UNIVERSIDADE DE BRASÍLIA
FACULDADE DE TECNOLOGIA
DEPARTAMENTO DE ENGENHARIA ELÉTRICA

ARQUITETURAS EM FPGA PARA COMPARAÇÃO DE SEQUÊNCIAS BIOLÓGICAS EM ESPAÇO LINEAR

JAN MENDONÇA CORRÊA

ORIENTADORA: ALBA CRISTINA MAGALHÃES ALVES DE MELO

TESE DE DOUTORADO EM ENGENHARIA ELÉTRICA

BRASÍLIA/DF: MAIO – 2008

UNIVERSIDADE DE BRASÍLIA
FACULDADE DE TECNOLOGIA
DEPARTAMENTO DE ENGENHARIA ELÉTRICA

ARQUITETURAS EM FPGA PARA COMPARAÇÃO DE SEQUÊNCIAS BIOLÓGICAS EM ESPAÇO LINEAR

JAN MENDONÇA CORRÊA

TESE DE DOUTORADO SUBMETIDA AO DEPARTAMENTO DE ENGENHARIA ELÉTRICA DA FACULDADE DE TECNOLOGIA DA UNIVERSIDADE DE BRASÍLIA

BANCA EXAMINADORA:

Profa. Alba Cristina Magalhães Alves de Melo, Doutora (CIC-UnB)
(Orientadora)

Prof. Adson Ferreira da Rocha, Doutor (ENE-UnB)
(Examinador Interno)

Prof. Ricardo Pezzuol Jacobi, Doutor (CIC-UnB)
(Examinador Interno)

Profa. Maria Emília Machado Telles Walter, Doutora (CIC-UnB)
(Examinadora Externa)

Prof. Sérgio Bampi, Doutor (UFRGS)
(Examinador Externo)

BRASÍLIA/DF, MAIO DE 2008

FICHA CATALOGRÁFICA

CORRÊA, JAN MENDONÇA

ARQUITETURAS EM FPGA PARA COMPARAÇÃO DE SEQUÊNCIAS BIOLÓGICAS EM ESPAÇO LINEAR

[Distrito Federal] 2008.

xvii, 138p., 210 x 297 mm (ENE/FT/UnB, Doutor, Tese de Doutorado – Universidade de Brasília. Faculdade de Tecnologia. Departamento de Engenharia Elétrica

1.Bioinformática 2.Hardware reconfigurável

3.FPGA 4.Alinhamento de seqüências

I. ENE/FT/UnB II. Título (série)

REFERÊNCIA BIBLIOGRÁFICA

CORRÊA, J. M. (2008). Arquiteturas em FPGA para Comparação de Seqüências Biológicas em Espaço Linear. Tese de Doutorado em Engenharia Elétrica, Publicação PPGENE TD-025/2008 Departamento de Engenharia Elétrica, Universidade de Brasília, Brasília, DF, 138p.

CESSÃO DE DIREITOS

AUTOR: Jan Mendonça Corrêa.

TÍTULO: Arquiteturas em FPGA para Comparação de Seqüências Biológicas em Espaço Linear.

GRAU: Doutor ANO: 2008

_____

Jan Mendonça Corrêa
Jan@cic.unb.br
Departamento de Ciências da Computação ASS-465/6
Campus Universitário Darcy Ribeiro - ICC Centro
Asa Norte - Brasília - DF - Brasil

AGRADECIMENTOS

Agradeço à minha orientadora Dra. Alba Cristina Magalhães Alves de Melo por tudo que aprendi com ela como pesquisadora e como ser humano. Ao Dr. Ricardo Pezzuol Jacobi, cujos comentários e idéias contribuíram bastante nesta tese. Ao Dr. Adson Ferreira da Rocha pelos conselhos durante o curso. Aos membros da banca Dra. Maria Emília Machado Telles Walter e Dr. Sérgio Bampi pelos valiosos comentários.

Agradeço a Cássia e Karla pelo atendimento atencioso. Agradeço aos amigos do departamento de Ciência da Computação e do departamento de Engenharia Elétrica. Agradeço a todos que de alguma forma contribuíram neste trabalho.

Agradeço aos meus pais José e Joana, à minha irmã Daphne ao meu avô Leibnitz pelo exemplo de vida. Agradeço à Ana Lúcia por seu companheirismo.

RESUMO

ARQUITETURAS EM FPGA PARA COMPARAÇÃO DE SEQUÊNCIAS BIOLÓGICAS EM ESPAÇO LINEAR

Autor: Jan Mendonça Corrêa

Orientadora: Alba Cristina Magalhães Alves de Melo

Programa de Pós-Graduação em Engenharia Elétrica

Brasília, maio de 2008

O alinhamento de seqüências biológicas é uma das operações mais básicas em bioinformática, tendo por objetivo determinar a similaridade entre as seqüências. A solução deste problema envolve geralmente a comparação de seqüências através de programação dinâmica. Este tipo de comparação gera resultados ótimos mas possui complexidade quadrática de tempo, justificando métodos para sua aceleração em hardware como o FPGA.

Na presente tese foram projetadas arquiteturas wavefront em FPGA utilizando espaço linear para três diferentes algoritmos. O primeiro algoritmo foi o de Smith-Waterman. Ele foi implementado na forma de um vetor wavefront e foi utilizado na aceleração da fase inicial de um algoritmo de alinhamento. Esta arquitetura foi capaz de recuperar o maior escore e posição em espaço linear. Esta arquitetura foi sintetizada em FPGA e o melhor resultado da arquitetura foi 246,9 vezes mais rápido que em software, demonstrando a utilidade da arquitetura.

A seguir, foi projetada uma arquitetura para a recuperação do escore ótimo do algoritmo de programação dinâmica DIALIGN também em espaço linear. Foram obtidos resultados até 383,41 vezes superiores ao programa em software. Para recuperar o alinhamento ótimo no DIALIGN é necessário espaço quadrático. Assim, foi projetada uma variante do DIALIGN capaz de recuperar o alinhamento de duas seqüências em espaço linear. Após a implementação em hardware, os resultados obtidos foram até 141,38 vezes mais rápido que a implementação em software.

ABSTRACT

FPGA ARCHITECTURES FOR BIOLOGICAL SEQUENCE COMPARISON IN LINEAR SPACE

Author: Jan Mendonça Corrêa

Supervisor: Alba Cristina Magalhães Alves de Melo

Programa de Pós-Graduação em Engenharia Elétrica

Brasília, may 2008

The alignment of biological sequences is one of the more basic operations in bioinformatics. Its purpose is to find the similarity between sequences. The solution to this problem generally involves sequence comparison through dynamic programming. This kind of comparison yields optimal results but has quadratic time complexity thus justifying its hardware acceleration in FPGA.

In this thesis, linear space wavefront architectures were designed in FPGA for three different algorithms. The first algorithm was Smith-Waterman. It was implemented in a wavefront array and utilized to accelerate the initial phase of a sequence alignment algorithm. This architecture was able to retrieve the largest score and its position in linear space. It was synthesized in FPGA and the best result was 246,9 times faster than software, showing the appropriateness of the architecture.

Also, an architecture to retrieve the optimal DIALIGN score in linear space was designed. The results were up to 383,41 times better than software. The retrieval of the optimal alignment for DIALIGN needs quadratic space. Therefore, a variant for the DIALIGN dynamic programming algorithm was proposed to retrieve the alignment in linear space. This variant was implemented in hardware and the results were up to 141,38 times faster than the software implementation.

SUMÁRIO

LISTA DE TABELAS

LISTA DE FIGURAS

LISTA DE SÍMBOLOS, NOMECLATURA E ABREVIAÇÕES

ALM - *Adaptive Logic Module*

ASIC - *Application Specific Integrated Circuit*

BISP - *Biological Information Signal Processor*

BLAST - *Basic Local Alignment Search Tool*

BLOSUM - *Blocks Substituion Matrix*

CLB - *Configurable Logic Block*

CUPS - *Cell Updates per Second*)

FPGA - *Field Programmable Gate Arrays*

FPX - *Field Programmable Port Extender*

HDL - *hardware description language*

IOBs - *Input Output Blocks*

LUT - *Lookup Table*

PAM - *Point Accepted Mutations*

PCI - *Peripheral Component Interconnect*

MCUPS - *Million Cell Updates per Second*

RAM - *Random access memory*

SAMBA - *Systolic Accelerator for Molecular Biological Application*

SRAM - *Static random access memory*

VHDL - *VHSIC Hardware Description Language*

VHSIC - *Very High Speed Integrated Circuits*

1 – INTRODUÇÃO

A bioinformática é uma área na qual são realizadas pesquisas para produzir, organizar, visualizar ou analisar informações biológicas [Nih08]. Assim, a bioinformática atualmente suporta pesquisas de novos medicamentos e novos materiais baseados em biotecnologia. Dentro da bioinformática, um problema importante é a comparação de seqüências biológicas. Neste problema, duas seqüências de DNA, RNA ou proteínas são comparadas. Através destas comparações é possível inferir muitas informações relevantes de um organismo a partir das informações já identificadas de organismos relacionados filogeneticamente [Eur02]. Quanto maior a similaridade sas seqüências, maior a chance de terem as mesmas funções biológicas.

Nesta tese, foi feito um levantamento dos conceitos de biologia molecular relevantes para o entendimento do problema da comparação de seqüências biológicas. Este problema é muito intensivo em computação, em muitos casos exigindo que o número de operações necessárias seja proporcional ao produto dos tamanhos das seqüências [Set97]. A comparação de seqüências pode ser feita de forma heurística ou exata. Nos métodos heurísticos, não há garantias de se encontrar o resultado ótimo.A vantagem neste caso, é que os resultados são obtidos em tempo menor. Nos métodos exatos, obtém-se o resultado ótimo que é, portanto, mais relevante do ponto de vista biológico. No entanto, os tempos de execução podem ser altos. Uma maneira de acelerar a obtenção de resultados exatos na comparação de seqüências biológicas é através do uso de arquiteturas dedicadas que têm o potencial de executar várias operações em paralelo.

Soluções comerciais de arquiteturas dedicadas para comparação de seqüências biológicas podem ser muito caras e normalmente são proprietárias, causando dependência de fornecedores específicos e dificuldades de comunicação com outras arquiteturas [Tim08]. Além disto, existem vários algoritmos para a comparação de seqüências e apenas uma pequena parte destes foi acelerada em hardware. Assim, a comparação de seqüências em hardware é uma área de intensa atividade de pesquisa.

Várias arquiteturas dedicadas já foram propostas para comparação de seqüências. A maioria destas arquiteturas implementa algoritmos de programação dinâmica como o Smith-Waterman [Smi81] e Needleman-Wunsch [Nee70], com bons resultados [Wor02], [Zah07] e [Put03]. Nestes algoritmos é utilizado o conceito de alinhamento que visa encontrar áreas semelhantes. A este alinhamento é associado um escore. Existem basicamente duas abordagens para a execução do Smith-Waterman e Needleman-Wunsh: o cálculo do escore ótimo e o cálculo do alinhamento ótimo. O escore ótimo é o valor atribuído ao melhor alinhamento das seqüências e seu cálculo pode ser feito em espaço linear. Devido a restrições de espaço em hardware reconfigurável, vários autores optaram por projetar arquiteturas que recupera apenas o maior escore, o que utiliza menos espaço e possibilita grande desempenho [Lav98] [Mar03] [Oli05b] [Yu03].

Para o cálculo do alinhamento, pode ser requerido espaço quadrático [Smi81], o que implica em grande espaço em hardware, mesmo para alinhar seqüências pequenas. Dentre as várias arquiteturas analisadas, a única arquitetura proposta para recuperar o alinhamento ótimo com o Smith-Waterman foi [Yam02]. Neste caso, a recuperação do alinhamento foi feita em espaço quadrático, o que limitou bastante os tamanhos das seqüências. Assim, é importante utilizar pouco espaço para a computação e as arquiteturas desenvolvidas nesta tese serão projetadas para utilizar espaço linear.

Na análise das arquiteturas já existentes, foi observado que hardware reconfigurável como o FPGA (*Field Programmable Gate Arrays*) [Alt08] [Xil08] foi uma solução muito utilizada para comparação de seqüências. Esta solução foi utilizada com bons resultados por vários autores [Guc02], [Wor02], [Mar03], [Mos98], [Yu03], [Yam02], [Oli05b] e possui grande flexibilidade [Wol04]. Assim, nesta tese será utilizada a síntese em FPGAs para os testes das arquiteturas.

A maioria das arquiteturas estudadas utilizou um vetor sistólico [Kun82] para comparar as seqüências. Este tipo de arquitetura tem elementos de computação que funcionam de forma síncrona, permitindo bom desempenho. Embora este tipo de arquitetura seja relativamente simples por ser síncrona, ela faz com que todos os elementos funcionem na mesma velocidade

independente da computação realizada no elemento. Na presente tese, optou-se por utilizar uma arquitetura wavefront [Kun87] que, embora tenha um projeto mais complexo, é mais eficiente quando as operações a serem realizadas nos elementos possuem tempos distintos. Até onde vai nosso conhecimento, a arquitetura projetada nesta tese é a primeira a utilizar um vetor wavefront para comparação de seqüências biológicas.

As principais contribuições da presente tese são as seguintes :

1) Arquitetura wavefront para o algoritmo Smith-Waterman:

Uma vez construída a arquitetura básica do vetor wavefront, o primeiro algoritmo implementado em hardware foi o Smith-Waterman (SW). Diferentemente de outras arquiteturas para o SW, a arquitetura aqui projetada fez a aceleração de um algoritmo completo para o alinhamento de duas seqüências [Bou07a], obtendo o melhor escore e a posição no qual o mesmo ocorre. A arquitetura foi sintetizada em FPGA e utilizada para comparar seqüências reais de DNA. Os testes mostraram muito bons resultados com o algoritmo SW sendo executado até 246,9 vezes mais rápido que em software.

2) Arquitetura wavefront para o escore do DIALIGN:

A mesma arquitetura básica foi modificada para executar o DIALIGN [Mor96] que é um algoritmo de comparação de seqüências bem mais complexo. O DIALIGN tem equações de recorrência bem mais complexas e utiliza mais dados, resultando em uma arquitetura com menos elementos. A arquitetura foi capaz de recuperar o escore do alinhamento ótimo e, até onde sabemos, foi a primeira a fazer isto para o DIALIGN [Bou07b]. Esta arquitetura pôde executar comparação de seqüências de tamanhos quaisquer. Para a síntese, foi utilizado um FPGA com mais recursos. Para comparações com seqüências reais de DNA, obteve-se resultados que foram 383,41 vezes mais rápido que o programa em software, indicando o potencial da arquitetura.

3) Variante do DIALIGN em espaço linear:

Nas arquiteturas analisadas foi constatado que nenhuma arquitetura conseguiu recuperar o alinhamento ótimo de duas seqüências em espaço linear. Fazer isto é importante para casos reais onde a matriz de similaridade é muito grande para ser armazenada dentro do FPGA. Assim, é importante a criação de uma arquitetura que faça o alinhamento utilizando espaço linear.

A recuperação do alinhamento para o algoritmo do DIALIGN necessita de espaço quadrático [Mor96]. Versões mais atualizadas deste algoritmo [Mor02] têm a necessidade de espaço reduzida mas, mesmo assim, quadrático para o caso geral. Assim, para recuperar o alinhamento do DIALIGN em hardware foi necessário o projeto de uma nova versão do DIALIGN que execute em espaço linear.

4) Arquitetura wavefront para recuperação do alinhamento DIALIGN: a variante do algoritmo em 3) foi implementada em hardware, aproveitando-se a arquitetura básica e adicionando-se um novo módulo no circuito. Os resultados obtidos mostraram que esta arquitetura pode executar até 141,38 vezes mais rápido do que em software.

Nesta tese, o capítulo 2 discorre sobre os fundamentos básicos da comparação de seqüências biológicas. No capítulo 3 são apresentadas as arquiteturas dedicadas, suas vantagens e desvantagens. Um levantamento das principais implementações de algoritmos de bioinformática em arquiteturas dedicadas é feito no capítulo 4. O capítulo 5 apresenta as arquiteturas wavefront projetadas nesta tese e as modificações feitas para cada um dos três casos: aceleração do Smith-Waterman, recuperação do escore do DIALIGN e recuperação do alinhamento do DIALIGN. O capítulo 6 apresenta os resultados obtidos na síntese de cada caso e as comparações com software. O capítulo 7 conclui esta tese.

2 –RECONHECIMENTO APROXIMADO DE PADRÕES EM COMPARAÇÃO DE SEQUÊNCIAS BIOLÓGICAS

2.1 FUNDAMENTOS DE BIOLOGIA MOLECULAR

A biologia molecular é o campo da ciência que estuda a estrutura e processos biológicos em nível molecular [Bri08]. Dentre as estruturas estudadas pela biologia molecular, o DNA é uma das mais importantes [Zah96]. O DNA ou ADN (ácido desoxirribonucléico) é a substância química que armazena informações informações genéticas essenciais para a maioria dos seres vivos. Estruturalmente, o DNA é composto de duas fitas em espiral. Cada uma destas fitas contém uma seqüência de nucleotídeos. Um nucleotídeo é composto de uma pentose, um grupo fosfato e de uma base nitrogenada, que pode ser purina: Adenina (A), Guanina (G) ou pirimidina: Citosina (C) e Timina (T).

As bases nitrogenadas são complementares nas duas fitas, isto é, a Adenina de uma fita geralmente se liga a uma Timina na outra fita e a Guanina em uma fita geralmente se liga a uma Citosina na outra fita. Esta ligação entre bases é feita por pontes de hidrogênio. As duas fitas se unem em uma estrutura helicoidal chamada de dupla hélice. Esta complementaridade possibilita que o DNA se duplique. Neste caso, cada uma das fitas serve como molde para se ligar a uma fita complementar. As fitas de DNA são lidas em um sentido específico chamado de 5’ para 3’. A figura 2.1 mostra uma representação do DNA, onde pode ser vista a estrutura da dupla hélice com as respectivas bases se ligando de forma complementar.

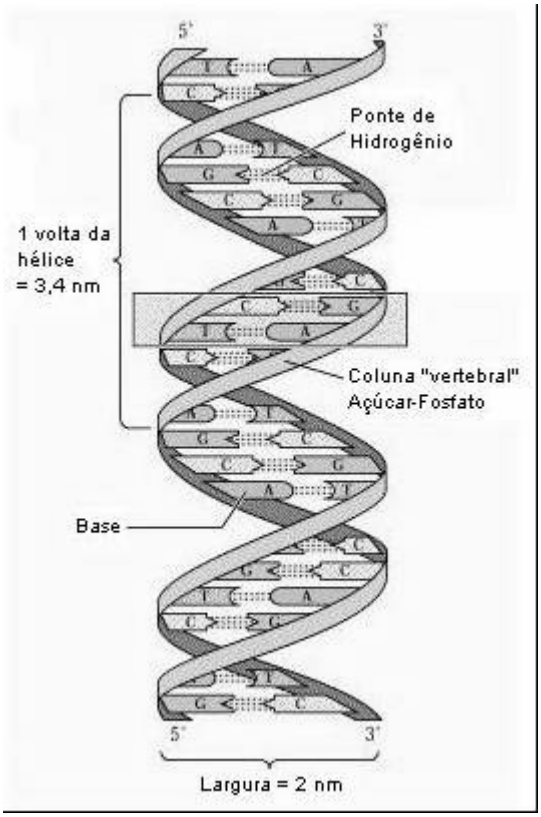

Figura 2.1 – Representação do DNA [Sfi06]

Uma molécula de DNA pode se organizar de forma compacta em cromossomos [Zah96]. Nos seres humanos e em outros eucariontes, os cromossomos existem em pares nas células somáticas, ou seja, as que não são reprodutivas. Apenas uma pequena parte do DNA dos seres eucariontes corresponde aos genes [Set97], que são considerados as unidades básicas de transmissão das características genéticas. Grande parte do DNA não possui função conhecida e é chamada de não codificante [Zah96]. Em cada gene na cadeia de DNA, existem trechos que codificam as proteínas e são chamados exons. Entre os exons, dentro de um mesmo gene, podem existir regiões consideradas como não codificadoras de proteínas que são chamadas de introns [Mey95].

As proteínas são responsáveis por várias funções dentro das células, compondo a maioria do peso seco das células. As funções podem ser, por exemplo, estruturais ou metabólicas [Mey95] [Leh00]. Como proteínas com função estrutural podem-se citar as proteínas que

fazem parte do citoesqueleto responsável pela forma da célula, sua locomoção e transporte intracelular de substâncias [Zah96]. Como função metabólica, pode-se citar as enzimas, que são proteínas especializadas para atuarem como catalisadores em várias reações químicas essenciais que ocorrem dentro das células, possibilitando que reações específicas para as quais elas são destinadas ocorram com maior rapidez.

Para executar a síntese de proteínas é necessário um novo componente chamado RNA (ácido ribonucléico) [Smi97]. O RNA é composto de uma cadeia de nucleotídeos em uma fita simples. Estes nucleotídeos podem ser Adenina (A), Guanina (G), Citosina (C) ou a base Uracila (U), ocupando o lugar da Timina (T) presente no DNA. Os passos para a síntese de proteínas são a transcrição e a tradução que aqui serão descritas conforme acontecem nos eucariontes [Mey95].

Na transcrição, a informação contida no gene será decodificada para uma fita de RNA. Este processo é semelhante ao da duplicação de DNA. Uma das fitas de DNA serve como molde e a ela é ligada uma fita de RNA. O lugar onde começa o gene a ser decodificado é indicado por marcadores especiais (seqüências de bases) chamadas de promotores. Enzimas especiais como a polimenrase RNA podem reconhecer estes promotores e se ligar a eles. Quando a decodificação chega ao fim do gene, marcadores especiais permitem que outras enzimas terminem a decodificação. Esta decodificação pode gerar três tipos de RNA necessários para a síntese protéica. Tanto a duplicação de DNA como a decodificação acontecem no sentido 5' para 3'.

O RNA ribossômico, chamado RNAr, quando associado a outras proteínas ribossômicas, formam o ribossomo, que é a unidade celular onde as proteínas são sintetizadas [Mey95]. O RNA mensageiro (RNAm) é a seqüência de RNA que contém a transcrição do gene, ou seja, um molde complementar do gene correspondente à proteína que esta sendo sintetizada. Este RNAm é transcrito no núcleo das células eucariontes. Ele não deixa o núcleo exatamente igual à forma como foi transcrito. O RNAm tem regiões codificantes que são os exons e não codificadores que são os introns [Smi97]. Antes de sair do núcleo, os introns são retirados e os exons são montados. Às vezes acontece o fenômeno da montagem alternativa (*alternative*

splicing) onde há formas diferentes de montar os exons, às vezes com a exclusão de alguns deles [Mey95].

O terceiro tipo de RNA é o RNA transportador (RNAt) [Smi97]. O RNA transportador tem a função de associar os aminoácidos necessários para a síntese protéica com as trincas de bases chamadas de códons. Como existem 64 possibilidades de trincas de 3 bases e apenas 20 aminoácidos, diferentes trincas acabam associadas ao mesmo aminoácido.

No processo de tradução, o RNAm, já fora do núcleo, liga-se aos ribossomos onde a proteína será sintetizada. Para cada códon no RNAm haverá um anticódon de um RNAt. A molécula de RNAt contém de um lado um anticódon e do outro um aminoácido associado. A cada códon lido no RNAm será trazido um aminoácido através de um RNAt específico que será ligado à cadeia de aminoácidos existente. Ao fim da leitura do RNAm, a cadeia de aminoácidos (também chamada de cadeia de polipeptídios) correspondente à proteína sintetizada estará pronta. Uma vez pronta, a cadeia de polipeptídios pode se juntar às proteínas já existentes no citoplasma ou ser enviada para alguma organela citoplasmática para processamento adicional [Mey95].

A figura 2.2 mostra a síntese de proteínas.

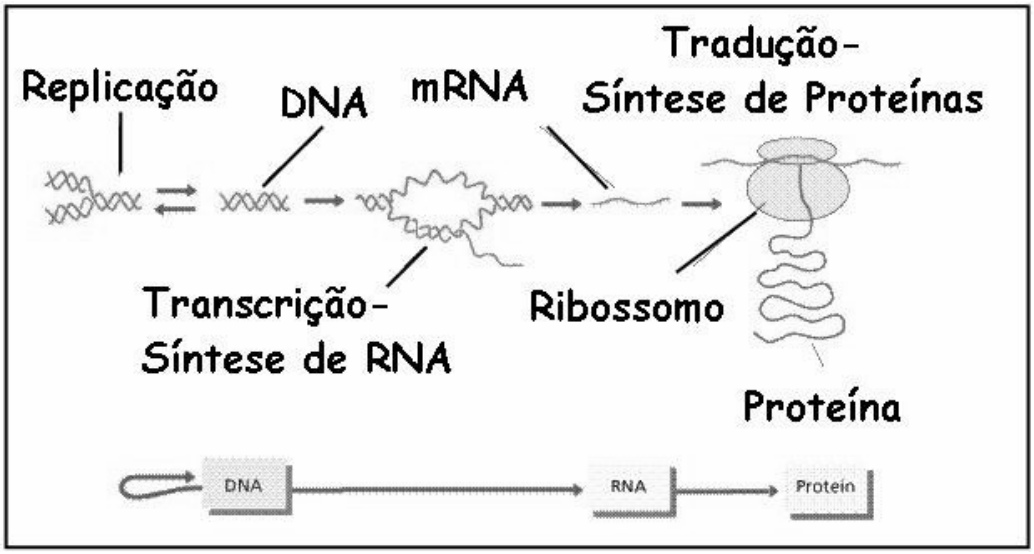

Figura 2.2 – Síntese de Proteína [Zah96]

A figura 2.2 mostra a síntese de proteínas. A transcrição é feita no DNA gerando o RNAm, a tradução é executada com RNAt no ribossomo composto de RNAr (e outros compostos) gerando uma cadeia polipeptídica.

Existem alguns RNAs que não codificam proteínas e são chamados de RNAs não codificadores [Mat06]. Embora a função da maioria dos RNAs não codificadores seja desconhecida, elas parecem estar relacionadas a diversas funções importantes na tradução, transcrição e expressão de genes.

2.2 RECONHECIMENTO APROXIMADO DE PADRÕES

Uma área de particular interesse para várias aplicações é o reconhecimento de padrões em seqüências [Nav01], que consiste em se procurar a ocorrência de uma determinada seqüência *P* em uma seqüência *B*. A seqüência *P* é considerada reconhecida em *B* se for encontrada em *B* uma seqüência suficientemente semelhante a *P*. Esta medida de semelhança irá variar dependendo dos objetivos a serem alcançados.

Nas aplicações de reconhecimento de padrões em seqüências, considera-se que o dado de interesse já foi previamente digitalizado e colocado na forma de uma seqüência de caracteres. Esta será a seqüência de base. A informação a ser procurada já foi também previamente digitalizada de forma que possa ser comparada comptacionalmente. Esta será chamada de seqüência de procura.

O reconhecimento de padrões pode ser dividido em dois tipos: exato e aproximado. No reconhecimento exato de padrões, duas seqüências são comparadas e são consideradas idênticas se elas coincidem em cada um dos seus elementos. Se eles coincidem exatamente, a seqüência foi reconhecida. Se isto não ocorre, não houve reconhecimento. Por exemplo, duas seqüências *abcde* e *abcde* são reconhecidas como sendo exatamente iguais mas as seqüências *abcde* e *abcdf* não são reconhecidas, embora sejam semelhantes.

No reconhecimento aproximado de seqüências, é importante obter uma medida do quanto duas seqüências são semelhantes para posteriormente se verificar o reconhecimento ou não do padrão. Por exemplo, as seqüências A=*abcde* e B=*abcdf*, embora não sejam idênticas, guardam uma semelhança que pode ser medida. Neste caso, a diferença pode ser contabilizada como sendo a substituição de um *e* por um *f* no final das seqüências. Cabe ressaltar que existem outras formas de se contabilizar a semelhança. Por exemplo, no caso anterior pode-se considerar que uma das seqüências teve seu último elemento retirado e que no lugar dele foi colocado um novo elemento. Isto pode contabilizar a semelhança entre as duas seqüências de forma diferente.

As complexidades no tempo e no espaço de um algoritmo para uma comparação aproximada de seqüências para o caso geral dependem de várias características das seqüências comparadas bem como o tipo e número de erros permitidos [Nav01].

No reconhecimento aproximado de padrões, procura-se não uma seqüência exata, mas uma seqüência, ou conjunto de seqüências, que mais se aproximam da seqüência de pesquisa. Para permitir esta análise, é criada uma função que permite medir a semelhança entre duas seqüências.

2.2.1 Probabilidade de Pareamento

O objetivo do reconhecimento aproximado de padrões é encontrar as ocorrências de uma seqüência procurada em uma seqüência de base. A posição onde um elemento da seqüência procurada coincide com um elemento na seqüência de base é chamada de pareamento. O pareamento de duas seqüências é a associação de todos os elementos das duas seqüências. Para se ter uma idéia da acurácia do algoritmo utilizado para esta tarefa, é interessante saber qual a probabilidade de ocorrência de um pareamento. Esta probabilidade depende da função de distância utilizada. Por exemplo, se for permitido um número indefinido de operações de inserção, remoção e substituição, então qualquer seqüência pode ser pareada.

Embora vários estudos teóricos tenham sido conduzidos, é bastante difícil conseguir uma fórmula analítica que determine qual a probabilidade de um pareamento para uma função de distância qualquer [Kur97]. Em muitos casos, são utilizadas fórmulas empíricas para estimar as probabilidades.

Supondo um alfabeto correspondendo as 4 bases nitrogenadas $\sum = \{A,T,G,C\}$, o número de elementos no alfabeto é dado por $\sigma = |\sum| = 4$. Supondo que todas as bases têm probabilidades iguais de ocorrência, teremos a probabilidade da ocorrência de uma base em uma posição qualquer da seqüência dada por $P_{base} = 1 / \sigma = 0{,}25$. Assim, o pareamento exato de duas seqüências de tamanho *n* tem a probabilidade dada por $(P_{base})^n$. Por exemplo, a probabilidade de pareamento exato de duas seqüências de tamanho 10 é dada por $(0{,}25)^{10} = 0{,}00000095$. Assim, pode ser visto que, para seqüências grandes, a probabilidade de pareamento das seqüências ao acaso é bastante pequena.

Para calcular a probabilidade de *m* pareamentos ao acaso de bases em uma seqüência de tamanho *n* assumindo uma distribuição de bases totalmente ao acaso utiliza-se a seguinte fórmula [Mey83] (2.1):

$$P_m = (n ! / (m! (n-m)!)) (P_{base})^m (1 - P_{base})^{n-m} \qquad (2.1)$$

2.3 COMPARAÇÃO DE SEQÜÊNCIAS BIOLÓGICAS

Comparar seqüências biológicas, sejam elas de DNA, aminoácidos ou outras, é uma operação extremamente útil em bioinformática [Set97]. Comparações podem servir de base para pesquisas envolvendo a descoberta da relação de parentesco entre seres vivos, indicando possibilidades de desenvolvimento de medicamentos, terapias e materiais biotecnológicos. Normalmente a comparação é feita através de uma métrica denominada de similaridade. Assim, é necessário medir o quanto uma seqüência é similar à outra.

Durante a evolução dos seres vivos, podem acontecer alterações em seus materiais genéticos. As alterações mais comuns são a mutação, a inserção e remoção de bases [Mey95]. Na

mutação de uma seqüência de DNA, uma ou mais bases podem ser modificadas. Esta alteração também é chamada de substituição pois nela uma base é substituída por outra. Na inserção, uma determinada seqüência de nucleotídeos de tamanho variável é inserida no DNA. Na remoção, uma seqüência de bases de tamanho variável é retirada do DNA.

Em uma comparação de seqüências, normalmente as duas seqüências são comparadas procurando-se descobrir o menor número de eventos de inserção, substituição e remoção que poderiam ter ocorrido para transformar uma seqüência na outra. Existem várias formas de contabilização. Em algumas delas, são atribuídos escores para cada um dos eventos possíveis (substituição, inserção e remoção). Estes valores podem ser bastante diferentes dependendo dos objetivos. Para melhor visualizar inserções e remoções, pode-se adicionar espaços (*gaps*) em uma determinada seqüência, de forma que os espaços em uma seqüência podem ser alinhados com bases em outra seqüência. Não é permitido que espaços em uma seqüência sejam alinhados com espaços na outra seqüência.

As mutações de uma seqüência à outra podem ser visualizadas na forma de um alinhamento, onde se pode ver como cada elemento (base) de uma das seqüências biológicas está relacionado a outras. O alinhamento é dito ótimo se ele minimiza o número de operações necessárias para transformar uma seqüência em outra [Fis92]. Todos os alinhamentos que tenham gerado o mesmo escore do alinhamento ótimo são considerados ótimos também.

A figura 2.3 apresenta um alinhamento entre as seqüências de DNA, GGATCT e GATACGTG, onde a pontuação para coincidências (*match*) é +1. A penalidade para não-coincidências (*mismatch*) é -1, a penalidade para a introdução de espaços (*gaps*) é -2. O escore total é obtido somando todos os valores das colunas.

```
 G  G  A  T  -  -  C  T  -
 -  G  A  T  A  C  G  T  G
-2 +1 +1 +1 -2 -2 -1 +1 -2=  -5
```

Figura 2.3 – Possível alinhamento entre duas seqüências

A comparação ilustrada na figura 2.3 utiliza matrizes unitárias, ou seja, somente as coincidências de bases possuem escores positivos. As outras três possibilidades (as outras três bases possíveis para a mesma posição) recebem o mesmo escore negativo.

2.4 ALGORITMOS PARA COMPARAÇÃO DE SEQÜÊNCIAS

Encontrar o alinhamento de melhor escore entre duas seqüências não é uma tarefa simples para seqüências grandes devido ao enorme número de possibilidades possíveis. Em [Lip84] foi estimado um número aproximado de alinhamentos possíveis, que é mostrado na fórmula (2.2):

$$A = (1 + 2^{1/2})^{2n+1}\, n^{1/2} \qquad (2.2)$$

Nesta fórmula, A é o número de alinhamentos possíveis e n o tamanho das seqüências (supondo tamanho igual). Embora o número de alinhamentos aumente exponencialmente com o tamanho das seqüências (aproximadamente $O(2^{2n+1})$) eles se acomodam em uma matriz de programação dinâmica que possui tamanho proporcional ao produto dos tamanhos das seqüências ($O(n^2)$) [Lip84]. Como a complexidade do número de alinhamentos possíveis é bem maior do que a matriz utilizada para calcular os alinhamentos, temos que em uma comparação de seqüências grandes, podemos ter vários alinhamentos que compartilham trechos em comum [Lip84].

2.4.1 Alinhamento Global (Needleman-Wunsch)

O alinhamento é dito global quando se tenta encontrar o melhor alinhamento possível utilizando todos os elementos das seqüências [Set97]. Para encontrar os possíveis alinhamentos, investigam-se os escores gerados pelas operações de tentativa de pareamento (sendo que os elementos podem coincidir ou não). Quando ocorre a coincidência, ela é contabilizada positivamente e quando ela não ocorre, é contabilizada negativamente. Também são penalizadas as inserções e remoções. Este tipo de alinhamento é interessante para se saber o quanto as duas seqüências são semelhantes. Este alinhamento também serve como um

indicador de quantos eventos genéticos são necessários para transformar uma seqüência em outra [Nav01].

Em [Nee70], Needleman e Wunsch propuseram um algoritmo baseado em programação dinâmica para o problema do alinhamento global.

Para solucionar o problema do alinhamento global utilizando a técnica de programação dinâmica [Cor90] são considerados os elementos iniciais (prefixos) das duas seqüências. Na medida em que estes prefixos são alinhados, os escores parciais calculados relativos aos prefixos são utilizados para calcular os escores para os prefixos maiores. Assim, a cada passo dois elementos (um de cada seqüência comparada) são comparados. O resultado da comparação é utilizado conjuntamente com os escores já obtidos (com o alinhamento prévio dos prefixos) para se obter o valor atual.

