadas usando tecnologias antigas.

Analisando os resultados da Tabela 6.11 se observa que as arquiteturas LNucleo e LProt são as que possuem o melhor desempenho, porém isso se deve a tecnologia do dispositivo FPGA utilizado. As arquiteturas de Puttegowda et al. (2003) e Yu et al. (2003) possuem as menores UPs e, se sintetizadas para o nosso dispositivo alvo, possivelmente

Tabela 6.11: Comparação entre as arquiteturas estudadas e arquiteturas projetadas para a distância Levenshtein

Arquitetura	*Tecnologia*	*UPs*	*Freq*	*GCUPS*	*Tam. Máx.*
Puttegowda et al. (2003)	FPGA-XC2V6000-4	7000	180	1260	7000
Yu et al. (2003)	FPGA-XCV1000E-6	4032	202	814	4032
Kundeti et al. (2008)	ASIC-0,13 um	15	1000	15	arbitrário
LNucleo	**FPGA-XC5VLX330T-1**	**8000**	**315**	**4880**	**524388**
LProt	**XC5VLX330T-1**	**7000**	**315**	**4270**	**524388**

apresentariam o melhor desempenho. Apesar disso, a arquitetura de Puttegowda et al. (2003) necessita reconfigurar o dispositivo para cada comparação, já que a sequência de alvo é colocada de forma fixa na própria descrição da arquitetura. Esses tempos de reconfiguração acarretariam um atraso significativo em um teste de desempenho utilizando um banco de dados real. A arquitetura apresentada por Kundeti et al. (2008) possui o pior desempenho porém é a mais flexível de todas, permitindo tamanhos de sequências e tipos de pontuação arbitrários.

Nossa contribuição nas arquiteturas de distância Levenshtein é a de estar entre o máximo desempenho e a pouca flexibilidade apresentada por Puttegowda et al. (2003) e Yu et al. (2003) e a máxima flexibilidade e o baixo desempenho apresentado por Kundeti et al. (2008). Nossas arquiteturas permitem que as sequências sejam particionadas, segundo nosso conhecimento a única que implementa essa característica para a distância Levenshtein. Isso permite que um pouco mais de meio milhão de sequências possam ser comparadas no hardware, número bastante superior ao apresentado por Yu et al. (2003) e Puttegowda et al. (2003).

A Tabela 6.12 mostra um comparativo entre as arquiteturas que implementam o Smith-Waterman com *affine gap*. Consideramos que seria desleal comparar arquiteturas que implementam o Smith-Waterman com *affine gap* com as arquiteturas que implementam utilizando *linear gap*. Como o escopo desse trabalho é implementar o Smith-Waterman com *affine gap*, a forma utilizada no uso prático do algoritmo, algumas arquiteturas estudadas no Capítulo 4 não constam nessa tabela, seja por apresentarem a versão *linear gap* do algoritmo ou por não apresentarem resultados de síntese e sim ideias de como otimizar a UP.

Para efeito de comparação com as demais arquiteturas, sintetizamos o SWAffine para o dispositivo XC2V6000 o que permite compararmos com a maioria das arquiteturas estado da arte apresentadas na Tabela 6.7. Como a maioria dos autores não informam o *speed grade* do FPGA alvo para suas arquiteturas, e essa variável afetar diretamente a frequência de operação do circuito e consequentemente a métrica GCUPS de desempenho, optamos por sintetizar o SWAffine para o *speed grade* mínimo e máximo desse dispositivo FPGA e relatar os dois valores. A síntese preservou as mesmas características da arquitetura sintetizada para o dispositivo XC5VLX330T-1, exceto os tamanhos das memórias implementadas em BRAM. O número de UPs também foi reduzido devido a limitada quantidade de lógica disponível nesse dispositivo - utilizamos 88% dos slices totais. Porém, nosso objetivo sintetizando para o dispositivo XC2V6000 é comparar somente o desempenho da arquitetura, de forma análoga as arquiteturas estado da arte apresentadas, sem considerar os atrasos referentes ao barramento e as escritas e leituras das memórias.

