Pós-Graduação em Ciência da Computação

Antonyus Pyetro do Amaral Ferreira

Aceleração da consulta a um grande banco de DNA forense: uma abordagem multiplataforma

Universidade Federal de Pernambuco
posgraduacao@cin.ufpe.br
www.cin.ufpe.br/~posgraduacao

RECIFE
2016

Antonyus Pyetro do Amaral Ferreira

Aceleração da consulta a um grande banco de DNA forense: uma abordagem multiplataforma

Este trabalho foi apresentado à Pós-Graduação em Ciência da Computação do Centro de Informática da Universidade Federal de Pernambuco como requisito parcial para obtenção do grau de Doutor em Ciência da Computação.

ORIENTADOR(A): **Prof. Manoel Eusébio de Lima**

RECIFE
2016

Catalogação na fonte
Bibliotecária Monick Raquel Silvestre da S. Portes, CRB4-1217

F383a Ferreira, Antonyus Pyetro do Amaral
Aceleração da consulta a um grande banco de DNA forense: uma abordagem multiplataforma / Antonyus Pyetro do Amaral Ferreira. – 2016.
152 f.: il., fig., tab.

Orientador: Manoel Eusébio de Lima.
Tese (Doutorado) – Universidade Federal de Pernambuco. CIn, Ciência da Computação, Recife, 2016.
Inclui referências.

1. Engenharia da computação. 2. Arquitetura de computador. 3. FPGA. I. Lima, Manoel Eusébio de (orientador). II. Título.

621.39 CDD (23. ed.) UFPE- MEI 2017-79

Antonyus Pyetro do Amaral Ferreira

Aceleração da Consulta a um Grande Banco de DNA Forense: Uma Abordagem Multiplataforma

Tese de Doutorado apresentada ao Programa de Pós-Graduação em Ciência da Computação da Universidade Federal de Pernambuco, como requisito parcial para a obtenção do título de Doutor em Ciência da Computação

Aprovado em: 16/06/2016.

_____

Orientador: Prof. Dr. Manoel Eusébio de Lima

BANCA EXAMINADORA

_____

Profa. Dra. Edna Natividade da Silva Barros
Centro de Informática / UFPE

_____

Profa. Dra. Veronica Teichrieb
Centro de Informática / UFPE

_____

Prof. Dr. Edward David Moreno Ordonez
Departamento de Computação / UFS

_____

Prof. Dr. Wang Jiang Chau
Departamento de Engenharia de Sistemas Eletrônicos / USP

_____

Prof. Dr. Victor Wanderley Costa de Medeiros
Departamento de Estatística e Informática / UFRPE

Agradecimentos

Gostaria de agradecer à minha família pelo apoio e por me dar suporte. Aos meus colegas de trabalho que estão diretamente relacionados com este trabalho: João Gabriel e Jefferson dos Anjos pelo trabalho, dedicação, por acreditar e embarcar neste projeto junto comigo. A Luís Figueirôa, com quem pude trocar experiências por algum tempo deste projeto.

À orientação do Professor Manoel Eusébio, tanto nessa jornada quanto desde o período do HPCin, no projeto Sísmica, que me acolheu e que, com orgulho, cooperei e aprendi bastante.

Ao CETENE, na pessoa da Coordenadora do LINCS, Edna Barros pelo auxílio e pela confiança em mim depositada.

Aos Professores Abel Guilhermino, Edna natividade e Wang Chau pelas correções e sugestões dadas na ocasião do exame de qualificação desta tese.

Ao Centro de Informática, representado por seus professores e funcionários, onde eu me graduei em Engenharia da Computação, fiz Mestrado, conheci pessoas excelentes e vivi anos preciosos da minha vida.

Ao CNPQ por dar suporte aos 2 anos iniciais da pesquisa e na aquisição de equipamentos através da taxa de bancada.

Resumo

A comparação de cadeias de DNA é um problema clássico em biologia molecular. Uma aplicação forense dessas comparações é usada no problema de identificação pessoal. Por exemplo, nos EUA, o sistema CODIS dispõe, hoje em dia, de 14,5 milhões de perfis de DNA armazenados em seu banco de dados. Visando acelerar essa recorrente tarefa da consulta em banco de dados similares ao CODIS, este trabalho apresenta implementações em software e em hardware digital do algoritmo de Needleman-Wunsch, que representa uma técnica global ótima para se medir a similaridade entre cadeias de DNA. Implementações em Multi-Threads, SIMD (Single Instruction Multiple Data) e OpenCL são investigadas para a plataforma dos GPPs (General Purpose Processors). A infraestrutura de OpenCL também foi usada para analisar o desempenho das GPUs (Graphics Processing Units) para essa tarefa. Adicionalmente, uma arquitetura de hardware digital customizada explorou o paralelismo dos FPGAs (Field Programmable Gate Arrays), buscando-se otimizar o uso dos recursos de hardware e a banda de memória. Os experimentos foram conduzidos usando um banco de DNA sintético com 8 milhões de indivíduos, em que cada um deles é representado por 15 sequências do tamanho de 240 nucleotídeos. Nesse caso de uso, a implementação em um único FPGA Stratix IV, rodando a 280MHz atingiu o maior speed-up de 1885x, em comparação com a implementação canônica em software. Como resultados secundários, as versões em OpenCL (GPU e GPP) e a versão SIMD obtiveram menores tempos de execução comparados com os softwares SWIPE e FASTA que são amplamente utilizados na área.

Palavras-chave: Bioinformática. DNA. Arquitetura de Hardware. Análise Forense. Computação de alto desempenho. FPGA.

Abstract

The comparison of DNA sequences is a classic problem in molecular biology. A forensic application of this comparison is used in the personal identification problem. For instance, in the USA, the CODIS system has today 14.5 million DNA profiles stored on its database. In order to accelerate the recurrent task to query into similar databases, this work presents implementations in software and hardware of the Needleman-Wunsch algorithm, that represents an optimal global technique for measuring similarity between DNA sequences. Multi-threaded, SIMD (Single Instruction Multiple Data), and OpenCL implementations were investigated in a GPP (General Purpose Processor) platform. The OpenCL infrastructure was also used to analyze the performance of GPUs (Graphics Processing Units) for this task. Additionally, a customized digital hardware architecture explored the parallelism of the FPGAs (Field Programmable Gate Arrays), optimizing the use of hardware resources and memory bandwidth. The experiments were conducted using a synthetic DNA database with 8 million individuals, in which, each of them are represented as 15 sequences with length of 240 nucleotides. In this case study the implementation in a single Stratix IV FPGA, running at 280MHz achieved the highest speed-up of 1885x, in comparison with the canonic software implementation. As collateral results, the OpenCL (GPU and CPU) and SIMD versions outperformed consolidated software implementations like SWIPE and FASTA.