Para cada comparação de elementos existem três possibilidades:

a) Alinhar os dois elementos, neste caso podendo haver coincidência ou não;

b) Alinhar um espaço da primeira seqüência com um elemento da segunda seqüência, isto corresponde a uma remoção na primeira seqüência ou uma inserção na segunda;

c) Alinhar um elemento da primeira seqüência com um espaço da segunda seqüência, isto corresponde a uma inserção na primeira seqüência ou uma remoção na segunda;

Sejam duas seqüências $A= a_1\ a_2\ a_3\ ...\ a_m$ e $B= b_1\ b_2\ b_3\ ...\ b_n$ onde m e n são os tamanhos das respectivas seqüências. Os escores parciais (correspondentes aos alinhamentos dos prefixos) são mantidos em uma matriz de programação dinâmica M, composta de mxn elementos M[1..m,1..n]. Cada um dos elementos M[i,j] da matriz é calculado utilizando um elemento anterior e as três possibilidades de alinhamento descritas. Estas operações possíveis são descritas pelas seguintes relações de recorrência (2.3):

$$M(i,j) = \textbf{máximo} \begin{cases} M(i-1,j) + CI \text{ (custo da inserção)} \\ M(i-1,j-1) + CC \text{ (custo da comparação)} \\ M(i,j-1) + CR \text{ (custo da remoção)} \end{cases} \quad (2.3)$$

Os valores iniciais M(1..*i*,0) são *i* * (custo da inserção) para cada elemento e M(0,1..*j*) é *j* * (custo da remoção) para cada elemento. Os custos de inserção (CI), comparação (CC) e remoção (CR) podem ser diferentes dependendo do objetivo a ser alcançado. Uma forma de pontuação é CI com valor -2 e CR com valor -2 também. O custo da comparação depende dos elementos avaliados. Por exemplo, se os elementos forem iguais o escore pode ser 1 e se os elementos forem diferentes o escore pode ser -1.

De acordo com as relações de recorrência, o cálculo de cada célula M[*i,j*] da matriz de programação dinâmica depende do valor de três células M[*i*-1,*j*],M[*i*-1,*j*-1],M[*i,j*-1]. Para cada caso, existem três possibilidades: a) pode-se alinhar $a_1 \ldots a_i$ com $b_1 \ldots b_{j-1}$ e alinhar b_j com um espaço, b) pode-se alinhar $a_1 \ldots a_{i-1}$ com $b_1 \ldots b_{j-1}$ e alinhar a_i com b_j e c) pode-se alinhar $a_1 \ldots a_{i-1}$ com $b_1 \ldots b_j$ e alinhar a_i com um espaço.

Para poder recuperar o alinhamento, é guardada a informação de quais entre as três células foram escolhidas na relação de recorrência. Esta informação pode ser armazenada em 3 bits, cada um indicando qual das 3 células foi utilizada. Dadas as relações de dependência no cálculo das células, necessitamos apenas de uma linha da matriz para calcular a próxima linha ou de uma coluna da matriz para calcular a próxima coluna. Alternativamente, pode-se utilizar uma anti-diagonal para se calcular a próxima [Nee70].

Ao final do cálculo dos elementos da matriz de programação dinâmica, a última célula à direita e abaixo M(*i*+1,*j*+1) conterá o escore total do alinhamento. Para descobrir o melhor alinhamento, é necessário descobrir qual foi a seqüência de operações que levou a este resultado. Para saber isto, é necessário que cada célula guarde a informação sobre qual das três células (M[*i*-1,*j*],M[*i*-1,*j*-1],M[*i,j*-1]) foi utilizada para o cálculo do escore naquela posição. Com esta informação é possível percorrer o caminho reverso (*backtracking*) do

elemento M[*i*+1,*j*+1] até o elemento M[0,0]. Este caminho (ou caminhos, caso haja mais de um) corresponde ao alinhamento global ótimo.

A complexidade deste algoritmo no tempo é proporcional a O(*mn*) pois é necessário o cálculo das células da matriz de programação dinâmica[Cor90]. O espaço necessário depende das relações de precedência que têm de ser respeitadas. Para calcular os elementos da matriz é necessário manter na memória apenas duas linhas ou colunas (uma com o resultado anterior e outra que está sendo calculada). Assim, o espaço necessário é proporcional a O(*n*) onde *n* é o tamanho da linha ou coluna utilizada. No entanto, para recuperar o alinhamento, deve-se fazer o percurso reverso da matriz (*backtraking*) sobre os elementos da matriz. Isto nos leva à complexidade O(*mn*) de espaço.

A seguir, será apresentado um exemplo de alinhamento global utilizando o algoritmo de Needleman-Wunsch. As seqüências a serem comparadas são A=TAGGAG e B=TGCTAG. Existem várias formas de pontuação dos escores [Set97]. Neste exemplo, as inserções de espaços (*gaps*) terão escore -1. As coincidências (*matches*) receberão escore +1 as substituições (*mismatches*) receberão escore -1. A matriz de programação dinâmica resultante é mostrada na figura 2.4.

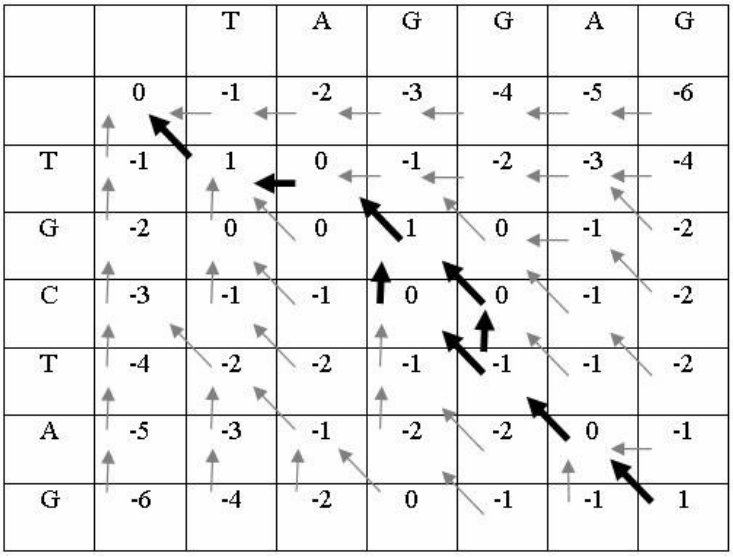

Figura 2.4 – Matriz de programação dinâmica do Alinhamento Global

Para recuperar um alinhamento global ótimo, começa-se da célula mais a direita e abaixo que possui o escore do alinhamento global que é 1, com a matriz M[0..6,0..6], percorre-se o caminho reverso para obter um ou mais alinhamentos ótimos.

As figuras 2.5(a) e 2.5(b) mostram os alinhamentos ótimos recuperados com a matriz de figura 2.4:

```
A =   T   A   G   -   G   A   G

B =   T   -   G   C   T   A   G
      1  -1   1  -1  -1   1   1  =  1
```

(a)

```
A =   T   A   G   G   -   A   G

B =   T   -   G   C   T   A   G
      1  -1   1   1  -1   1   1  =  1
```

(b)

Figura 2.5 – Dois alinhamentos globais possíveis

2.4.2 Alinhamento Local (Smith-Waterman)

O alinhamento local difere do alinhamento global por alinhar trechos das seqüências ao invés de alinhar as seqüências inteiras, procurando por regiões de alta similaridade dentro das mesmas [Set97].

O algoritmo proposto por Smith-Waterman [Smi81] é baseado no algoritmo de Needleman-Wunsch (seção 2.4.1), porém adaptado para tratar o problema do alinhamento local. No Needleman-Wunsch, o melhor alinhamento pode ter um escore bastante negativo, desde que seja o maior obtido. Para impedir isto, a equação 2.7 foi modificada impedindo valores menores que zero.

Como o Smith-Waterman tem caráter local, não são permitidos alinhamentos com um número muito grande de inserções, remoções e substituições. Isto é feito atribuindo um valor zero caso o escore se torne negativo. Isto faz com que o algoritmo busque a maior região de similaridade entre duas seqüências.

O alinhamento local de Smith-Waterman [Smi81] utiliza uma pontuação de espaços lineares. Neste caso, possui complexidade no tempo e no espaço de $O(n^2)$ onde n é o tamanho das seqüências. O cálculo de cada célula da matriz de programação dinâmica é feito, então, de acordo com a fórmula mostrada em (2.4):

$$M(i,j) = \textbf{máximo} \begin{cases} 0 \\ M(i-1,j) + CI \text{ (custo da inserção)} \\ M(i-1,j-1) + CC \text{ (custo da comparação)} \\ M(i,j-1) + CR \text{ (custo da remoção)} \end{cases} \qquad (2.4)$$

Os valores iniciais M(1..i,0) e M(0,1..j) são zero para cada elemento. De mesma forma que no alinhamento global (seção 2.4.1), o cálculo de cada célula M[i,j] da matriz de programação dinâmica depende do valor de três células M[i-1,j],M[i-1,j-1],M[i,j-1].

Além da introdução do valor zero na equação de recorrência e da inicialização da primeira linha e coluna da matriz M com valor zero, a recuperação do melhor alinhamento local, pode ocorrer a partir de qualquer lugar da matriz de programação dinâmica. Assim, para descobrir o alinhamento ótimo é preciso encontrar a célula (ou células) com maior escore da matriz e percorrer o caminho inverso até encontrar uma célula que tenha valor zero. No alinhamento local, também é possível haver vários alinhamentos ótimos a partir de uma mesma célula, sendo portanto interessante percorrer todas as possibilidades de operações para achá-los.

A seguir será apresentado um exemplo do alinhamento local utilizando o algoritmo de Smith-Waterman utilizando também as seqüências A=TAGGAG e B=TGCTAG, com as seguintes pontuações: será atribuído -1 para as inserções de espaço (inserções e remoções de elementos)

e também -1 para o caso do pareamento onde os elementos não coincidem. Será dado escore +1 para o pareamento onde os elementos coincidem. A matriz de programação dinâmica correspondente está na figura 2.6.

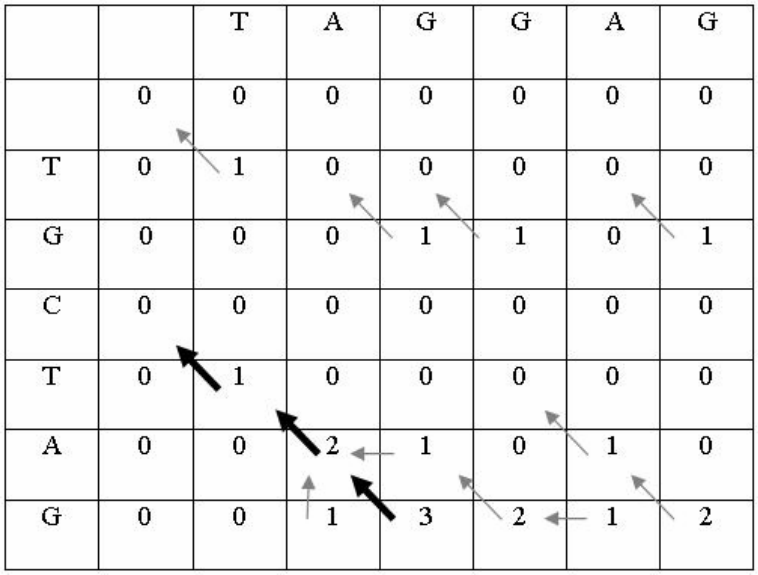

Figura 2.6 - Matriz de programação dinâmica do Alinhamento Local

Para achar o melhor alinhamento percorre-se o caminho reverso a partir da célula com maior valor M(6,3), até encontrar uma célula com valor zero.

O resultado final do alinhamento local ótimo será dado pelo inverso do alinhamento feito durante o percurso da matriz. Isto é mostrado na figura 2.7:

```
A =    T  A  G
B =    T  A  G
```

Figura 2.7 – Alinhamento Local

Nos exemplos das figuras 2.5 e 2.7, podemos ver claramente a diferença de um alinhamento global e um local. No alinhamento global foi feita uma tentativa de alinhar as seqüências como

um todo. Como resultado o algoritmo mediu o grau de similaridade entre as seqüências mas não reconheceu uma subseqüência comum (TAG). Já o algoritmo de alinhamento local reconheceu esta subseqüência comum mas não mediu a similaridade total das seqüências.

2.4.3 Algoritmo Global em Espaço Linear

Hirschberg [Hir75] propôs um algoritmo capaz de obter alinhamento global ótimo em espaço linear O(n), onde n é o tamanho da menor seqüência. Isto é feito percorrendo-se a matriz de programação dinâmica várias vezes para as várias posições por onde passa o alinhamento ótimo. Assim o algoritmo troca um aumento no tempo de execução por menor complexidade no espaço. A complexidade no tempo continua sendo O(nm) embora o tempo total seja multiplicado por uma constante próxima de 2.

O algoritmo de Hirschberg trabalha com a idéia de partição binária. Dada uma matriz M mxn onde m é o número de linhas e n número de colunas, o primeiro objetivo é determinar por qual linha i passa o alinhamento ótimo na coluna n/2. Para descobrir isto, os escores são calculados tanto partindo da célula mais à esquerda e acima como da célula mais abaixo e à direita. Quando o cálculo dos escores chegar na coluna n/2, o melhor valor obtido pelos dois cálculos dos escores, tanto o que vem da esquerda como o que vem da direita, será na célula por onde passa o alinhamento global ótimo M(i,n/2). A partir deste ponto, a matriz pode ser particionada em duas matrizes M(1..i,1..(n/2)) e M(i..m,(n/2)..n) e o procedimento descrito acima é repetido para as submatrizes resultantes, recursivamente, até que todos os elementos do alinhamento ótimo sejam determinados. O número de células a serem calculadas aproximadamente igual à metade do número da matriz anterior. Assim, se a matriz original tem mn elementos, no algoritmo de Hirschberg calcula-se o valor de (mn + (mn/2) + (mn/4) + ...) células, o que dá aproximadamente 2mn células no total.

A seguir será apresentado um exemplo do alinhamento global utilizando o algoritmo de Hirschberg utilizando também as seqüências A=TAGGAG e B=TGCTAG. Será atribuído -1 para as inserções de espaço (inserções e remoções de elementos) e também -1 para o caso do pareamento onde os elementos não coincidem. Será dado escore +1 para o pareamento onde os elementos coincidem. A matriz de programação dinâmica correspondente está na figura 2.8.

		T	A	G	G	A	G		
	0	-1	-2	-3	-4	-5	-6		
T	-1	1	0	-1	-2+0	-2	-4	-6	T
G	-2	0	0	1	0-1	-1	-3	-5	A
C	-3	-1	-1	0	0+0	0	-2	-4	G
T	-4	-2	-2	-1	-1+1	1	-1	-3	G
A	-5	-3	-1	-2	-2+1	2	0	-2	A
G	-6	-4	-2	0	-1-1	0	1	-1	G
		-6	-5	-4	-3	-2	-1	0	
		T	G	C	T	A	G		

Figura 2.8 - Matriz de programação dinâmica do algoritmo de Hirschberg 1

Na figura 2.8 pode ser visto que o alinhamento é calculado nas duas direções. No canto superior esquerdo temos o alinhamento de TAGGAG com TGCTAG. No canto inferior direito temos o alinhamento de GATCGT e GAGGAT (as duas bases iniciais G são alinhadas na célula inferior direita). A figura mostra o cálculo de duas matrizes diferentes sendo que uma é o inverso da outra. Assim o resultado do alinhamento das duas tem de ser igual por se tratarem das mesmas seqüências em ordem inversa. A vantagem deste cálculo é que o cálculo pode ser feito em duas direções simulataneamente. A coluna que aparece em cinza é a coluna onde os elementos coincidem (sendo comum para os pares de seqüências alinhadas) e onde foi descoberta uma célula pertencente ao alinhamento ótimo em ambas as matrizes. O ponto de encontro é dado pela célula que tem o maior escore resultante da soma do percorrimento normal e reverso. No caso de haver mais de uma, como no exemplo onde as células M[3,4] e M[4,4] tem o maior escore da soma que é zero, se decide pela célula que faz parte do alinhamento parcial ótimo (direto e reverso) que no caso é M[4,4]. Assim, o elemento M[4,4] já é marcado como por uma linha forte, fazendo parte do alinhamento ótimo. O próximo passo é mostrado na figura 2.9. Uma vez descoberto o elemento que faz parte do alinhamento ótimo, a matriz agora pode ser dividida

em duas submatrizes mostradas em cinza, uma superior esquerda e outra inferior direita ambas marcadas em cinza. A submatriz no canto superior esquerdo mostra a comparação de TAG com TGC. A submatriz no cando inferior direito mostra a comparação de GAT com GAG. Isto faz com que o número de novos escores a serem calculados seja a metade do passo anterior, pois agora as submatrizes têm aproximadamente a metade do número de células da matriz original.

		T	A	G	G	A	G		
	0	-1	-2	-3	-4	-5	-6		
T	-1	1	0	-1		-2	-4	-6	T
G	-2	0	0	1		-1	-3	-5	A
C	-3	-1	-1	0		0	-2	-4	G
T	-4	-2	-2	-1	0	1	-1	-3	G
A	-5	-3	-1	-2		2	0	-2	A
G	-6	-4	-2	0		0	1	-1	G
		-6	-5	-4	-3	-2	-1	0	
		T	G	C	T	A	G		

Figura 2.9 - Matriz de programação dinâmica do algoritmo de Hirschberg 2

O processo é repetido dentro das submatrizes, procurando dentro delas o elemento pertencente ao alinhamento ótimo que fica aproximadamente na coluna do meio, fazendo uma partição binária.

A figura 2.10 mostra o resultado final.

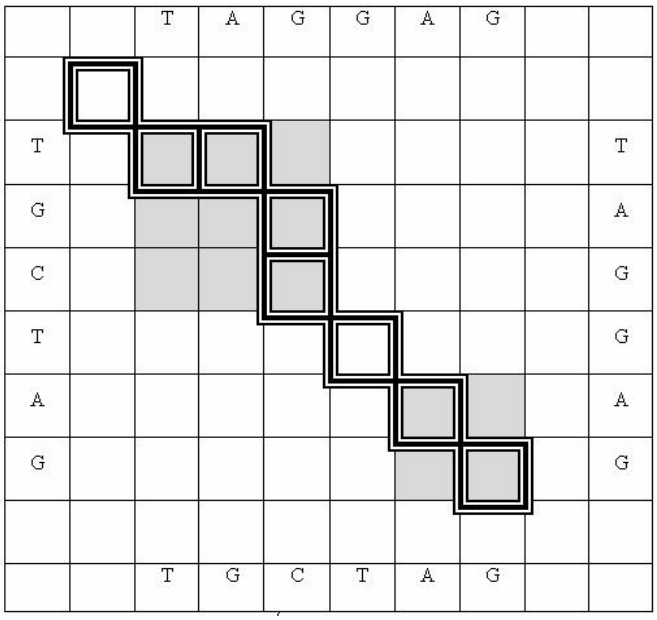

Figura 2.10 Alinhamento ótimo produzido na Matriz do algoritmo de Hirschberg

O resultado é o alinhamento dado na figura 2.11:

A = T A G – G A G

B = T – G C T A G

+1 -1 +1 -1 -1 +1 +1 = 1

Figura 2.11 – Alinhamento produzido pelo algoritmo de Hirschberg

Vale ressaltar que os escores calculados nas submatrizes e nas células no meio da matriz (posição onde deve passar o alinhamento ótimo) não têm utilidade para calcular o escore total do alinhamento global. Sendo assim, para obter este escore é necessário computar o escore baseado no alinhamento global encontrado.

2.4.4 Alinhamento Local em Espaço Linear

Existe um método para se calcular alinhamento ótimo local em espaço linear [Gus97]. O primeiro passo é calcular a matriz de escores do alinhamento local (seção 2.4.2) por completo,

porém somente armazenando a última linha (ou coluna) e o melhor escore obtido até o momento, bem como sua posição na matriz. Depois de terminar o cálculo da matriz, a posição onde se encontra o maior escore indica o final de um alinhamento ótimo. Com isto, falta descobrir onde começa o alinhamento. Isto é feito invertendo as seqüências e calculando os escores novamente. A célula descoberta anteriormente (maior escore) é o ponto de partida para o cálculo. A partir daí, quando se encontra uma célula da matriz com o mesmo valor do maior escore, encontrou-se o início de um alinhamento ótimo. Nesta fase, tem-se duas subseqüências que quando alinhadas produzem um alinhamento ótimo. O passo seguinte é fazer um alinhamento global das duas subseqüências onde se obtém o mesmo alinhamento conseguido no alinhamento local [Cha95]. A vantagem é que este alinhamento global pode ser realizado em espaço linear com o algoritmo de Hirschberg (seção 2.4.3)**.**

Como exemplo, temos a matriz gerada pelo alinhamento local das seqüências TAGGAG e TGCTAG mostrada na seção 2.4.3 que é mostrada na figura 2.12.

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

Figura 2.12 - Matriz de programação dinâmica do Alinhamento Local

Na figura 2.12 podemos notar que o maior escore (3) é gerado na terceira coluna e na sexta linha. Sabendo onde o alinhamento local termina, este se torna o início de um novo alinhamento sobre o reverso das seqüências [Gus97]. O algoritmo local é executado novamente apenas para saber onde começa o alinhamento, parando após a obtenção do maior escore (3). Com o início e o final do melhor alinhamento local, o problema transforma-se em

um problema de alinhamento global entre partes da seqüência e pode-se aplicar um algoritmo de espaço linear como o algoritmo de Hirschberg (seção 2.4.3) para a obtenção do alinhamento.

A figura 2.13 mostra a matriz gerada pela inversão das seqüências. Nela podemos ver que a posição do maior escore encontrado na figura 2.12 (3) está marcada na figura e se torna o início de um alinhamento. A partir daí, é feito o cálculo da matriz até se encontrar novamente o maior escore 3. Este indicará o final da subseqüência que está sendo alinhada.

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

Figura 2.13 - Matriz de programação dinâmica do Alinhamento Local

O passo seguinte é fazer o alinhamento global entre as duas subseqüências. Isto é mostrado na figura 2.14.

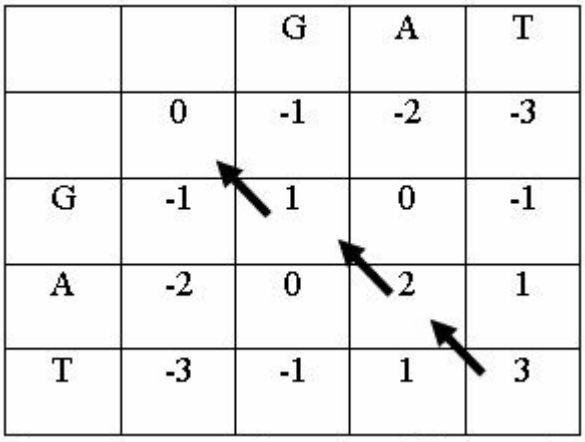

Figura 2.14 – Alinhamento global parcial

O alinhamento global mostrado na figura 2.14 pode ser feito em espaço linear usando Hirschberg e é o mesmo obtido pelo alinhamento local [Gus97].

2.4.5 DIALIGN

O DIALIGN é um método para alinhamento de seqüências que pode ser utilizado tanto para alinhar pares de seqüências quanto para alinhamento múltiplo [Mor98] [Sch04]. Este método procura por sub-alinhamentos sem "*gaps*" chamados de fragmentos (ou diagonais) e depois procura alinhar estes fragmentos da melhor forma possível. Como o DIALIGN não tem "*gaps*" não é necessário o tratamento destes, facilitando a implementação e diminuindo o número de parâmetros que podem influenciar os resultados. Além disto, o algoritmo desconsidera regiões fora das diagonais que possuem baixa similaridade, evitando problemas do alinhamento global que podem acontecer no algoritmo de Needleman-Wunch [Mor99].

O DIALIGN é capaz de detectar pequenas regiões de similaridade que não podem ser detectadas por outros algoritmos e as entender na forma de um alinhamento global [Mor99].

O primeiro passo para o alinhamento múltiplo é a comparação entre pares de seqüências [Mor96]. Dadas duas seqüências se faz o "*Dot-matrix*" delas, que é simplesmente uma matriz de tamanho *M*x*N* onde *M* é o tamanho da seqüência *A* e *N* é o tamanho da seqüência *B*. Cada posição (*i*,*j*) desta matriz contém o valor 1 se o elemento na posição *i* na seqüência *A* é igual

ao elemento na posição *j* da seqüência *B* e 0, caso contrário. Esta matriz apresenta os fragmentos que são sub-alinhamentos de subseqüências de *A* e *B*. Estes fragmentos, também chamados de diagonais não utilizam "*gaps*" e, por isto, as subseqüências alinhadas neles são do mesmo tamanho. A seguir, é necessário medir a significância estatística destas diagonais. Só serão utilizadas no alinhamento aquelas que ultrapassarem um dado limiar.

Para uma diagonal *D* com tamanho *l*, a probabilidade *P*(*l,m*) de uma diagonal *D* de tamanho *l* ter pelo menos *m* "*matches*" é dada pela equação 2.5 [Mor96]:

$$P(l,m) = \sum_{i=m}^{l} \left(\frac{l}{i} \right) p^{i} (1-p)^{l-i} \qquad (2.5)$$

Neste caso, *p* é a probabilidade de um "*match*". Para seqüências de DNA temos *p*=0,25 e para seqüências de aminoácidos temos *p*=0,05.

A probabilidade pode ser convenientemente representada pela fórmula 2.6 [Mor96]:

$$E(l,m) = - \ln(P(l,m)) \qquad (2.6)$$

Para cada diagonal *D* é atribuído um peso *w*(*D*) dada pela fórmula 2.7 [Mor96]:

$$\mathrm{w}(D) = \begin{cases} E(l,m), \text{ se } E(l,m) > T \\ 0 \text{ caso contrário} \end{cases} \qquad (2.7)$$

Este peso dá uma medida da significância das diagonais. Se o limiar *T* definido pelo usuário for alcançado, então a diagonal é considerada significante, senão ela é descartada. Quanto maior o valor de *w*(*D*), menor a probabilidade de *m* "*matches*" acontecerem em uma seqüência de tamanho *l* por acaso.

O alinhamento total de duas seqüências é obtido através do alinhamento das diagonais. Este alinhamento é feito de forma consistente, ou seja, se uma posição (i,j) pertence a um fragmento D_x então para qualquer diagonal (fragmento) D_y no mesmo alinhamento não existe nela posição (k,l) tal que $k=i$ ou $l=j$. Assim, um elemento de uma seqüência não pode ser alinhado mais de uma vez no mesmo alinhamento. Um alinhamento com k fragmentos D_1, D_2,.., D_k tem seu escore total S dado pela fórmula 2.8 [Mor96]:

$$S = \sum_{i=1}^{k} w(D_i) \qquad (2.8)$$

Para descobrir o alinhamento e o escore é necessário alinhar os fragmentos utilizando um algoritmo de programação dinâmica. O alinhamento entre duas seqüências $A=a_1a_2...a_M$ e $B=b_1b_2...b_N$ com tamanhos M e N respectivamente para cada par (i,j) com $1 \leq i \leq M$ e $1 \leq i \leq N$ se determina todos os inteiros k com $k \leq \min(i,j)$ onde o fragmento $(a_{i-k}b_{i-k},...,a_ib_j)$ começando da posição $(i-k,j-k)$ até a posição (i,j) tem um peso w positivo. Assim, para cada posição (i,j), será definido um escore(i,j) para o alinhamento de fragmentos D_1, D_2,.., D_k nos prefixos $(a_1a_2...a_i)$ e $(b_1b_2...b_j)$. O último fragmento D_k alinhado na posição (i,j) é recuperado pela função prec$(i,j)= D_k$. Para cada fragmento $D=(a_{i-k}b_{i-k},...,a_ib_j)$ com peso positivo é definido $\sigma(D)$ como o peso total acumulado pela soma dos escores de todos os fragmentos até D, incluindo o próprio D. A função $\pi(D)$ é definida como o fragmento que precede D.

Os valores de $\sigma(D)$ e $\pi(D)$ são calculados pelas relações de recorrência representadas na fórmulas 2.9 e 2.10 [Mor96]:

$$\sigma(D) = \text{escore}(i-k-1,j-k-1) + w(D) \qquad (2.9)$$

$$\pi(D)= \text{prec}(i-k-1,j-k-1)$$

$$\text{escore}(i,j) = \max\{\text{escore}(i-1,j), \text{escore}(i,j-1), \sigma(D_{i,j}) \} \qquad (2.10)$$

Onde $D_{i,j}$ é definido como o maior fragmento que termina no ponto (i,j) que satisfaz a fórmula 2.11 [Mor96]:

$$\sigma(D_{i,j}) = \max \{ \sigma(D) : D \text{ termina no ponto } (i,j) \} \qquad (2.11)$$

O valor de prec(i,j) será definido segundo a fórmula 2.12 [Mor96]:

$$\text{prec}(i,j) = \begin{cases} \text{prec}(i,j\text{-}1) & \text{Se escore}(i,j)=\text{escore}(i,j\text{-}1) \\ \text{prec}(i\text{-}1,j) & \text{Se escore}(i,j\text{-}1) < \text{escore}(i,j) = \text{escore}(i\text{-}1,j) \\ D_{i,j} & \text{Se escore}(i,j\text{-}1),\ \text{escore}(i\text{-}1,j) < \text{escore}(i,j) = \sigma(D_{i,j}) \end{cases} \qquad (2.12)$$

Uma vez calculada a matriz, basta fazer o precurso inverso para encontrar o alinhamento. Na figura 2.15 é mostrado como um alinhamento pode ser gerado pelos fragmentos.

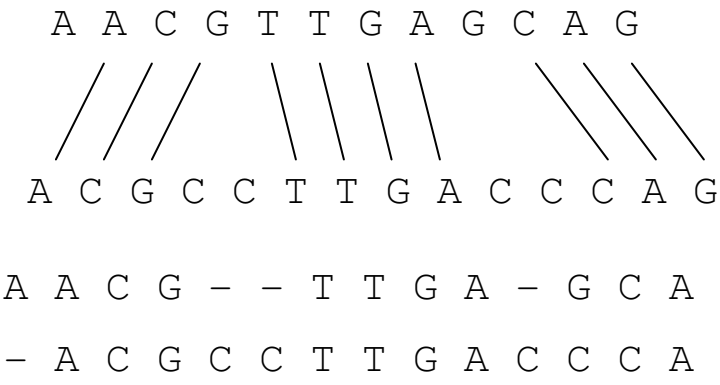

Figura 2.15 – Alinhamento de diagonais

A figura 2.15 na parte (A) mostra os fragmentos que foram alinhados indicados pelas linhas conectando as bases. A parte (B) mostra o alinhamento final.

Para fazer o alinhamento múltiplo, são feitos inicialmente $n(n\text{-}1)/2$ alinhamentos onde n é o número de seqüências. Após os alinhamentos, são guardados os fragmentos encontrados que têm o maior peso w. Para aumentar o desempenho podem ser guardados apenas os fragmentos que pertencem a alinhamentos ótimos [Mor96] mas outras abordagens utilizando todos os fragmentos são possíveis [Mor99]. Estes fragmentos são ordenados de acordo com o peso. Para fazer um alinhamento múltiplo, o fragmento com maior peso é colocado inicialmente. Os

fragmentos com pesos menores são colocados um a um em ordem decrescente de peso desde que seja consistente com o alinhamento múltiplo já existente. Os que não forem consistentes são descartados. O procedimento é repetido até que nenhum fragmento possa mais ser adicionado.

3 – SISTEMAS DEDICADOS

Grande parte das arquiteturas de computação disponíveis atualmente são baseadas em Processadores de Propósito Geral (PPGs). Esta classe de arquiteturas é caracterizada por processadores que executam um conjunto genérico de instruções [Pat98]. A computação a ser executada nestas arquiteturas é ditada por um programa em código de máquina. Assim, os mesmos processadores podem executar ações diferentes dependendo do programa utilizado. Para que esta abordagem seja possível, é necessário que o PPG tenha flexibilidade suficiente para executar as várias instruções que um programa necessita. Além disto, para ser eficiente um PPG deve executar as instruções rapidamente.

Apesar de amplamente utilizados, os PPGs apresentam algumas dificuldades em seu desenvolvimento [Ada99]:

a) Gasto de Energia: Na medida em que o número de transistores aumenta e a velocidade de relógio (*clock*) sobe, aumenta também o gasto de energia para o processamento. Em muitos casos, muitos circuitos internos são desnecessários durante grande parte do processamento.

b) Custo de fabricação: Os PPGs são feitos em muitos casos com a melhor tecnologia de fabricação disponível para serem rápidos, mas isto também os tornam caros.

c) Custo de projeto: Como os PPG devem ser flexíveis e devem executar várias instruções diferentes, o seu projeto é complexo.

Para tentar executar aplicações específicas com maior velocidade e menor consumo de energia, foram criadas arquiteturas específicas ASICs (*Application Specific Integrated Circuits*). Estas arquiteturas são otimizadas para obterem um alto desempenho em relação à velocidade de processamento e ao gasto de energia apresentando. Por outro lado, têm algumas desvantagens tais como [Enz99]:

a) alto custo de projeto
b) grande custo de produção para volumes pequenos
c) ausência de flexibilidade

O custo de projeto de uma arquitetura dedicada deve ser compensado por uma maior eficiência na execução da aplicação.

A ausência de flexibilidade impede que a arquitetura possa ser modificada para se enquadrar em novas especificações. Esta é uma grande limitação, uma vez que a arquitetura toda deve ser descartada se, por alguma razão, ela não satisfaz mais as especificações.

Uma importante aplicação de sistemas dedicados é na aceleração de algoritmos por hardware. Neste caso, os sistemas dedicados comportam-se como um co-processador dedicado acoplado a uma máquina hospedeira através de alguma tecnologia de comunicação [Com02]. Uma das limitações deste tipo de arquitetura é o gargalo de comunicação. Apesar do processamento em um circuito poder ser realizado em grande velocidade, se há necessidade de comunicação freqüente, o desempenho total acaba sendo limitado pela largura de banda da interconexão entre o circuito dedicado e a máquina hospedeira. Esta largura de banda, na maioria dos casos, não é comparável à capacidade de processamento disponível.

Uma arquitetura dedicada possui um paralelismo inerente onde cada unidade funcional que compõe o circuito pode atuar em paralelo com outras. Este paralelismo inerente é bem difícil de ser utilizado em toda a sua extensão por vários motivos. Em primeiro lugar, muitas aplicações possuem dependências de dados que impossibilitam a paralelização de várias operações. Um exemplo disto são os cálculos iterativos onde, para calcular o valor atual *f(n)*, é necessário ter calculado o valor *f(n-1)*. Nestes casos, é impossível calcular *f(n)* e *f(n-1)* em paralelo. Outra limitação para a utilização total do paralelismo inerente é a dificuldade de projetar hardware paralelo. Esta dificuldade se dá pela dificuldade de transpor algoritmos planejados para serem seriais para uma execução em paralelo em hardware [Com02].

Geralmente, quanto maior o paralelismo de operações, maior será o desempenho do circuito,

mas também maior deverá ser a área do mesmo circuito. Assim, uma maior complexidade do projeto acarretará um maior gasto de energia. Normalmente em várias arquiteturas, quanto maior o tamanho do circuito, menor a velocidade máxima do relógio. Assim, encontrar a solução de melhor desempenho é encontrar uma solução de compromisso entre o quanto se pode paralelizar o problema e qual a velocidade máxima de execução.

O paralelismo de operações pode ser classificado quanto ao grau de granulosidade [Elr98]. A granulosidade está relacionada à complexidade dos elementos de computação que realizam as operações. Na granulosidade grossa, as operações realizadas nos elementos de computação são mais complexas como, por exemplo, somadores e comparadores. Na granulosidade fina, as operações executadas são mais simples como funções lógicas de duas entradas. Quanto maior a granulosidade, normalmente menor o grau de comunicação. Assim, se a granulosidade for fina [Kou91], o grau de comunicação e controle entre estes elementos é geralmente alto. Desta forma, o projeto de sistemas dedicados deve levar em consideração não apenas o máximo de paralelismo que pode ser conseguido em suas operações, mas também qual será o custo necessário para a comunicação entre os elementos de computação e o seu controle efetivo, para que sejam respeitadas as precedências de dados e o fluxo de controle. Segundo [Har01], uma granulosidade excessivamente fina em uma arquitetura dedicada (por exemplo de 1 bit) é ineficiente devido às dificuldades de roteamento de dados entre os elementos.

Assim, uma característica importante no projeto de sistemas dedicados é a implementação de fluxos de controle e comunicações que sejam simples e regulares.

3.1 ARQUITETURAS SISTÓLICAS E WAVEFRONT

O nome da arquitetura sistólica vem da analogia com a sístole cardíaca onde o sangue é bombeado através das artérias. De forma semelhante, em uma arquitetura sistólica, os dados são bombeados através dos elementos de computação [Kun82], cadenciados por um relógio.

Os vetores sistólicos e os vetores wavefront são ambos compostos por vários elementos de processamento interligados. A principal diferença entre eles está na sincronização do fluxo de

dados. Nas arquiteturas sistólicas os dados são transmitidos de forma síncrona, a cada transição do relógio. Nas arquiteturas wavefront a comunicação é assíncrona, feita através de um protocolo de *handshaking* [Kun87].

A idéia básica da arquitetura sistólica ou wavefront é dividir a tarefa a ser realizada entre elementos de computação simples, que conseqüentemente poderão executar rapidamente a sub-tarefa designada. A seguir, os dados computados poderão ser novamente utilizados, sendo que para isto serão enviados através da rede de interconexão.