Tabela 6.12: Comparação entre as arquiteturas estudadas e arquiteturas projetadas para o Smith-Waterman com affine gap

Arquitetura	*Tecnologia*	*UPs*	*Freq (MHz)*	*GCUPS*	*Tam. Máx.*
Chow et al. (1991)	ASIC-1 mm	16	12,5	0,2	4194304
Han e Parameswaran (2002)	ASIC-0,5 um	64	55	3,2	64
Oliver et al. (2005a)-01	FPGA-XC2V6000	168	45	7,6	504
Oliver et al. (2005a)-02	FPGA-XC2V6000	119	45	5,2	1428
Benkrid et al. (2009)	FPGA-XC2VP100-6	168	47,6	8,0	168
Cheng e Parhi (2008)	FPGA-XC2V6000	<56	55	<9,2	-
Jiang et al. (2007)	FPGA-XC2V6000	80	88	7,4	900000
Court e Herbordt (2007)	FPGA-XC2VP70	138	39,2	5,4	-
SWAffine-01	**ASIC-0,18 um**	**64**	**100**	**6,4**	**64**
SWAffine-02	**FPGA-XC5VLX330T-1**	**650**	**143**	**92,9**	**65536**
SWAffine-03	**FPGA-XC2V6000**	**189**	**60-84**	**11,3-15,8**	**65536**

Na arquitetura de Cheng e Parhi (2008) colocamos uma estimativa do número de UPs e o desempenho já que o autores não revelam esses dados. Eles apenas citam integrar menos de 1/3 do número de UPs da arquitetura de Oliver et al. (2005a), porém processam três sequências por ciclo. Assim, calculamos uma estimativa máxima hipotética, onde os autores integrariam exatamente 1/3 das UPs de Oliver et al. (2005a)-01, e atribuímos o sinal de menor para informar que os resultados reais são inferiores a esse valor.

A arquitetura SWAffine-03 apresentou desempenho entre 52% e 113% superior, em GCUPS, a arquitetura de Jiang et al. (2007) - o melhor desempenho reportado para o dispositivo XC2V6000. Essa variação, como citada anteriormente, refere-se a síntese para os *speed grades* mínimos e máximos do dispositivo XC2V6000. Atribuímos esse significativo aumento no desempenho da arquitetura em relação as demais a nova UP que propomos no capítulo 5.5 que reduz em 50% o número de somadores de 16 bits e em 87,5% o número de bits de dois dos comparadores - adicionando nesse sistema dois multiplexadores de duas entradas e um registrador de dois bits. Essa estratégia se mostrou vantajosa pois, além de permitir integrar cerca de 12% mais UPs que a arquitetura de Oliver et al. (2005a)-01, é capaz de reduzir o caminho crítico da UP e aumentar em 33% a frequência do circuito comparado a essa mesma arquitetura. Ainda, observamos que o dispositivo XC5VLX330T-1 se mostra como uma excelente escolha para implementação desse algoritmo pois o SWAffine-02 apresenta desempenho 5,8 vezes superior ao apresentado pelo SWAffine-03 utilizando o máximo *speed grade* do dispositivo XC2V6000.

Em relação às arquiteturas ASIC, por utilizarem tecnologias diferentes, tornam a comparação direta bastante difícil. Em relação à capacidade de integração de UPs, a tecnologia que utilizamos no SWAffine-01 permitiria integrar mais UPs, porém devido aos tempos de projeto elevados optamos por projetar uma arquitetura menor. Obtivemos 6,4 GCUPS de desempenho e poderíamos dobrar esse valor utilizando essa mesma tecnologia caso aumentássemos o número de UPs. A arquitetura de Chow et al. (1991) é a que possui a UP mais complexa, permitindo em um mesmo projeto configurar o tipo de algoritmo (alinhamento local ou global), diversas matrizes de substituição e entrada de dados. Isso a faz ter a maior UP das arquiteturas estudadas, o que prejudica bastante a capacidade de integração e a métrica de desempenho.

Em relação ao tamanho máximo de caracteres que cada arquitetura permite calcular,

Chow et al. (1991) é a que possui a maior capacidade seguida pela de Jiang et al. (2007). Como o objetivo da arquitetura SWAffine-02 é processar proteínas, permitimos um valor ligeiramente maior (1,8 vezes) que o maior tamanho de sequência do Swiss-Prot que é de 32213 caracteres. Ainda, observamos, analisando a Tabela 6.9, que as arquiteturas de Oliver et al. (2005a) e Benkrid et al. (2009) também especializadas em processar proteínas, não seriam capazes de calcular 100% das sequências do Swiss-Prot.