Key-words: Bioinformatics. DNA. Hardware Architecture. Forensics. high performance computing. FPGA.

Lista de Ilustrações

Lista de Tabelas

Lista de Abreviaturas

1. **Base nitrogenada** – Composto químico cíclico que contém nitrogênio. As bases nitrogenadas do DNA são Adenina, Guanina, Citosina, Timina.
2. **DNA** - ácido desoxirribonucleico em português, ADN ou em inglês DNA, deoxyribonucleic acid. Molécula em dupla hélice que carrega a informação genética dos seres vivos.
3. **FPGA** – Field Programing Gate Array
4. **GPU** – Graphics Processing Unit
5. **GPP** – General Purpose Processor, processador de propósito geral.
6. **HDL** – Hardware Description Language, linguagem de descrição de hardware.
7. **MPI** – Message Passing Interface, protocolo de troca de mensagens utilizado em computação paralela e distribuída em cluster.
8. **Nucleotídeo** – Também chamado de base nitrogenada.
9. **OpenCL** – Open Computing Language, linguagem utilizada para aplicações paralelas em sistemas heterogêneos.
10. **OpenMP** - Open Multi-Processing, API de software para paralelização de threads
11. **Sequenciamento de DNA** – Codificação completa ou parcial do material genético de um ser vivo em sequência de caracteres

Sumário

1

Introdução

Em países desenvolvidos, como os EUA e o Reino Unido existem bases de dados forenses de DNA que ajudam as autoridades em investigações de crimes. Nos EUA, por exemplo, o sistema é chamado de CODIS (Combined DNA Index System) e foi criado em 2007. Ele faz parte de um programa do FBI de suporte para a justiça criminal, bem como de um software usado para realizar as consultas ao banco. Atualmente o banco de dados do CODIS conta com quase 14,5 milhões de perfis de DNA de indivíduos que cometeram diversos tipos de crimes e ofensas legais. Até junho de 2015, o sistema produziu mais de 288.298 acertos, ajudando em mais de 274.648 investigações (CODIS—NDIS Statistics n.d.). O sistema define 13 regiões do DNA humano (chamadas de loci) a serem armazenadas no banco de dados para propósitos de identificação pessoal. A Figura 1 mostra os marcadores de DNA escolhidos, bem como sua localização cromossômica.

Já no Reino Unido, até junho de 2015, a NDNAD (National DNA Database) armazena 5,7 milhões de perfis de DNA (Office 2015) (15 sequências por indivíduo), o que representa perto de 10% da população do Reino Unido.

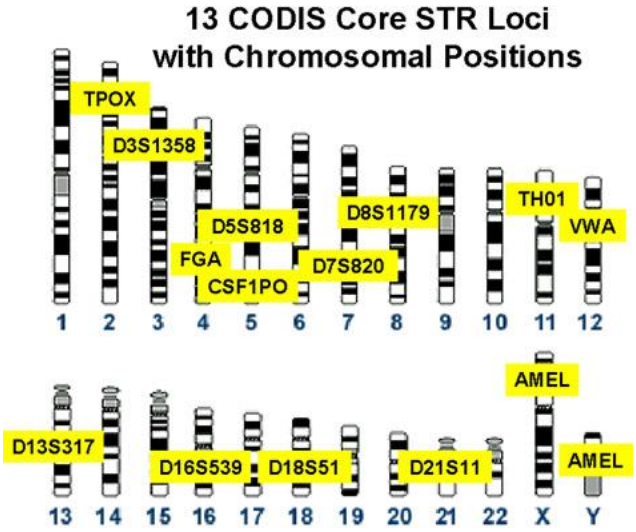
No contexto brasileiro, segundo a ENASP (Estratégia Nacional de Justiça e Segurança Pública), a taxa de resolução de inquéritos de homicídio no Brasil é muito baixa, por volta de 5 a 8% (Conselho Nacional do Ministério Público s.d.).

Esse dado se torna alarmante quando em comparação com países na Europa que usam bancos de dados de DNA e apresentam taxas da ordem de 90%.

Outro dado que se soma a esse cenário é que a População carcerária brasileira é a terceira maior do mundo com 715.655 presos em todos os regimes de prisão mais 373.991 mandados de prisão em aberto, totalizando 1,08 milhão de pessoas (Conselho Nacional de Justiça 2014). Já o número de inquéritos policiais ou notícias-crime sem conclusão é da ordem de 3,8 milhões. Essa quantidade equivale a 72% do total de 5,3 milhões de inquéritos recebidos pelas Promotorias e Procuradorias estaduais e federais (Conselho Nacional do Ministério Público s.d.).

Nesse sentido, a lei 12.654 de 28 de maio de 2012 torna obrigatória a identificação genética, por meio de DNA, de condenados por crimes hediondos ou crimes violentos contra a pessoa, como homicídio, extorsão mediante sequestro, estupro, entre outros. O objetivo é utilizar os dados colhidos nas investigações de crimes cometidos por ex-detentos. A lei estabelece a criação do banco nacional sigiloso com o material genético colhido pelos estados da federação.

Figura 1 Os 13 Loci dos marcadores utilizados no CODIS



Nesse Banco Nacional de Perfis Genéticos, serão centralizados não só dados de

criminosos como os de desaparecidos. Esses dados terão de ser cedidos voluntariamente pelos familiares e só poderão ser utilizados para ajudar em eventuais buscas.

A legislação define que 15 marcadores serão armazenados para cada indivíduo. Logo uma busca num banco da dimensão da população carcerária brasileira seria $15\ x\ 10^6$ alinhamentos de DNA por consulta ao banco (no pior caso) resultando em 3 bilhões de comparações de nucleotídeos. Este é um problema grande, especialmente quando se observa o número de inquéritos de homicídios em aberto no Brasil até 2007 que é de 151.819.