Em uma arquitetura sistólica ou wavefront, os elementos de computação normalmente são organizados em estruturas regulares como, por exemplo, matrizes unidimensionais ou bidimensionais. Existe a necessidade da arquitetura poder se comunicar com subsistemas externos através de operações de entrada e saída. Normalmente, esta comunicação é feita pelos elementos de computação que estão nas bordas da arquitetura sistólica ou wavefront.

A figura 3.1 mostra uma representação de uma arquitetura bidimensional onde os quadrados são os elementos de computação e as setas indicam os possíveis fluxos de dados destes elementos. Podemos ver que cada elemento se comunica com até 6 elementos vizinhos. Os elementos que ficam na borda podem se comunicar com outras estruturas como, por exemplo, uma memória RAM.

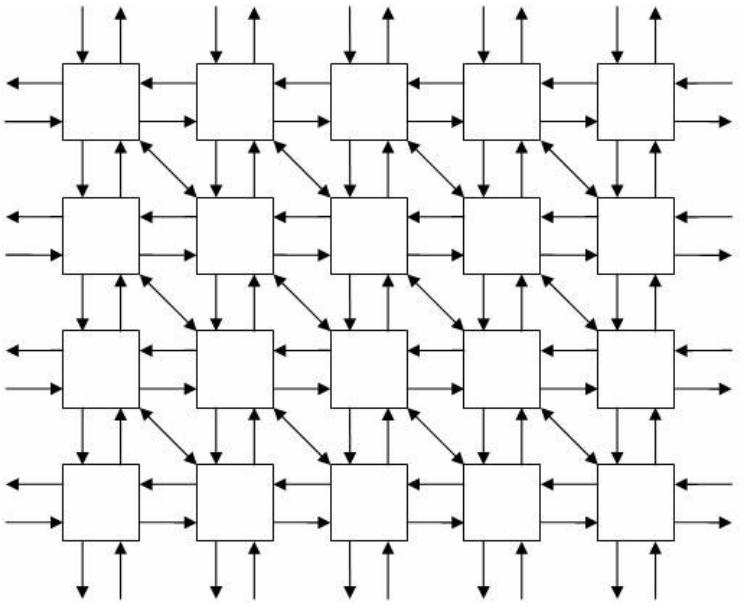

Figura 3.1- Arquitetura bidimensional [Kun87]

A figura 3.2 mostra uma representação de uma arquitetura unidimensional onde cada elemento de computação se comunica no máximo com dois outros elementos. Neste exemplo, a comunicação com o restante do hardware se dá apenas nos dois elementos de computação das bordas.

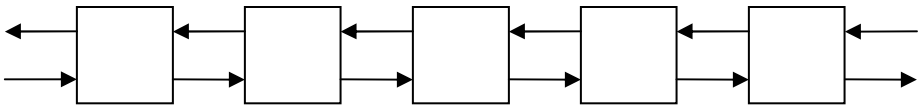

Figura 3.2- Arquitetura unidimensional [Kun87]

Nem toda a aplicação pode ser implementada eficientemente em arquiteturas sistólicas ou wavefront. A primeira característica da aplicação a ser avaliada é se ela pode ser paralelizada. Para que isto ocorra, não deve haver muitas dependências de dados dentro da aplicação [Elr98]. Se uma dada aplicação for eminentemente serial, ou seja, suas instruções têm dependências de dados que impõem uma execução seqüencial, torna-se difícil o seu

mapeamento em uma arquitetura deste tipo. Se, por outro lado, existir um alto grau de paralelismo entre as operações, então a aplicação pode ser candidata a ser implementada em uma arquitetura sistólica ou wavefront.

O passo seguinte é avaliar se é possível decompor o problema da aplicação em subproblemas mais simples que possam ser implementados pelos elementos de computação. Normalmente, quanto mais simples o circuito, maior a velocidade de relógio na qual ele pode operar [Tau84]. A arquitetura sistólica ou wavefront deve ser, na medida do possível, composta por elementos de computação semelhantes. Isto torna o projeto da aplicação mais simples e modular, contribuindo também para a equalização dos atrasos nas células. Para fazer isto, deve ser possível dividir a aplicação em subproblemas semelhantes que possam ser mapeados para elementos de computação semelhantes. Nem todos os elementos de computação poderão ser iguais, uma vez que os que fazem a comunicação da arquitetura com o resto da placa podem ser capazes de fazer algum controle de entrada e saída, sendo, portanto, diferentes.

Para que haja um maior desempenho, o controle do fluxo de dados deve ser simples e regular, com cada elemento de computação necessitando de poucas linhas de controle. Preferencialmente, o número de linhas de controle globais deve ser reduzido ao mínimo necessário. Neste caso, o relógio (*clock*) é uma linha de sincronização realmente indispensável. As demais comunicações devem ser executadas de forma local, principalmente entre elementos de computação adjacentes. Para isto ser possível, é necessário que a aplicação possa ser dividida em subproblemas e os resultados obtidos possam ser reutilizados como entrada de outros subproblemas, de forma que quando um elemento de computação obtiver um resultado, este possa ser transmitido como entrada para um elemento de computação vizinho.

Com elementos de computação simples e interconectados de forma regular, uma arquitetura sistólica ou wavefront pode ser facilmente expandida conforme as necessidades do projeto, pois uma vez projetados os elementos de computação e suas interconexões, torna-se uma tarefa mais simples adicionar novos elementos à arquitetura, respeitadas as limitações tecnológicas.

Uma arquitetura sistólica ou wavefront deve ter poucas operações de entrada e saída, uma vez que a comunicação com o restante do sistema é feita através de alguns poucos elementos de computação na arquitetura. Em termos da aplicação, isto significa que o problema a ser modelado deve ser intensivo em computação, mas não em entrada e saída. Aplicações comuns para arquiteturas sistólicas são: programação dinâmica [Jac04], processamento de imagens [Dos87], operações com matrizes (somas, multiplicações, resolução de sistemas lineares, entre outras) [Mel89], processamento de sinais (FFT (*Fast Fourier Transform*)), convoluções, entre outras [Moe99].

3.2 SISTEMAS RECONFIGURÁVEIS

Muitas arquiteturas dedicadas, não podem ser modificadas após a sua fabricação. Esta é uma grande limitação, pois, muitas vezes os projetos sofrem modificações e se o hardware dedicado utilizado não puder atender a novas especificações, então não poderá mais ser utilizado. Desta forma, seria necessário projetar um novo circuito com as modificações necessárias. Além disto, ainda há o custo de reinstalação do hardware dedicado e de testes de compatibilidade.

O hardware reconfigurável [Enz99] permite tanto executar as operações de forma temporal como em um processador (através da reconfiguração) e de forma espacial como em um ASIC (*Application-specific Integrated Circuit*). Assim uma arquitetura reconfigurável pode ter uma seqüência de operações distintas ao longo do tempo bem como executar estas operações em paralelo.

Este tipo de arquitetura possibilita outras abordagens de computação [Enz99]:

a) Hardware de múltiplos modos: Vários algoritmos diferentes são executados seqüencialmente ou concorrentemente no mesmo hardware reconfigurável.

b) Particionamento Temporal: Um algoritmo é particionado em várias seções, cada uma destas é implementada por um circuito individual, e então é executado de forma seqüencial.

c) Co-processador: A execução do algoritmo é acelerada pela implementação dos trechos críticos em hardware reconfigurável.

d) Hardware sob Demanda: Neste caso, os trechos críticos são executados em software ou em hardware na medida em que for necessário.

e) Adaptação Dinâmica: Neste caso, o circuito da arquitetura reconfigurável é modificado ao longo da execução dependendo dos dados tratados.

Um processador de propósito geral, PPG [Tan90] tem a desvantagem de que, para cada instrução em código de máquina a ser executada, são necessárias as fases de busca da instrução, decodificação e execução, o que torna o processo de execução destas instruções mais lento [Pat98] quando comparado com uma arquitetura dedicada.

Além disto, quando um programa é executado em PPG geralmente ele não executa sozinho. A aplicação que está sendo executada normalmente é acompanhada por um sistema operacional. Este sistema operacional pode permitir a execução concomitante de outras aplicações. Isto causa uma perda de desempenho devido ao tempo gasto pelo sistema operacional em gerência de memória, gerência de processos, gerência de dispositivos, entre outros [Tan03].

Em um PPG, grande parte do circuito é utilizada para implementar instruções que não são efetivamente utilizadas para uma dada aplicação, implicando em um grande desperdício de recursos computacionais e de projeto. Isto é particularmente verdadeiro para arquiteturas do tipo CISC [Pat98]. Sendo assim, as arquiteturas reconfiguráveis se apresentam como uma alternativa para um processamento específico.

Um sistema reconfigurável é basicamente composto de duas partes [Enz99]. Uma é o hardware reconfigurável que pode ser modificado assumindo novas funções lógicas, fluxos de controle, entre outras características. A segunda parte é responsável pelo gerenciamento do hardware reconfigurável e que permite que o hardware possa ser modificado. Nesta segunda

parte, é necessária a comunicação com outro sistema que envia informações sobre a configuração.

Apesar das arquiteturas de hardware reconfigurável poderem ser construídas de várias formas, existem algumas características que permitem uma classificação das mesmas [Enz99]:

a) Composição do Sistema: Várias composições de sistemas reconfiguráveis são possíveis. Estes sistemas podem utilizar lógicas configuráveis bem como processadores internos especializados.

b) Reconfiguração durante a execução (*Run-time Reconfiguration*): Reconfigurar o hardware durante a execução da aplicação é uma característica interessante. Para esta característica ser útil, é necessário que o tempo de reconfiguração seja o menor possível. A sobrecarga adicionada pela reconfiguração tem de ser levada em conta na implementação da aplicação.

c) Configuração da Granulosidade: A granulosidade diz respeito ao tamanho e à complexidade dos elementos de computação que a arquitetura reconfigurável possui. Esta tem um grande impacto na forma como o algoritmo da aplicação poderá ser dividido entre os elementos de computação.

d) Configuração de Escalonamento: As arquiteturas reconfiguráveis necessitam receber informações tanto a respeito dos dados da aplicação que está sendo executada como da configuração que ela deve ter em um determinado momento.

De acordo com [Pag96], existem várias formas como uma arquitetura reconfigurável pode executar um programa:

a) Hardware puro: Neste modelo, uma configuração de hardware correspondente a uma aplicação é projetada e colocada no hardware reconfigurável. Neste aspecto, esta forma é semelhante ao projeto de um ASIC.

b) Microprocessador de Aplicação Específica: Este modelo corresponde a criar em hardware reconfigurável um processador com instruções específicas para uma determinada aplicação. Este processador executaria um programa específico. Assim, é preciso criar tanto o processador abstrato como o código abstrato a ser executado sobre ele. Os dois necessitam ser otimizados simultaneamente para um maior desempenho. A seguir, a descrição do processador é implementada no hardware reconfigurável. Esta forma é mostrada na figura 3.3.

c) Reutilização Seqüencial: Este modelo pode ser utilizado quando a aplicação toda não couber dentro do hardware reconfigurável. Para resolver este problema, pode-se reconfigurar a arquitetura várias vezes, uma para cada trecho da aplicação. Para medir o custo-benefício desta reconfiguração deve-se levar em conta o tempo gasto para reconfigurar o hardware em comparação a aceleração da aplicação que será obtida pela reconfiguração. Esta forma é mostrada na figura 3.4.

d) Múltipla Utilização Simultânea: Neste modelo, há recursos suficientes no hardware reconfigurável para executar mais de uma aplicação ao mesmo tempo. Cada uma destas aplicações pode se comunicar com o computador hospedeiro de forma independente. Para isto, seria necessário desenvolver estratégias para projetar circuitos específicos para aplicações que sejam independentes da mesma forma que as aplicações devem ser independentes. Esta forma é mostrada na figura 3.5.

e) Uso Sob Demanda: Neste modelo, o hardware reconfigurável pode ser utilizado por várias aplicações de forma independente, dependendo da demanda de processamento de cada uma. Para fazer isto, a cada reconfiguração do hardware, os elementos de computação (e conseqüentemente o poder de processamento) são divididos de acordo com a necessidade das aplicações. É necessário portanto definir uma política de alocação dos recursos do hardware reconfigurável. Esta forma é mostrada na figura 3.6.

As figuras correspondentes aos modelos de utilização foram retiradas de [Oli99].

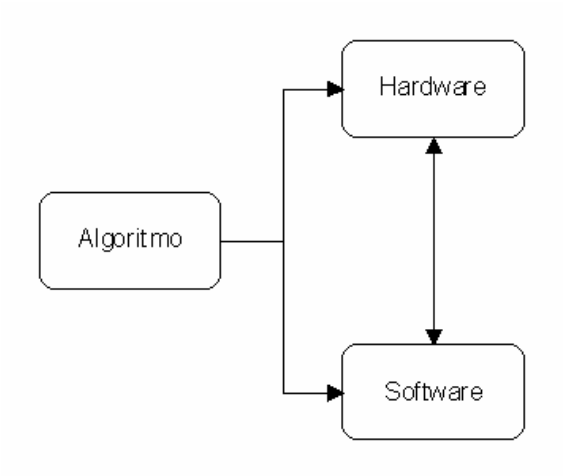

Figura 3.3 - Microprocessador de Aplicação Específica

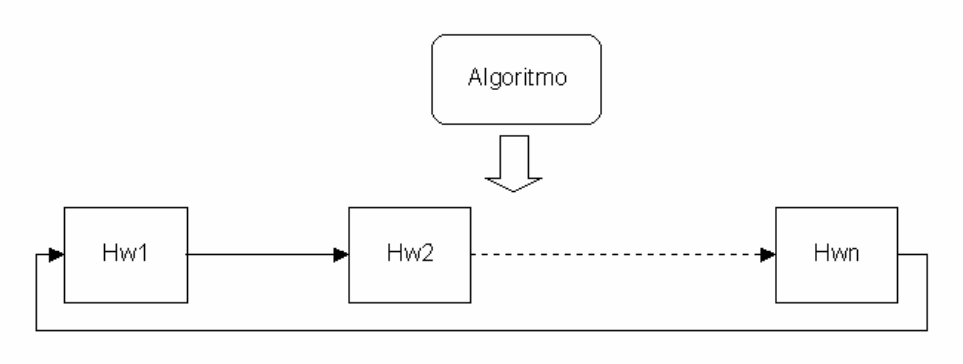

Figura 3.4 - Reutilização Seqüencial

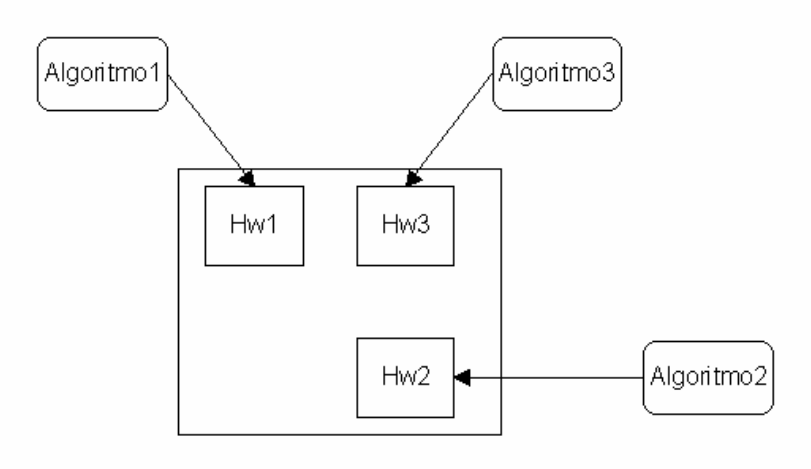

Figura 3.5 - Múltipla Utilização Simultânea

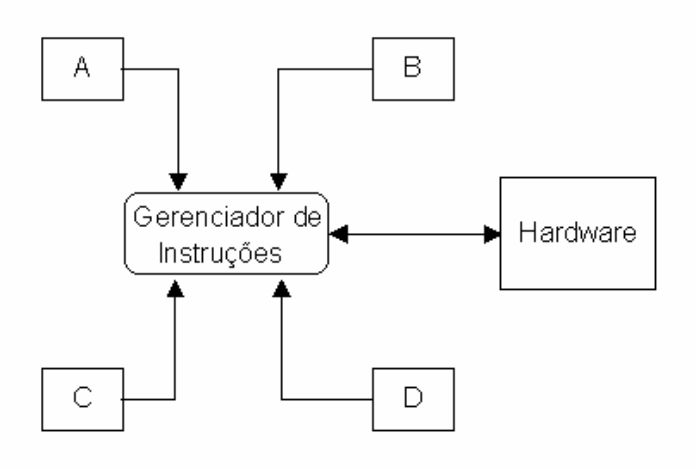

Figura 3.6 - Uso Sob Demanda

Em [Ada99], o modelo de [Pag96] é estendido levando em consideração as formas como a reconfiguração do hardware reconfigurável podem ser feitas. Assim estas formas estariam divididas em três classes de programabilidade do hardware:

a) Projeto Estático: Neste modelo, o hardware é programado uma única vez. Esta configuração não muda durante toda a vida útil do hardware. Este projeto utiliza as vantagem de prototipação e ferramentas de síntese de hardware reconfigurável mas não utiliza a flexibilidade proporcionada pela reconfiguração.

b) Projeto Estaticamente Reconfigurável: Neste modelo, já existe a reconfiguração do hardware entre tarefas de processamento. Dependendo do tamanho destas tarefas, se estas forem relativamente pequenas e freqüentes, a reconfiguração pode ser considerada como sendo durante a execução (*runtime reconfiguration*). Este modelo reflete em uma melhor utilização dos recursos computacionais disponíveis. Estes recursos podem também ser divididos entre aplicações diferentes, dependendo da necessidade.

c) Projeto de Reconfiguração Dinâmica: Neste modelo, o hardware pode ser reconfigurado várias vezes, inclusive para uma mesma tarefa computacional. O hardware pode ser configurado para executar uma aplicação, parte dela, ou ter seus recursos divididos entre aplicações diferentes. O tempo de reconfiguração da arquitetura limitará a granulosidade das tarefas. Se o tempo de reconfiguração for grande, a granulosidade das tarefas deverá ser maior para compensar a sobrecarga da reconfiguração.

A diferença principal entre a reconfiguração estática e a dinâmica é que na estática o dispositivo é reconfigurado apenas quando está inativo, enquantoque na dinâmica, pode ser reconfigurado a qualquer momento, mesmo durante uma execução [Com00].

Em muitos casos, é interessante utilizar uma arquitetura reconfigurável conjuntamente com um processador de propósito geral (PPG). Neste caso, a parte da aplicação que pode ser acelerada por hardware é mapeada para a arquitetura reconfigurável e a parte que não pode ser eficientemente mapeada em hardware é executada no PPG.

Normalmente, as partes da aplicação que executam muitas operações de entrada e saída e acesso à memória são mais bem executadas em PPG pois o barramento utilizado para as placas de hardware dedicado podem ser relativamente lentos. Partes da aplicação que são eminentemente seriais, ou seja, não podem ser paralelizadas com um ganho satisfatório de desempenho [Fos95], normalmente são executadas mais rápido em PPG do que em arquiteturas reconfiguráveis já que os PPG funcionam em velocidades bem maiores.

As arquiteturas reconfiguráveis podem ser utilizadas na forma de arquiteturas híbridas onde

tentam relacionar a flexibilidade proporcionada pela arquitetura reconfigurável com o alto poder de processamento que os circuitos integrados específicos proporcionam [Kav96].

3.3 FPGA

Os FPGAs (*Field Programmable Gate Arrays*) [Wol04] [Pel05] são circuitos integrados em larga escala que podem ter sua configuração interna modificada após a sua fabricação. Os FPGAs podem ser construídos em uma grande variedade de tamanhos e com características diferentes. Normalmente, a capacidade do FPGA é medida pelo número de blocos lógicos configuráveis que ele possui ou pelo seu número de portas lógicas equivalentes. Outras características importantes são o consumo de energia e a velocidade de operação dos FPGAs.

Uma característica comum a quase todos os FPGAs é que eles são compostos de blocos programáveis. Estes blocos são compostos de registradores, elementos lógicos configuráveis e interconexões. Estes elementos normalmente estão arranjados na forma de grade (*Grid*). Alguns FPGAs mais complexos podem ter estruturas mais complexas como multiplicadores, que são especificamente úteis para aplicações como processamento de sinais. Podem existir ainda algumas estruturas nas bordas do dispositivo capazes de fazer operações de entrada e saída programadas.

Existem características que são comuns aos FPGAs [Wol04]:

a) Elementos de computação: Os elementos de computação são normalmente compostos por alguns registradores e elementos lógicos de baixo nível. Eles realizam as funções lógicas e como são simples, muitas vezes é necessário mapear uma determinada função lógica para vários destes elementos através de um processo conhecido como mapeamento tecnológico.

b) Tabelas de *Lookup*: Estas tabelas, também chamadas de LUT (*Lookup Table*), podem implementar funções lógicas através de suas *n* entradas. A LUT também geralmente está associada a um ou mais registradores programáveis como um *flip-flop*. Os detalhes de implementação da LUT variam de fabricante para fabricante e dentro do mesmo fabricante

pode variar entre famílias.

c) Memória: A maioria dos dispositivos modernos possui blocos de memória internos. A memórias podem ser do tipo SRAM ou outros. Esta memória pode ser global para todos os elementos lógicos ou locais para um determinado grupo de elementos.

d) Recursos de Roteamento: Estes recursos fazem a interligação entre os elementos de processamento, a memória interna do FPGA e outras estruturas que podem fazer parte do FPGA como os elementos de entrada e saída. Estes elementos podem ter velocidades diferentes bem como níveis de flexibilidade diferentes. Os recursos de roteamento podem se organizar de forma hierárquica interligando tanto elementos de processamento próximos como fazer ligações distantes entre blocos e a memória do FPGA.

e) Entrada e Saída Configurável: A placa de FPGA deve se comunicar com o computador hospedeiro. Assim, a maioria das placas de FPGAs possui gerenciamento de relógios (*clocks*) para aumentar o desempenho. Outra função desta entrada e saída é manter a compatibilidade entre a interface da placa e os circuitos internos do FPGA. Normalmente, os FPGAs têm circuitos chamados de blocos de entrada e saída que a função de fazer o gerenciamento da comunicação.

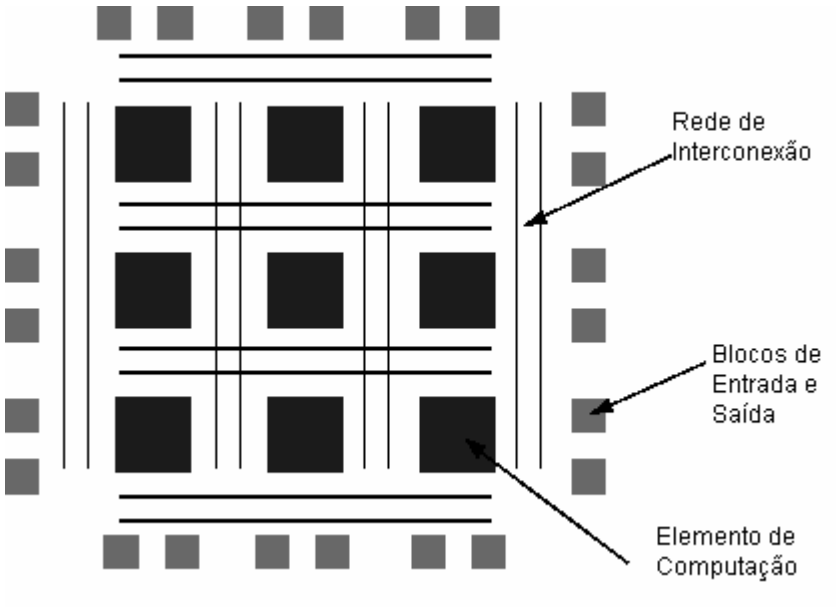

Figura 3.7 - Estrutura do FPGA

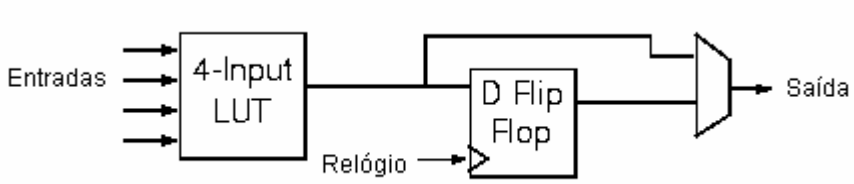

Figura 3.8 - Elemento de Computação

A figura 3.7 mostra que os elementos de computação em um FPGA são separados por redes de interconexão. Nas bordas do FPGA estão os elementos de entrada e saída. Cada um dos elementos de computação mostrados na figura 3.8 é composto basicamente de uma LUT (*Lookup Table*) e de um registrador do tipo *flip-flop*. Os elementos de computação podem ser mais complexos se a granulosidade for maior.

O número de entradas da LUT pode variar dependendo do tipo do FPGA. Tipicamente possui entre 4 e 7 entradas [Wol04]. O elemento de computação contém entradas, correspondendo às entradas da LUT e saídas correspondendo aos resultados da computação ou do valor armazenado. Cada circuito projetado deve ser acomodado no menor elemento de computação capaz de processá-lo. Caso o circuito seja mais complexo que o comportado por um elemento de computação, ele deve ser mapeado para vários destes elementos.

Para conectar os elementos de computação, é necessária uma rede de interconexão. As linhas correspondentes ao relógio (*clock*) não fazem parte desta interconexão programável por que já estão embutidas no processo de fabricação. Os elementos de comunicação podem se relacionar associando seus pinos de entradas e saídas. Este processo é chamado também de roteamento. Cada um dos elementos de computação pode se ligar a uma das linhas de interconexão que está adjacente a ele. Isto é mostrado na figura 3.9. Nos cruzamentos entre estas linhas estão os blocos de roteamento (*Switch Block*). Estes blocos são responsáveis por interligar linhas adjacentes de forma que elementos de computação distantes possam se comunicar. Estes blocos são mostrados na figura 3.10.

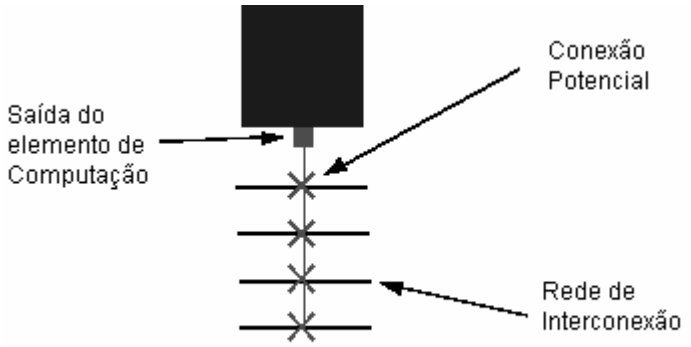

Figura 3.9 - Saída do elemento de computação conectada à rede de interconexão

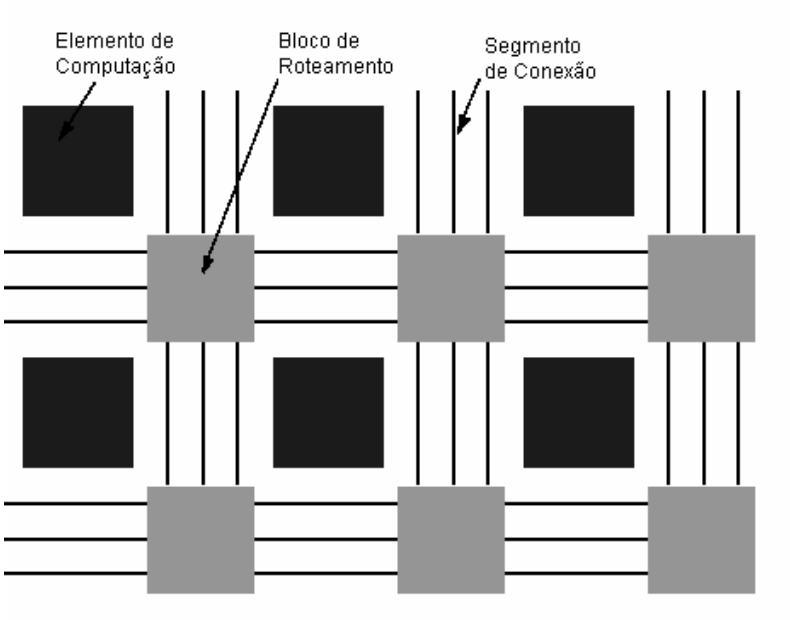

Figura 3.10 - Roteamento dos elementos de computação

Os FPGAs podem variar bastante quanto a granulosidade dos seus elementos de computação. Uma granulosidade fina geralmente corresponde a poucas portas lógicas. Uma granulosidade grossa pode corresponder até a somadores, comparadores e multiplicadores ou em casos extremos até uma ULA. Existem características que devem ser levadas em consideração [Kou91]. O tamanho grande do elemento de computação tem impacto negativo na sua velocidade. Por outro lado, elementos de computação pequenos necessitam de mais recursos de interconexão e conseqüentemente há um maior atraso devido ao roteamento. Assim, é difícil encontrar um tamanho ótimo de elemento de computação.

As aplicações em FPGAs normalmente são mais lentas e consomem mais energia do que as mesmas aplicações implementadas em um ASIC específico. Os FPGAs têm a vantagem sobre o ASIC de ter um preço muito mais acessível para pequenos volumes de produção, o que os tornam uma solução muito atraente em relação ao custo/benefício obtido. Além disto, o custo e tempo necessários para o projeto de aplicações em FPGAs é bastante menor que a de um circuito integrado específico. Assim, mesmo para um projeto de um ASIC específico pode ser útil implementar um protótipo em FPGA.

Com novas ferramentas de projeto e teste de aplicações em FPGAs, o tempo necessário para

um projeto de aceleração em hardware foi reduzido em centenas de vezes [Har01]. Assim, há uma tendência cada vez maior de migração de ASICs para FPGAs e portanto há também a necessidade de ferramentas de compilação de alto nível para particionamento da computação entre o hardware reconfigurável e o PPG.

3.4 SÍNTESE DE SISTEMAS RECONFIGURÁVEIS

Para que um FPGA possa efetivamente executar aplicações, é necessário fazer um mapeamento do algoritmo a ser implementado para os elementos de computação e o roteamento entre estes. Existem diversos tipos de elementos de computação. Por exemplo, a Xilinx [Xil08] possui elementos de computação chamados CLB (*Configurable Logic Block*), a Altera [Alt08] possui elementos de computação chamados ALM (*Adaptive Logic Module*). Cada um destes blocos pode ter particularidades que causam impacto no desempenho da aplicação a ser modelada no FPGA. O mapeamento e roteamento das funções lógicas dentro destes elementos de computação dependem também da tecnologia adotada e tem grande impacto sobre o desempenho final. Como este mapeamento é complexo, é interessante haver ferramentas que ao mesmo tempo permitam que o projetista se abstraia destes detalhes, bem como permitam otimizações feitas a mão, caso necessário.

A síntese em FPGA pode ser dividida em várias fases [Wol04]:

a) Especificação do Circuito: Nesta fase, o circuito a ser implementado no FPGA é descrito utilizando uma linguagem apropriada como Verilog ou VHDL, entre outros.

b) Otimização Lógica: Nesta fase, as equações lógicas são otimizadas visando uma implementação mais eficiente do circuito.

c) Mapeamento Tecnológico: Nesta fase é feita a implementação dos circuitos gerados na fase anterior nos elementos de computação presentes no FPGA. É feita uma verificação formal antes do mapeamento no circuito. Em alguns casos, circuitos ocuparão mais de um elemento de computação. Em outros casos, alguns destes elementos serão subutilizados. Assim é

necessário haver uma otimização do número de elementos utilizados.

d) Posicionamento: Nesta fase faz-se o mapeamento dos elementos de computação gerados na fase anterior para os blocos físicos onde eles ficarão. Como o tempo de processamento depende do tempo de comunicação entre os elementos, é necessário otimizar a escolha das linhas de interconexão utilizadas. Isto pode ser feito nesta fase fazendo com que os elementos de computação que se comunicam sejam mapeados mais próximos.

e) Roteamento: Esta fase diz respeito à escolha de como os blocos de interconexão farão a conexão entre os elementos de computação mapeados.

Estas fases podem ser simuladas por software antes da síntese propriamente dita. O software de simulação é bastante útil para verificar eventuais problemas de síntese que podem ocorrer. O passo seguinte, caso a simulação por software não apresente problemas, é configurar o FPGA para atuar de acordo com o circuito sintetizado. O processo de síntese normalmente gera um arquivo de configuração que deve ser passado para o FPGA propriamente dito. Esta configuração pode tanto ser armazenada em uma memória SRAM ou em uma memória flash [Wol04].

3.5 LINGUAGENS PARA DESCRIÇÃO DE HARDWARE

Existem várias abordagens para se fazer síntese de hardware reconfigurável [Com02]. Uma abordagem seria a utilização de diagramas esquemáticos. Estes diagramas feitos com o auxílio de softwares específicos contêm representações visuais de elementos como ULAs, memória, portas lógicas, entre outros, ligados por uma rede de interconexão. Ficaria a cargo do projetista elaborar o esquemático que implementa a aplicação. Neste caso, ficariam também a cargo do projetista tarefas como o particionamento dos elementos de computação. Uma vez elaborado o circuito, a ferramenta geraria uma configuração que poderia ser enviada para o FPGA. Este tipo de projeto exige do projetista grande nível de conhecimento sobre projeto de circuitos e não é adequado para projetos grandes e complexos.

Outra opção de projeto é a utilização de linguagens para descrição de hardware. Algumas linguagens comuns são o VHDL (VHSIC (*Very High Speed Integrated Circuits*) *Hardware Description Language*) [Ash90], VERILOG [Wol04], HANDELC e SystemC [Pel05], entre outras.

Após a escrita do programa em uma destas linguagens, programas específicos podem gerar o arquivo de configuração do FPGA correspondente. Estes programas normalmente permitem a simulação do circuito gerado em FPGA bem como diversos níveis de otimização manual.

Em relação às linguagens de descrição de hardware reconfigurável pode ser feita a seguinte divisão, considerando a forma de modelagem [Com02]:

a) Modelagem Comportamental: Nesta modelagem não é necessário descrever o sistema fisicamente, basta apenas dar uma descrição do seu comportamento. Este tipo de modelagem se assemelha mais a descrição de uma linguagem de alto nível onde se pode utilizar estruturas de decisão e repetição para modelar o comportamento desejado.

b) Modelagem Estrutural: Nesta abordagem a linguagem descreve o circuito em termos da sua estrutura básica, ou seja, em termos das portas utilizadas e das interligações entre elas bem como os sinais que compõem o sistema. Nesta modelagem, pode-se utilizar elementos mais complexos como somadores e outras estruturas desde que elas já tenham sido descritas estruturalmente.

A modelagem comportamental, embora possibilite um maior grau de abstração, nem sempre é sintetizável, ou seja, nem sempre é possivel gerar um circuito baseado com uma descrição puramente comportamental. Mesmo quando é possível a geração, normalmente o circuito não é tão otimizado como seria se fosse descrito de forma estrutural [Com02].

Os sistemas descritos em linguagens de descrição de hardware (HDL *hardware description language*) são compostos de módulos que realizam o processamento e sinais responsáveis pela comunicação entre estes. Os sinais modelam as interconexões em hardware, que transmitem

uma informação proveniente de uma fonte a um ou mais destinos. Um aspecto que diferencia as HDLs de linguagens de software usuais é a modelagem do caráter reativo do hardware. Enquanto que em software uma função é chamada explicitamente pelo fluxo de execução de um programa, em hardware o fluxo de processamento é determinado pela reação dos módulos a estímulos transmitidos pelos sinais. As HDLs em geral provêem diversos recursos para modelar o hardware. Descrições estruturais, comportamentais e *dataflow* são alternativas para expressar o seu comportamento. Em termos de simulação, estes elementos são representados por processos. Os processos são modelos de componentes físicos que possuem uma lista de sinais aos quais são sensíveis. Assim, quando um destes sinais muda de valor, o processo pode tomar uma ação correspondente, simulando a reatividade do hardware. Estes processos podem ser síncronos ou assíncronos em relação aos sinais de entrada. Os processos são projetados para atuarem em paralelo.

O uso de linguagens para especificação do hardware reconfigurável permite um maior grau de abstração em comparação com a descrição direta dos circuitos. Com elas, é possível fazer modelagens hierárquicas que simplificam o projeto e a construção de bibliotecas que permitem a reutilização do código. Além disto, a padronização da linguagem permite a portabilidade do código para arquiteturas diferentes [Com02].

Durante o processo de simulação, o componente pode ser avaliado quanto à geração de saídas esperadas em relação aos valores de entrada. Isto pode ser visto através do monitoramento dos sinais gerados. A verificação dos circuitos gerados é importante pois em várias arquiteturas reconfiguráveis podem ocorrer danos aos circuitos devido a curtos-circuitos gerados por conexões indevidas [Com02].