7 CONCLUSÕES

O uso de dados biológicos já é uma realidade e tende a aumentar drasticamente nos próximos anos. A indústria de biotecnologia cresce anualmente em uma taxa bastante superior a da economia tradicional. Além disso, novas aplicações que façam uso de informações biológicas são desenvolvidas em uma velocidade impressionante. Todo o ecossistema necessário para desenvolver esse tipo de indústria está formado. De um lado existem grandes empresas que estão em franca competição no desenvolvimento de novos sequenciadores e tecnologias que permitam que genomas possam ser transformados em cadeias de caracteres. Ainda, já estão institucionalizado os bancos de dados genéticos públicos que permitem que qualquer pessoa possa fazer uso da informação genética já conhecida. Na outra ponta, empresas de biotecnologia estão nascendo e propondo produtos novos desde melhoramento genético de sementes, criação de pesticidas inteligentes para controle de pragas até a criação de novos fármacos.

Entre essas duas pontas está a tecnologia da informação e a forma como esses dados devem ser processados. O processamento de dados biológicos em processadores de propósito geral, como já foi explicitado nesse trabalho, não possui o desempenho necessário para essa indústria emergente. Assim, propomos o uso de computação híbrida para acelerar os algoritmos que processam dados biológicos e permitir que mais dados possam ser processados na mesma quantidade de tempo ou que seja reduzido o tempo de execução desses algoritmos. Os testes realizados entre a placa FPGA e um computador mostram uma aceleração de 270 vezes para um tamanho médio de proteína e em torno de 470 vezes para sequências de até 2500 caracteres.

Apresentamos nesse trabalho 2 arquiteturas distintas que implementam o algoritmo distância Levenshtein e o algoritmo Smith-Waterman com a versão *affine gap* e matrizes de substituição. A arquitetura da distância Levenshtein projetada é a primeira, segundo nosso conhecimento, que permite que sequências maiores que 500K possam ser comparadas em hardware. Essa característica é especialmente importante quando se utiliza DNA ou RNA como dados de entrada devido ao aumento no atual tamanho dessas sequências. Em relação a arquitetura que implementa o Smith-Waterman apresentamos uma nova unidade de processamento capaz de reduzir o caminho crítico do circuito e a área, sendo entre as arquiteturas estudadas a que apresenta o melhor desempenho.

Nosso estudo mostrou que, principalmente o algoritmo Smith-Waterman, pode ser utilizado como um módulo para outros algoritmos como é o caso do Clustal. Ainda, pode ser utilizado como um filtro para acelerar o Blast. Por esses motivos, a nova unidade de processamento projetada permite obter ganhos de desempenho em várias atividades de bioinformática. Como trabalhos futuros pretendemos modificar a forma como as matrizes

de substituição são armazenadas, adicionando um circuito para compactar e descompactar os dados (o que permitiria integrar mais UPs). Ainda, o protocolo de comunicação projetado pode ser modificado para que a sequência de busca não necessite ser enviada a cada quebra da sequência de alvo, o que reduziria significativamente o *throughput* do sistema. Em relação aos tempos para a comunicação entre FPGA e computador, o projeto de uma hierarquia de memória utilizando a RAM do computador hospedeiro, a RAM da placa FPGA e a BRAM também aumentaria o desempenho do sistema.

Por fim, avaliamos que para o atual mercado de biotecnologia dispositivos FPGAs são as melhores escolhas tecnológicas. O mercado ainda necessita que os algoritmos sejam adaptados para formas quase personalizadas de uso, o que faz com que seja ideal para implementar em FPGA. Porém, a medicina personalizada pode mudar este panorama. A ideia por traz disso é que cada indivíduo possa ter o seu genoma pessoal mapeado e, com isso, verificar a probabilidade de desenvolver novas doenças ou desenvolver fármacos personalizados. Com o desenvolvimento desse mercado, o processamento de dados biológicos alcançaria um novo patamar; com bilhões de pessoas de posse do seu genoma necessitando comparar com bancos de dados cada vez maiores. Nesse contexto já não tão hipotético, pois essa indústria tem se desenvolvido rapidamente, a tecnologia ASIC se apresenta como a melhor escolha em termos de custo e desempenho.

REFERÊNCIAS

BENKRID, K.; LIU, Y.; BENKRID, A. A highly parameterized and efficient fpga-based skeleton for pairwise biological sequence alignment. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, v. 17, n. 4, p. 561 –570, april 2009. ISSN 1063-8210.

BOUKERCHE, A. et al. A hardware accelerator for the fast retrieval of dialign biological sequence alignments in linear space. *Computers, IEEE Transactions on*, v. 59, n. 6, p. 808 –821, june 2010. ISSN 0018-9340.

CHENG, C.; PARHI, K. High-speed implementation of smith-waterman algorithm for dna sequence scanning in vlsi. In: *Signals, Systems and Computers, 2008 42nd Asilomar Conference on*. [S.l.: s.n.], 2008. p. 1528 –1533. ISSN 1058-6393.