1.1 O contexto biológico

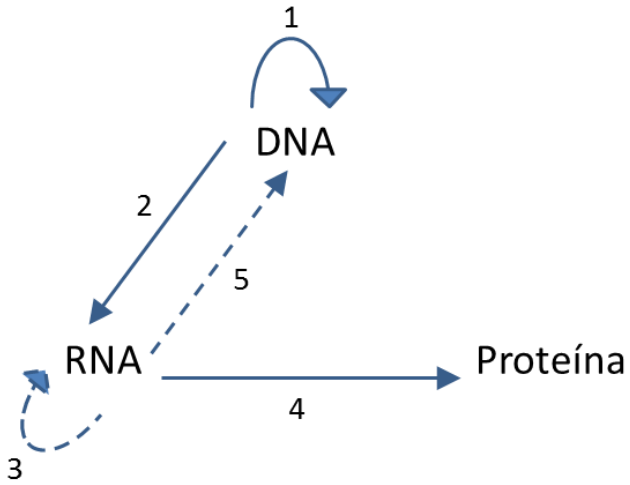
O genoma é a totalidade da informação genética de um indivíduo, codificada na forma de DNA (ou na forma de RNA em alguns vírus). O DNA (Ácido Desoxirribonucleico) está contido no núcleo celular nos seres eucariontes (animais, plantas, fungos, etc.) e disperso no citoplasma nos seres procariontes (bactérias, etc.). O genoma dos seres eucariontes é mais complexo e está distribuído na forma de cromossomos. Nos humanos todas as células contêm 22 pares de cromossomos autossomos e um par de cromossomos ligados ao sexo. A única exceção a essa regra são os gametas que apresentam metade dessa informação genética.

A Biologia Molecular é o ramo da biologia que estuda a estrutura genética e as funções no nível molecular. O dogma central da Biologia Molecular enunciado por Crick (Crick 1970) trata da "detalhada transferência resíduo-por-resíduo de informação sequencial. O dogma define que tal informação não pode ser transferida de volta a partir de uma proteína para outra proteína ou ácido nucleico" (Crick 1970).

O dogma central da Biologia Molecular, enunciado por Crick (Crick 1970) em 1958 e depois revisitado em 1970, pode ser traduzido pelo diagrama da Figura 2, em que as setas cheias representam os eventos predominantes nos seres vivos: a dinâmica de multiplicação celular (1), produção do RNA mensageiro (2)

e produção (síntese) de proteínas (4). As setas tracejadas 3 e 5 representam fenômenos raros que ocorrem na reprodução dos vírus que não contêm DNA, apenas RNA.

Figura 2 Diagrama do Dogma Central da Biologia Molecular (Crick 1970).



Nos seres eucariontes, o DNA encontra-se protegido pela membrana nuclear, segmentado e enovelado na forma de cromossomos, como mostra a Figura 3. Ao se desenrolar os cromossomos pode-se ver a estrutura de dupla hélice do DNA.

A dupla hélice do DNA ou do RNA são longas cadeias poliméricas formadas por compostos químicos chamados nucleotídeos. Os nucleotídeos são compostos por um grupo açúcar-fosfato (que é a espinha dorsal do DNA) e por uma base nitrogenada. As bases nitrogenadas classificadas em Pirimidinas (com estrutura de aro simples – Citosina e Timina) e Purinas (com estrutura de aro duplo – Adenina e Guanina), como pode ser visto na Figura 4. As bases nitrogenadas se ligam por pontes de hidrogênio e possuem polaridade anti-paralela o que torce a molécula no formato helicoidal.

Figura 3 Localização e estrutura do DNA (DNA s.d.)

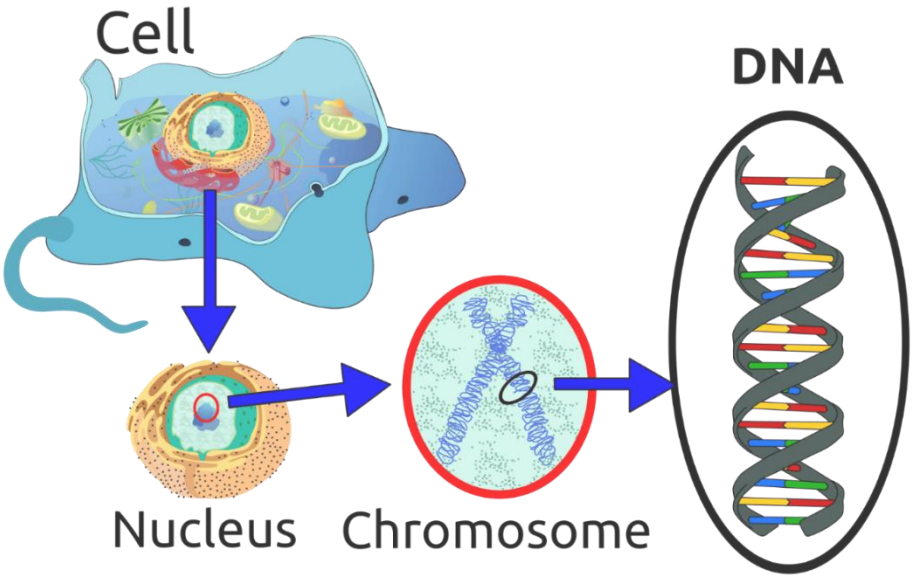
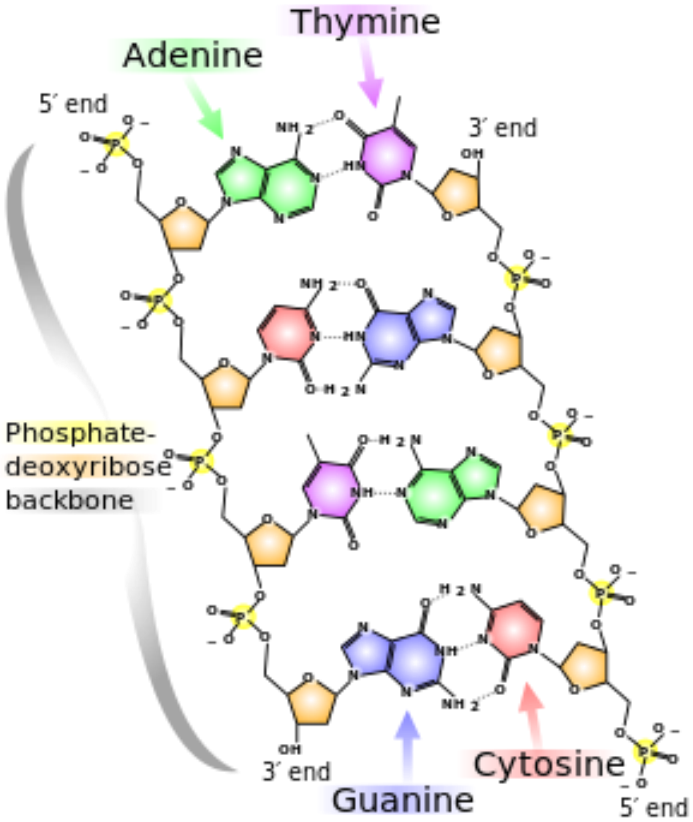