Existem várias tentativas de utilizar linguagens de alto nível, principalmente subconjuntos da linguagem C, para fazer a descrição do hardware [Com02] [Pel05]. Com o aumento crescente do número de portas presentes nos FPGAs e com o aumento da complexidade das aplicações que executam sobre eles existe um grande aumento do custo de projeto nestas arquiteturas [Pag04]. Sendo assim, é interessante a utilização de linguagens de alto nível para o projeto de hardware. Esta abordagem possui várias características interessantes como um alto nível de

abstração e a possibilidade da utilização de técnicas de engenharia de software para a implementação das aplicações [Pag04].

Porém, muitas destas linguagens têm natureza seqüencial, o que dificulta, e em alguns casos impossibilita, a geração da configuração do hardware

3.5.1 SYSTEMC

O SystemC [Ope05] é uma biblioteca em C++ para várias plataformas, distribuída gratuitamente. Ela é utilizada para a modelagem de vários tipos de sistemas principalmente em hardware. O SystemC pode modelar hardware de forma semelhante ao VHDL e Verilog mas com várias outras funcionalidades permitidas pela programação orientada a objeto. O SystemC pode utilizar funções presentes em bibliotecas C++ como acesso a arquivos, sistema operacional entre outros, o que é bastante interessante para o desenvolvimento do projeto. Após a compilação de um programa em SystemC, pode ser gerado um programa executável que faz a simulação do sistema, lendo arquivos de configuração e gerando arquivos com o resultado, bem como possibilita a interação com o usuário durante a execução, o que outras linguagens de descrição de hardware não permitem.

O SystemC contém todos os tipos de dados existentes no C++, como ponto flutuante, ponteiros, entre outros. Além destes tipos, o SystemC possui também tipos específicos para modelagem de hardware como o *sc_bit* e o *sc_logic*.

A simulação do SystemC funciona ciclo a ciclo [Ope05]. Após o início da simulação, os processos são agendados para execução. Todos os processos têm a chance de executar a cada ciclo de execução. Todas as mudanças de valores de portas e sinais não são feitas imediatamente, mas agendadas para o próximo ciclo de execução, para que a simulação possa ser feita de forma correta. Os processos que são colocados em execução são executados até o fim, ou até que se encontre um comando *wait*, que faz com que o processo seja suspenso por um determinado número de ciclos. Se nenhum número de ciclos for especificado, o processo é

suspenso até o início do próximo ciclo. Quando se termina a simulação dos processos, todos os sinais são atualizados e inicia-se o próximo ciclo.

No início da simulação, o procedimento de teste começa a executar, gerando mudanças de sinais, que são os eventos da simulação. Estes eventos causam a execução de outros processos que são sensíveis a estes eventos. Esta execução é agendada para os próximos ciclos. Quando todos os processos agendados para um determinado ciclo são executados, todos os sinais são atualizados e o próximo ciclo de execução começa.

4 – HARDWARE DEDICADO PARA COMPARAÇÃO DE SEQÜÊNCIAS BIOLÓGICAS

Os algoritmos de bioinformática normalmente exigem grande poder computacional, pois tratam grandes volumes de dados. Neste contexto, uma solução particularmente interessante para a resolução de alguns problemas de bioinformática é a utilização de hardware configurável, como o FPGA. Nesse capítulo serão analisadas várias propostas de arquiteturas dedicadas para comparação de seqüências.

4.1 PROJETO BÁSICO DE HARDWARE SISTÓLICO PARA COMPARAÇÃO DE SEQÜÊNCIAS

Uma vantagem de uma arquitetura sistólica dedicada é que ela pode explorar o paralelismo dos circuitos com seus diferentes componentes atuando em paralelo, para obter um resultado mais rápido. Isto é particularmente útil para a implementação de um algoritmo como o alinhamento local com Smith-Waterman (seção 2.4.2), pois uma vez respeitada a dependência de dados, vários escores podem ser calculados em paralelo por diferentes partes do circuito de uma arquitetura.

Existem algumas dificuldades na transformação do problema de comparação de seqüências biológicas para um hardware dedicado capaz de resolver o problema de forma eficiente. Por exemplo, no caso do algoritmo de Smith-Waterman, a primeira exigência a ser atendida é a relação de recorrência. Neste caso, para o cálculo de cada um dos elementos da matriz são necessários os valores de três outros elementos e esta dependência tem de ser levada em conta no vetor sistólico.

4.1.1 Vetor Sistólico Unidirecional

A solução mais imediata consiste da utilização de um vetor sistólico unidirecional e unidimensional para fazer a comparação de seqüências [Bou07a]. Nestes casos, a arquitetura sistólica obedece às relações de recorrência de Smith-Waterman, calculando a cada passo uma

anti-diagonal da matriz de similaridade (seção 2.4.2). Para construir a matriz, os elementos da seqüência procurada são armazenados nos elementos enquanto a seqüência de base é colocada de forma invertida. Cada um destes elementos possui também um registrador para um elemento da seqüência que está sendo processado, em um determinado momento. Em cada passo, duas bases das seqüências são comparadas dentro do elemento de computação do vetor sistólico. Após esta comparação, a base no registrador é passada para o elemento de computação seguinte. A figura 4.1 mostra um exemplo deste tipo de arquitetura.

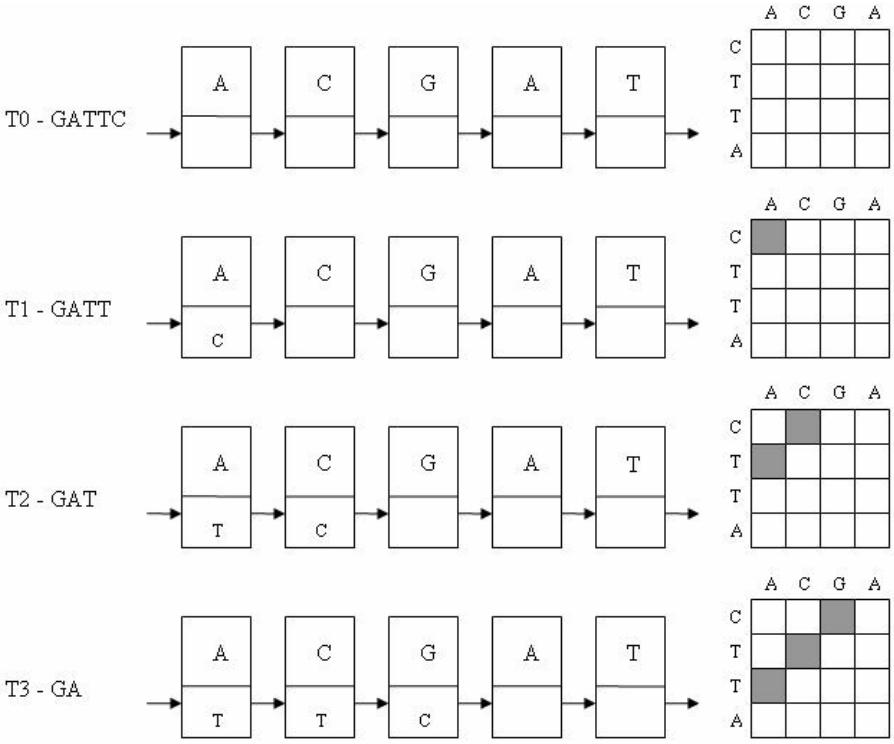

Figura 4.1 Vetor Sistólico Para Comparação de Seqüências

Na figura 4.1, é feita a comparação de duas seqüências **ACGAT** e **CTTAG**. A seqüência procurada (**ACGAT)** foi colocada nos elementos de computação do vetor sistólico. A seqüência de base (**CTTAG)** vai ser colocada invertida, uma base de cada vez, no vetor sistólico. A figura 4.1 mostra do lado esquerdo a segunda seqüência invertida, no meio o vetor sistólico e do lado direito os elementos da matriz de similaridade que são calculados a cada passo. A seqüência de base encontra-se inicialmente na memória interna do FPGA e suas

bases são enviadas uma a uma para os circuitos que formam o vetor sistólico. No tempo *T0* não há nenhuma comparação. No instante *T1* a primeira base (**C**) é colocada no vetor sistólico e a sua comparação com a base **A** que se encontra ali corresponde ao cálculo do primeiro elemento da matriz de similaridade marcado em cinza na figura. À medida que as bases vão sendo deslocadas, vão sendo calculadas as antidiagonais da matriz. A cada iteração do vetor, uma nova anti-diagonal é calculada. Assim, podemos ver que cada elemento é capaz de calcular uma célula da matriz e que estes elementos são capazes de efetuar os cálculos em paralelo, e quando o vetor estiver cheio, é capaz de calcular *N* células em paralelo onde *N* é o tamanho do vetor sistólico.

Cada um dos elementos de computação do vetor sistólico deve conter as três células da matriz, necessárias para o cálculo da célula atual. Além disto, algumas células devem ser repassadas para outros elementos do vetor para que estes também possam calcular a próxima célula.

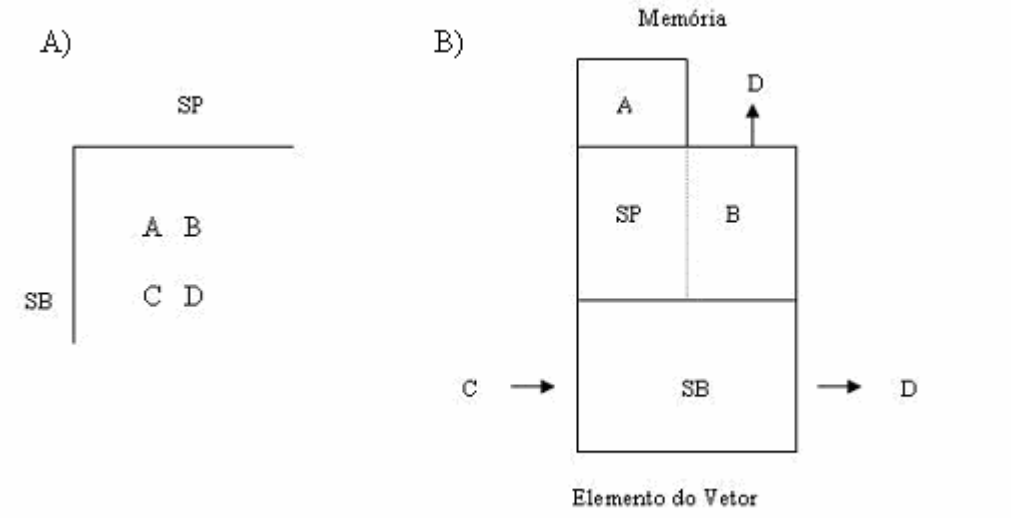

Figura 4.2- Arquitetura interna do Elemento do Vetor [Car03]

Na figura 4.2 (A) é representado um trecho da matriz de similaridade. No eixo horizontal da matriz está a seqüência procurada (*SP*). No eixo vertical está a seqüência de base (*SB*). Em um determinado momento, um elemento de *SP* está sendo comparado com um elemento de *SB*. Para calcular o valor da célula atual *D* são necessários os três escores *A*, *B* e *C*.

Na figura 4.2 (B), é mostrado um elemento do vetor sistólico. Nele, estão armazenados em registradores todos os valores necessários para a computação. Os elementos das seqüências comparadas (*SB, SP*) e os valores das três células *A*, *B* e *C* que são respectivamente M(i-1,j-1), M(i-1,j) e M(i,j-1) (seção 4.2.2). Com estes valores, pode-se calcula o valor desejado de *D* correspondente a M(i,j). Em cada instante, uma base de *SB* é movida para o próximo elemento do vetor sistólico. Junto com ela também é passado o valor da célula *C*. Para o cálculo de *D* é necessário apenas o valor passado de *C* e os elementos a serem comparados. Os valores de *A* e *B* já foram calculados previamente e estão no elemento. O valor calculado de *D* será passado tanto para o próximo elemento do vetor como para a memória do FPGA onde será armazenado junto com as outras células calculadas da matriz.

Os detalhes de como os valores das células são movimentados nos elementos do vetor sistólico são mostrados na figura 4.3.

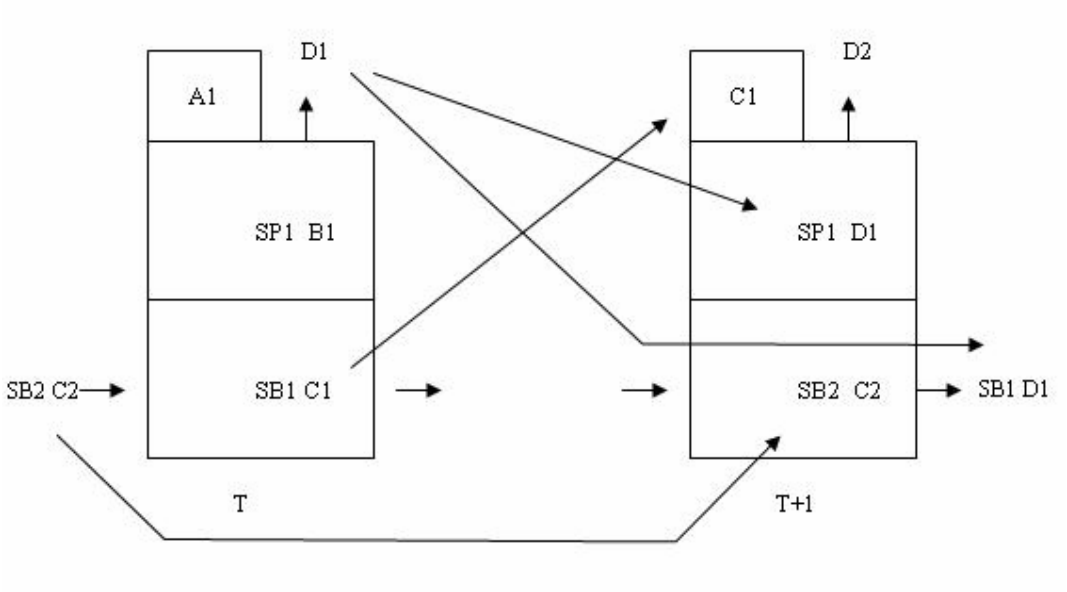

Figura 4.3 – Fluxo dos valores dentro de um elemento

Na figura 4.3 é apresentado o fluxo de um único elemento do vetor sistólico e como seus valores mudam no tempo. No tempo *T*, temos os campos *A1*, *B1*, *C1*, *D1*, *SB1*, *SP1* que são

análogos aos campos mostrados do lado direito da figura 4.2. No instante *T* o elemento utiliza os três valores *A1*, *B1* e *C1* para calcular o campo *D1*.

No tempo *T+1* os valores dos campos do elemento são modificados. O valor *A1*, correspondente à célula da diagonal (ver figura 4.2 na parte esquerda), pode ser descartado uma vez que não vai mais ser utilizado. O seu registrador no elemento será ocupado por *C1*. Isto significa que o valor da relação de recorrência correspondente ao *A1* na próxima iteração será o mesmo de *C1*.

O novo valor da célula *C* será *C2* que entrará no elemento neste instante vindo do elemento à esquerda no vetor. Junto com este valor entrará também o novo elemento da seqüência de base *SB2* que será utilizado na próxima comparação. O valor da célula *D1* calculada no instante *T* substituirá o valor da célula *B1*. No instante *T+1* o elemento da seqüência de base *SB1* é passado para o próximo elemento do vetor sistólico (não mostrado na figura) para ser utilizado em outra comparação. O valor calculado *D1* também é passado para o próximo elemento do vetor onde ocupará um registrador correspondente a uma célula *C1*. Assim podemos ver que a cada interação, é necessária apenas a transmissão de um elemento da seqüência de base e o valor de uma célula da matriz de similaridade. O restante dos valores necessário dependem apenas do rearranjo dos valores dentro do elemento. Em uma implementação típica, cada instante é indicado por um ciclo do relógio do sistema (*clock*).

A figura 4.4 mostra como o vetor sistólico executa a comparação das seqüências e atualiza seus registradores internos.

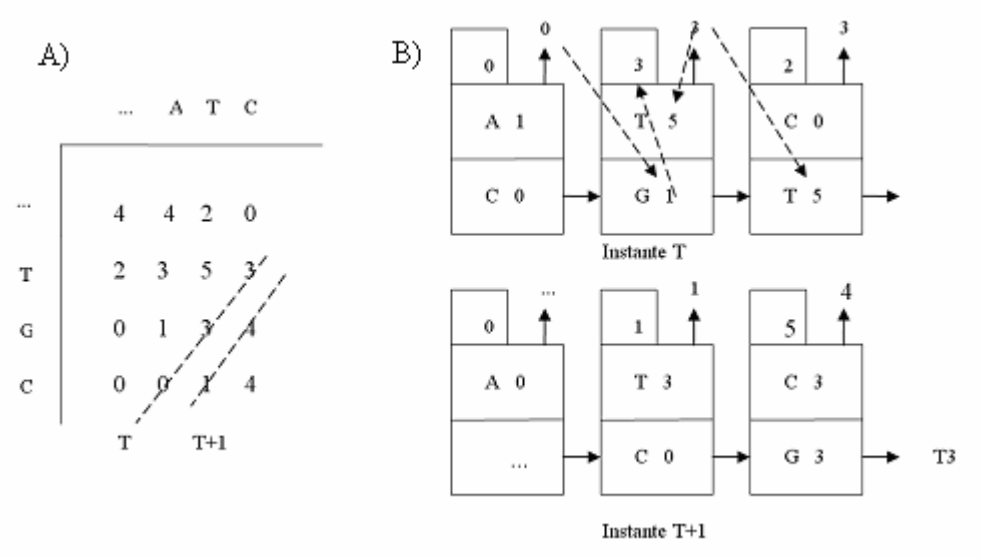

Figura 4.4 – Execução no vetor sistólico

Na figura 4.4 (A), é apresentado um trecho de uma matriz de similaridade. As linhas indicam as anti-diagonais que serão calculadas respectivamente nos instantes *T* e *T+1*. No vetor sistólico (B) no instante *T* estão presentes os elementos da seqüência procurada (**ATC**) que ficam fixos. A seqüência de base é colocada de forma invertida (**CGT**). A disposição dos valores dentro de cada elemento é a mesma da figura 4.2.

Durante o instante *T*, no primeiro elemento do vetor sistólico são comparadas as bases **A** e **C**. Para calcular os escores são necessários os valores das células na diagonal, acima e à esquerda (*A*, *B* e *C* respectivamente). Neste caso, *A* tem valor 0, *B* tem valor 1 e *C* tem valor 0. O valor de *D* calculado é também 0. De forma semelhante todos os outros elementos do vetor tem os valores necessários para os cálculos.

No instante *T+1*, as bases da seqüência de base são movidas para o elemento do vetor à direita bem como o valor calculado de *D* que passa a ser o valor *C* para o elemento da direita. Os demais valores das células são rearranjados. O valor de *C* passa a ser o novo *A*. O valor de *D* passa a ser o novo *B*. O novo valor de *C* vem do elemento do vetor à esquerda. As setas pontilhadas sobre o segundo elemento de computação mostram quais são as mudanças que ocorreram sobre ele da mudança do instante *T* para o *T+1*.

Podemos ver que no instante *T+1* o terceiro elemento de computação do vetor passa a base **T** e o valor 3 para um elemento de computação à direita não mostrado na figura. Neste instante também o primeiro elemento de computação deverá receber uma base do elemento a sua esquerda e calcular um valor *D* e estes não aparecem na figura 4.4 por não estarem no trecho mostrado da matriz.

4.1.2 Vetor Sistólico Bidirecional

Alguns projetos de arquiteturas para comparação de seqüências utilizam um vetor sistólico bidirecional [Hoa92], ao invés do unidirecional (seção 4.1.1). Neste caso, duas seqüências são movimentadas no vetor conforme mostrado na figura 4.5.

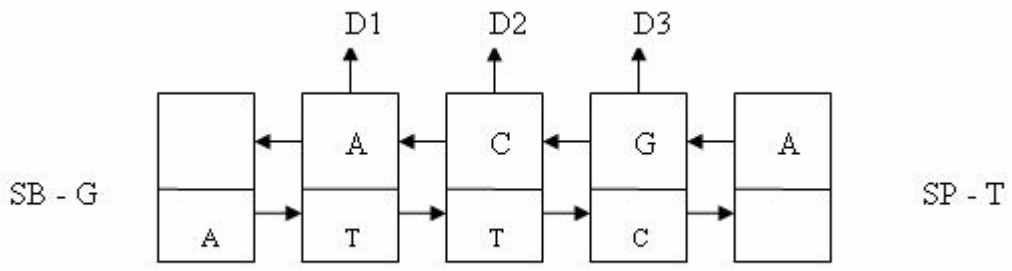

Figura 4.5 – Vetor Sistólico onde as duas seqüências se movimentam

A figura 4.5 mostra a mesma comparação de seqüências da figura 4.1. Neste caso, a seqüência procurada *SP* é movimentada da direita para a esquerda no vetor. A seqüência de base *SB* é movimentada da esquerda para a direita. Os elementos do vetor onde as seqüências se sobrepõem correspondem às anti-diagonais da matriz de similaridade, e nestes elementos de computação são calculados os valores das células (*D1*,*D2* e *D3*). Este tipo de projeto é pouco utilizado, pois necessita de muita comunicação entre os elementos.

4.1.3 Particionamento das Seqüências

Ao se optar por um projeto de vetor sistólico, a seqüência procurada é colocada diretamente no vetor, sendo que cada elemento do vetor conterá um elemento de SB. Isto limita o tamanho da seqüência procurada que pode ser colocada no vetor sistólico.

Em termos práticos, isso quer dizer a seqüência procurada deve ser pequena (< 300 bases). No entanto, os biólogos necessitam comparar seqüências bem maiores. Por essa razão, é aconselhável que seja projetada uma estratégia de particionamento que permita superar esta limitação.

Existem várias formas de particionamento das seqüências de base e procurada. A figura 4.6 ilustra uma destas formas.

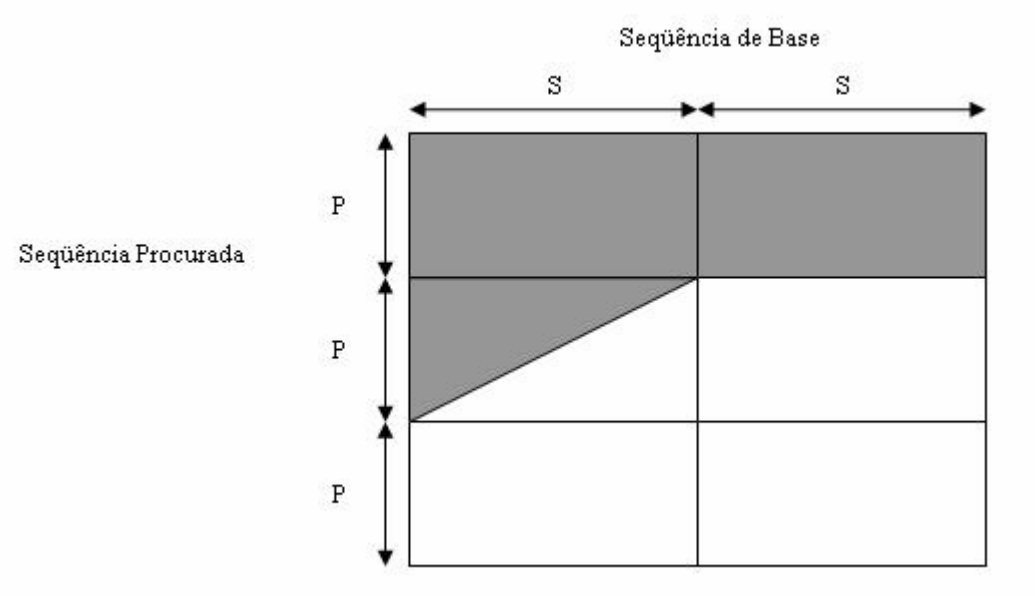

Figura 4.6 – Particionamento das seqüências

A figura 4.6 apresenta uma estratégia de particionamento onde a seqüência de base pode ser dividida em trechos de tamanho *S*. Se a seqüência procurada for grande, pode ser dividida em trechos de tamanho *P* que caibam dentro dos elementos de computação [Lav98]. Assim, a cada rodada de computação, um trecho da seqüência de base *S* é enviado junto com um trecho da seqüência procurada *P* para o vetor sistólico. Lá, eles são comparados e são geradas várias

antidiagonais correspondentes às células da matriz. Estes valores de células são mostrados como retângulos na figura 4.6.

Uma forma alternativa de particionamento consiste de em se manter mais de uma base fixa em cada elemento do vetor (figura 4.7) [Gra01].

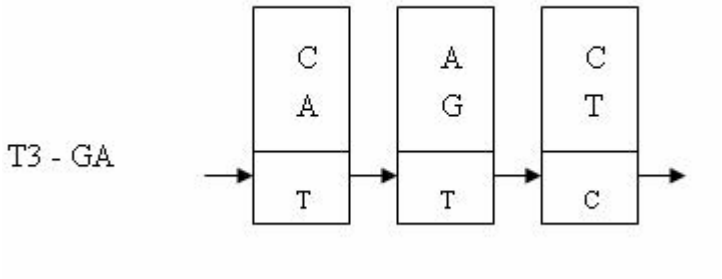

Figura 4.7 – Várias bases por elemento do vetor sistólico

Na figura 4.7, é apresentado o instante 3 da figura 4.1 onde os elementos do vetor possuem duas bases da seqüência procurada ao invés de uma. Embora esta abordagem possa colocar mais bases nos circuitos, tem a complexidade interna aumentada para tratar as bases adicionais, o que torna o circuito mais lento.

4.1.4 Matriz de Similaridade

Quanto à utilização da matriz de similaridade existem basicamente três tipos de arquiteturas:

a) Calculam a matriz de similaridade da forma mais rápida possível sem utilizar os dados gerados
b) Calculam toda a matriz e armazenam apenas o maior escore obtido na matriz
c) Calculam e armazenam toda a matriz (caso caiba no dispositivo) para utilização posterior como o cálculo do alinhamento

Em alguns casos, o desempenho no cálculo da matriz é dado em CUPS (*cell updates per second*), onde cada atualização é o cálculo de uma posição da matriz, que pode ser calculado

multiplicando-se o número de elementos de processamento (PE) pela freqüência de relógio (*clock*) na qual elas operam. Esta medida não leva em consideração os tempos necessários para fazer as transferências da memória RAM para os elementos de processamento.

4.2 ARQUITETURAS AVALIADAS

São conhecidas na literatura propostas de hardware para a execução de algoritmos heurísticos, como o BLAST [Alt90] [Cha04] [Kno04] [Kri05]. Porém, a discussão destas propostas está fora do escopo da presente tese. Assim, apresentamos e discutimos arquiteturas que utilizam programação dinâmica.

4.2.1 Carvalho [Car03]

Carvalho [Car03] propõe uma arquitetura sistólica que implementa o algoritmo de Smith-Waterman (seção 2.4.2). Esta arquitetura calcula a matriz de similaridade para uso posterior. A arquitetura é similar àquela mostrada na seção 4.1.1.

A matriz de coeficientes do Smith-Waterman possui muitos zeros. Para diminuir o tamanho da matriz a ser armazenada no vetor sistólico, foram eliminados zeros desnecessários utilizando-se a representação de matrizes JDS (*Jagged Diagonal Storage*) que permite um corte significativo do número de zeros. Isto permitiu uma redução de 75 a 80% do tamanho da matriz armazenada. Os resultados obtidos em simulações mostraram que, à medida em que se aumenta o número de células, menor é a freqüência máxima de operação. Isto acontece devido a atrasos no roteamento no FPGA, que tendem a se estabilizar a partir de um certo limite (em torno de 56Mhz neste caso). Em [Car03], os elementos do vetor, além de calcularem os escores, também calculam um vetor que indica qual das células foi utilizada na relação de recorrência. Isto possibilita encontrar o melhor alinhamento após o calculo de todas as células.

4.2.2 Grate e outros [Gra01]

Em [Gra01], foi proposto um hardware para o algoritmo Smith-Waterman (seção 2.4.2), utilizando um vetor sistólico unidirecional (seção 4.1.1). O cálculo da matriz de similaridade

se é feito através de um vetor linear onde as antidiagonais da matriz de distâncias são calculadas a cada iteração do algoritmo. A seqüência no banco de dados possui um caractere de aviso de fim para que um elemento de processamento (PE) possa fazer as instruções necessárias e atualizar a matriz. O algoritmo Smith-Waterman implementado trata banco de dados com até 10 milhões de bases. A arquitetura pode acomodar mais de 512 bases colocando várias no mesmo elemento (seção 4.1.3).

Esta abordagem produz o escore do alinhamento. A implementação do Smith-Wateman em FPGA em uma única placa Kestrel (512 elementos de computação 20Mhz) teve um *speedup* de até 20 vezes em relação a uma DEC Alpha 433 MHz.

4.2.3 Guccione e outros [Guc02]

Em [Guc02] é mostrada uma adaptação do algoritmo Smith-Waterman para FPGA. A arquitetura utilizada é um vetor sistólico (seção 4.1.1) de elementos de computação, onde são calculadas as antidiagonais.

Em [Guc02], a seqüência procurada é colocada diretamente nos elementos de computação do FPGA, utilizando a sua capacidade de reconfiguração dinâmica. Isto resultou em uma diminuição de 25% do tamanho do circuito tirando a necessidade de dois *flip-flops* por elemento de computação para colocar os dados. Além disto, como a cadeia procurada já é colocada junto com a configuração do FPGA, nenhuma sobrecarga é necessária para armazena-la nos elementos de computação.

A implementação em JBits em uma placa Xilinx Virtex XC2V6000-5 (11000 elementos de computação a 280 MHz) atingiu 3225 GCUPS, enquanto uma placa TimeLogic conseguiu apenas 50 GCUPS, e uma placa Splash 2 (XC4010) apenas 43 GCUPS. Neste artigo, não foi descrita a utilização da matriz de similaridade nem as seqüências utilizadas para teste.

4.2.4 Hoang [Hoa92]

Em [Hoa92] é descrita uma arquitetura de um vetor sistólico em FPGA em uma placa Splash (Xilinx XC3090) com 248 elementos de computação. O vetor sistólico permite que duas seqüências sejam movimentadas no vetor (seção 4.1.2).

Os algoritmos implementados têm relações de recorrência semelhantes as do Smith-Waterman e Needleman-Wunsch [Nee70]. Não há quebra da seqüência procurada. A computação é dividida em duas fases. Numa primeira fase, computa-se a matriz de similaridades. Isto é feito com um vetor que computa e armazena na RAM local apenas os escores.

Numa segunda fase, as distâncias são utilizadas para se procurar o melhor alinhamento. Isto é feito com o mesmo tipo de vetor sistólico (linear) associado a uma máquina de estados finita. As seqüências são novamente percorridas no vetor sistólico, uma anti-diagonal de cada vez começando do canto inferior direito e tentando chegar no canto superior esquerdo da matriz obtendo o melhor alinhamento. Dependendo do vetor associado (vetor para esquerda, para cima ou diagonal) é gerada uma representação binária. Esta representação binária pode significar que o vetor do melhor alinhamento está acima, à esquerda ou na diagonal. No vetor, se estiver sendo seguido o vetor horizontal, isto corresponde a mover a marca de melhor alinhamento para elemento de computação à esquerda no vetor. Se estiver seguindo um vetor vertical, corresponde a mover a marca de melhor alinhamento para a direita no vetor. Se estiver seguindo um vetor na diagonal a marca de melhor alinhamento continua no mesmo elemento de computação.

Os resultados se mostraram centenas de vezes mais rápidos do que os obtidos em um supercomputador Cray 2 para 10.000 alinhamentos de seqüências de tamanho 100. O artigo não mostra como esta arquitetura pode ser escalada para lidar com seqüências de tamanhos grandes que não caibam diretamente no FPGA.

4.2.5 Lavenier [Lav98]

Em [Lav98], é descrita uma arquitetura do algoritmo de Smith-Waterman em uma placa SAMBA (*Systolic Accelerator for Molecular Biological Applications*). Com pequenas modificações, esta placa pode servir para aumentar a velocidade dos algoritmos BLAST, FASTA, SSEARCH, além do próprio Smith-Waterman.

A placa SAMBA é um vetor sistólico de 128 processadores de 12 bits. A arquitetura do FPGA é de um vetor sistólico linear onde se calculam os coeficientes das anti-diagonais da matriz (seção 4.1.1). A seqüência procurada é particionada (figura 4.6) e cada parte é comparada com todo o banco de dados para gerar a matriz. Esta arquitetura calcula o escore.

Em uma comparação entre o SAMBA e uma DEC Alpha 150 MHz para procurar em uma seqüência de base de 21210389 aminoácidos e seqüência procurada de 3000, o SAMBA conseguiu um tempo de 3:20 minutos e a DEC Alpha de 280:00 sendo portanto 83 vezes mais rápida.

Num trabalho anterior, em [Lav96] são feitas as comparações entre 3 tipos de hardware para implementação do Smith-Waterman. O VLSI dedicado, o VLSI programável e o FPGA. Todos utilizaram um vetor sistólico para o cálculo da matriz (seção 4.1.1).

Para fazer esta comparação, foi utilizado o desempenho de pico para cada uma das tecnologias. A unidade de medida foi em milhões de atualizações de células por segundo(*Million Cell Updates per Second*) MCUPS. Cada atualização de célula corresponde ao cálculo da similaridade entre as duas posições nas duas seqüências. O VLSI dedicado conseguiu resultados da ordem de 20 vezes melhores que o FPGA (5120 MCUPS na Splash-2) e o VLSI programável, mas não implementou o Smith-Waterman completo, não incluindo os espaços. Além disto, o VLSI dedicado não pode ser modificado nem melhorado.

4.2.6 Marongiu e outros [Mar03]

No artigo de [Mar03], é utilizada uma ferramenta PHG (*Parallel Hardware Generator*), para gerar um código VHDL e sintetizar um vetor sistólico para o Smith-Waterman (seção 4.1.1).

O problema resolvido consistiu em procurar uma pequena seqüência de aminoácidos (peptídeo) em um conjunto grande de aminoácidos.

Implementado em um FPGA Xilinx XV1000-4, conseguiu um resultado 5.6 vezes mais rápido do que um Pentium III 1 GHz, para procurar 24 aminoácidos em uma base de 2M aminoácidos. Esta abordagem encontrou o escore e não fez particionamento.

4.2.7 Mosanya [Mos98]

A tese de doutorado de [Mos98] propõe um vetor sistólico unidimensional (seção 4.1.1) para a execução do algoritmo de Smith-Waterman. A forma de contagem dos escores é o chamado perfil generalizado de Gribskov. O FPGA usa um vetor sistólico para fazer o cálculo dos escores das anti-diagonais da matriz. A implementação também é capaz de calcular o melhor escore de alinhamento dado pelo perfil generalizado. As seqüências procuradas podem ser particionadas (figura 4.6).

Após a implementação de um algoritmo que encontra escore para comparação de seqüências em FPGA usando VHDL para Xilinx XC4000 (Genstrom), foi constatado que se conseguiu entre 25 e 60 milhões de escores por segundo nas implementações em FPGA (placas Renco e Genstorm) e apenas 600.000 em uma Sun Sparc e 5 milhões em uma rede de oito máquinas.

4.2.8 Oliver e outros [Oli05c]

Em [Oli05c], é proposto um vetor sistólico linear unidimensional (seção 4.1.1). A LUT (seção 3.3) foi projetada para tratar diferentes valores de penalidades associados ao Smith-Waterman. Na penalidade linear o custo do primeiro espaço (*gap*) é idêntico aos seguintes. Na penalidade

utilizada (*affine gap*) [Goh82] o primeiro vazio tem um valor e os seguintes podem ter valores diferentes (na maioria dos casos menor).

A seqüência de base foi dividida em várias partes de tamanho fixo com o mesmo tamanho do vetor de comparação. Se a seqüência procurada for maior que o vetor de comparação, cada elemento do vetor pode guardar até quatro elementos da seqüência procurada (figura 4.7). Se o tamanho da seqüência procurada não for múltiplo do tamanho do vetor, o excedente é preenchido com zeros.

O tamanho máximo da pesquisa foi de 2016 bases. A implementação em um FPGA Virtex II XC2V6000 conseguiu tratar até 252 elementos de computação a 55 MHz, resultando em 1390 MCUPS. Nesta implementação, o FPGA calcula a matriz e a envia para a CPU responsável pela tarefa de encontrar o melhor alinhamento. Comparando os resultados obtidos com um programa otimizado em C executando em um Pentium 4 1.6 GHz, o *speedup* chegou a 170 para escore linear e 125 para "*affine gap*".

Em [Oli05b], pesquisadores do mesmo grupo propuseram uma modificação do vetor de [Oli05c] para executar parte do alinhamento múltiplo do Clustalw [Tho94] que corresponde a alinhamentos de seqüências duas a duas. Isto foi feito também com "*affine gap*" e o vetor foi modificado para guardar o número de coincidências (*matches*). O desempenho máximo chegou a 3.1 GCUPS. Isto gerou um *speedup* de 50.9 vezes para 200 seqüências de globina quando comparado com um Pentium4 3GHz 1GB de RAM.