CHOW, E. et al. Biological information signal processor. In: *Application Specific Array Processors, 1991. Proceedings of the International Conference on*. [S.l.: s.n.], 1991. p. 144 –160.

CORRÊA, J. M. *Arquiteturas em FPGA para Comparação de Sequências em Espaço Linear*. Tese (Doutorado) — Universidade de Brasília, Brasília, Brasil, 2008.

COURT, T. V.; HERBORDT, M. C. Families of fpga-based accelerators for approximate string matching. *Microprocess. Microsyst.*, Elsevier Science Publishers B. V., Amsterdam, The Netherlands, The Netherlands, v. 31, n. 2, p. 135–145, 2007. ISSN 0141-9331.

CRISTIANINI, N.; HAHN, M. W. *Introduction to Computational Genomics: A case studies approach*. 1. ed. Cambridge: Cambridge University Press, 2007.

EMBL-EBI. *SSEARCH Protein Similarity Search*. 2009. Disponível em: <http://www.ebi.ac.uk/Tools/fasta33/index.html?program=SSEARCH>. Acesso em: novembro 2009.

GILLELAND, M. *Levenshtein Distance, in Three Flavors*. 2009. Disponível em: <http://www.merriampark.com/ld.htm>. Acesso em: novembro 2009.

GOTOH, O. An improved algorithm for matching biological sequences. *J. Molecular Biology*, v. 162, n. 3, p. 705 – 708, dezembro 1982.

GUCCIONE, S. A.; KELLER, E. Gene matching using jbits. In: *FPL '02: Proceedings of the Reconfigurable Computing Is Going Mainstream, 12th International Conference on Field-Programmable Logic and Applications*. London, UK: Springer-Verlag, 2002. p. 1168–1171. ISBN 3-540-44108-5.

GUSFIELD, D. *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology*. 1. ed. Cambridge: Cambridge University Press, 1997.

HAN, T.; PARAMESWARAN, S. Swasad: an asic design for high speed dna sequence matching. In: *Design Automation Conference, 2002. Proceedings of ASP-DAC 2002. 7th Asia and South Pacific and the 15th International Conference on VLSI Design. Proceedings.* [S.l.: s.n.], 2002. p. 541 –546.

HENIKOFF, S.; HENIKOFF, J. Amino acid substitution matrices from protein blocks. *Proceedings of the National Academy of Sciences USA*, n. 89, p. 10915–10919, 1992.

JIANG, X. et al. A reconfigurable accelerator for smith x2013;waterman algorithm. *Circuits and Systems II: Express Briefs, IEEE Transactions on*, v. 54, n. 12, p. 1077 –1081, dec. 2007. ISSN 1549-7747.

JONES, N. C.; PEVZNER, P. A. *An Introduction to Bioinformatics Algorithms*. 1. ed. Cambridge, MA: MIT Press, 2004.

KLEIWEG, P. *Levenshtein demo*. 2009. Disponível em: <http://odur.let.rug.nl/ kleiweg/lev/>. Acesso em: novembro 2009.

KUNDETI, V.; FEI, Y.; RAJASEKARAN, S. An efficient digital circuit for implementing sequence alignment algorithm in an extended processor. In: *ASAP '08: Proceedings of the 2008 International Conference on Application-Specific Systems, Architectures and Processors*. Washington, DC, USA: IEEE Computer Society, 2008. p. 156–161. ISBN 978-1-4244-1897-8.

KUNG, H. T. Why systolic architectures? *Computer*, IEEE Computer Society Press, Los Alamitos, CA, USA, v. 15, n. 1, p. 37–46, 1982. ISSN 0018-9162.

LECOMPTE, O. et al. Multiple alignment of complete sequences (macs) in the post-genomic era. *Gene*, v. 270, n. 1-2, p. 17 – 30, 2001. ISSN 0378-1119. Disponível em: <http://www.sciencedirect.com/science/article/B6T39-436W3KY-2/2/6d061757ee16f522ff74c4734281c3bd>.

LIN, X. et al. To accelerate multiple sequence alignment using fpgas. In: *HPCASIA '05: Proceedings of the Eighth International Conference on High-Performance Computing in Asia-Pacific Region*. Washington, DC, USA: IEEE Computer Society, 2005. p. 176. ISBN 0-7695-2486-9.

LIPTON, R.; LOPRESTI, D. A systolic array for rapid string comparison. In: *Proceedings Chapel Hill Conference on VLSI*. Rockville, MD, EUA: Computer Science Press, 1985. p. 363–376.