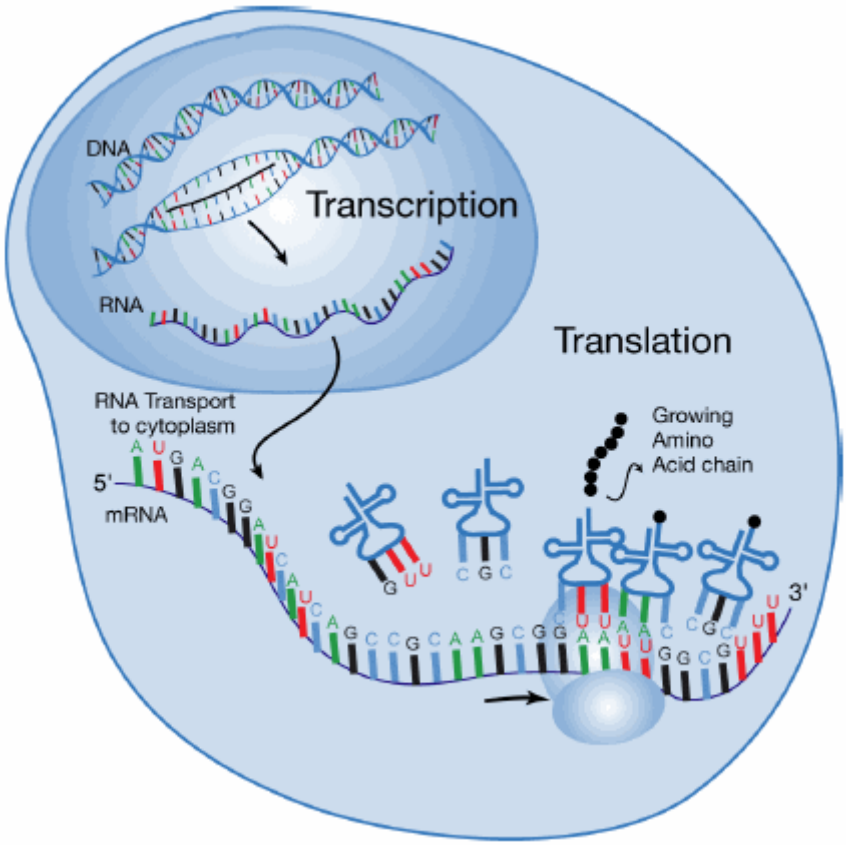
Figura 4 Estrutura química do DNA (DNA s.d.)



O processo de transcrição e tradução do DNA é mostrado na Figura 5. A Transcrição é a cópia do DNA em RNA, este transporta a cópia da informação

genética através do citoplasma da célula até as organelas onde as proteínas são formadas, os ribossomos. Nos ribossomos o RNA é traduzido através do encaixe dos aminoácidos correspondentes em uma proteína.

Figura 5 Transcrição e tradução do DNA em proteínas ((NHGRI) s.d.).

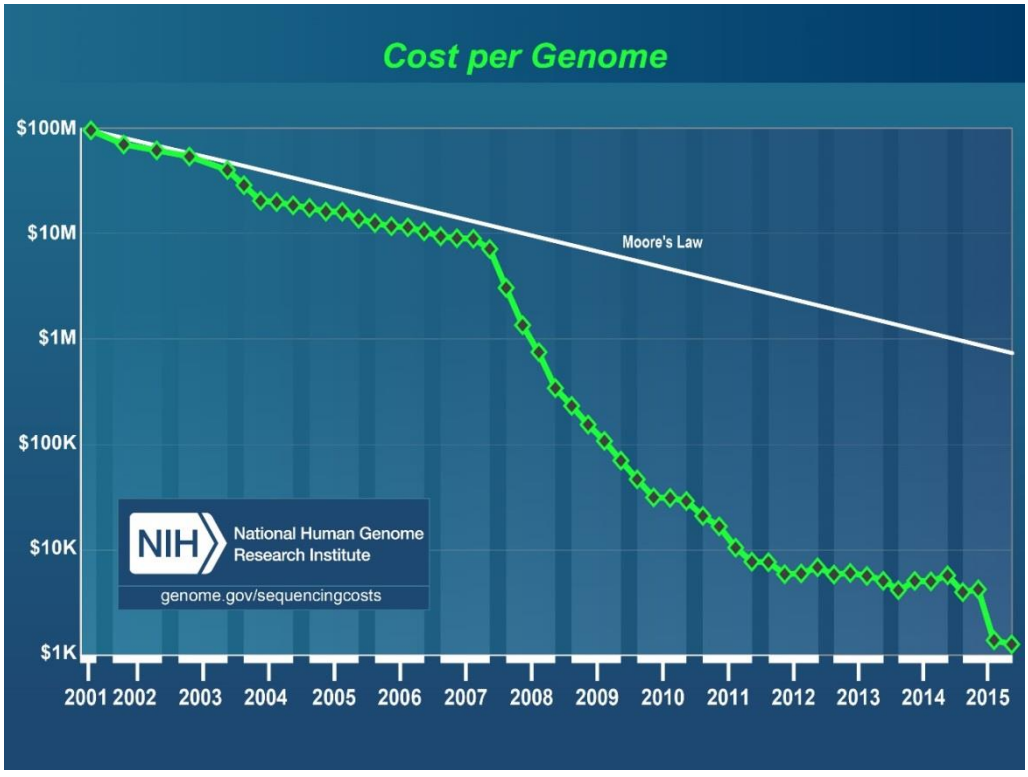


Desde que a estrutura do DNA foi revelada em 1953 (WATSON e CRICK 1953), a biologia molecular testemunhou um grande avanço (Setubal e Meidanis 1997). Com o crescimento na capacidade de manipular sequências de DNA, uma grande quantidade de dados tem sido gerada a todo instante. O sequenciamento de DNA corresponde na transcrição de toda ou parte da informação genética de um indivíduo em uma cadeia linear de símbolos correspondentes às bases nitrogenadas que compõem o DNA (a = adenina, c = citosina, t = timina, g = guanina). E desde o sequenciamento do primeiro DNA na década de 1970, o custo de sequenciamento por indivíduo vem decrescendo e o número de organismos completamente sequenciados é crescente (Setubal e

Meidanis 1997).