4.2.9 Puttegowda e outros [Put03]

Em [Put03], é proposto um vetor sistólico unidirecional (seção 4.1.1) para a execução do Smith-Waterman em uma placa Osíris. Neste caso, somente a matriz de similaridade é calculada. Ela possibilita quebra da seqüência procurada e consegue 1260 GCUPS com um "*speed-up*" de 500 sobre uma placa Timelogic. Foram utilizadas seqüências do GenBank com tamanhos de milhões de bases.

Para aumentar o desempenho, foi feita a reconfiguração dinâmica do FPGA onde as seqüências que são procuradas são colocadas nos elementos do vetor sistólico. Este FPGA apresentou 7000 elementos de computação por componente com até 180 MHz e 1260 MCUPS. Para procurar uma seqüência (que coube no FPGA) em uma base de 48 MB demorou menos de 0.5 segundos.

4.2.10 West e outros [Wes03]

Em [Wes03] é proposta a execução do algoritmo Smith-Waterman em um vetor sistólico unidirecional (seção 4.1.1). A seqüência procurada pode ter até 38 bytes. O objetivo do autor foi utilizar um FPGA para fazer procura rapidamente em um banco de dados de genes. Para ser capaz de processar um grande número de dados, o FPGA foi colocado no barramento ATAPI/IDE para ter uma maior velocidade de transferência de dados com o disco.

A arquitetura proposta foi generalizada para tratar com bytes ao invés das representações binárias comumente utilizada no Smith-Waterman. Assim o algoritmo pode realizar também procura em textos.

Esta implementação calcula o Smith-Waterman pelas linhas da matriz ao invés de pelas antidiagonais como é feito normalmente. A seqüência de base teve de ser particionada (figura 4.6), pois poderia ter vários megabytes de tamanho. A placa de FPGA utilizada é uma FPX (*Field Programmable Port Extender*). A máquina hospedeira utilizada foi um Intel Pentium-III 933 MHz com dois processadores, 512 MB RDRAM e sistema operacional Redhat Linux 7.2. As seqüências alvo e do banco de dados foram textos gerados sinteticamente.

Para uma velocidade de transferência de 40,5 MB/s e padrão de 38 bytes, o ganho de tempo do FPGA foi de 50,1 em relação a implementação em software. Utilizando outra métrica (o número de atualização de células por segundo) a implementação em software conseguiu 30.7 MCUPS e a implementação em FPGA conseguiu 3,8 GCUPS.

4.2.11 Worek [Wor02]

A dissertação de mestrado de Worek [Wor02] mostra o projeto do vetor sistólico unidirecional do Smith-Waterman (seção 4.1.1). Neste caso, um conjunto de elementos do vetor calcula e armazena a matriz de similaridade.

Foi testada a reconfiguração durante a execução para melhorar o desempenho. Caso as seqüências sejam grandes demais para caberem nos elementos de computação do FPGA, elas são carregadas e salvas da memória interna da placa do FPGA havendo uma comunicação com o computador hospedeiro para que possa utilizar e substituir os valores calculados. A quebra da seqüência procurada foi feita através da reconfiguração do FPGA. Como resultado, foi obtido que um Pentium III 1.4 GHz conseguiu atingir 82 MCUPS e um FPGA em uma placa OSIRIS conseguiu atingir 389 GCUPS.

4.2.12 Yamaguchi e outros [Yam02]

Em [Yam02], o Smith-Waterman é executado em um vetor unidirecional (seção 4.1.1). Caso a seqüência a ser procurada seja maior que o número de elementos de computação disponíveis, a seqüência é quebrada em várias partes (figura 4.6). Isto é feito de forma simultânea (*multithreaded*), onde se calcula primeiramente a parte superior da anti-diagonal e guarda-se a seqüência procurada em registradores temporários. A seguir, coloca-se a segunda parte da seqüência nos elementos de computação e se calcula a parte interior da anti-diagonal. Este procedimento é útil, pois se aproveita o ciclo ocioso que se tem entre uma comparação e outra. Isto é feito até terminar o cálculo e armazenamento de matriz de similaridade.

Na fase seguinte, o melhor alinhamento é computado pelo FPGA utilizando a matriz gerada. O alinhamento em hardware encontrado foi de 50 vezes mais rápido quando comparado ao software.

A execução do Smith-Waterman (apenas a primeira fase que é o cálculo do escores) em um FPGA Xilinx XCV2000E (144 elementos de computação a 40MHz) alinhou uma seqüência de

2048 bases em um banco de dados de 64 milhões de elementos em 34 segundos, o que é quase 330 vezes mais rápido do que um PentiumIII de 1GHz atingiu. Para a segunda fase, a implementação em FPGA para uma seqüência de 1024 elementos e banco de dados de 4096 elementos demorou 0,007 segundos enquanto o Pentium III executou em 0,35 segundos.

4.2.13 Yu e outros [Yu03]

Yu e outros [Yu03], propõem uma arquitetura unidirecional para o Smith-Waterman (seção 4.1.1) em um FPGA Xilinx Virtex. Utilizando VHDL Xilinx Foundation 5 em um Xilinx XCV1000E-6, com 4032 elementos de computação operando a 202 MHz este projeto atingiu 814 MCUPS. O objetivo da arquitetura é encontrar o maior escore e não há particionamento das seqüências.

4.2.14 Zhang e outros [Zah07]

Em [Zah07] é proposto um vetor sistólico unidimensional (seção 4.1.1) em um FPGA XD1000. Para aumentar o desempenho, a arquitetura utiliza vários estágios, cada um com seu próprio *clock* e com latências diferentes. No caso da comparação de aminoácidos, uma otimização do armazenamento reduziu em 80% o espaço gasto na tabela de substituição. A arquitetura calcula a matriz de similaridade e a armazena para uso posterior e faz também o particionamento (figura 4.6).

Os resultados de um programa em C foram comparados com um AMD 64 Opteron 2.2GHz com 8 GB de memória RAM. O desempenho máximo do FPGA foi 25,6 GCUPS e o *speedup* máximo sobre o software foi de 249,52 para duas seqüências de tamanhos de 65536 bases.

4.3 ANÁLISE DAS PROPOSTAS

As propostas analisadas diferiram muito no tipo de implementação, marca e características da placa de FPGA utilizada, número de elementos de computação utilizados, entre outros aspectos. Desta forma, é muito difícil fazer uma comparação abrangente entre as

implementações pois os artigos podem omitir detalhes de implementação, como o relacionamento entre o FPGA e o computador hospedeiro, a forma de particionamento das seqüências, testes com tamanhos variados de seqüências e desempenho segundo métricas utilizadas em outros trabalhos. Mesmo assim procuramos neste trabalho avaliar as características comuns a várias implementações.

A seguir, é apresentado um quadro comparativo entre as várias implementações do algoritmo Smith-Waterman e algoritmos relacionados em FPGA de 14 das propostas analisadas na seção 4.2.2.

A estrutura da tabela 4.1 é a seguinte: na primeira coluna temos o artigo. Na segunda coluna temos o FPGA utilizado (ou principal, se foi testado mais de um). Na terceira coluna, são apresentados os tamanhos das seqüências procurada e do banco de dados, respectivamente. A quarta coluna indica se foi implementada a quebra da seqüência procurada, quando ela não coube no FPGA. Na quinta coluna, está a capacidade computacional da placa em GCUPS. Na sexta coluna, está o *speed-up*, ou seja, quantas vezes a implementação foi mais rápida do que outra. Na sétima coluna, é apresentada a arquitetura com a qual a implementação foi comparada. Na oitava coluna consta se foi utilizada reconfiguração dinâmica na implementação. A nona coluna contém o tipo de utilização da matriz de similaridade. Existem as seguintes possibilidades: ALINHAMENTO indica que foi calculado o melhor alinhamento. ARMAZENA indica que a matriz foi armazenada para uso posterior, CÁLCULO a matriz é calculada mas o artigo não indica sua utilização. E ESCORE indica o valor do maior escore é recuperado.

Nos casos de comparação, foi escolhido o melhor resultado para ser colocado na tabela 4.1. Em alguns casos, os resultados foram inconclusivos ou não estiveram disponíveis no artigo sendo que estes estão indicados por “Não disponível” na tabela.

Tabela 4.1 Quadro Comparativo

Artigo	FPGA	Tamanhos Seqüências	Particiona-mento	GCUPS	"Speed-Up"	Arquitetura	Recon. Din.	Utilização da matriz
[Lav98]	SAMBA	3K x 2.1M	Sim	Nao Disp.	83	DEC Alpha 150 MHz	Não	Escore
[Guc02]	Virtex XC2V6000	Não Disp.	Sim	3220	75	Splash-2	Sim	Cálculo
[Wor02]	XC2V1000	ND x MBs .	Sim	389	4743	Pentium III 1,4 GHz	Sim	Armazena
[Mar03]	Xilinx XV1000	24x 2M	Não	Não Disp.	5,6	Pentium III 1 GHz	Não	Escore
[Lav96]	Splash-2	Não Disp.	Não	5	1	VLSI prog.	Não	Cálculo
[Wes03]	FPX	38 x MBs	Não	3,8	50,1	Pentium-III 933 MHz	Não	Escore
[Hoa92]	Splash Xilinx XC3090	100 x MBs	Não	Não Disp.	325,3	Cray-2	Não	Alinha-mento
[Mos98]	Xilinx XC4000	ND x MBs	Sim	0,06	100	Sun Spark	Não	Escore
[Yu03]	Xilinx XCV1000	Não Disp.	Não	0,8	Não disp.	Não Conclusivo	Não	Escore
[Put03]	OSIRIS	ND x 48MB	Sim	1260	500	Timelogic	Sim	Cálculo
[Yam02]	Xilinx XCV2000E	2K x 64MB	Sim	Não Disp.	330	PentiumIII de 1GHz	Não	Alinha-mento
[Gra01]	Kestrel	512x10M	Sim	Não Disp.	20	DEC Alpha 433 MHz	Não	Escore
[Oli05b]	Virtex II XC2V6000	2.0k x ND	Sim	9,2	170	Pentium 4 1,6 GHz	Não	Escore
[Zah07]	XD1000	65536 x 65536	Sim	25,6	249,52	AMD 64 2.2GHz	Não	Armazena

Analisando a tabela 4.1, podemos ver que em 9 casos houve a quebra da seqüência procurada. Isto é muito importante nos casos de aplicações reais onde as seqüências não cabem todas no FPGA. Mas muitas vezes ([Lav98] [Gra01]), quando foram executados os testes, as seqüências procuradas foram muito pequenas, o que evitou a sua quebra e eventuais comunicações entre o FPGA e computador hospedeiro. Isto fez com que os resultados apresentados ficassem melhores.

Apenas duas arquiteturas encontraram o alinhamento [Hoa92] [Yam02] e, ainda assim, para seqüências procuradas pequenas. Isto indica a dificuldade de armazenar matrizes grandes dentro do FPGA para calcular o alinhamento.

A reconfiguração dinâmica como em [Guc02] onde se coloca a seqüência procurada implementada diretamente no FPGA junto com as constantes de inserção, remoção e comparação entre as bases, pode apresentar resultados muitos rápidos, mas como a

reconfiguração demanda tempo, ela é útil nos casos a seqüência procurada cabe no FPGA e é comparada a uma grande seqüência no banco de dados.

Uma característica muito importante é o tamanho das seqüências avaliadas nas implementações do algoritmo de Smith-Waterman. A maioria das aplicações procura comparar seqüências grandes de vários KBs. Por exemplo, duas seqüências de 20KB teria como resultado uma matriz de 400 MB que está muito além da capacidade de armazenamento das placas de FPGA atuais. Isto gera a necessidade de se quebrar tanto a seqüência procurada quanto a seqüência de base. Quebrar estas seqüências é uma operação complexa, pois é necessário manter partes da matriz de coeficientes na placa para que novos coeficientes possam ser calculados. Assim há a necessidade de envio constante de pedaços de seqüências (tanto as procuradas como a do banco de dados) para o FPGA e envio constante de pedaços da matriz calculada que não cabem mais no FPGA para o computador hospedeiro. Além disto, a comunicação entre a placa de FPGA e o computador hospedeiro geralmente é lenta, pois elas podem estar no barramento PCI, que é lento em relação ao barramento que liga a CPU do computador hospedeiro e a sua memória.

Existe a expectativa de que, no futuro, as placas de FPGA possam ter espaço de armazenamento suficiente para grandes seqüências e possam ser ligados a barramentos de alta velocidade. Assim, grande parte dos problemas relacionados à quebra de seqüências poderia ser melhorado.

Artigos diferentes podem utilizar diferentes métricas de avaliação de desempenho. Uma forma de avaliação é o tempo gasto para se obter os escores na matriz do algoritmo Smith-Waterman para duas seqüências. Normalmente, avalia-se o tempo gasto para a comparação na arquitetura implementada e também o tempo gasto para avaliar as mesmas seqüências em um computador dedicado, *cluster* de computadores ou outra arquitetura de FPGA. Esta métrica normalmente é muito dependente dos tamanhos de seqüências escolhidas, pois normalmente aumentar o tamanho das seqüências gera um impacto muito grande no desempenho do FPGA, como já foi discutido. Por outro lado, se o algoritmo é implementado em uma máquina dedicada ou em um

cluster, é muito mais fácil lidar com seqüências maiores, pois estas máquinas possuem muito mais memória RAM e ela está acessível através de um barramento de alta velocidade.

Após avaliar várias implementações do algoritmo Smith-Waterman e assemelhados em FPGAs pode-se ver que os resultados obtidos foram muito significativos em relação às arquiteturas comparadas. Nas várias métricas, foram obtidos resultados centenas e às vezes milhares de vezes mais rápidos que as soluções em *software*. Isto indica um grande potencial para este tipo de arquitetura.

O algoritmo Smith-Waterman não é de implementação simples em FPGAs uma vez que a matriz gerada pode ser muito grande. Várias otimizações visando aumentar a capacidade de armazenamento de coeficientes da matriz no FPGA foram tentadas. Existe muito espaço para otimização deste algoritmo em FPGA, principalmente no que diz respeito a tentar manter dentro do FPGA apenas o número mínimo suficiente de coeficientes para calcular os escores do melhor alinhamento.

Na medida em que os FPGAs evoluem, eles devem aumentar a sua capacidade de armazenamento e velocidade tornando cada vez mais fácil a implementação do Smith-Waterman. Pode-se então armazenar as seqüências procuradas inteiras diretamente no FPGA, simplificando bastante o processo de procura. Se, além disto, a placa puder ser ligada em um barramento rápido, seqüências poderão ser comparadas com grande velocidade.

5–PROJETO DAS ARQUITETURAS WAVEFRONT PARA COMPARAÇÃO DE SEQÜÊNCIAS

5.1 DECISÕES DE PROJETO

O estudo das diversas arquiteturas de hardware dedicado para comparação de seqüências, apresentadas na seção 4.3, mostra que a maioria das propostas é de vetores sistólicos que são capazes de calcular vários escores da matriz de programação dinâmica simultaneamente. Quanto maior o tamanho do vetor sistólico, maior o número de escores que podem ser calculados simultaneamente e, portanto maior a sua capacidade de paralelização. Estes vetores têm a capacidade de reduzir a complexidade do algoritmo de programação dinâmica de comparação de seqüências de O(*mn*) para O(*mn/k*) onde *k* é o número de elementos de computação no vetor. Assim, uma arquitetura na forma de vetor sistólico é muito útil, do ponto de vista do poder computacional, para este problema.

No entanto, o vetor sistólico trabalha de maneira síncrona, onde todas as operações são realizadas sempre em um número fixo de ciclos de *clock*. Isto faz com que o circuito trabalhe sempre com o número de ciclos necessários para tratar o pior caso (seção 3.1).

Como o objetivo da arquitetura proposta nessa tese é que a mesma seja utilizada para implementar mais de um algoritmo de comparação de seqüências, consideramos que o vetor sistólico, por sua característica síncrona, não seria o mais adequado. Propomos, então, que o vetor de elementos de processamento seja wavefront e não sistólico, como a maioria das abordagens propostas na literatura (seção 4.2).

O vetor wavefront (seção 3.1) é diferente do sistólico por ser assíncrono e, portanto, é de projeto mais complexo pois necessita de um protocolo de *handshaking*. Embora o vetor wavefront utilize parte do seu tempo realizando o *handshaking*, sua arquitetura é mais flexível e particularmente útil nos casos onde diferentes elementos necessitem executar operações que demoram tempos diferentes. Neste caso, cada elemento pode operar utilizando um número

diferente de ciclos para completar sua operação e o vetor não necessariamente funnciona com o desempenho do pior caso.

Uma das principais limitações para a utilização de hardware para o alinhamento de seqüências biológicas é o tamanho da matriz de similaridade. Para aplicações práticas, a matriz de similaridade pode ser grande (seção 2.4.1). Isto gera alguns problemas para o projeto em FPGA [Wol04], causados principalmente pelo tamanho do FPGA e pelos gargalos de comunicação. O tamanho do FPGA limita o tamanho das seqüências que podem ser comparadas dentro dele. Soluções, como colocar matriz de similaridade na memória da placa do FPGA ou na memória do computador hospedeiro, esbarram nos gargalos de comunicação. Por estas razões, optamos por implementar somente algoritmos com complexidade linear de espaço.

Assim, projetamos uma arquitetura de base utilizando um vetor wavefront que possa evitar estas limitações e seja flexível e eficiente para calcular escores para o Smith-Waterman [Smi81] e DIALIGN [Mor96] e o alinhamento para o DIALIGN.

Inicialmente, foram implementadas variantes do Smith-Waterman (seção 2.4.2) e do DIALIGN (seção 2.4.5). Como, até onde vai nosso conhecimento, não existia uma variante em espaço linear para o DIALIGN, propusemos tal variante na presente tese (seção 5.7.1) e a implementamos. Nada impede, no entanto, que outros algoritmos baseados em programação dinâmica sejam implementados nesta mesma arquitetura de base.

5.2 VISÃO GERAL DA SOLUÇÃO

A arquitetura proposta é um vetor wavefront composto de vários elementos. A utilização da arquitetura é composta de 3 fases: inicialização, cálculo e coleta dos resultados.

Na fase de inicialização, é feita a inserção da seqüência procurada nos elementos do vetor, bem como a inicialização dos valores internos. Na fase de cálculo, a seqüência de base é

inserida no vetor e os resultados são calculados. Na fase de coleta de resultados, os resultados são retirados do vetor. A três fases são ilustradas nas figuras 5.1, 5.2 e 5.3.

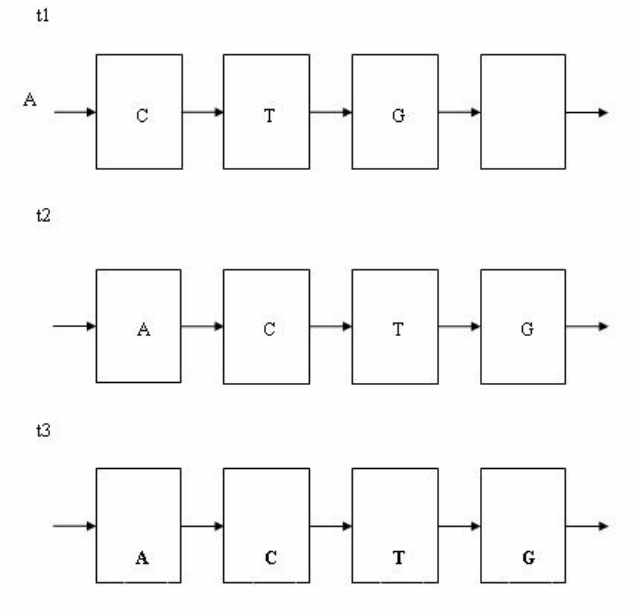

Figura 5.1 – Inicialização do vetor wavefront

Na figura 5.1, é representada a inicialização do vetor wavefront. No instante *t1*, a seqüência procurada ACTG entra no vetor de forma invertida. No instante *t2*, todos os elementos da seqüência procurada são colocados em suas posições finais. No instante *t3*, os elementos da seqüência procurada são colocados em seus registradores definitivos, que são mostrados na parte de baixo e em negrito.

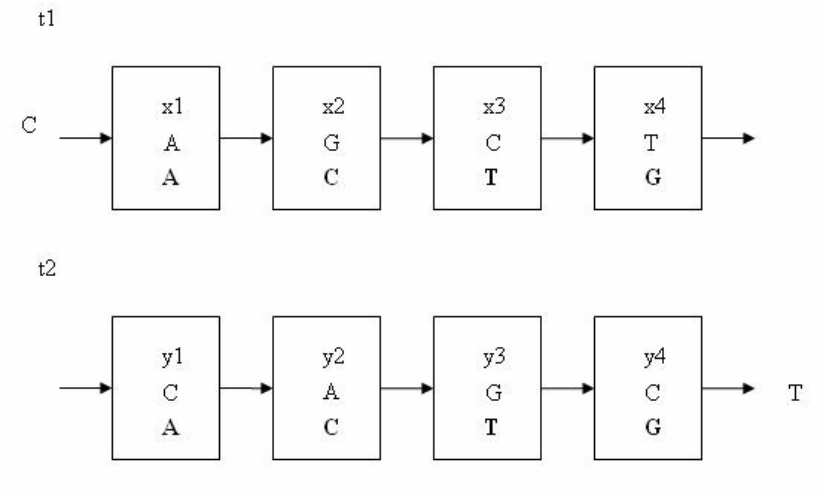

Figura 5.2 – Cálculo no vetor wavefront

Na figura 5.2, é ilustrado o cálculo de posições da matriz de programação dinâmica. Os elementos da matriz são calculados na forma de antidiagonais (seção 4.1.1). A seqüência procurada ACTG está na parte de baixo do elemento (em negrito). Os elementos da seqüência de base TCGAC entram um por um no vetor. No instante *t1*, os elementos do vetor utilizam uma relação de recorrência para calcular a antidiagonal *x* com valores de *x1* a *x4*. No instante *t2*, a base T foi retirada do vetor e a base C é colocada. A seguir, é calculada a antidiagonal *y* com valores de *y1* a *y4*.

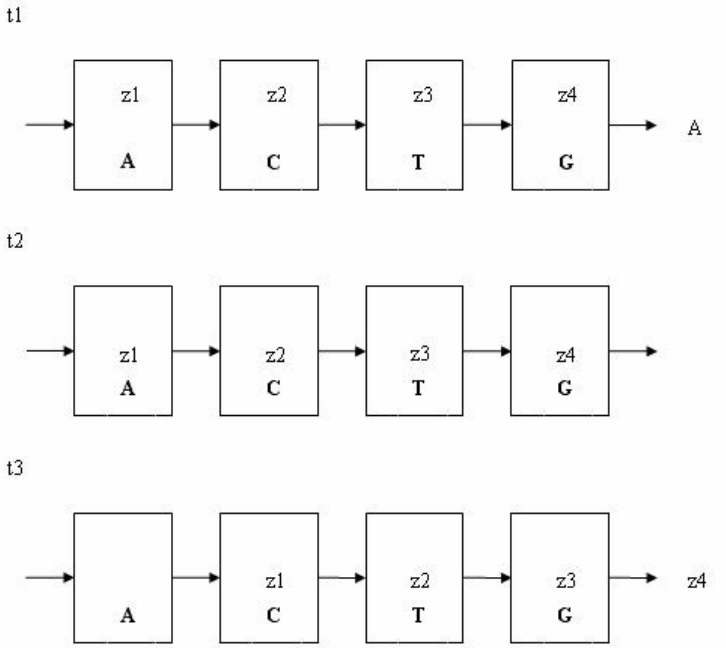

Figura 5.3 – Retirada dos resultados do vetor wavefront

A figura 5.3 mostra como os resultados são retirados do vetor. Os valores dos resultados finais de (*z1* a *z4*) foram armazenados nos elementos de computação. Estes valores podem ser, por exemplo, o maior escore em cada coluna matriz. No instante *t1*, sai o último elemento da seqüência de base do vetor. No instante *t2*, o vetor pára de efetuar os cálculos e vai retirar os resultados gerados. No instante *t3*, os resultados vão saindo um por um, sendo que o resultado gerado no último elemento (*z4*) sai primeiro.

5.3 PROJETO DA ARQUITETURA BÁSICA

A arquitetura básica proposta na presente tese define linhas de controle, dados de entrada e saída e o protocolo de *handshaking* entre os elementos. As operações executadas na equação de recorrência (nomeado CRR na figura 5.4) dependem do algoritmo de comparação de seqüências utilizado e serão detalhados nas seções 5.6, 5.7 e 5.8.

Para o projeto da arquitetura wavefront foi utilizado o SystemC (seção 3.5.1) e o Forte. A combinação de SystemC e Forte foi escolhida por sua facilidade de projeto. O SystemC utiliza a sintaxe do C++ assim, existe a possibilidade de utilizar todas as primitivas do C++ e facilidade de depuração [Ope05]. O Forte gera um código em Verilog a partir do SystemC e possui um ambiente integrado de simulação do código.

A arquitetura projetada é mostrada figura 5.4.

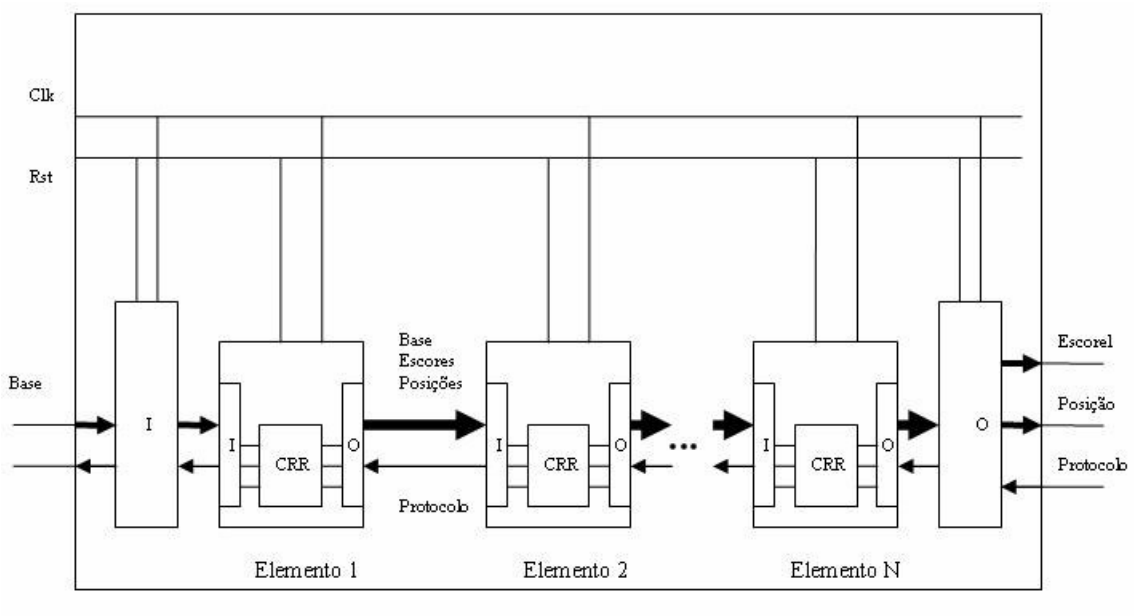

Figura 5.4 - Arquitetura projetada em SystemC

Na figura 5.4, é apresentado um diagrama do circuito do vetor wavefront. Nele, os dados entram pelo lado esquerdo do circuito. Uma parte do circuito, indicada por "I" executa um protocolo bidirecional que é capaz de ler os dados de forma consistente. O dado de entrada do circuito, mostrado com uma seta grossa no canto inferior esquerdo, é a Base da seqüência. O dado de saída "protocolo" representado por uma linha fina é utilizado no protocolo de "*handshaking*". Além disto, o circuito possui uma entrada de *clock* chamada de Clk que é o *clock* global de todos os elementos do vetor. A entrada de *reset* é chamada de Rst. O Clk é repassado para todos os elementos do wavefront. O Rst faz a inicialização dos circuitos no vetor wavefront, inclusive em cada um dos *N* elementos. Uma vez que os dados entram no circuito do vetor wavefront, eles são repassados para o primeiro elemento do vetor, que também possui um circuito dedicado para o protocolo de comunicação, marcado com "I". Uma vez recebidos os dados, a computação do elemento é feita no circuito indicado por "CRR" que significa "Circuito da Relação de Recorrência". O CRR pode ser modificado para conter diferentes relações de recorrência correspondentes a diferentes algoritmos de comparação de seqüências.

A seguir, os dados necessários para a relação de recorrência são enviados para o próximo elemento do vetor utilizando o protocolo de comunicação marcado por "O". A seta grossa indica os dados que são enviados que são a base, escores, e posições. A seta fina de volta é

utilizada no protocolo. Ao final do processamento, os dados são retirados pelo último elemento, “Elemento N”, que se comunica através do protocolo com o circuito marcado com “O”. O circuito de cálculo de recorrência (CRR) é composto por uma máquina de estados com os seguintes estados : configuração, cálculo, obtenção de resultado e inativo.

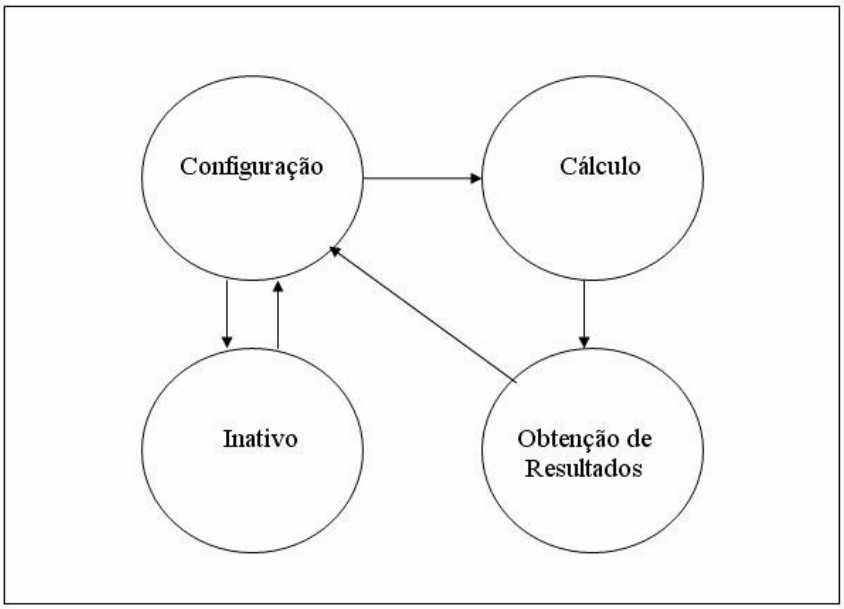

Figura 5.5 – Estados de cada elemento do vetor wavefront

A figura 5.5 ilustra a máquina de estados de cada elemento de processamento, onde o estado inicial é a configuração. Cada elemento do vetor wavefront pode ser configurado de forma diferente. Por exemplo, o elemento pode ter uma base fixa diferente a cada execução. Alternadamente, o elemento pode ser configurado para mudar para o estado inativo, onde ele simplesmente recupera o resultado da sua entrada e o coloca na saída, ficando assim até que seja configurado novamente. Este estado é útil quando nem todos os elementos do vetor são utilizados, no caso onde, por exemplo, a seqüência procurada é menor que o vetor. Neste caso, os últimos elementos do vetor (que não contêm bases de seqüência procurada) são configurados como inativos. Assim, o vetor wavefront, através do seu protocolo de comunicação, permite que os valores calculados pelos elementos ativos do vetor sejam repassados rapidamente pelos elementos inativos. O elemento do vetor pode ser configurado para fazer o cálculo da relação de recorrência e, neste caso, obrigatoriamente deverá passar

pelo estado “obtenção de resultados” onde os resultados gerados são retirados do vetor. O vetor pode fazer novos cálculos após uma nova configuração.

5.3.1 Dados e sinais utilizados

No vetor wavefront proposto, vários elementos de processamento são interligados. Para um melhor desempenho, o controle e comunicação entre estes elementos devem ser reduzidos ao máximo [Kun82]. Para que o vetor wavefront seja eficiente, é interessante que cada elemento seja simples de forma que possa executar em uma maior freqüência de relógio, resultando em um maior número de escores calculados por unidade de tempo [Hoa92].

O primeiro passo para a implementação é determinar quais sinais serão utilizados na comunicação. O primeiro sinal é justamente o sinal de relógio (*clock*), que diz qual a velocidade máxima na qual o circuito pode funcionar. Este sinal é muito importante para os circuitos em FPGA, pois as sincronizações entre portas lógicas e outros componentes são feitas através deste sinal. A velocidade final do relógio é dada pela arquitetura. Diferentes arquiteturas podem executar o mesmo circuito em diferentes velocidades.

Um outro sinal que é comum a todos os elementos é o sinal de “*reset*”. Este é um dos poucos sinais globais do circuito. Ele é necessário para sincronizar o início da execução de todos os elementos. Os elementos podem ter uma pequena inicialização, que é executada justamente nos ciclos após este sinal.

Os elementos têm entradas e saídas *data_in* e *data_out*. O *data_in* é composto por vários bits que incluem todas as entradas de um elemento e o *data_out* os bits de saída de cada elemento.

Os bits de *data_in* dependem da utilização da arquitetura e basicamente contêm:

a) Valores utilizados na relação de recorrência.
b) A base que está entrando no elemento.

c) O bit *flag_ativa* que diz quando o elemento deve começar a calcular a relação de recorrência.

d) Sinais de controle para o estado do elemento.

Os bits de *data_out* contêm:

a) Valores calculados no elemento utilizando a relação de recorrência.

b) A base que está saindo.

c) O bit *flag_ativa*, que indica que o próximo elemento também deve ser ativado.

d) Sinais de controle para o estado do elemento.

O escore que sai de cada elemento é o *escore_d* (que é descrito com o nome de "D" na seção 4.2). O escore que entra no elemento é colocado no lugar do *escore_c*.

Cada elemento contém dentro dele registradores. Os principais são:

a) Registradores com os valores utilizados na relação de recorrência

b) A base fixa no elemento e a base que está passando

c) Registradores para guardar as informações que serão retiradas como resultados

d) Registradores de estado para o funcionamento do elemento

A figura 5.6 mostra os sinais de um elemento da figura 5.4 em SystemC [Ope05].

```
1: SC_MODULE(elemento)
2: {
3:   sc_in< bool >       clk;
4:   sc_in< bool >       rst;
5:   sc_in< sc_int<tam_extra> > data_in;
6:   sc_out< sc_int<tam_extra> >  data_out;
7:   sc_in< bool >  valido_data_in;
8:   sc_out< bool > pronto_data_in;
9:   sc_out< bool > valido_data_out;
10: sc_in< bool >  pronto_data_out;
11: SC_CTHREAD(faz_elemento, clk.pos());
12:}
```

Figura 5.6 – Trecho do módulo do elemento

O trecho mostrado na figura 5.6 mostra três portas de entrada para o *clock* o *reset* e os dados de entrada. Mostra também os dados de saída. Além disto, cada elemento conta com duas entradas e duas saídas (valido_data_in, pronto_data_in, valido_data_out, pronto_data_out respectivamente) para a implementação do protocolo de comunicação. As mesmas portas são utilizadas para a comunicação entre os elementos.

Na última linha, é mostrada a lista de sensibilidade do módulo. A *thread faz_elemento* é executada a cada transição positiva do *clock* sendo que ela pode demorar mais de um ciclo para ser executada.

5.3.2 Protocolo de *handshaking*

A arquitetura wavefront projetada necessita de um protocolo de *handshaking*. O circuito que implementa este protocolo é mostrado na figura 5.4, nos blocos marcados com "I" e "O". No código implementado, os circuitos "I" e "O" são descritos respectivamente por *data_in* e *data_out* e descrevem um protocolo de comunicação feito para evitar leitura e escrita inconsistente entre os elementos. A figura 5.7 apresenta as principais operações deste protocolo.

```
1: Data_in()
2: {
3:   avisa que pode receber dados
4:   espera enquanto não recebeu dados válidos
5:   lê dados
6:   avisa que não pode receber mais dados
7: }
8: Data_out()
9: {
10: escreve dados válidos
11: avisa que enviou dados válidos
12: espera  até o  próximo elemento acusar recebimento
13: avisa que não vai mais enviar dados válidos
14:}
```

Figura 5.7 – Protocolos de *Handshaking*

Na figura 5.7, é importante notar que as portas sempre vão possuir um valor quando lidas, mas os valores só devem ser considerados válidos quando tiverem uma ordem específica para isto.

Os protocolos de comunicação são extremamente importantes, pois senão, corre-se o risco de fazer a computação dos elementos sobre dados inconsistentes, o que geraria resultados errados.

5.4 SÍNTESE DO VERILOG

Após o teste do projeto em SystemC e a verificação do seu funcionamento, passou-se para o passo seguinte, que é a tradução do projeto para a linguagem Verilog para que o projeto possa ser sintetizado. Para fazer a transformação do projeto em SystemC para Verilog foi utilizada a ferramenta Cynthesizer da Forte [For05].