LLOYD, S.; SNELL, Q. Sequence alignment with traceback on reconfigurable hardware. In: *Reconfigurable Computing and FPGAs, 2008. ReConFig '08. International Conference on*. [S.l.: s.n.], 2008. p. 259 –264.

MCMAHON, P. L. *Accelerating Genomic Sequence Alignment using High Performance Reconfigurable Computers*. Dissertação (Mestrado) — Universidade de Cape Town, 2008.

MOLDOVAN, D.; FORTES, J. Partitioning and mapping algorithms into fixed size systolic arrays. *Computers, IEEE Transactions on*, C-35, n. 1, p. 1 –12, jan. 1986. ISSN 0018-9340.

NAWAZ, Z. et al. Acceleration of smith-waterman using recursive variable expansion. In: *Digital System Design Architectures, Methods and Tools, 2008. DSD '08. 11th EUROMICRO Conference on*. [S.l.: s.n.], 2008. p. 915 –922.

NCBI. *350 KB Sequence Length Limit Removed by Sequence Database Collaboration*. 2008. Disponível em: <http://www.ncbi.nlm.nih.gov/Web/Newsltr/Spring04/350kb.html>. Acesso em: novembro 2008.

NCBI. *GenBank Overview*. 2009. Disponível em: <http://www.ncbi.nlm.nih.gov/genbank/>. Acesso em: abril 2009.

OLIVER, T.; SCHMIDT, B.; MASKELL, D. Hyper customized processors for bio-sequence database scanning on fpgas. In: *FPGA '05: Proceedings of the 2005 ACM/SIGDA 13th international symposium on Field-programmable gate arrays*. New York, NY, USA: ACM, 2005. p. 229–237. ISBN 1-59593-029-9.

OLIVER, T. et al. Multiple sequence alignment on an fpga. In: *ICPADS '05: Proceedings of the 11th International Conference on Parallel and Distributed Systems - Workshops*. Washington, DC, USA: IEEE Computer Society, 2005. p. 326–330. ISBN 0-7695-2281-5.

PARK, J.; QIU, Y.; HERBORDT, M. Caad blastp: Ncbi blastp accelerated with fpga-based accelerated pre-filtering. In: *Field Programmable Custom Computing Machines, 2009. FCCM '09. 17th IEEE Symposium on*. [S.l.: s.n.], 2009. p. 81 –87.

PUTTEGOWDA, K. et al. A run-time reconfigurable system for gene-sequence searching. In: *VLSID '03: Proceedings of the 16th International Conference on VLSI Design*. Washington, DC, USA: IEEE Computer Society, 2003. p. 561. ISBN 0-7695-1868-0.

SACCONE, C.; PESOLE, G. *Handbook os Comparative Genomics: Principles and Methodology*. [S.l.]: Wiley-Liss, 2003.

SERNA, A. de la. Differential scoring for systolic sequence alignment. In: *Bioinformatics and Bioengineering, 2007. BIBE 2007. Proceedings of the 7th IEEE International Conference on*. [S.l.: s.n.], 2007. p. 1204 –1208.

SMITH, T.; WATERMAN, M. Identification of common molecular subsequences. *J. Molecular Biology*, v. 147, n. 1, p. 195 – 197, março 1981.

TICONA, W. G. C. *Aplicação de Algoritmos Genéticos Multi-Objetivo para o Alinhamento de Sequências Biológicas*. Dissertação (Mestrado) — Instituto de Ciências Matemáticas e de Computação, Universidade de São Paulo, São Carlos, 2003.

VENTER, J. C. Multiple personal genomes await. *Nature*, Nature Publishing Group, v. 464, n. 7289, p. 676–677, abril de 2010.

WESTE, N. H. E.; HARRIS, D.; BANERJEE, A. *CMOS VLSI Design: A Circuits and Systems Perspective*. 3. ed. Boston, MA: Pearson Education, 2005.

WOREK, W. J. *Matching Genetic Sequences in Distributed Adaptive Computing Systems*. Dissertação (Mestrado) — Instituto Politécnico da Virginia, 2002.

WUNSCH, S. B. N. M. C. D. A general method applicable to the search for similarities in the amino-acid sequence of two proteins. *J. Molecular Biology*, v. 48, n. 3, p. 443 – 453, março 1970.

YU, C. W. et al. A smith-waterman systolic cell. In: *FPL '03: Proceedings of the 13th International Workshop on Field Programmable Logic and Applications*. [S.l.]: Springer, 2003. p. 375–384.