Esse fato pode ser visto na Figura 6 que relaciona o custo do sequenciamento de genoma completo ao longo dos anos. Em 2002 esse custo era de 100 milhões de dólares e em 2015 chega por volta de 5 mil dólares. O gráfico da Figura 6 é mono-log no eixo y, de forma que a lei de Moore é representada por uma reta. Observa-se que até meados de 2006 o custo caia aproximadamente na taxa da lei de Moore, mas a partir com a difusão dos NGS (Next Generation Sequencers) o ritmo de queda foi bem mais intenso. Porém, a partir de 2011 o ritmo tende novamente a se aproximar da lei de Moore, sem que nenhuma nova tecnologia tenha trazido avanço maior que o ritmo ditado pela lei de Moore.

Figura 6 Gráfico mono-log da evolução do custo de sequenciamentos de DNA (em milhares de dólares) (National Human Genome Research Institute (NHGRI) n.d.)



Contudo, a partir de outubro de 2015, novamente, o custo cai mais rapidamente aproximando-se de 1 mil dólares com o lançamento de sequenciadores que aumentam o volume de sequências processadas por lote e

inclusive podendo processar diversos organismos por vez. O marco dos sequenciamentos pelo custo de 1 mil dólares é tido por especialistas como um marco da massificação das análises genéticas personificadas.

A necessidade de processar essas informações, de maneira que elas possam ser úteis em avanços científicos, criou uma nova gama de problemas. Por exemplo, bases de dados são necessárias para armazenar essa informação gerada e o entendimento das sequências requerem sofisticadas técnicas de reconhecimento de padrões. O exemplo clássico de um problema em biologia computacional equacionável por um algoritmo é a comparação de sequências de DNA, em que duas sequências representando biomoléculas são comparadas a fim de se determinar a similaridade entre elas.

Estre problema é resolvido milhares de vezes todos os dias. A comparação entre sequências é a mais importante das operações primitivas em biologia computacional servindo como base para diversas outras manipulações mais complexas (Singh 2015). O algoritmo que resolve o problema do alinhamento ótimo usa uma técnica computacional chamada programação dinâmica. Ela consiste em resolver uma instância do problema a partir de computações já realizadas para instâncias menores do mesmo problema (Singh 2015). O gargalo mais importante desse algoritmo é a intrínseca dependência de dados e a complexidade de espaço da solução.

Esse volume de dados gerado demanda um poder de processamento grande (Singh 2015), visto que após o sequenciamento, um intenso pós-processamento é necessário para analisar e entender os dados, verificar semelhanças entre indivíduos, diagnosticar doenças, por exemplo. Dentre esses processamentos necessários encontra-se os alinhamentos locais e globais de sequências.

1.2 Problemática e Escopo

Nesse contexto, o problema de verificar a identidade de um criminoso consiste em comparar o material coletado na cena do crime com todos os perfis

existentes que estão armazenados no banco de dados, no formato de consulta 1-por-n. Adicionalmente, cada indivíduo é identificado por p sequências curtas de DNA. No caso do banco nacional do Reino Unido $p = 15$ e no CODIS $p = 13$.

Entretanto, nesse caso, as comparações das sequências de DNA implicam em realizar um alinhamento global a cada par de sequências. O algoritmo de Needleman-Wunsch resolve este problema de maneira ótima e provê uma medida global de similaridade entre duas sequências (Needleman e Wunsch 1970).

Embora algoritmos heurísticos sejam uma solução computacional bastante promissora em temos de redução de complexidade, há aplicações em que não se pode admitir margens de erro, como em aplicações de segurança e aplicações forenses. A solução proposta neste trabalho pretende realizar com rapidez alinhamentos ótimos (sem o uso de heurísticas) de cadeias de DNA, permitindo que um número bem maior de alinhamentos seja feito por unidade de tempo.

Visando acelerar esse número crescente de comparações de sequências, muitos trabalhos propuseram soluções, desde soluções em hardware digital (Aluru e Jammula 2014), passando por implementações em GPUs (Graphic Processing Units) (Savran, Gao e Bakos 2014) (Liu, Wirawan e Schmidt 2013) e também em melhorias nas aplicações que fazem uso de processadores de propósito geral (GPPs).

Este trabalho apresenta implementações nas três arquiteturas mencionadas e as compara, utilizando como base a implementação canônica em GPP.

A implementação baseada em GPP proposta maximiza a localidade de dados, explora recursos em sistemas multiprocessados e com multithreads. Também foram utilizadas instruções intrínsecas de CPU do conjunto SSE3 e AVX para explorar o paralelismo no nível de instruções. Para explorar o paralelismo multithread/multiprocessador, foi utilizada a API de OpenMP (Open Multi-Processing), além da utilização da linguagem OpenCL (Open Computing Language) nas implementações para a arquitetura dos GPPs.

As implementações em GPU buscam maximizar a ocupação dos núcleos, distribuindo um core por alinhamento, fazendo uso da memória privada das threads em detrimento do uso da memória global. A linguagem utilizada para programar as GPUs foi OpenCL.

A implementação em hardware digital explora o paralelismo intrínseco dos FPGAs para implementar uma arquitetura customizada para resolver o problema. A arquitetura proposta explora o paralelismo espacial e temporal através de um caminho de dados completamente em pipeline (usando arrays sistólicos), que prioriza o uso da memória interna do FPGA e que foi projetada para atingir uma alta frequência de operação.

Diferentemente das abordagens relatadas na literatura, a arquitetura proposta, neste trabalho, além de explorar o paralelismo temporal e espacial, também realiza a fase de *traceback* do algoritmo dentro do FPGA para múltiplos alinhamentos em paralelo. Pretende-se, com isso, a minimização do acesso à memória externa ao FPGA e a aceleração da aplicação final como um todo.

1.3 Objetivo

Esse trabalho tem como objetivo propor implementações otimizadas da consulta em um banco de DNA de grandes dimensões utilizando as arquiteturas dos GPPs, GPUs e FPGAs. Este banco de DNA forense tem características similares ao do CODIS.