Apesar do Forte ser uma ferramenta poderosa, ele não é capaz de sintetizar qualquer programa em SystemC. Para um código em SystemC poder ser sintetizado, é necessário passar por várias transformações tornando o código mais parecido com uma linguagem de descrição de hardware, como VHDL ou Verilog.

As principais modificações no código feitas no SystemC original foram as seguintes:

a) Foram retiradas todas as instruções de alto nível, como acesso a arquivos, escritas na tela e interação com o usuário, que são utilizadas para depurar o programa.

b) Colocou-se o código SystemC no formato do Forte com restrições de portas, sinais, nomes de métodos, entre outros.

c) Foram retirados os tipos de dados que não eram sintetizados, como ponteiros.

d) Os arquivos foram separados em módulos de acordo com o Forte.

e) Todas as configurações necessárias para a execução do Forte foram feitas.

f) Foi necessário delimitar, através da primitiva *wait()*, o que deveria ser sintetizado em cada ciclo de relógio.

O circuito do vetor wavefront composto de um determinado número de elementos é chamado de *DUT (Design Under Test)* segundo a terminologia do Forte [For05]. Cada um dos elementos tem uma função como, por exemplo, calcular sua relação de recorrência e passar um resultado parcial para o próximo elemento. O *DUT* protocolos de *handshaking* que tem como função passar informações para o primeiro elemento e retirar informações do último elemento do vetor. Além disto, o *DUT* executa algumas funções de controle, como a inicialização do vetor e troca de estado dos elementos.

Para que a arquitetura gerada fosse executada corretamente, foi necessária a criação de um protocolo de comunicação entre os elementos, entre o *DUT* e o primeiro elemento e entre o último elemento e o *DUT*. Também foi necessário um protocolo para fazer a comunicação entre o *DUT* e o *Testbench*, que envia as informações necessárias para a execução. Foi feito, também, um ajuste manual de qual instrução podera ser executada em cada ciclo, pois sendo o SystemC uma linguagem de alto nível, não são tratadas as particularidades da síntese, como o período do ciclo no qual uma instrução pode ser executada. Por exemplo, um código em SystemC pode fazer uma multiplicação em ponto flutuante em um único ciclo de simulação, mas é muito difícil fazer um circuito de hardware real que faça isto. Mesmo que fosse possível executar esta instrução em um único ciclo, isto aumentaria bastante o período do relógio e, portanto diminuiria o desempenho do circuito. O SystemC não tem limite da complexidade da instrução que pode ser executada em cada ciclo, mas a síntese no Forte tem.

Existem algumas fases para a execução do Forte [For05]:

a) Simulação comportamental: o Forte executa o SystemC, gerando o resultado.

b) Simulação de Transferência em Nível de Registrador: O Forte transforma o código SystemC original em um código da forma de transferência entre registradores.

c) Síntese do Verilog: O Forte sintetiza o código da fase anterior fazendo otimizações e gera o código em alguns arquivos em Verilog.

A transferência em nível de registrador [Tha00] tenta separar os dados (registradores) do controle da intercomunicação entre eles para fazer uma melhor especificação.

O Verilog é uma linguagem de descrição de hardware que comporta construções de alto nível, porém o Forte faz várias otimizações do código e gera um código em Verilog com uma máquina de estados que pode ser mais eficientemente sintetizada em hardware [For05]. Isto faz com que o código gerado seja de difícil compreensão. Na máquina de estados, cada um dos estados é uma instrução que pode ser executada em um ciclo de relógio e muda a variável de estado para o estado seguinte. No código gerado, várias máquinas implementando vários circuitos são executadas em paralelo.

O Forte tem parâmetros de otimização em relação ao tamanho do circuito e à velocidade com que ele é executado. Normalmente, para se fazer um circuito mais rápido, é necessário utilizar mais recursos de hardware. E, ao contrário, se o objetivo for economizar recursos, provavelmente o circuito executará de forma mais lenta [For05].

No caso do projeto aqui apresentado, a síntese de um vetor wavefront é uma solução de compromisso. Quanto mais elementos no vetor, maior o número de cálculos de relações de recorrência que podem ser executados por unidade de tempo. Porém, a implementação destes elementos utiliza os recursos do FPGA, que são escassos. Se o vetor for mal projetado, pode acontecer a falta de um recurso importante, como vias de interconexão, e este pode limitar o número de elementos.

Na figura 5.8 é apresentado o circuito de um dos elementos de computação conforme sintetizado pelo Forte.

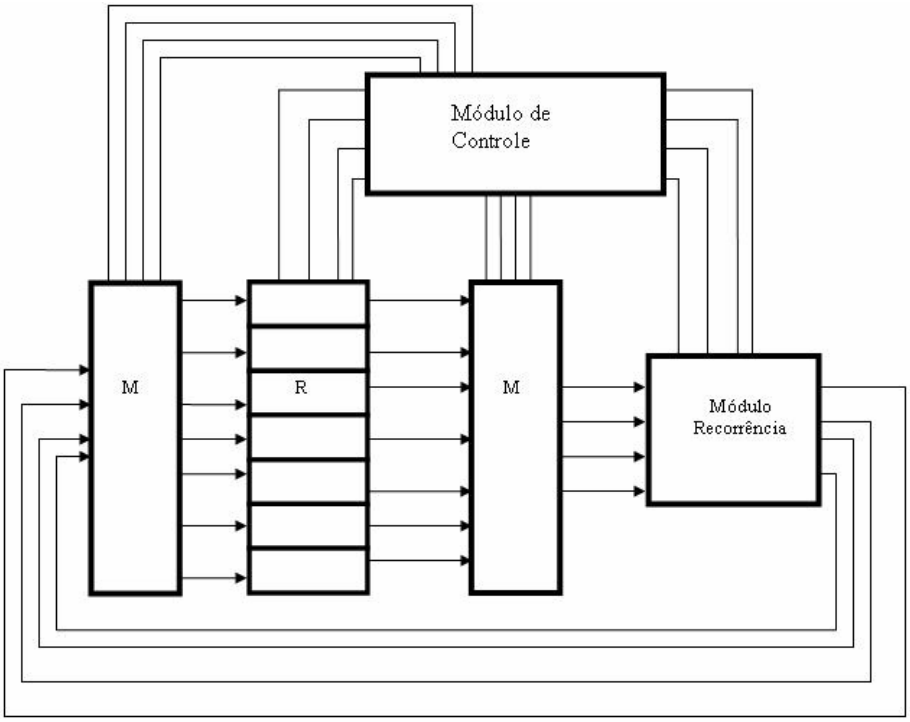

Figura 5.8 - Caminho de dados de cada elemento de computação

A figura 5.8 mostra o circuito sintetizado. Os valores calculados nas relações de recorrência bem como os valores temporários necessários são armazenados em um banco de registradores marcados com "R" no circuito. Os valores nos registradores são selecionados de acordo com uma rede de multiplexadores indicada no circuito por "M". Os valores são enviados para o circuito chamado de "Módulo Recorrência" que calcula as relações de recorrência. As relações de recorrência são especificadas para cada algoritmo e têm de ser especificada detalhadamente em SystemC, inclusive quais operações devem ser feitas em cada ciclo de relógio. Os valores calculados são colocados novamente no banco de registradores, em seu local adequado, determinado por uma outra rede de multiplexadores. O fluxo de dados e as operações realizadas no "Módulo de Recorrência" são definidos pelo "Módulo de Controle", que é uma máquina de estados e que determina o que deve ser executado a cada momento, sendo sintetizada com base nos estados especificados em SystemC.

A seguir, é apresentado um exemplo do código em Verilog gerado pelo Forte. A figura 5.9 mostra um trecho do código gerado para o elemento.

```
1: module elemento(clk, rst, data_in, data_out, valido_data_in, pronto_data_in, valido_data_out,
pronto_data_out);
2:     input clk;
3:     input rst;
4:     input [63:0] data_in;
5:     input valido_data_in;
6:     input pronto_data_out;
7:     output [63:0] data_out;
8:     output pronto_data_in;
9:     output valido_data_out;
```

Figura 5.9 – Exemplo do código gerado em Verilog para o elemento

A figura 5.9 mostra o equivalente em Verilog para o código das portas descritas em SystemC na figura 5.7. Pode ser visto que os nomes especificados em SystemC são mantidos no código em Verilog.

5.5 ARQUITETURA PARA O ALGORITMO DE SMITH-WATERMAN

Uma vez construída a arquitetura de base, ela pode ser modificada para funcionar com diferentes relações de recorrência. Nesta seção, são utilizadas as relações de recorrência do algoritmo Smith-Waterman (seção 2.4.2). O objetivo foi executar a primeira fase do algoritmo que recupera o alinhamento em espaço linear (seção 2.4.4). Para isto, foi necessário calcular todos os elementos da matriz de programação dinâmica e armazenar qual é o maior escore e onde ele ocorre. Esta informação pode ser utilizada posteriormente por um programa em software para fazer o alinhamento somente da região onde acontece o alinhamento local [Bat08]. Esta abordagem evita as limitações da placa de FPGA (seção 5.1), pois as duas seqüências são colocadas no vetor wavefront, é executado um processamento de complexidade O(*mn/p*) e são armazenados dados de tamanho proporcional a *p*, onde *p* é o tamanho do vetor.

O vetor wavefront projetado está ilustrado na figura 5.10 é baseado na seção 4.1.1.

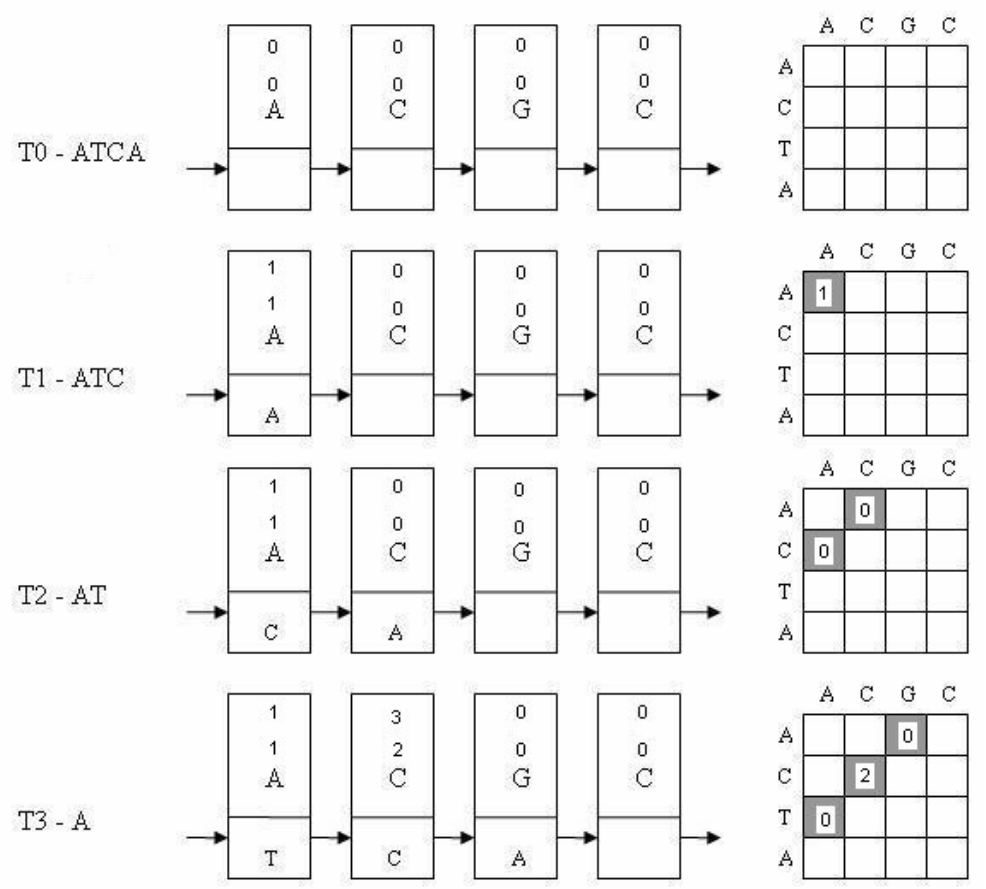

Figura 5.10 – Processamento do vetor wavefront

Na figura 5.10, cada um dos elementos armazena uma base da seqüência procurada, uma base da seqüência que está passando pelo vetor wavefront e também dois valores. Os números no topo do elemento de computação são as informações armazenadas. O valor mais acima representa o ciclo onde o maior escore ocorreu e o valor abaixo é o maior escore que já ocorreu no elemento. Os dois valores são inicializados em 0. No instante *t1*, o escore 1 é calculado no primeiro elemento e no primeiro ciclo e, portanto o primeiro elemento armazena os valores 1 e 1. No instante *t3*, o segundo elemento calcula um escore 2 no ciclo 3. Assim, o segundo elemento armazena os valores 3 e 2.

Após o cálculo da matriz completa, os valores são retirados dos elementos da mesma forma que as bases estão passando. Pela ordem na qual o escore é retirado do elemento, sabe-se qual elemento continha o escore e, portanto, em qual coluna da matriz o maior escore ocorreu. Pelo

valor do ciclo, sabe-se em que anti-diagonal foi calculado o escore e, portanto, em qual linha o escore foi calculado.

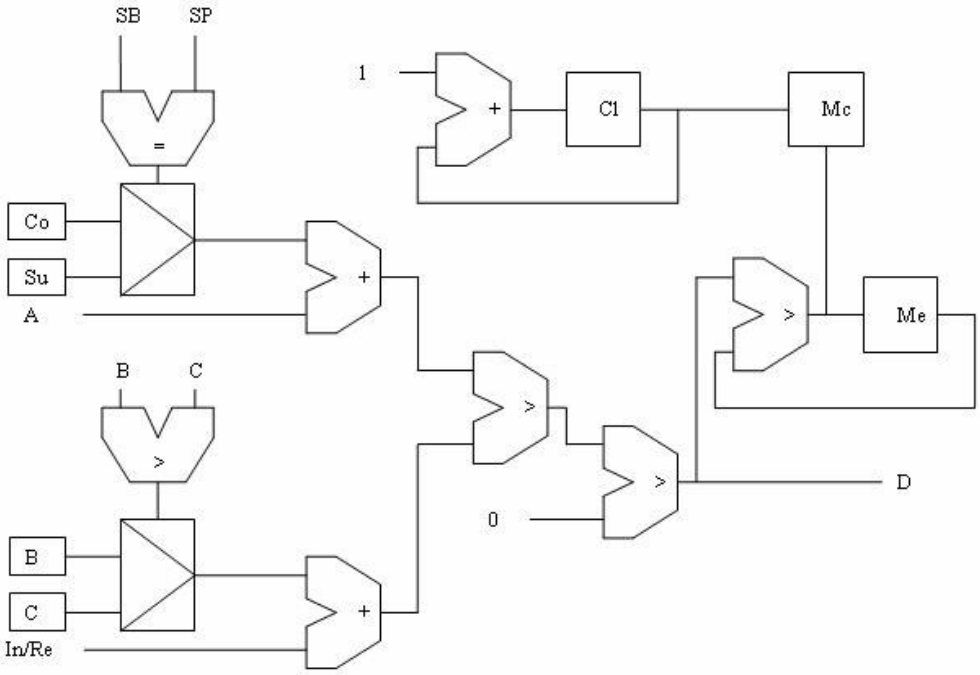

Figura 5.11 – Circuito para relação de recorrência do Smith-Waterman

A figura 5.11 mostra o circuito para o cálculo de equação de recorrência do Smith-Waterman (seção 4.1.1). Para calcular cada célula (D) da matriz, são necessárias três células. A célula da diagonal (A), a célula acima (B) e a célula à esquerda (C) na matriz de similaridade. Primeiramente, um elemento da seqüência procurada (SP) é comparado com um elemento da seqüência de base através de um comparador de igualdade marcado com "=". Se houve igualdade, o multiplexador logo abaixo tem como saída a constante (Co) que é o escore de uma coincidência. Se não houve igualdade, a saída do multiplexador é a constante (Su) que é o escore de substituição. A saída do multiplexador é somada com o valor de (A) através de um somador marcado com "+".

Em paralelo, os valores de (B) e (C) são comparados com um comparador marcado por ">" na figura. A saída do multiplexador abaixo dele será o maior valor entre (B) e (C). Este valor será somado com o escore associado à inserção e remoção (In/Re). A seguir um comparador

escolhe o maior valor dentre os dois calculados anteriormente e este valor é comparado com zero. Se for maior que zero, este será o valor (D) e, caso seja menor, o valor de (D) será zero.

Para calcular o melhor escore, um comparador é conectado a célula (D). O novo valor (D) é comparado com o registrador (Me) que guarda o melhor escore (D) que já ocorreu naquela coluna. Se o valor (D) recém-calculado é maior que o valor no registrador Me então o valor de (Me) é substituído por (D). O registrador (Cl) é incrementado cada vez que um novo valor de (D) é calculado e, portanto, contém o número de antidiagonais que foram calculadas até o momento. Saber o número da antidiagonal permite, portanto, saber a linha da matriz em que (D) foi calculado. Se o valor de Me foi alterado então é permitida a escrita do valor de (Cl) no registrador (Mc) que conterá a linha onde foi encontrado o escore (D) na coluna.

5.6 ARQUITETURA PARA O ALGORITMO DE CÁLCULO DE ESCORE DO DIALIGN

A fase mais intensiva de computação do DIALIGN (seção 2.4.5) é o alinhamento de pares de seqüências, respondendo por mais de 90% do tempo de processamento [Abd01] [Sch04] e, portanto, é a fase mais interessante de ser paralelizada em hardware. A principal dificuldade é colocar em um vetor wavefront as relações de recorrência do DIALIGN que são bastante diferentes das do Smith-Waterman ou Needleman-Wunch. A arquitetura desenvolvida aqui é, portanto, diferente de todas as arquiteturas vistas na seção 4.2.

O primeiro passo é desenvolver uma arquitetura capaz de identificar os fragmentos (também chamados de diagonais). Da fórmula 2.5 foi observado que o menor fragmento que atinge o limiar T=16 utilizado em [Mor96] é uma seqüência de 12 coincidências. Quanto maiores os fragmentos, maior a chance de um alinhamento ótimo não ser encontrado devido a inconsistências entre os mesmos [Mor98]. Assim, um parâmetro interessante no DIALIGN é fazer m=l na fórmula 2.5 fazendo com que o algoritmo encontre as menores diagonais que satisfazem o limiar T. No vetor wavefront, o escore E será o mesmo utilizado na fórmula 2.6 com a diferença que o logaritmo utilizado será em base 2, mais adequada para a representação em hardware.

A arquitetura para o cálculo do escore no DIALIGN foi feita utilizando como base a arquitetura descrita na seção 5.3.

A figura 5.12 mostra como a matriz de escore é calculada pela arquitetura wavefront [Bou07b].

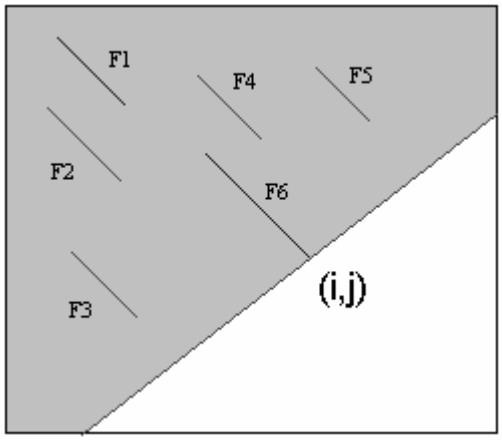

Figura 5.12 – Matriz de similaridade do DIALIGN

A figura 5.12 mostra que o cálculo da matriz é feito através de antidiagonais. Os escores já calculados são mostrados em cinza. As diagonais cujos escores ultrapassam o limiar *T* são mostradas como linhas escuras nomeadas de F1-F6.Os fragmentos F1 e F2 fazem parte do alinhamento ótimo na posição (*i,j*) e os fragmentos F2-F5 não fazem parte do alinhamento por serem inconsistentes com o alinhamento ótimo. Em um dado ponto (*i,j*) da antidiagonal que está sendo calculada no momento é necessário decidir se a diagonal será estendida ainda mais ou se ela terminou. Caso ela tenha terminado, será alinhada com as outras diagonais já encontradas.

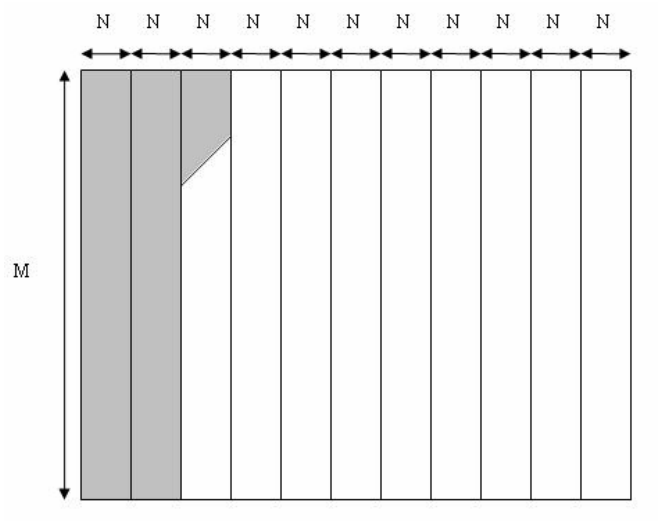

Figura 5.13 – Particionamento da seqüência procurada

A figura 5.13 mostra o particionamento da seqüência procurada em segmentos de tamanho N onde N é o tamanho do vetor wavefront. Cada segmento N é comparado com toda a seqüência de base de tamanho M.

Uma arquitetura do DIALIGN em hardware deve conter todas as fórmulas de 2.5 até 2.12. Além disto, para aumentar o desempenho, o cálculo da diagonal e o alinhamento das diagonais são feitos simultaneamente dentro dos elementos do vetor wavefront. Assim é necessário o cálculo de dois escores: um para a diagonal que está sendo calculada no momento e outro para o alinhamento das diagonais. Para fazer este cálculo pelas relações de recorrência, são necessários 3 valores para cada escore. O algoritmo dentro de cada elemento funciona conforme a figura 5.14.

1 :Descobre o melhor alinhamento de diagonais para a posição utilizando a fórmula 2.12
2: **Se** possível estende diagonal utilizando as fórmulas 2.5 e 2.6
3: **Senão** a diagonal chegou ao fim
4: **Se** diagonal não atinge limiar *T* (fórmula 2.7) ela é descartada
5: **Senão**
6: **Se** diagonal é consistente com melhor alinhamento dado em na linha 1 (fórmulas 2.8-2.11)
7: Incorpora diagonal no alinhamento
8: **Senão**
9: **Se** diagonal é superior ao melhor alinhamento dado na linha 1
10: Diagonal passa a ser o novo alinhamento
11: Armazena o maior escore de diagonal e alinhamento

Figura 5.14 – Algoritmo para cálculo da recorrência do DIALIGN

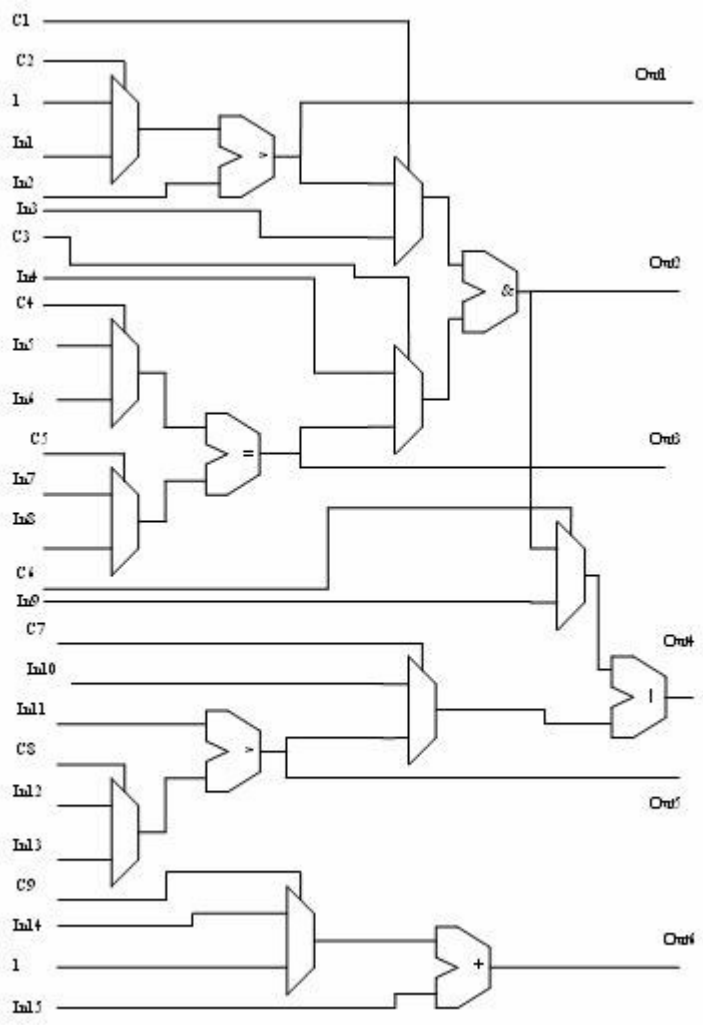

Figura 5.15 – Circuito para as relações de recorrência

Na figura 5.15, é mostrado o circuito projetado para calcular as relações de recorrência em um elemento do vetor wavefront. Todas as entradas do circuito estão do lado esquerdo e são numeradas de “In1” até “In15”. Como o circuito deve fazer operações diferentes em momentos diferentes, são necessárias linhas de controle nos multiplexadores. Além disto, deve fazer operações em paralelo para aumentar o desempenho. As linhas de controle são numeradas de “C1” até “C9”. As saídas do circuito são numeradas de “Out1” até “Out6”.

Na figura 5.15, os blocos indicados por “>” são comparadores capazes de indicar se uma entrada é maior ou menor que a outra. Os blocos indicados por “=” também são comparadores

que verificam se as duas entradas são iguais. Os blocos indicados por "&" e "|" fazem respectivamente o "*and*" e o "*or*" lógico das entradas. Os outros blocos sem sinais são multiplexadores. O circuito também faz as somas necessárias para as relações de recorrência. O bloco que faz a soma é marcado com "+", onde o bloco à esquerda é um multiplexador. Este circuito é capaz de executar todas as relações de recorrência necessárias para o DIALIGN. Para que o circuito funcione adequadamente, é necessário também armazenar os valores calculados em registradores, inclusive os valores temporários.

5.7 ARQUITETURA PARA O CÁLCULO DO ALINHAMENTO DO DIALIGN

A arquitetura proposta na seção 5.6 produz como saída o escore DIALIGN. Na presente seção, é descrito o projeto de uma arquitetura que recupera o alinhamento propriamente dito.

O algoritmo para obtenção do alinhamento, como originalmente definido por [Mor96], utiliza espaço quadrático para guardar as diagonais que serão utilizadas no alinhamento. Em [Mor00], foram feitas modificações para reduzir o espaço utilizado, sendo que o algoritmo pode executar em um espaço proporcional a #*F* onde *F* é o número de fragmentos. Embora o espaço tenha sido bastante reduzido, não há garantias que o algoritmo seja executado em espaço linear. Por exemplo, dependendo do grau de similaridade de duas seqüências de tamanho 20kbp, o valor de *#F* chegou a 661373 [Mor00]. A versão apresentada em [Mor02] contém o mesmo problema.

Para que o alinhamento de duas seqüências com o DIALIGN possa ser feito em FPGA, é importante que o espaço seja linear, independente das seqüências alinhadas, devido às restrições de espaço no FPGA.

Assim, para o projeto da arquitetura, foi necessário também projetar um algoritmo capaz de fazer o alinhamento em espaço linear.

5.7.1 Algoritmo projetado para recuperação do alinhamento no DIALIGN

Nesta seção, é descrita a variante do algoritmo do DIALIGN (seção 2.4.5) que foi projetada de modo a permitir que o alinhamento possa ser recuperado pelo hardware em espaço linear.

5.7.1.1 Visão geral

A idéia principal é a seguinte. Na primeira fase, são calculadas as matrizes *escore* e *prec* do algoritmo original (seção 2.4.5). Além disto, um vetor contendo um elemento para cada coluna armazena o maior escore encontrado nesta coluna, a linha onde o mesmo ocorre, e a linha e coluna onde ocorre o fragmento anterior. Ao término da primeira fase, verifica-se qual é o maior escore global, bem como suas posições na matriz (linha e coluna). Essas posições marcam o final do último fragmento que compõe o alinhamento ótimo. A partir daí, recuperam-se os fragmentos anteriores e compara-se com a posição do fragmento de maior escore encontrado naquela coluna. Se as posições forem as mesmas, o fragmento é adicionado ao alinhamento ótimo. Caso contrário, é feito o re-processamento de parte das matrizes até que seja atingido o final do fragmento que deveria ter sido o último recuperado. Esse procedimento é repetido até que o alinhamento seja totalmente obtido.

A figura 5.16 mostra um exemplo de alinhamento de duas seqüências CCCTTACCGGCCCT e CCCGGACCTACCT utilizando a variante projetada nesta tese para o DIALIGN.

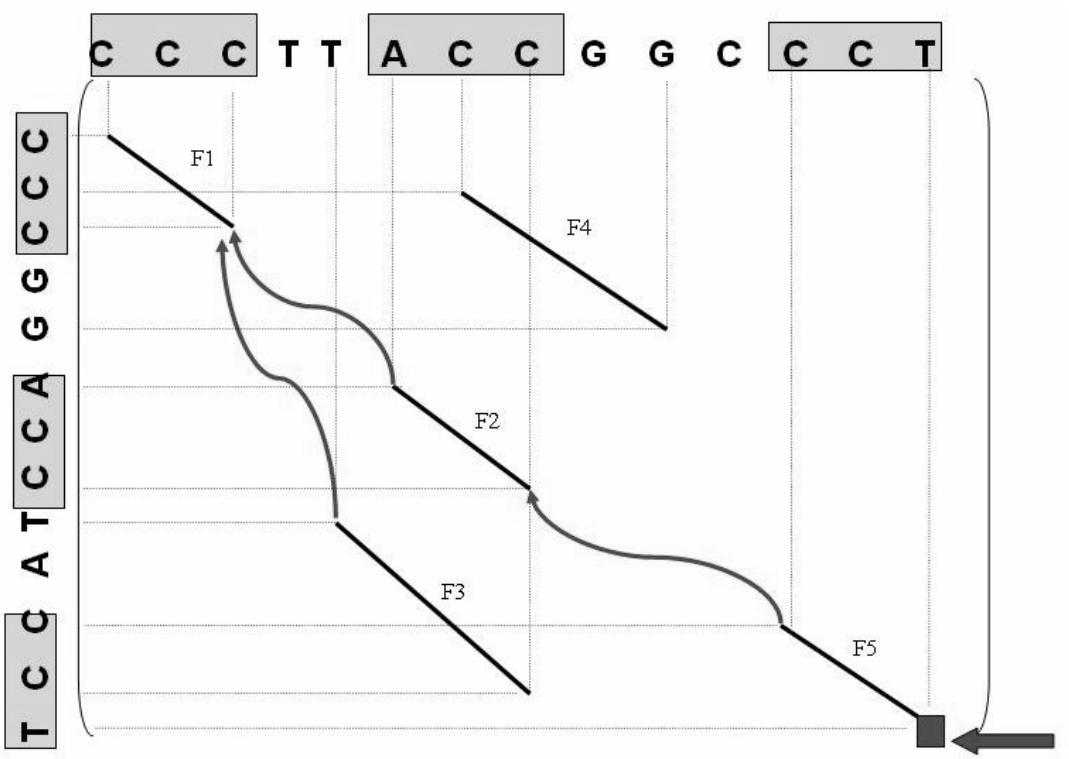

Figura 5.16 - Exemplo de alinhamento utilizando a variante do DIALIGN

Na figura 5.16, os fragmentos são mostrados em linhas pretas e a eles são atribuídos nomes como F1, F2, F3, F4 e F5. Estão indicados na figura o alinhamento ótimo F1-F2-F5 e dois alinhamentos sub-ótimos F1-F3 e F4.

Na última coluna, marcado com um quadrado e uma seta está o fragmento F5 que é o último fragmento do melhor alinhamento. O fragmento F5 possui informação sobre o fragmento anterior que é F2, indicado por uma seta. Na coluna em que termina F2, existe outro fragmento (F3) cujo escore do alinhamento parcial é maior que o escore do alinhamento parcial que termina em F2. O vetor guarda o maior alinhamento parcial na coluna e, portanto, F3. Assim, não é possível recuperar o alinhamento na primeira rodada pois F2 não está na oitava coluna. Na segunda rodada, será feito o alinhamento de duas subseqüências cujo final (linha e coluna) é o mesmo de F2, garantindo que este fragmento estará no alinhamento.

A segunda rodada é mostrada na figura 5.17.

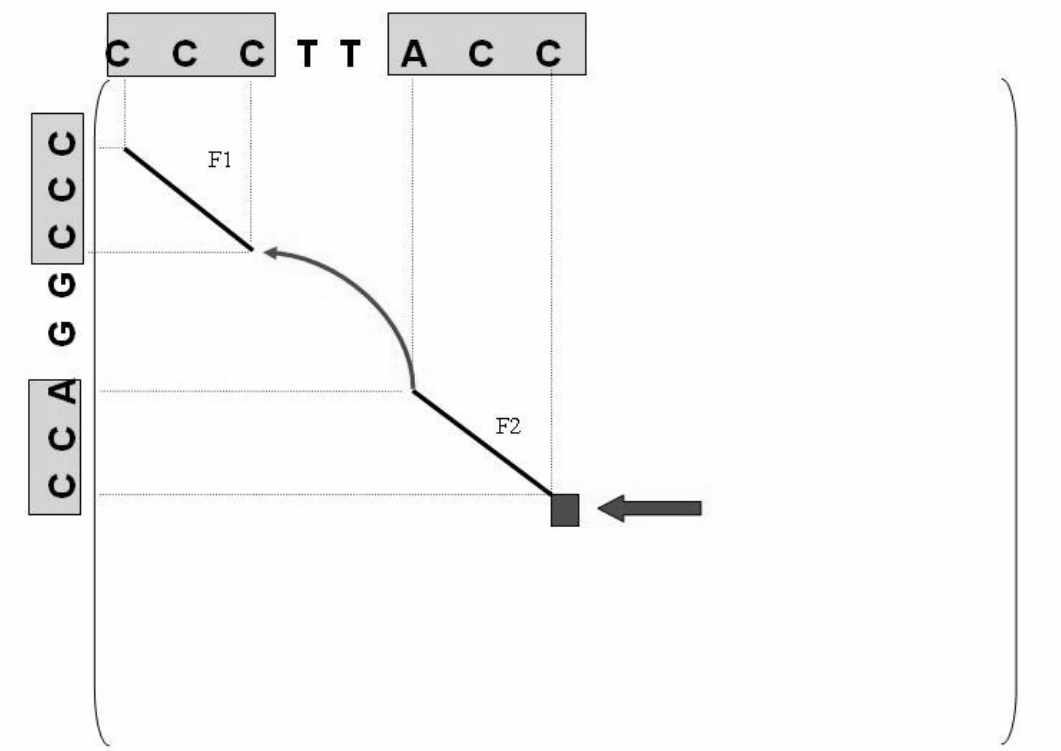

Figura 5.17 - Segunda rodada no alinhamento utilizando a variante do DIALIGN

A figura 5.17 mostra que o alinhamento ótimo das subseqüências termina com F2. O fragmento anterior é F1 que pôde ser recuperado diretamente pois F1 é o melhor subalinhamento na terceira coluna. Como F1 não tem fragmento anterior então ele é o último fragmento. Uma vez recuperados todos os fragmentos, o alinhamento final é dado na figura 5.18.

```
C  C  C  t  t  A  C  C  g  g  c  C  C  T
C  C  C  g  g  A  C  C  t  -  a  C  C  T
```

Figura 5.18 – Alinhamento final

Na figura 5.18, as posições onde há coincidência são indicadas por bases com letras maiúsculas, as substituições são indicadas por ambas as bases estarem com letra minúscula e as inserções e remoções são indicadas por uma letra minúscula e um "-". Como o DIALIGN não contabiliza espaços no seu escore, a posição dos espaços inseridos não é levada em consideração no cálulo do escore. O escore total do alinhamento será 9.

5.7.1.2 Detalhamento do algoritmo

O objetivo da variante do DIALIGN proposta é obter o alinhamento de duas seqüências *A* e *B* de tamanhos *R* e *S* respectivamente. O alinhamento final será o alinhamento de vários fragmentos (seqüência contendo apenas coincidências) que dão o melhor escore em termos do número total de coincidências. Para serem adicionados ao alinhamento, estes fragmentos deverão estar acima de um limiar *T*, definido pelo usuário. O algoritmo é composto de duas matrizes M[*i*,*j*] e N[*i*,*j*] sendo que $0 \leq i \leq R$, $0 \leq j \leq S$ onde *R* e *S* são os tamanhos máximos da linha e coluna respectivamente.

A matriz M corresponde a matriz de escore do DIALIGN original (seção 2.4.5) onde é feito o cálculo dos escores dos fragmentos individuais (seqüência de coincidências). É calculada através de programação dinâmica utilizando a fórmula 5.1:

$$M(i,j) = \begin{cases} M(i-1,j-1) + 1 \text{ , se } A_i = B_j \\ 0, \text{ caso contrário} \end{cases} \qquad (5.1)$$

Os fragmentos são estendidos enquanto o elemento da seqüência *A* na posição *i* coincidir com o elemento da seqüência *B* na posição *j*. O valor 1 é atribuído para cada coincidência.

Os valores das diagonais w(*D*), de acordo com a fórmula 2.7, serão calculados na matriz M com a diferença que os valores abaixo do limiar *T* simplesmente não são utilizados em Cálculo_fase_1, como será detalhado depois. Para o cálculo de *E*(*l*,*m*) (fórmula 2.6), será utilizado $\log_2$ ao invés de ln pois a base dois é mais adequada para representação em computadores digitais. No caso, $l = m$ pois só existem "*matches*" (coincidências) nos fragmentos recuperados por esta versão do algoritmo. Assim, $E(l,m) = -\log_2((0.25)^m) = -\log_2((2^{-2})^m)$ que sempre será um número inteiro que exige menor espaço de representação no FPGA e tem operações aritméticas mais rápidas [Wol04].

A matriz N corresponde à matriz *prec* (fórmula 2.12) no DIALIGN original. Cada posição *i*,*j* na matriz N contém três valores: o maior escore de alinhamento naquela posição, incluindo o último fragmento, linha_anterior e coluna_anterior que indicam a posição onde o penúltimo fragmento do alinhamento foi obtido. Eles são chamados respectivamente de N.escore, N.linha_anterior e N.coluna_anterior.

No primeiro passo, os campos de N(*i,j*) terão seus valores iniciais calculados em hardware através de programação dinâmica segundo a fórmula 5.2:

$$N(i,j) = \textbf{Max} \begin{cases} N(i\text{-}1,j\text{-}1).\text{escore} \\ N(i\text{-}1,j).\text{escore} \\ N(i,j\text{-}1).\text{escore} \end{cases} \qquad (5.2)$$

No cálculo inicial do valor de N(*i,j*), este terá seus três campos de valores equivalentes aos três campos de valores da posição entre N(*i*-1,*j*-1), N(*i*-1,*j*), N(*i,j*-1) que tiver o maior campo escore. Por exemplo, se N(*i*-1,*j*-1).escore for o maior em (5.2) então N(*i,j*).escore = N(*i*-1,*j*-1).escore, N(*i,j*).linha_anterior = N(*i*-1,*j*-1).linha_anterior e N(*i,j*).coluna_anterior = N(*i*-1,*j*-1). coluna_anterior. Em caso de igualdade de escore, os valores que estão em N(*i,j*-1) tem preferência sobre N(*i*-1,*j*) e este sobre N(*i*-1,*j*-1) como em [Mor96]. O valor de cada posição N(*i,j*) pode mudar posteriormente no algoritmo.

Na variante proposta, o valor de $\sigma(D_{i,j})$ (fórmula 2.11) é calculado e colocado em N(*i,j*).escore. O fragmento precedente ao atual prec(*i,j*) (fórmula 2.12) é indicado através dos campos N[*i,j*].linha_anterior e N[*i,j*].coluna_anterior.

Além disto, são calculados quatro vetores de tamanho *S* que correspondem ao número de colunas. O vetor Maior_Escore, com posições de 1 até *S* =, armazena o maior escore obtido em cada coluna bem como a coluna do último fragmento do alinhamento. O vetor Linha_Maior_Escore armazena a linha onde foi obtido o último fragmento adicionado ao alinhamento parcial de maior escore. O vetor Coluna_Frag_Anterior armazena a coluna do penúltimo fragmento adicionado ao alinhamento. O vetor Linha_Frag_Anterior armazena a

linha do penúltimo fragmento do alinhamento. Para recuperar o alinhamento, é necessário recuperar todos os fragmentos que o compõem bem como suas posições e seus escores. O conteúdo dos vetores está resumido na tabela 5.1.

Tabela 5.1 Conteúdo dos vetores

Nome	Conteúdo
j	coluna onde termina um fragmento
Maior_Escore[*j*]	maior escore de alinhamento cujo último fragmento termina na coluna *j*
Linha_Maior_Escore[*j*]	linha do último fragmento do alinhamento na coluna *j*
Coluna_Frag_Anterior[*j*]	coluna do fragmento anterior ao que termina em *j*
Linha_Frag_Anterior[*j*]	linha do fragmento anterior ao que termina em *j*

O cálculo dos vetores para cada posição *i* e *j* é feito conforme o algoritmo da figura 5.19:

```
Cálculo_fase_1 (Seqüência A, Seqüência B, Tam_Linha, Tam_Coluna, Limiar  T)
1: Inicializa todas as posições de M, N e dos vetores com zero
2:  Para todo  i
3:      Para todo j
4:          Calcula  M(i,j) usando (5.1)
5:          Calcula  N(i,j).escore, N(i,j).linha_anterior, N(i,j).coluna_anterior  usando (5.2)
6:          linha_anterior = N(i,j).linha_anterior
7:          coluna_anterior = N(i,j).coluna_anterior
8:           Se M(i,j)=0        /* Não houve coincidência */
9:               Se M(i-1,j-1) ≥ T   /* Teste de relevância */
10:                      Então
11:                       Se Consistente(N(i,j),M(i-1,j-1))  /* Teste de Consistência */
12:                           N(i,j).linha_anterior = i
13:                           N(i,j).coluna_anterior = j
14:                           N(i,j).escore = N(i,j).escore + M(i-1,j-1)
15:                           adicionou_fragmento = verdadeiro
15:                       Senão
16:                            Se M(i-1,j-1) > N(i,j)
17:                               N(i,j).linha_anterior = i
18:                               N(i,j).coluna_anterior = j
19:                               N(i,j).escore = M(i-1,j-1)
20:                               linha_anterior=0
21:                               coluna_anterior=0
22:                               adicionou_fragmento = verdadeiro
23:          Se (adicionou_fragmento = verdadeiro ) e (N(i,j).escore > Maior_Escore[j])
24:              Maior_Escore[j] = Escore em N(i,j)
25:              Linha_Maior_Escore[j] = i
26:              Coluna_Frag_Anterior[j] =  coluna_anterior
27:              Linha_Frag_Anterior[j] =  linha_anterior
28:      Fim Para todo j
29: Fim Para todo  i
30:  Retorna Maior_Escore, Linha_Maior_Escore, Coluna_Frag_Anterior Linha_Frag_Anterior
```

Figura 5.19 – Primeira fase da variante do DIALIGN

O algoritmo começa inicializando as matrizes M e N. A seguir, para cada posição *i* e *j* serão calculados os valores de M[*i*,*j*] e N[*i*,*j*]. Os valores iniciais de N[*i*,*j*] são calculados de acordo com a fórmula 5.2 mas podem ter seus valores mudados depois. Se M(*i*,*j*)=0 (linha 8) significa que um fragmento termina na coluna *j*. Resta decidir se ele deve fazer parte do alinhamento ou não. Isto é feito através do teste de relevância utilizando o limiar *T* (linha 9).

Ultrapassado o limiar, é necessário verificar se o fragmento pode ser colocado no alinhamento. Isto é feito com o teste de consistência (linha 11). São adicionados apenas fragmentos que são consistentes, ou seja, se uma posição (*i,j*) pertence a uma diagonal D_y que está sendo adicionada então para qualquer diagonal D_x que já está no alinhamento não existe nela posição (*k,l*) tal que *k*=*i* ou *l*=*j*. Assim, um elemento de uma seqüência não pode ser alinhado mais de uma vez no mesmo alinhamento.

Existe ainda a possibilidade do fragmento atual ser maior que o alinhamento até a posição (linha 16). Neste caso, o próprio fragmento se torna o alinhamento. Os valores de Maior_Escore[*j*], Linha_Maior_Escore[*j*], Coluna_Frag_Anterior[*j*] e Linha_Frag_Anterior[*j*] mudam apenas se um novo fragmento é adicionado na coluna *j* e se o valor do alinhamento na coluna é maior que os anteriores (linha 23).

Cada posição *j* do vetor Maior_Escore contém a soma de todas as diagonais alinhadas até aquela posição correspondendo à soma das diagonais alinhadas até a coluna *j* (fórmula 2.8). Assim, em Maior_Escore estão contidos os valores de vários alinhamentos parciais bem como o alinhamento ótimo segundo o algoritmo. Maior_Escore juntamente com Linha_Maior_Escore permite saber a linha e coluna *i* e *j* da última diagonal do alinhamento $D_{i,j}$. As coordenadas de *prec*(*i,j*) (fórmula 2.12) estão nos vetores Coluna_Frag_Anterior e Linha_Frag_Anterior.

Podem acontecer casos onde o alinhamento completo pode ser obtido desta forma em uma única rodada. O algoritmo proposto, diferentemente de [Mor98], foi projetado para funcionar em espaço linear. Logo, não é armazenada a matriz quadrada contendo todas as informações

para recuperar o alinhamento em uma única passada. Assim, podem ser necessárias mais de uma rodada para recuperar todo o alinhamento. O vetor Maior_Escore, com o maior escore obtido na coluna, garante que se encontrará o escore final do alinhamento bem como o último fragmento deste alinhamento mas não garante que o fragmento anterior estará armazenado nele. Existe a possibilidade de um fragmento de um alinhamento sub-ótimo ser colocado no vetor. Assim, ao final de uma rodada, tem-se certeza do último fragmento e do penúltimo mas não necessariamente do antepenúltimo.

O algoritmo ilustrado na figura 5.20 recebe como parâmetros as duas seqüências *A* e *B*, seus tamanhos *R* e *S* e o limiar *T* e gera como saída um vetor Alinhamento_Final de tamanho *S*, contendo o alinhamento das seqüências. Cada posição do vetor Alinhamento_Final possui três campos: o maior escore de alinhamento, a linha e a coluna onde eles foram obtidos. Eles são chamados respectivamente de Alinhamento_Final.escore, Alinhamento_Final.linha e Alinhamento_Final.coluna.

```
Cálculo_alinhamento(A,B,R,S,T)

1:   Inicializa Alinhamento_Final.escore,  Alinhamento_Final.linha_anterior,
     Alinhamento_Final.coluna_anterior
2:   Cálculo_fase_1 (A,B,R,S,T) retornando Maior_Escore, Linha_Maior_Escore, Coluna_Frag_Anterior
     e Linha_Frag_Anterior
3:   Escore_parcial = 0
4:   Escore_final =  Maior_valor(Maior_Escore)
5:   a = Posição_maior_valor(Maior_Escore)
6:   b = Coluna_Frag_Anterior[a]
7:   Enquanto Escore_parcial < Escore_final   /*Recupera alinhamento*/
8:      Enquanto Linha_Frag_Anterior[a]  = Linha_Maior_Escore[b]  /*Recupera fragmentos da rodada*/
9:         Escore_parcial = Escore_parcial+  Maior_Escore[a]
           /* Adiciona fragmento da posição a em Alinhamento_Final */
10:        Alinhamento_Final[a].escore =  Escore_final
11:        Alinhamento_Final[a].coluna =  a
12:        Alinhamento_Final[a].linha =  Linha_Maior_Escore[a]
13:        a = b
14:        b = Coluna_Frag_Anterior[a]
15:     Fim Enquanto
16:     Cálculo_fase_1 (A,B,Linha_Maior_Escore[a], a, T)
17:     a = posição de maior valor em Maior_Escore
18:     b = Coluna_Frag_Anterior[a]
19:  Fim Enquanto
20:  Retorna Alinhamento_final
```

Figura 5.20 – Obtenção do alinhamento com a variante do DIALIGN

Primeiramente, é inicializado o vetor Alinhamento_Final com o valor zero em todos os seus campos e posições. A seguir, na linha 2, é feito o cálculo inicial dos vetores Maior_Escore, Linha_Maior_Escore, Coluna_Frag_Anterior e Linha_Frag_Anterior segundo o Cálculo_fase_1 (figura 5.19).

Após o cálculo dos vetores, procede-se a fase de recuperação do alinhamento. Assim, o primeiro passo é descobrir qual o maior escore presente em Maior_Escore[*S*] e sua posição *a* no vetor. A posição *a* é então a coluna onde ocorre o maior escore gerado pelo algoritmo e é também a posição do último fragmento incluído no alinhamento.

A variável Escore_parcial contém o escore dos fragmentos recuperados até o momento e é inicializada com zero (linha 3). Escore_final contém maior escore presente no vetor Maior_Escore que é o escore total do alinhamento (linha 4). Assim, o processo de recuperar os fragmentos continuará até que o total do escore dos fragmentos recuperados seja igual ao escore do alinhamento total, o que corresponde a recuperação total do alinhamento (linha 7).

Em Linha_Maior_Escore, na posição [*a*] está a linha onde termina o último fragmento do alinhamento. Nos vetores Coluna_Frag_Anterior e Linha_Frag_Anterior na posição *a* estão respectivamente a coluna e a linha do fragmento anterior no alinhamento, ou seja, o penúltimo fragmento. Em Coluna_Frag_Anterior[*a*] será obtida a nova coluna *b*. Assim, esta coluna *b* servirá para obter novos valores de Linha_Maior_Escore[*b*], Coluna_Frag_Anterior[*b*] e Linha_Frag_Anterior[*b*].

Para saber se o fragmento anterior deverá ser incluído, é feito o seguinte procedimento: Descobrir *b*=Coluna_Frag_Anterior[*a*] (linha 6). Se Linha_Frag_Anterior[*a*]= Linha_Maior_Escore[*b*] (linha 8) então o fragmento anterior pode ser adicionado, pois o fragmento anterior ao atual é realmente o que ficou armazenado no vetor. Caso contrário, é necessária mais uma rodada onde a linha e coluna final são respectivamente Linha_Maior_Escore[*a*] e *a*. O laço de repetição termina quando Linha_Frag_Anterior[*a*] é diferente de Linha_Maior_Escore[*b*], ou seja, nenhum outro fragmento pode ser recuperado nesta rodada. Se todos os fragmentos do alinhamento final ainda não foram recuperados, então

o Cálculo_fase_1 é executado novamente (linha 16) e desta vez o cálculo dos vetores só ocorrerá até a Linha_Maior_Escore[*a*] e a coluna *a*, que são as coordenadas do último fragmento do alinhamento. O algoritmo executará até que todos os fragmentos do tenham sido recuperados, ou seja, Escore_parcial = Escore_final (linha 7).

5.7.2 Circuito para recuperação do alinhamento no DIALIGN

A figura 5.21 mostra o vetor wavefront projetado para a recuperação para a recuperação do alinhamento em espaço linear, de acordo com o algoritmo descrito na seção 5.7.1.

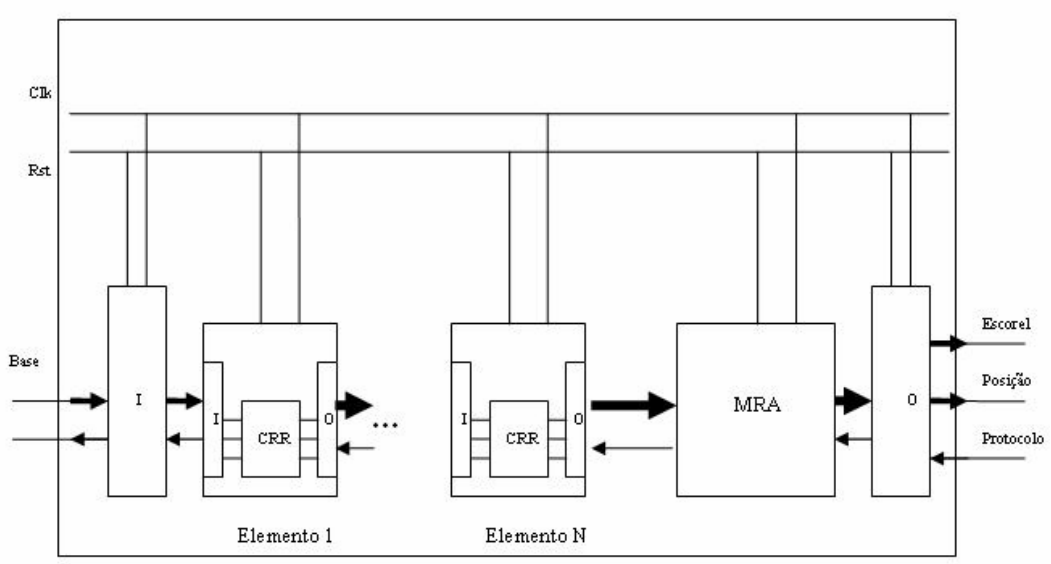

Figura 5.21 Arquitetura projetada para recuperar o alinhamento com a variante do DIALIGN

A arquitetura na figura 5.21 é parecida com a arquitetura mostrada na seção 5.3, tendo o mesmo protocolo e possuindo N elementos. A principal diferença é que, depois do último elemento, foi inserido um circuito chamado de Módulo para Recuperação do Alinhamento (MRA). Este circuito possui uma máquina de estados semelhante ao elemento (figura 5.5). Durante a fase de cálculo, ele simplesmente obtém a sua entrada e passa para a saída. Durante a fase onde os resultados são recuperados, ele recupera os dados dos resultados e repassa para o circuito, com a diferença de que estes dados são utilizados para calcular o maior escore obtido e, a partir dele, a posição do último fragmento que pôde ser recuperado no alinhamento. Esta informação será utilizada na próxima fase de configuração, onde a comparação das seqüências será executada novamente. Desta vez só serão comparadas as subseqüências que

recuperarão os fragmentos restantes do alinhamento, conforme mostrado o algoritmo mostrado na figura 5.20.

Dentro do CRR serão executadas as relações de recorrência conforme o algoritmo da figura 5.19. O algoritmo em hardware que é executado no MRA, para recuperação do alinhamento é dado na figura 5.22.

```
Recupera Alinhamento
1:  Recebe resultados
2:  Incrementa contador
3:  Se contador<=tamanho_vetor
4:      Se escore_dialign > maior_escore
5:        Coluna_final = tamanho_vetor – contador
6:        Linha_final = linha do fragmento atual
7:        Posição_fragmento_anterior =   precedente do fragmento_atual
8:      Senão
9:        Se fragmento atual==  Posição_fragmento_anterior
10:           Coluna_final = tamanho_vetor – contador
11:           Linha_final = linha do fragmento atual
12:           Posição_fragmento_anterior =   precedente do fragmento_atual
13:    Passa resultado
14: Senão
15:     Passa Coluna_final e Linha_final
```

Figura 5.22 - Algoritmo projetado para recuperar o alinhamento na arquitetura

A figura 5.22 mostra o algoritmo no módulo MRA. Os valores de contador, linha_final, coluna_final, maior_escore, posição_fragmento_anterior são inicializados com zero durante a fase de configuração. Durante a fase de recuperação dos resultados, o MRA recebe os resultados do elemento à esquerda a cada iteração (linha 1). Um contador é utilizado para saber quantos resultados já foram processados e qual era a posição no resultado no vetor (linha 2). Se o contador for menor ou igual ao número de elementos no vetor, então o resultado é válido. A seguir, descobre-se se o escore obtido no alinhamento parcial é maior que maior_escore (linha 4). Se for este o caso, então ele se torna o novo final de alinhamento (linhas 5, 6 e 7). Se o resultado atual não é o maior escore de alinhamento então ele pode ser um subalinhamento que faz parte do alinhamento ótimo. Verifica-se se o fragmento precedente do fragmento anterior é o mesmo do fragmento do resultado atual (linha 9). Se isto for verdade, então o fragmento atual faz parte do alinhamento ótimo e a sua linha e coluna será a linha_final e coluna_final obtida até o momento. Se os valores forem ambos zero, a saída foi o alinhamento completo. Caso contrário, os valores de linha_final e coluna_final serão

utilizados na próxima fase de configuração para indicar as subseqüências que serão utilizadas na próxima rodada do alinhamento.

O algoritmo processa os resultados de todos os elementos. Quando o contador for maior que o número de elementos no vetor então a saída do MRA é a posição do último fragmento recuperado do alinhamento. A figura 5.23 apresenta o circuito implementado no MRA.

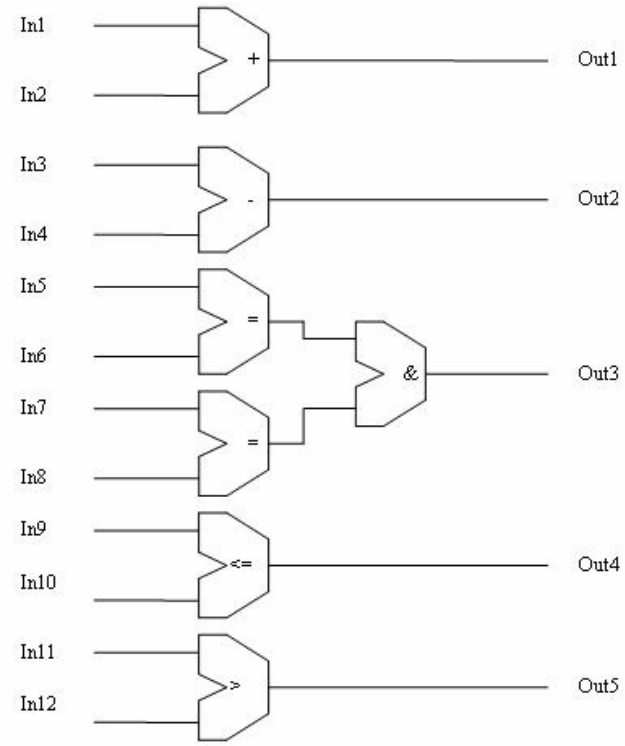

Figura 5.23 - Circuito de recuperação do alinhamento

Na figura 5.23, o circuito tem suas entradas nomeadas de In1 a In12 e as saídas nomeadas de Out1 a Out5. O circuito é capaz de realizar as operações do algoritmo da figura 5.22. O componente marcado com "+" é um somador e incrementa o valor do contador. O componente marcado com "-" é um subtrator e é utilizado subtrair o valor do contador de tamanho_vetor. Os comparadores "=", ">" e "<=" foram utilizados nas comparações. Os dois comparadores "=" ligados por um *and* lógico são utilizados para verificar se a linha e a coluna do fragmento atual são de um fragmento que faz parte do alinhamento.

Após a execução do alinhamento em hardware, o resultado obtido é o conjunto de vetores mostrados na tabela 5.1 e estes vetores possuem tamanho *n*, onde *n* é o tamanho do vetor

wavefront. Nestes vetores estão todas as informações sobre o alinhamento. O alinhamento pode ser impresso rapidamente em software em tempo linear O(*n*). O algoritmo para a impressão do alinhamento é mostrado na figura 5.24.

```
Imprime Alinhamento
1:  F = fragmento da posição de maior escore
2:  Faça
3:     Imprime elementos até o fragmento F
4:     Imprime os elementos que compõe o  fragmento F
5:     F = fragmento anterior a F
6:  Enquanto F possuir fragmento anterior válido
7:  Imprime elementos que ocorrem depois de F
```

Figura 5.24 – Impressão do alinhamento em software

O primeiro passo é descobrir onde está o maior escore do vetor (linha 1). Como o alinhamento já foi descoberto em hardware, todos os fragmentos podem ser recuperados de uma só vez. Dentro do laço de repetição, é feita a impressão dos elementos das seqüências até o fragmento F (linha 3). Estes elementos são impressos como um sub-alinhamento. Como estes elementos estão fora dos fragmentos, eles podem ser alinhados de diversas formas pois não são considerados para o cálculo do escore do DIALIGN [Mor96]. A seguir, é feita a impressão do fragmento (linha 4). Ela é realizada mostrando os elementos das duas seqüências que compõe os *matches* no fragmento. A seguir, encontra-se o fragmento anterior (linha 5). O laço é repetido enquanto houver fragmentos a serem colocados no alinhamento (linha 6). Finalmente, os elementos restantes são impressos (linha 7).

6 – RESULTADOS EXPERIMENTAIS

Neste capítulo são apresentados os resultados experimentais obtidos com as 3 arquiteturas propostas nesta tese. Para cada arquitetura, são apresentados inicialmente dados referentes à síntese para FPGA tais como o número de elementos de processamento, freqüência de *clock* e utilização dos recursos do FPGA. A seguir, são discutidos os tempos de execução obtidos bem como os *speedups* alcançados quando comparados com um programa confeccionado em linguagem C executando em um Pentium 4 GHz. Os programas em C também foram desenvolvidos nesta tese e implementam em software o algoritmo proposto para hardware.

Os programas utilizados para a síntese foram o ISE 8.1 [Xil08] e o Quartus II [Alt08] utilizados para os FPGAs da Xilinx e da Altera, respectivamente. Os programas de síntese foram executados em um Pentium 4 3 GHz .

Todas as arquiteturas sintetizadas aqui utilizaram a arquitetura de base (seção 5.3). A figura 6.1 ilustra um elemento sintetizado com seus sinais de entrada e saída de acordo com o código especificado na seção 5.3.1 utilizando o ISE 8.1.

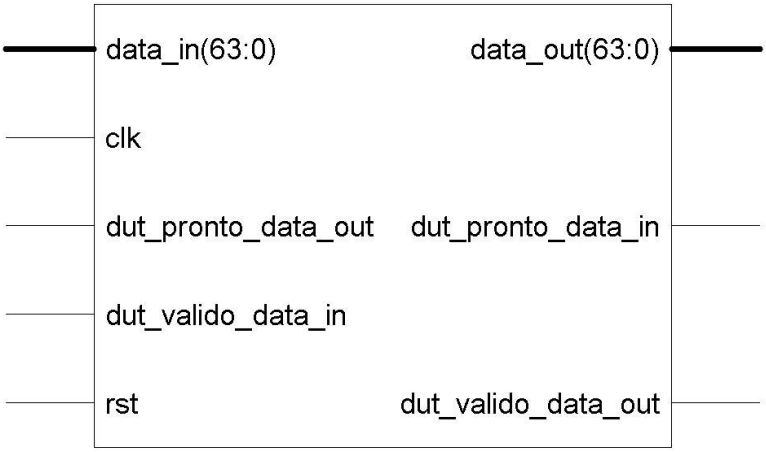

Figura 6.1 – Bloco sintetizado para o elemento

Na figura 6.1, vemos as entradas do elemento que estão do lado esquerdo e as saídas do elemento que estão do lado direto. A linhas finas indicam sinais de um único bit. As duas linhas grossas data_in e data_out possuem 64 bits e seu tamanho pode variar dependendo do tamanho do circuito.

6.1 ARQUITETURA WAVEFRONT PARA O SMITH-WATERMAN

6.1.1 Síntese da arquitetura

A arquitetura proposta para o Smith-Waterman (seção 5.5) foi sintetizada para ao FPGA Xilinx xc2vp70-6ff1517 utilizando o ISE 8.1. A seguir serão mostradas as saídas geradas pelo simulador para diversos tamanhos de vetores wavefront.

Após a síntese, foram obtidos os resultados mostrados na tabela 6.1.

Tabela 6.1 – Síntese no FPGA Xilinx xc2vp70

Elementos	Slices	Flip-flops	LUTS	IOBs	GCLKs	Freq.
20	14%	5%	13%	7%	1	181,4
100	69%	25%	65%	7%	1	174,7

Na primeira coluna é ilustrado o número de elementos que o vetor wavefront possui. Na segunda coluna, é apresentado o percentual utilizado de *Slices*. Cada *Slice* contém 2 LUTS e 2 registradores de 1 bit e multiplexadores para fazer a interligação. O número de *Slices* é limitado pelos recursos do FPGA que neste caso possui 33088 destes.

Na terceira coluna, pode-se ver o percentual utilizado de *Slice Flip Flops.* Os *Slice Flip Flops* permitem que o FPGA armazene bits em registradores. O FPGA tem 66176 *Slice Flip Flops*.

A quarta coluna apresenta o percentual utilizado de *LUTs*. As *LUTs* permitem escolher uma função lógica e tem 4 entradas. O FPGA possui 66176 *LUTs*.

Na quinta coluna temos o percentual utilizado de *IOBs*. Os *IOBs* (*Input Output Blocks*) são responsáveis pela entrada e saída servindo como armazenamento temporário para entrada e saída, caso necessário. O FPGA possui 964 *IOBs*.

A sexta coluna contém o número de *GCLKs* utilizados. Os *GCLKs* são os *clocks* globais disponíveis. No projeto, foi utilizado apenas um *clock* global dos 16 disponíveis.

Na sétima coluna, temos a velocidade máxima que o circuito executa. Esta freqüência é dada em MHz.

Na tabela 6.1 pode ser visto que para o caso do cálculo do escore do Smith-Waterman, a utilização dos recursos do FPGA é proporcional ao número de elementos sintetizados. Pode ser observado que quanto maior o número de elementos, menor a freqüência de operação do FPGA, o que está de acordo como o esperado. O número de *IOBs* (*Input Output Block*) é sempre o mesmo pois o número de entradas e saídas para o vetor não muda quando o número de elementos é alterado. Também pode ser visto na tabela 6.1 que, para 100 elementos no vetor wavefront, menos de 70% do FPGA foi utilizado.

Na figura 6.2 é mostrado o circuito do *DUT* sintetizado com 20 elementos para o cálculo do escore do Smith-Waterman utilizando o ISE 8.1. Os elementos do vetor aparecem como blocos conectados na horizontal.

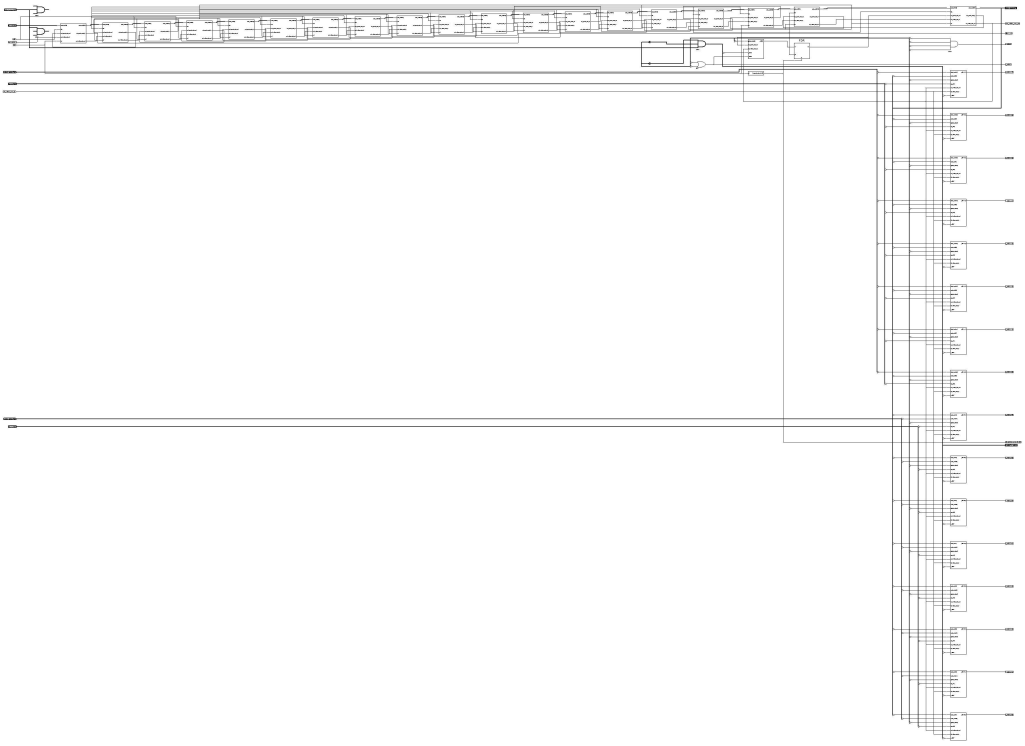

Figura 6.2 – Circuito de 20 elementos no Xilinx xc2vp70

6.1.2 Tempos e *speedups*

Para os testes, foi utilizada uma seqüência de tamanho 100 bp (*base pairs*), que foi comparada com um banco de dados de 10Mbp. Neste caso, o FPGA (Xilinx xc2vp70) demorou 0,744 segundos para calcular uma matriz de similaridade de tamanho 100bpx10.000.000bp bem como a posição e valor do melhor escore encontrado. Um programa otimizado em C implementando o mesmo algoritmo, executado em um Pentium 4 3 GHz, levou 183,72 segundos. Isto significa um *speedup* de 246,9. Nestes testes, para uniformizar os resultados, somente o tempo de CPU é considerado, ou seja, o tempo de entrada e saída bem como o tempo de leitura dos arquivos com as seqüências não é considerado.

Neste caso, a aceleração do Smith-Waterman feita pelo vetor wavefront apresentado na seção 5.3, utilizando as relações de recorrência da seção 2.4.2, conseguiu um *speedup* de 246,9 sobre um programa em software equivalente. Depois da computação, os resultados a serem transmitidos pela placa de FPGA para o computador hospedeiro são apenas alguns poucos bytes com os escores e posições, o que pode ser feito em poucos milisegundos pelo

barramento PCI, por exemplo. Assim, a arquitetura conseguiu fazer a computação dos resultados de forma eficiente e evitando as limitações de comunicação do FPGA [Bou07a].

6.2 ARQUITETURA WAVEFRONT PARA RECUPERAÇÃO DO ESCORE NO DIALIGN

6.2.1 Síntese da arquitetura

A arquitetura proposta na seção 5.6 foi sintetizada para 3 FPGAs: Xilinx xc2vp70, Altera STRATIX 2 EP2S15F672I4 e Altera STRATIX 2 EP2S180F1508I4.

Na síntese feita para o Xilinx xc2vp70, foram obtidos os resultados mostrados na tabela 6.2.

Tabela 6.2 – Síntese no FPGA Xilinx xc2vp70

Elementos	Slices	Flip-flops	LUTS	IOBs	GCLKs	Freq.
12	99%	36%	88%	60%	1	132,540MHz

Como pode ser visto na tabela 6.2, só conseguiu-se colocar 12 elementos de processamento no FPGA , o que ocupou 99% dos *slices*.

Foi feita também a síntese para o Altera STRATIX 2 EP2S15F672I4. Na figura 6.3 é apresentado um circuito gerado para o Altera STRATIX 2 EP2S15F672I4 com 16 elementos utilizando o Quartus II.

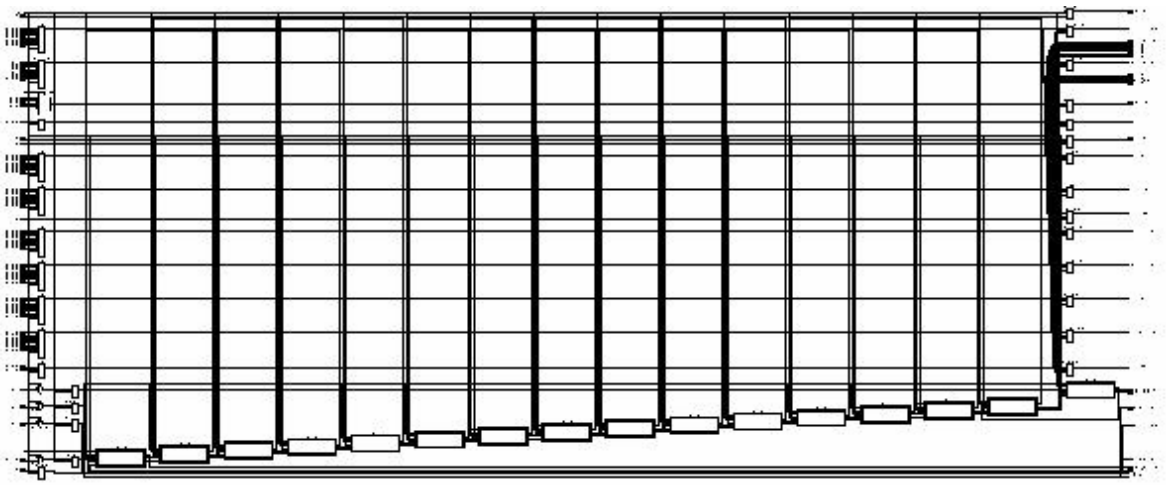

Figura 6.3 – Circuito com 16 elementos no Altera STRATIX 2 EP2S15F672I4

O resultado da síntese com 20 elementos no Altera STRATIX 2 EP2S15F672I4 é mostrado na tabela 6.3.