1.4 Contribuições

Com este trabalho, espera-se contribuir no desenvolvimento de aplicações de bioinformática para o alinhamento de múltiplas sequências de DNA para as arquiteturas alvo escolhidas: GPPs, GPU e FPGA. Na plataforma de GPPs foram explorados os paralelismos de threads e o SIMD. Para as GPUs diversas configurações foram utilizadas incluindo o paralelismo SIMD com o uso dos tipos vetoriais de OpenCL. Em FPGA uma arquitetura de hardware foi desenvolvida para estressar essa plataforma para se obter o maior desempenho

possível.

1.5 Estrutura do trabalho

O capítulo 2 apresenta as bases conceituais para comparações entre duas cadeias de DNA, apresentando o algoritmo de Needleman-Wunsch, Smith-Waterman e o BLAST. No capítulo 3 é apresentada arquitetura das GPUs para a aceleração de aplicações. No capítulo 4 aborda-se a motivação da utilização dos FPGAs em sistemas de alto desempenho. No capítulo 5 descreve-se a revisão da literatura dividida em três tipos: trabalhos de implementação em processadores de propósito geral, trabalhos de implementação em GPUs e trabalhos de implementação em FPGA. Já no capítulo 6 são mostradas as implementações propostas para Processadores de propósito geral (GPPs). No Capítulo 7 sequem as implementações propostas em OpenCL. No Capítulo 8 são detalhadas a arquitetura proposta para FPGA e sua implementação na aplicação forense. Já no Capítulo 9 os resultados das implementações propostas são comparados entre si e com os resultados da literatura. Por fim em 10 e 11 apresentam-se as conclusões finais e as atividades futuras a serem desenvolvidas, respectivamente.

2

Comparando sequências de DNA

Dadas duas cadeias $A = (a_1, a_2, \cdots, a_n)$ e $B = (b_1, b_2, \cdots, b_m)$ de DNA de tamanho n e m, com alfabeto $\Sigma = \{a, c, g, t\}$. Um alinhamento entre essas duas cadeias é a operação que conceitualmente segue casando regiões similares alinhando um caractere de A com um caractere em B.

Assim o alinhamento de DNA resultante R é uma tupla $R = < A', B' >$, em que $A' = (a'_1, a'_2, \cdots, a'_l)$ e $B' = (b'_1, b'_2, \cdots, b'_l)$ tem alfabeto $\Sigma' = \{a, c, t, g\} \cup \{-\}$ que corresponde aos nucleotídeos mais o símbolo de vazio ' $-$ '. Dessa forma o alinhamento $R = < a_i, b_i >$, pode gerar os resultados:

$$R < a_i, b_i >= \begin{cases} se\ a_i = b_i, acerto \\ se\ a_i \neq b_i, troca \\ se\ a_i = \text{'}-\text{'}, inserção \\ se\ b_i = \text{'}-\text{'}, remoção \end{cases}$$

Uma inserção ocorre quando a segunda sequência é alinhada com um espaço. De outra forma, quando a primeira sequência é alinhada com um espaço, uma deleção ocorre. A cadeia de referência é sempre a cadeia A que é chamada a cadeia original. Em relação à cadeia A se busca uma relação evolutiva, com o

menor número de operações de inserção, remoção e troca, para se chegar à cadeia B.

Adicionalmente precisa-se seguir as seguintes regras:

- Os caracteres de A e B precisam ser alinhados na mesma ordem em que eles aparecem nas sequências. Ou seja, os símbolos não podem ser permutados.
- Um símbolo pode se alinhado com um vazio, representado por um espaço $' - '$.
- Dois vazios não podem ser alinhados.

Dessa forma, acertos (matches) devem ser recompensados e ter pesos maiores que zero. Trocas (mismatches) devem ser penalizadas com pesos menores que zero, porque representam mutações e biologicamente não ocorrem com grande frequência. As inserções e remoções precisam ser mais fortemente penalizadas porque ocorrem biologicamente com muito menos frequência e normalmente implicam em mudanças estruturais no DNA.

2.1 Alinhamento Ótimo Global de DNA

O número de possíveis alinhamentos globais entre duas sequências de tamanho n e m é dado por:

$$\binom{n+m}{m} = \frac{(n+m)!}{(m!)^2} \cong \frac{2^{2n}}{\sqrt{\pi n}}.$$

Dessa maneira, a estratégia de força bruta para resolver esse problema com duas sequências de 250 nucleotídeos realizaria 10^{149} alinhamentos para determinar o melhor alinhamento (Setubal e Meidanis 1997).

Por exemplo, os três exemplos na Figura 7 são tentativas de se encontrar o melhor alinhamento dadas as sequências $\boldsymbol{aagt}$ e $\boldsymbol{accgt}$.

Figura 7 Três exemplos de alinhamentos para as sequências $aagt$ e $accgt$

$a\ a\ g\ t\ -$ $\vert$ $a\ c\ c\ g\ t$	$-\ a\ a\ g\ t$ $\vert\ \vert$ $a\ c\ c\ g\ t$	$a\ a\ -\ g\ t$ $\vert\ \vert\ \vert$ $a\ c\ c\ g\ t$

Uma maneira não exaustiva de reduzir o número exponencial de possibilidades que precisam ser consideradas e, ainda assim, garantir que a solução ótima será encontrada foi proposto por Needleman e Wunsch (Needleman e Wunsch 1970). O algoritmo de Needleman-Wunsh (NW) usa programação dinâmica para quebrar o problema em subproblemas menores. Então se resolve os subproblemas de forma ótima e finalmente utilizam-se as soluções dos subproblemas para construir uma solução ótima para o problema original (Needleman e Wunsch 1970).

Segundo Ian Parberry (Parberry 1995), "a programação dinâmica é um nome elegante para a recursão com uma tabela. Ao invés de resolver subproblemas recursivamente, os resolva sequencialmente e os armazene em uma tabela". O autor continua: "a programação dinâmica é particularmente útil em problemas nos quais a estratégia dividir para conquistar é usada para resolver um exponencial número de subproblemas. Nesse caso, faz sentido se computar cada solução na primeira vez e armazená-la numa tabela para uso futuro, ao invés de recomputá-la recursivamente toda vez em que ela for necessária".

Dessa forma, o algoritmo reduz a complexidade de tempo e de espaço do problema original de exponencial para quadrática $O(nm)$, para duas sequências de tamanho n e m respectivamente. Essa redução na complexidade de tempo e espaço faz do algoritmo de NW um algoritmo ótimo. Outra característica é o fato dele não utilizar heurísticas para resolver o problema, uma vez que algoritmos heurísticos introduzem imprecisões no processo a fim de obter menor complexidade.

Iterativamente, constrói-se a matriz M_{nxm}, através da Equação 2.1, usando as posições dos nucleotídeos na cadeia, de cada sequência, como índices da matriz.

$$\boldsymbol{M}(i,j) = max\begin{cases}\boldsymbol{M}(i-1,j-1) + \boldsymbol{S}(x_i, y_i) \\ \boldsymbol{M}(i-1,j) + g_1 \\ \boldsymbol{M}(i,j-1) + g_2\end{cases} \quad (2.1)$$

$$\boldsymbol{S}(x_i, y_i) = \begin{cases}\delta, & Se\ x_i = y_i \\ -\delta, & Se\ x_i \neq y_i\end{cases} \quad (2.2)$$

Em que $\boldsymbol{M}(i,j)$ é a matriz nxm de escores; $\boldsymbol{S}(x_i, y_i)$ é o escore de substituição para os nucleotídeos i e j; e g_1 e g_2 são chamados de penalidades de *gap*. A penalidade de gap é o peso (normalmente negativo) associado com deleções e inserções de nucleotídeos durante o processo de alinhamento. Um exemplo desses parâmetros definiria $g_1 = g_2 = -1$. Já $\boldsymbol{S}(x_i, y_i)$ é mostrado na Equação 2.2, em que δ é o custo de substituição.

O algoritmo de Needleman-Wunsh segue três passos:

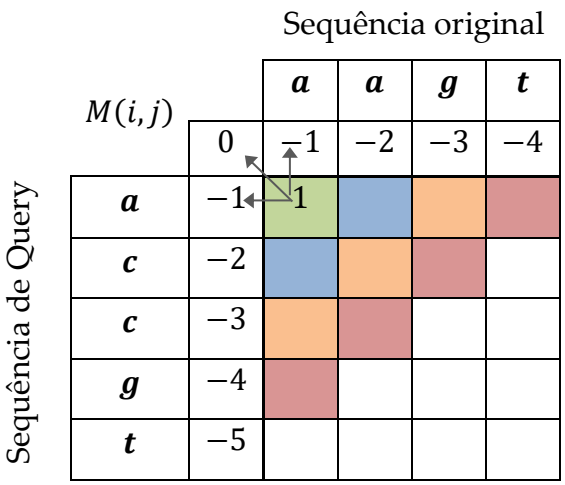
1. Inicialização da matriz de escores.
2. Computação das matrizes de *traceback* e de escores.
3. Obtenção do alinhamento resultante a partir da matriz de *traceback*.

No passo 1 as bordas da matriz são preenchidas iniciando-se com $\boldsymbol{M}(0,0) = 0$ e adicionando-se a partir daí a penalidade de gap g ao longo da linha 0 e da coluna 0 ($\boldsymbol{M}(i,0) = i * g_1$ e $\boldsymbol{M}(0,j) = i * g_2$). Esse passo é necessário por causa da dependência de dados do algoritmo, como mostrada na Equação 2.1, em que cada ponto necessita do resultado do ponto à esquerda, acima e da diagonal esquerda superior. A partir desse ponto o segundo passo do algoritmo pode iniciar como mostra a Figura 8. Na Figura 8 também pode ser vista a dependência de dados do algoritmo de NW, em que cada cor representa as células com dependência de dados resolvida no mesmo instante. Em outras palavras, as células na anti-diagonal podem ser computadas ao mesmo tempo.

Por essa característica, o algoritmo pode ser classificado com um problema de frente de onda (wave front problem).

A sequência na vertical é chamada de sequência de query (ou sequência de consulta) $accgt$ e a na horizontal a sequência original $aagt$. Note que no caso do exemplo da Figura 8 tem-se um acerto $(a - a)$ então $S(1,1) = 1$ e o operador $max(0 + 1, -1 - 1, -1 - 1)$, definido na Equação 2.1, resulta em 1.

Figura 8 Passo 1 do algoritmo de Needleman-Wunsch



Na Figura 9 pode-se ver que os dois primeiros laços FOR são responsáveis pelo passo 1 e os dois laços FOR seguintes realizam a computação da matriz de escores M.

A Figura 10(a) ilustra a matriz de escores resultante do exemplo dado acima. As células destacadas mostram o passo de traceback, iniciando por $M(5,4)$ (o canto inferior direito) e caminhando na matriz a escolha do máximo local a cada passo até chegar no canto superior esquerdo da matriz.

Figura 9 Pseudocódigo do algoritmo de Needleman-Wunsch

```
Similaridade(X,Y):
        for i = 0,...,m:  M[i,0] = i*g;
        for j = 1,...,n:   M[0,j] = j*g;
        for i = 1,...,m:
                for j = 1,...,n:
                        M[i,j] = max( M[i-1,j-1] + S(X[i],Y[j]),
                                      M[i-1,j] + G,
                                      M[i,j-1] + G)
                end
```

Figura 10 Resultado do alinhamento para a query $accgt$ e a sequência $aagt$

$M(i,j)$		$\boldsymbol{a}$	$\boldsymbol{a}$	$\boldsymbol{g}$	$\boldsymbol{t}$
	0	-1	-2	-3	-4
$\boldsymbol{a}$	-1	1	0	-1	-2
$\boldsymbol{c}$	-2	0	0	-1	-2
$\boldsymbol{c}$	-3	-1	-1	-1	-2
$\boldsymbol{g}$	-4	-2	-2	0	-1
$\boldsymbol{t}$	-5	-3	-3	-1	1

(a)

$$\begin{array}{ccccc} a & a & - & g & t \\ | & & & | & | \\ a & c & c & g & t \end{array}$$

(b)

Na Figura 10(b) pode ser visto o alinhamento global obtido no exemplo da Figura 10(a).

Primeiro ocorre um acerto $(a-a)$, depois uma troca $(a-c)$, então uma inserção de (c) que não estava na cadeira original, mas foi inserida na query e por fim ocorrem dois acertos $(gt-gt)$.

Com o algoritmo de NW é possível se obter vários alinhamentos ótimos para um par de cadeias e todos são corretos (Setubal e Meidanis 1997). Por

exemplo, na Figura 11(a) é ilustrado outro possível alinhamento com o algoritmo de NW.