Tabela 6.3 – Síntese no Stratix 2 EP2S15F672I4

Elementos	Total	ALUTs	Registradores	Pinos	Freqüência
20	84%	57%	50%	46 %	161,21 MHz

Pode ser visto na síntese no Xilinx xc2vp70 e no Altera STRATIX 2 que poucos elementos puderam ser colocados no vetor wavefront, quando comparado com o algoritmo de Smith-Waterman (tabelas 6.2 e 6.3). Isto ocorre porque as relações de recorrência do DIALIGN são mais complexas que as do Smith-Waterman (seção 5.7), exigindo mais recursos do FPGA. Assim, justifica-se a utilização de FPGAs maiores como por exemplo o Altera STRATIX 2 EP2S180F1508I4. Os resultados são mostrados na tabela 6.4.

Tabela 6.4 – Síntese no Stratix 2 EP2S180F1508I4

Elementos	ALUTs	Registradores	Pinos	Freqüência
50	19%	15%	14%	132,84 MHz
100	39%	31%	14%	116,40 MHz
200	77%	62%	14%	74,48 MHz

Na tabela 6.4 podemos ver que a utilização de ALUTS é aproximadamente proporcional ao número de elementos no vetor. Vemos também que a freqüência de operação caiu bastante como o aumento do número de elementos.

6.2.2 Tempos e *speedups*

O FPGA Xilinx xc2vp70 com 12 elementos funcionando a 132,54 MHz é capaz de comparar duas seqüências de tamanhos 12bp e 10Mbp em 0,96 segundos. Um algoritmo em C executou em 33,75 segundos levando em consideração apenas o tempo de CPU. Assim o *speedup* foi de 34,44.

O FPGA Altera STRATIX 2 com 20 elementos funcionando a 161,21 MHz é capaz de comparar duas seqüências de tamanhos 20bp e 10Mbp em 0.81 segundos. O mesmo algoritmo em C para seqüências do mesmo tamanho executou em 55,11 segundos levando em consideração apenas o tempo de CPU. Assim o "*speedup*" foi de 68,37.

Comparando o xc2vp70 e o STRATIX 2 em termos de poder de processamento dado pelo número de elementos possíveis multiplicado pela freqüência de operação, o STRATIX 2 tem praticamente o dobro do poder de processamento do xc2vp70.

A tabela 6.5 mostra os resultados para o Altera STRATIX 2 EP2S180F1508I4 utilizando 50, 100 e 200 elementos. Seqüências sintéticas de tamanhos 50bp, 100bp e 200bp são comparadas com uma seqüência também sintética (geradas aleatoriamente) de tamanho 10MBbp. O objetivo é verificar o desempenho comparativo destas arquiteturas.

Tabela 6.5 – Comparação de arquiteturas de tamanhos diferentes

Seq Proc	Seq Base	Tempo FPGA	Tempo Software	Speed-Up
50	10 MB	0,978 s	139,8 s	142,9
100	10 MB	1,17 s	312,32 s	266,9
200	10 MB	1,74 s	661,39 s	343,0

Pode ser visto na tabela 6.5 que, quanto maior o número de elementos, maior o *speedup*, mesmo com a diminuição da freqüência de *clock* do FPGA vista na tabela 6.4. Para a arquitetura com 200 elementos, o *speedup* obtido sobre a arquitetura de 100 elementos foi superior em 28,5% apesar da grande redução da freqüência de operação.

A arquitetura foi projetada para particionar a seqüência no vetor wavefront (seção 5.7). Assim, foram comparadas seqüências reais de DNA com tamanhos maiores que o da arquitetura. As seqüências utilizadas foram retiradas do NCBI [Ncb08] e são: *Aspergillus niger contig An18c0160* (AM270408), *Aspergillus niger contig An16c0230* (AM270375) e *Encephalitozoon cuniculi* (AL590443), com tamanhos 121589bp, 169786bp e 194439bp respectivamente. Foram utilizadas também duas seqüências de Anellovirus (NC_009225 e AB290918) com tamanhos 3245 bp e 3242 bp respectivamente.

Os resultados obtidos nas comparações com o programa em C são mostrados na tabela 6.6.

Tabela 6.6 – Comparação das seqüências

Seq Proc(bp)	Seq Base(bp)	Tempo FPGA	Tempo Software	*Speedup*
169786	194439	28,83 s	11053,70 s	383,41
121589	169786	17,9 s	6812,00 s	380,55
3245	3242	0,01 s	3,48 s	348,00

Pode ser visto na tabela 6.6 que os maiores *speedups* foram obtidos para as maiores seqüências comparadas. Além disto, para as maiores seqüências, o tempo em software se torna muito grande. Na primeira linha da tabela, o tempo em software é de mais de 3 horas enquanto o tempo em FPGA é de menos de 30 segundos, justificando portanto sua utilização.

Para mostrar um exemplo da aceleração da fase de comparação de seqüências duas a duas, utilizaremos as seguintes seqüências de DNA retiradas do NCBI e que são variantes do adenovirus humano: NC_004001, NC_001405, NC_002067 e NC_003266 com tamanhos 34794bp, 35937bp, 35100bp e 35994bp respectivamente. Os resultados são mostrados na tabela 6.7.

Tabela 6.7 – Comparação das seqüências de Adenovirus

	Tempo (segundos) FPGA / Software / *Speedup*		
Seqüências	34794bp	35937bp	35100bp
35937bp	1,09/ 415,31/ 381,0	---	---
35100bp	1,07/ 406,81/ 380,2	1,11/ 419,35/ 377,8	---
35994bp	1,10/ 416,33/ 378,5	1,14/ 431,42/ 378,4	1,11/ 420,12/ 378,5

Pode ser visto na tabela 6.7 que a faixa de *speedups* obtidos foi entre 377,8 e 381, indicando que o desempenho é consistente para esta faixa de tamanhos de seqüências.

O FPGA STRATIX 2 EP2S180F1508I4 tem um custo estimado em 10.688 dólares. Comparando com um Pentium 4 3 GHz com um custo de 1.000 dólares, a razão entre o preço e o desempenho (no melhor caso) é 10688/ 383,41 = 27,87 contra 1000/1 para o Pentium. Assim a razão preço/desempenho do FPGA é cerca de 35,88 vezes melhor que o do Pentium.

Assim, a arquitetura projetada mostrou um grande desempenho quando comparada com a opção em software, além de possuir um excelente custo/benefício [Bou07b].

6.3 ARQUITETURA WAVEFRONT PARA RECUPERAÇÃO DO ALINHAMENTO NO DIALIGN

6.3.1 Síntese da arquitetura

A arquitetura para recuperação do alinhamento do DIALIGN é ainda mais complexa do que a arquitetura utilizada para recuperar os escores (seção 5.7). Assim, esta arquitetura foi sintetizada diretamente no FPGA de maior capacidade, o Altera STRATIX 2 EP2S180F1508I4. Além disto, foram feitas várias otimizações no código visando diminuir o tamanho do vetor sintetizado e aumentar a freqüência de operação do FPGA. Isto foi feito aumentando o número de ciclos necessários para o cálculo das relações de recorrência.

A arquitetura para recuperar o alinhamento do DIALIGN em espaço linear foi feita em SystemC [Sys05] e depois traduzida para o Verilog com o Forte [For05]. O circuito foi sintetizado para o FPGA Altera STRATIX 2 EP2S180F1508I4 utilizando o QUARTUS II. O resultado da síntese é mostrado na Tabela 6.8.

Tabela 6.8 – Síntese no Stratix 2 EP2S180F1508I4

Elementos	ALUTs	Registradores	Pinos	Freqüência
128	81%	91%	14%	100,1 MHz

Pode ser visto na tabela 6.8 que a arquitetura tem 128 elementos funcionando na freqüência de 100,1 MHz. O número de elementos que foram sintetizados foi menor que no caso da recuperação dos escores, indicando uma maior utilização de recursos por elemento do vetor. Isto se deve ao maior número de informações que a arquitetura precisa armazenar para recuperar o alinhamento (seção 5.7.1).

6.3.2 Tempos e *speedups*

Para análise de desempenho, foram utilizadas as seqüências de RNA não codificadoras retiradas de [Mir08]: MIR156d|stem (*Arabidopsis thaliana)*, cel-let-7 MI0000001 (*Caenorhabditis elegans*), hsa-let-7a-2 MI0000061 (*Homo sapiens*) com tamanhos de 118bp, 99bp e 72bp.

A tabela 6.9 mostra os resultados com um programa otimizado em C para obter o alinhamento executando em um Pentium 4 3 GHz.

Tabela 6.9 – Alinhamento de seqüências de RNA não codificador

Seq Proc (bp)	Seq Base(bp)	Limiar	Rodadas	Escore	Tempo FPGA	Tempo Software	Speed-Up
118	99	0	6	57	0,000428 s	0,007 s	16,35
72	99	0	5	52	0,000374 s	0,004 s	10,69
72	118	0	10	46	0,000702 s	0,009 s	12,82

Na tabela 6.9, pode ser vista a comparação de seqüências de RNA não codificador de tamanhos menores que o tamanho máximo do vetor. Na tabela 6.9, a primeira e segunda colunas mostram os tamanhos das sequências comparadas. A terceira coluna mostra o limiar utlizado no DIALIGN. Na quarta coluna tem-se o número de rodadas necessárias para obter o alinhamento completo das duas sequências. A quinta coluna mostra o escore do DIALIGN obtido no alinhamento. A sexta e sétima colunas mostram respectivamente os tempos gastos no FPGA e em software. A oitava coluna mostra o *speedup* obtido pelo FPGA. O alinhamento das menores sequências (72 e 99) teve o *speedup* de 10,69 enquanto o alinhamento das maiores seqüências conseguiu *speedup* de 16,35. Apesar do tempo total depender das

sequências alinhadas, isto é um indicativo de que sequências maiores que ocupem todos os elementos do FPGA possam conseguir um maior *speedup*.

Para utilizar todo o poder de processamento da arquitetura, foram escolhidas sequências de tamanho próximo a 128. Para o teste da arquitetura foram utilizadas 5 sequências de RNA não codificador retiradas de [Ncr08]. São elas: FR000859 (*Methanocaldococcus jannaschii*) de tamanho 127, FR003139 (*Pseudomonas putida*) de tamanho 127, FR220660 (*Mus musculus*) de tamanho 128, FR063329 (*Archaeoglobus fulgidus*) de tamanho 128 e FR379163 (*Homo sapiens*) de tamanho 128.

O alinhamento de várias seqüências de RNAs não codificadores utilizando o DIALIGN é útil devido à capacidade do DIALIGN de descobrir fragmentos com grande similaridade separados por áreas com pouca similaridade [Cos03].

O resultado da comparação entre a arquitetura e o programa otimizado em C que executa a mesma tarefa do alinhamento linear é mostrado na tabela 6.10.

Tabela 6.10 – Alinhamento de seqüências de RNA não codificador

Seq Proc	Seq Base	Limiar	Rodadas	Escore	Tempo FPGA	Tempo Software	*Speed-Up*
FR000859	FR003139	0	14	67	0,001025	0,016	15,6
FR000859	FR220660	0	13	69	0,000945	0,014	14,8
FR000859	FR063329	0	9	71	0,000660	0,011	16,6
FR000859	FR379163	0	11	73	0,000803	0,012	14,9
FR003139	FR220660	0	10	73	0,000738	0,013	17,6
FR003139	FR063329	0	11	70	0,000806	0,013	16,1
FR003139	FR379163	0	13	73	0,000933	0,013	13,9
FR220660	FR063329	0	13	72	0,001046	0,016	15,3
FR220660	FR379163	0	11	77	0,000890	0,015	16,8
FR063329	FR379163	0	11	72	0,000897	0,013	14,5

Pode ser visto na tabela 6.10 que os *speedups* para o alinhamento encontram-se entre 13,9 e 17,6. Embora estes speedups sejam consideráveis, são menores que os obtidos pela arquitetura do DIALIGN para os escores (seção 2.4.5). Uma das razões é que as sequências são pequenas e, por esta razão, na execução em software elas podem ser totalmente colocadas na cache aumentando bastante a velocidade do programa. O escore obtido nos alinhamentos e o número

de rodadas necessárias para recuperar o alinhamento são dependentes das sequências comparadas.

A seguir, foram feitas comparações com sequências maiores. As seqüências utilizadas são seqüências reais de DNA retiradas do NCBI, *Aspergillus niger contig An18c0160* (AM270408), *Aspergillus niger contig An16c0230* (AM270375), *Encephalitozoon cuniculi* (AL590443) e da bactéria *Rhizobium leguminosarum* (AM236080) com tamanhos 121589bp, 169786bp, 194439bp, 5057142bp. Foram também utilizadas seqüências sintéticas de tamanhos 128 e 10.000.000.

Procurar por RNA não codificador dentro de seqüências genômicas é um problema relevante do ponto de vista biológico, sendo útil para entender os mecanismos de regulação genéticos [Cal04].

A tabela 6.11 mostra o resultado para o alinhamento com seqüências maiores.

Tabela 6.11 – Alinhamento com seqüências maiores

Seq Proc	Seq Base	Limiar	Rodadas	Escore	Tempo FPGA	Tempo Software	Speed-Up
72	169786	0	12	72	0,028396 s	2,23 s	78,53
72	121589	0	12	72	0,020441 s	1,57 s	76,80
72	194439	0	12	72	0,032188 s	2,59 s	80,46
118	194439	0	15	118	0,032543 s	4,05 s	130,40
118	5057142	0	11	118	0,809720 s	102,52 s	126,61
128	10000000	0	10	128	1,599409 s	226,14 s	141,38

Pode ser visto na tabela 6.11 que os *speedups* obtidos foram consideravelmente maiores, chegando a 141,38. A recuperação do alinhamento em software demorou 3 minutos e 46 segundos enquanto a recuperação em FPGA demorou 1,59 segundos.

A seguir, foi feita a comparação da versão 2 do DIALIGN que faz o alinhamento com a versão 1 que apenas calcula o escore. Como a versão 1 é executada em apenas uma rodada, será utilizado um limiar 4 que fará a versão 2 utilizar apenas uma rodada, para que possam ser comparados. O resultado é mostrado na tabela 6.12.

Tabela 6.12 – Comparação entre as duas arquiteturas do DIALIGN

Versão	Seq Proc	Seq Base	Escore	Tempo FPGA
1	118	19443	111	0,031403s
2	118	19443	111	0,032543s
1	118	5057142	107	0,814657s
2	118	5057142	107	0,808397s
1	128	10000000	128	1,610822s
2	128	10000000	128	1,598464s

No mesmo FPGA, couberam 200 elementos da versão 1 que só calcula o escore e 128 na versão 2 que faz o alinhamento. A versão 2 é mais complexa e utiliza proporcionalmente mais espaço no FPGA.

Os tempos de execução para as versões 1 e 2 são bastante semelhantes, sendo que a adição do módulo de recuperação do alinhamento e as modificações das relações de recorrências dos elementos não causaram uma grande sobrecarga. Além disto, foram feitas otimizações no circuito visando aumentar a velocidade, o que reduziu o tempo de execução.

7 - CONCLUSÕES

Nesta tese, foi feita uma extensa pesquisa sobre os principais projetos de arquiteturas dedicadas para algoritmos de bioinformática. A maioria das arquiteturas dedicadas analisadas implementou algoritmos de comparação de seqüências utilizando programação dinâmica, sendo implementadas em FPGA. Dado o bom desempenho destas arquiteturas, de acordo com a literatura, o projeto feito na tese utilizam este tipo de arquitetura.

Assim, foi projetado um vetor wavefront utilizando SystemC e Forte. O vetor wavefront projetado se mostrou bastante útil pois, uma vez que a arquitetura de base foi construída, ela pôde ser modificada de forma a utilizar três algoritmos diferentes. As principais modificações feitas para estes algoritmos foram na estrutura interna dos elementos de computação. Assim, esta arquitetura é flexível e pode ser adaptada posteriormente para outros algoritmos de programação dinâmica.

A implementação da arquitetura utilizando a combinação de SystemC e Forte se mostrou uma escolha acertada. O SystemC possui uma sintaxe acessível diminuindo o tempo necessário para o projeto. O Forte permitiu a simulação do programa bem como a geração dos arquivos em Verilog utilizados na síntese, utilizando as ferramentas de síntese dos fabricantes de FPGA.

A escolha de FPGAs como arquiteturas de destino se mostrou útil devido à sua grande flexibilidade bem como a disponibilidade de bons ambientes de síntese e simulação dados pelos fabricantes Xilinx [Xil08] e Altera [Alt08].

O primeiro algoritmo projetado foi a aceleração da fase mais intensiva de computação do alinhamento do Smith-Waterman em espaço linear. A síntese em um FPGA Xilinx xc2vp70 feita com 100 elementos resultou em um *speedup* de 246,9 sobre um programa em software equivalente executado em um Pentium 4 3 GHz [Bou07a].

O segundo algoritmo testado na arquitetura foi o cálculo do escore ótimo do DIALIGN em espaço linear. Esta arquitetura foi diferente de todas as analisadas na literatura. Devido a complexidade deste algoritmo, foi utilizado um FPGA de maior capacidade, o Stratix 2 EP2S180F1508I4. Neste FPGA foram sintetizados 200 elementos. O *speedup* chegou a 383,41 sobre o software [Bou07b].

O passo seguinte foi projetar uma arquitetura para a recuperação do alinhamento do DIALIGN. Foi constatado na revisão bibliográfica que o cálculo do alinhamento armazenando a matriz de programação dinâmica é inviável para seqüências grandes uma vez que estas matrizes estão muito acima da capacidade de armazenamento dos FPGAs atuais. Assim, foi projetada uma variante do DILIAGN capaz de recuperar o alinhamento em espaço linear. A síntese desta variante foi feita no Stratix 2 EP2S180F1508I4 gerando 128 elementos. O *speedup* obtido sobre o software chegou a 141,38.

Analisando os resultados obtidos sobre os programas em C pode-se concluir que a escolha da arquitetura wavefront, ferramentas de desenvolvimento, FPGA como arquitetura de destino e escolha dos algoritmos implementados resultou em *speedups* expressivos sobre os algoritmos equivalentes em software.

Ao propor uma arquitetura básica wavefront e implementar algoritmos de comparação de seqüências em espaço linear, a presente tese abre diversas perspectivas de trabalhos futuros. Como trabalho futuro imediato, a arquitetura wavefront pode ser modificada para fazer o alinhamento no DIALIGN para seqüências de tamanho qualquer através do particionamento das seqüências reaproveitando parte da solução projetada para a recuperação do escore do DIALIGN.

A versão em software do algoritmo da variante do DIALIGN para o alinhamento em espaço linear pode se tornar uma aplicação independente e pode ser utilizada para alinhamento de seqüências de tamanho qualquer. Esta versão pode ser paralelizada em várias máquinas.

Devido à flexibilidade da arquitetura wavefront, ela pode ser utilizada para a implementação de outros algoritmos de comparação de seqüências utilizando programação dinâmica como a previsão de estrutura secundária de RNAs. Os algoritmos implementados futuramente podem ser tanto da área de bioformática quanto como de outras áreas, tais como mineração de dados em textos.

REFERÊNCIAS BILIOGRÁFICAS

[Abd01] S. Abdeddaïm, B. Morgenstern, "Speeding up the DIALIGN multiple alignment program by using the 'Greedy Alignment of BIOlogical Sequences LIBrary' (GABIOS-LIB)", Lecture Notes in Computer Science, v. 2066:, p. 1–11, 2001

[Ada99] A. M. S. Adário, E. L. Roehe, S. Bampi, "Dynamically Reconfigurable Architecture for Image Processor Applications", Proceedings of Design Automation Conference, ACM, p. 623-628, 1999

[Alt08] Altera, http://www.altera.com, acessado em maio de 2008

[Alt90] S.F. Altschul, W. Gish, , W. Miller, E.W. Myers, D.J. Lipman, "Basic local alignment search tool", Journal of Molecular Biology, v. 215, p. 403-410, 1990

[Ani03] A. Anish, "Hardware Accelerated Protein Identification", dissertação de mestrado, Electrical and Computer Engineering Department, University of Toronto, 2003

[Ash90] P.J. Ashenden, "The VHDL Cookbook". First Edition, Departament of Computer Science, University of Adelaide South Australia, 1990

[Bat08] R. B. Batista, A. Boukerche, A. C. M. A. de Melo, "A parallel strategy for biological sequence alignment in restricted memory space", Journal of Parallel and Distributed Computing, v. 68, p. 548-561, 2008

[Bri08] Encyclopedia Britannica http://www.britannica.com/eb/article-9053245/molecular-biology/, acessado em maio de 2008

[Bou07a] A. Boukerche, J. M. Correa, A. C. M. A. de Melo, R. P. Jacobi, A. F. Rocha: "Reconfigurable Architecture for Biological Sequence Comparison in Reduced Memory Space" Proceedings of the International Parallel and Distributed Processing Symposium - Nature Inspired Distributed Computing workshop, p. 1-8, 2007

[Bou07b] A. Boukerche, J. M. Correa, A. C. M. A. de Melo, R. P. Jacobi, A. F. Rocha: “An FPGA-Based Accelerator for Multiple Biological Sequence Alignment with DIALIGN” Proceedings of the International Conference on High Performance Computing, p. 71-82, 2007

[Cal04] G.A. Calin, C. Sevignani, C.D. Dumitru, T. Hyslop , E. Noch, S. Yendamuri, M. Shimizu, S. Rattan, F. Bullrich, M. Negrini, C.M. Croce, “Human microRNA genes are frequently located at fragile sites and genomic regions involved in cancers” Proceedings of National Academy of Science, USA. v. 101 p. 2999-3004, 2004

[Car03] L. G. A. Carvalho, “Uma abordagem em hardware para algoritmos de comparação de seqüências baseados em programação dinâmica”, Departamento de Ciência da Computação, Universidade de Brasília, 2003

[Cha95] K. Chao, W. Miller, “Linear-Space Algorithms that Build Local Alignments from Fragments”, Algorithmica, v.13 p.106-134, 1995

[Cha04] C. Chang, “BEE2: A High-End Reconfigurable Computing System”, Relatório de Pesquisa, Berkeley Wireless Research Center, Universidade da Califórnia em Berkeley, 2004

[Com00] K. Compton, S. Hauck, “An introduction to Reconfigurable Computing”, Invited Paper IEEE Computer, abril, 2000

[Com02] K. Compton, S. Hauck, "Reconfigurable Computing: A Survey of Systems and Software", ACM Computing Surveys, v. 34, p. 171-210, 2002

[Cor90] T.H. Cormen, C.E. Leiserson, R.L. Rivest, “Introduction to Algorithms “, MIT Press, 1990

[Cos03] J. Costas, C.P. Vieira, F. Casares, J. Vieira, “Genomic characterization of a repetitive motif strongly associated with developmental genes in Drosophila”, BMC Genomics, 2003

[Day78] M. Dayhoff, R.M. Schwartz, B.C. Orcutt, “A model of evolutionary change in Proteins”, Atlas of Protein Structure, National Biomedical Research Foundation, 1978

[Dur98] R. Durbin, “Biological Sequence Analysis: Probabilistic Models of Proteins and Nucleic Acids”, Cambridge Univ Press, 1998

[Dos87] K. Doshi, P. Varman, “A modular systolic architecture for image convolutions”, International Conference on Computer Architecture Proceedings of the 14th annual international symposium on Computer architecture, p. 56-63, 1987

[Elr98] H. El-Rewini, T.G. Lewis, “Distributed and Parallel Computing”, Manning Publications, 1998

[Enz99] R. Enzler, “The Current Status of Reconfigurable Computing”, Technical Report, Electronics Laboratory, Swiss Federal Institute of Technology (ETH), 1999

[Eur02] European Commission, “Prospective Analysis of the relationship and synergy between Medical Informatics (MI) and Bioinformatics (BI)” White Paper EC-IST, 2002

[Fis92] V. Fischetti, G. Landau, J. Schmidt, P. Sellers, “Identifying Periodic Occurrences of a Template with Applications to Protein Structure Combinatorial Pattern Matching”, Lecture Notes in Computer Science, v. 644, p. 111-120, 1992.

[For05] Forte Design Systems, “Cynthesizer User’s Guide For Cynthesizer 2.4.0”, 2005

[Fos95] I. Foster, “Designing and Building Parallel Programs. Concepts and Tools for Parallel Software Engineering”, Addison Wesley, 1995

[Goh82] O. Gotoh, “An improved algorithm for matching biological sequences”. Journal of Molecular Biolog, v. 162, p. 705–708, 1982

[Gra01] L. Grate, M. Diekhans, D. Dahle, R. Hughey, “Sequence Analysis With the Kestrel SIMD Parallel Processor”, Proceedings of the Pacific Symposium on Biocomputing, Hawaii, Estados Unidos, p. 263-274, 2001

[Guc02] S.A. Guccione, E. Keller, “Gene Matching Using JBits, Field-Programmable Logic and Applications, Reconfigurable Computing Is Going Mainstream”, Proceedings of 12th International Conference, Lecture Notes in Computer Science v. 2438, p. 79-88, 2002

[Gus97] D. Gusfield, “Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology”, Cambridge University Press, Cambridge, UK, 1997

[Had95] J.D. Hadley, B.L., Hutchings, “Design Methodologies for Partially Reconfigured Systems”, Proceedings of IEEE Symposium on FPGAs for Custom Computing Machines, p. 78-84, 1995

[Har01] R.W. Hartenstein, “A Decade of Reconfigurable Computing: a Visionary Retrospective”, Proceedings of Conference on Design Automation and Testing in Europe, p. 642 –649, 2001

[Hen97] J. Henderson, S. Salzberg, K.H. Fasman, “Finding genes in DNA with a hidden Markov model”, Journal of Molecular Biology , p. 127-141, 1997

[Hen92] S. Henikoff, J. G. Henikoff, “Amino acid substitution matrices from protein blocks”, Proceedings of National Academy of Science USA, v. 89 p.10915-10919, 1992

[Hir75] D.S. Hirschberg, “A linear space algorithm for computing maximal common subsequences” Communications of the ACM, v. 18 p. 341–343, 1975

[Hoa92] D.T. Hoang, D. P. Lopresti, “FPGA Implementation of Systolic Sequence Alignment”, Field-Programmable Gate Arrays: Architectures and Tools for Rapid Prototyping, p. 183-191, Springer-Verlag, 1992

[Jac04] R.P. Jacobi, M. Ayala, L.G.A. Carvalho, C.H. Llanos, R.W. Hartenstein, “Reconfigurable Systems for Sequence Alignment and for General Dynamic Programming”, Proceedings of the 3rd Brazilian Workshop on Bioinformatics - WOB 2004, p. 25-32, 2004

[Kah05] T. Kahveci, V. Ramaswamy, H. Tao, T. Li, “Approximate Global Alignment of Sequences”, Proceeding of the IEEE Symposium on Bioinformatics and Bioengineering, p. 81-88, 2005

[Kav96] A. Kaviani, S. Brown, “Hybrid FPGA Architecture”, Proceeding of the Fourth International ACM Symposium on Field-Programmable Gate Arrays, p. 3-9, 1996

[Kno04] G. Knowles, P. Gardner-Stephen, “A New Hardware Architecture for Genomic Sequence Alignment”, Proceeding of the 3rd International IEEE Computer Society Computational Systems Bioinformatics Conference , p. 16-19, 2004

[Kou91] J.L. Kouloheris, A. El Gamal, "FPGA Performance versus Cell Granularity," Proceeding of the IEEE Custom Integrated Circuits Conference, p. 621-624, 1991

[Kri05] K. Muriki, K. Underwood, R. Sass, “RC-BLAST: Towards an Open Source Hardware Implementation”, Proceeding of the HiCOMB 2005 Fourth IEEE International Workshop on High Performance Computational Biology, 2005

[Kun82] H.T. Kung, “Why Systolic Architectures?”, Computer, vol. 15, p. 37- 46, 1982.

[Kun87] S.Y. Kung; S.C. Lo; S.N. Jean; J.N. Hwang, “Wavefront Array Processors-Concept to Implementation”, Computer v. 20 p. 18 - 33, 1987

[Kur97] S. Kurtz, G. Myers, “Estimating the probability of approximate matches”, Proceedings of the 8th Annual Symposium on Combinatorial Pattern Matching, p 52-64, 1997

[Lav96] D. Lavenier, “Dedicated Hardware for Biological Sequence Comparison”, Journal of Universal Computer Science, p. 77-86, 1996

[Lav98] D. Lavenier, “Speeding up genome computations with a systolic accelerator”, SIAM news, v.31, p.6-7, 1998

[Leh00] A.L. Lehninger, D. L. Nelson, “Principios de bioquímica”, Sarvier, 2a ed, 2000

[Lip84] D.J. Lipman, W.J. Wilbur, T.F. Smith, M. S. Waterman, "On the statistical significance of nucleic acid similarities", Nucleic Acids Research v.12 p 215-226, 1984

[Mar03] A. Marongiu, P. Palazzari, V. Rosato, “PROSIDIS: a Special Purpose Processor for PROtein SImilarity DIScovery”, Proceeding of the Second IEEE International Workshop on High Performance Computational Biology, p.155, 2003

[Mat06] J.S. Mattick, I.V. Makunin, “Non-coding RNA”, Human Molecular Genetics v. 15, p. R17-R29, 2006

[Mel89] R. A. Melhem, “Systolic Accelerator for the Iterative Solution of Sparse Linear Systems”, IEEE Transactions on Computers, v. 38, p. 1591-1595, 1989

[Mey83] P. Meyer, “Probabilidade - aplicações à Estatística”, Livros Técnicos e Científicos Editora, 1983

[Mey95] R. A. Meyers, “Molecular biology and biotechnology: A comprehensive desk reference”, New York Vch Publishers, 1995

[Mir08] miRBase, http://microrna.sanger.ac.uk/sequences/index.shtml acessado em maio de 2008

[Moe99] T. Moeller, “Field Programmable Gate Arrays for Radar Front-End Digital Signal Processing”, IEEE Symposium on FPGAs for Custom Computing Machines, p.178-187, 1999

[Mor96] B. Morgenstern, A. Dress, T. Werner, “Multiple DNA and protein sequence alignment based on segment-to-segment comparison”, Proceeding of the National Academy of Science, p.12098–12103, 1996

[Mor98] B. Morgenstern, K. Frech, A. Dress, T. Werner, "DIALIGN: finding local similarities by multiple sequence alignment”, Bioinformatics, v.14 p.290-294, 1998

[Mor99] B. Morgenstern, “DIALIGN 2: improvement of the segment-to-segment approach to multiple sequence alignment”, Bioinformatics, v.15 p.211-218, 1999

[Mor00] B. Morgenstern, “A space-efficient algorithm for aligning large genomic sequences”, Bioinformatics, v.16 p.948 – 949, 2000

[Mor02] B. Morgenstern, “A simple and space-efficient fragment-chaining algorithm for alignment of DNA and protein sequences”, Applied Mathematics Letters v.15 p.11-16, 2002

[Mos98] E. Mosanya, “A Reconfigurable Processor for Biomolecular Sequence Processing”, tese de doutorado, Faculté informatique et communications, Ecole Polytechnique Fédérale de Lausanne, 1998

[Nav01] G. Navarro, “A guided tour to approximate string matching”, ACM Computing Surveys, v.33 p.31-88, 2001

[Ncb08] National Center for Biotechnology Information (NCBI), www.ncbi.nlm.nih.gov, acessado em maio de 2008

[Ncr08] Funtional RNA Project, http://www.ncrna.org/ acessado em maio de 2008

[Nee70] S. B. Needleman, C. D. Wunsch, “A General Method Applicable to the Search for Similarities in the Amino-Acid Sequence of Two Proteins”, Journal of Molecular Biology, v.48 p.443–453, 1970

[Nih08] National Institutes of Health, Biomedical Information Science and Technology Initiative (BISTI), http://www.bisti.nih.gov/ acessado em maio de 2008

[Oli99] F. A. D. D. Oliveira, “Arquiteturas reconfiguráveis de computadores: características gerais e um estudo de caso”, Universidade Federal do Rio Grande do Sul, 1999

[Oli04] T. Oliver, B. Schmidt, “High Performance Biosequence Database Scanning on Reconfigurable Platforms”, Proceeding of the Third IEEE International Workshop on High Performance Computational Biology (HiCOMB), Estados Unidos, 2004

[Oli05] T. Oliver, B. Schmidt, D.L. Maskell, “Hyper Customized Processors for Bio-Sequence Database Scanning, on FPGAs”, Proceeding of the International Symposium on Field Programmable Gate Arrays, p.229-237, Estados Unidos, 2005

[Oli05b] T. Oliver, B. Schmidt, D.L. Maskell, D. Nathan, R. Clemens: “Multiple Sequence Alignment on an FPGA”, Proceeding of the International Conference on Parallel and Distributed Systems, p.326-330, 2005

[Oli05c] T. Oliver, B. Schmidt, D.L. Maskell, A.P. Vinod, “A Reconfigurable Architecture for Scanning Biosequence Databases”, Proceeding of the International Symposium on Circuits and Systems, p.4799-4802, Japão, 2005

[Ope05] Open SystemC Initiative Draft, Standard SystemC Language Reference Manual, 2005

[Pag96] I. Page, “Reconfigurable processor architectures”, Microprocessors and Microsystems, v.20 p.185-196, 1996

[Pag04] I. Page, “Compiling software to gates”, Embedded Systems Programming Magazine, 2004

[Pat98] D.A. Patterson, J.L. Hennessy, “Computer Organization and Design: The Hardware/Software Interface”, Morgan Kaufmann, 1998

[Pel05] D. Pellerin, S. Thibault, “Practical FPGA Programming in C”, Prentice Hall, 2005

[Put03] K. Puttegowda, W. Worek, N. Pappas, A. Dandapani, P. Athanas, “A run-time reconfigurable system for gene-sequence searching”, Proceedings. 16th International Conference on VLSI Design, p.561-566, 2003

[Set97] J. Setubal, J. Meidanis, “Introduction to Computational Molecular Biology”, PWS Publishing Company, 1997

[Sfi06] Sfiec Biologia, http://www.sfiec.org.br acessado em maio de 2008

[Sch04] M. Schmollinger, K. Nieselt, M. Kaufmann, B. Morgenstern, “DIALIGN P: Fast pair-wise and multiple sequence alignment using parallel processors”, BMC Bioinformatics, 2004

[Smi81] T.F. Smith, M.S. Waterman, “Identification of common molecular sub-sequences”, Journal of Molecular Biology, v.147 p.195-197, 1981

[Smi97] A.D. Smith, “Oxford dictionary of biochemistry and molecular biology”, Oxford: Oxford Univ Press, 1997

[Str99] T. Strachan, A. P. Read, “Human Molecular Genetics 2”, John Wiley & Sons, 1999

[Tan90] A.S. Tanenbaum, “Organização Estruturada de Computadores”, Prentice/Hall do Brasil, 1990

[Tan03] A.S. Tanenbaum, “Sistemas Operacionais Modernos”, Pearson, 2003

[Tau84] H. Taub, “Circuitos Digitais e Microprocessadores”, Editora Mcgraw - Hill 1984

[Tha00] P. A. Thaker, “Register-Trasnfer Fault Modeling and Test Evaluation Technique for VLSI Circuits”, Tese de Doutorado, The Department of Electrical and Computer Engineering, George Washington University, 2000

[Tho94] J. Thompson, D. Higgins, T. Gibson, "CLUSTAL W: improving the sensitivity of progressive multiple sequence alignment through sequence weighting, position specific gap penalties and weight matrix choice," Nucleic Acids Research, v.22 p.4673-4680, 1994

[Tim08] Timelogic Decypher http://www.timelogic.com acessado em maio de 2008

[Wes03] B. West, R.D. Chamberlain, R.S. Indeck, Q. Zhang, “An FPGA-based Search Engine for Unstructured Database”, Proceeding of the Workshop on Application Specific Processors, p. 25-32, 2003

[Wor02] W.J. Worek, “Matching Genetic Sequences in Distributed Adaptive Computing Systems”, dissertação de mestrado, Computer Engineering, Virginia Polytechnic Institute and State University, 2002

[Wol04] W. Wolf, “FPGA-Based System Design, Modern Semicondutor Series”, Prentice-Hall, 2004

[Xil08] Xilinx, http://www.xilinx.com acessado em maio de 2008

[Yam02] Y. Yamaguchi, T. Maruyama, A. Konagaya, “High Speed Homology Search with FPGAs”, Proceeding of the Pacific Symposium on Biocomputing, p. 271-282, Estados Unidos, 2002

[Yu03] C.W. Yu, K.H. Kwong, K.H. Lee, P.H.W. Leong, “A Smith-Waterman Systolic Cell”, Proceeding of the International. Conference on Field-Programmable Logic and Applications, p. 375-384, Portugal 2003

[Zah96] A. Zaha, “Biologia molecular básica”, Mercado Aberto, 1996

[Zah07] P. Zhang, G. Tan, G.R., Gao, “Implementation of the Smith-Waterman algorithm on a reconfigurable supercomputing platform”, Proceeding of the Conference on High Performance Networking and Computing, p.39-48, 2007