Figura 11 Resultado alternativo para o alinhamento das sequências $accgt$ and $aagt$

$M(i,j)$		$\boldsymbol{a}$	$\boldsymbol{a}$	$\boldsymbol{g}$	$\boldsymbol{t}$
	0	−1	−2	−3	−4
$\boldsymbol{a}$	−1	1	0	−1	−2
$\boldsymbol{c}$	−2	0	0	−1	−2
$\boldsymbol{c}$	−3	−1	−1	−1	−2
$\boldsymbol{g}$	−4	−2	−2	0	−1
$\boldsymbol{t}$	−5	−3	−3	−1	1

(a)

$$\begin{array}{cccccc} a & - & a & g & t \\ | & & & | & | \\ a & c & c & g & t \end{array}$$

(b)

Nesse caso, a célula ressaltada mostra a alternativa pela troca ($a - c$) ao invés da inserção no mesmo ponto com igual escore zero.

Uma vez obtido o alinhamento, uma estratégia de escore, para medir a similaridade entre as duas sequências precisa ser escolhida.

$$Escore = \sum_{i=1}^{l} escore(a_i, b_i) \tag{2.3}$$

$$escore(x,y) = \begin{cases} \alpha & x = y \\ \beta & x \neq y \\ \gamma & x = - \; ou \; y = - \end{cases} \tag{2.4}$$

As Equações 2.3 e 2.4 definem uma estratégia de escore para um dado alinhamento. Na Equação 2.3 é definido que o escore final do alinhamento é o somatório dos escores individuais. Em que o tamanho do alinhamento l é definido no intervalo $\max(n,m) \leq l \leq n + m$, para as sequências $A = (a_1, a_2, \cdots, a_{\mathrm{n}})$ e $B = (b_1, b_2, \cdots, b_{\mathrm{m}})$. Na Equação 2.4 são definidos os pesos α para um acerto, em que os nucleotídeos são iguais, β para uma troca, e γ para uma remoção ou inserção.

Por exemplo, pode-se definir os pesos da equação abaixo. Utilizando-se esses pesos, na Figura 12 observam-se os cálculos das similaridades para os alinhamentos da

Figura 10(a) e da Figura 11(a).

$$escore(x,y) = \begin{cases} 3 & x = y \\ -1 & x \neq y \\ -5 & x = - \; ou \; y = - \end{cases}$$

Observa-se que o resultado dos escores não mudaram para os dois alinhamentos possíveis obtidos com o algoritmo de NW para as cadeias $accgt$ e $aagt$.

Figura 12 Escore de similaridade para dois possíveis alinhamentos das sequências $accgt$ and $aagt$

$$\begin{array}{ccccc} a & a & - & g & t \\ | & & & | & | \\ a & c & c & g & t \end{array}$$

Escore de Similaridade

$$3 - 1 - 5 + 3 + 3 = 3$$

$$\begin{array}{ccccc} a & - & a & g & t \\ | & & & | & | \\ a & c & c & g & t \end{array}$$

Escore de Similaridade

$$3 - 5 - 1 + 3 + 3 = 3$$

2.2 Alinhamento Ótimo Local de DNA

O alinhamento local é uma variação do algoritmo de Needleman-Wunsch que encontra regiões locais de alto nível de similaridade proposto por Smith-Waterman (Smith e Waterman 1981). Os alinhamentos locais visam buscar subsequências relativamente conservadas entre duas sequências. Ou seja, na comparação com duas sequências cadeias $A = (a_1, a_2, \cdots, a_{\mathrm{n}})$ e $B = (b_1, b_2, \cdots, b_{\mathrm{m}})$ de DNA de tamanho n e m, com alfabeto $\Sigma = \{a, c, g, t\}$, pretende-se encontrar as sub-regiões com menor número de alterações evolutivas entre A e B, incluindo as operações de acerto, troca, remoção e inserção.

O alinhamento de DNA local resultante R é um conjunto $R = \{ < A'_1, B'_1 >, < A'_2, B'_2 >, < A'_j, B'_j > \}$, em que cada $A'_i = (a'_1, a'_2, \cdots, a'_k)$ e $B'_i = (b'_1, b'_2, \cdots, b'_k)$ tem alfabeto $\Sigma = \{a, c, t, g\} \cup \{-\}$ que corresponde aos nucleotídeos mais o símbolo de vazio ' $-$ ', e cada $< A'_i, B'_i >$ é uma sub-cadeia com escore acima do limiar θ. Dessa forma no alinhamento local, um único alinhamento pode resultar em várias subsequências com relevante similaridade.

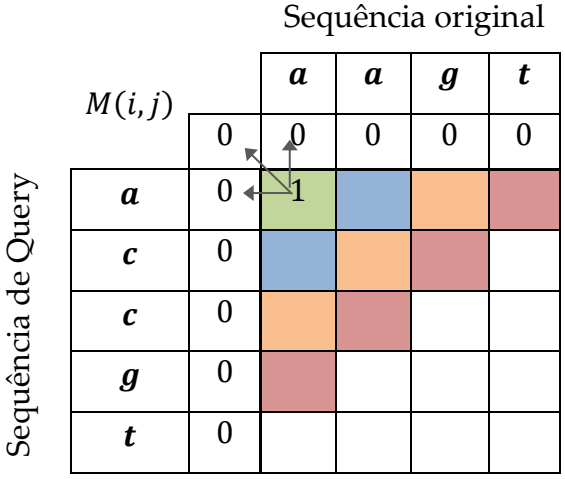
No algoritmo de Smith-Waterman a primeira linha e a primeira coluna da matriz são inicializadas com zeros (vide Figura 13). A matriz $a(i,j)$, construída seguindo a Equação 2.5, contém apenas escores maiores ou iguais a zero. Na Figura 13 se observa que a dependência de dados e a caraterística de frente de onda se dá de forma igual ao é igual do algoritmo de NW.

Como pode ser visto na Equação 2.5, o algoritmo de Smith-Waterman segue a mesma dependência de dados do seu antecessor, dependendo de três valores previamente calculados (à esquerda, acima e na diagonal superior esquerda).

$$a(i,j) = max \begin{cases} a(i,j-1) + G \\ a(i-1,j-1) + S(i,j) \\ a(i-1,j) + G \\ 0 \end{cases} \tag{2.5}$$

Definindo-se $S(i,j)$ como na Equação 2.2 e a penalidade de gap $G = -1$, se obtém o primeiro valor da matriz através do operador $max(0-1, 0+1, 0-1, 0)$ que resulta em 1.

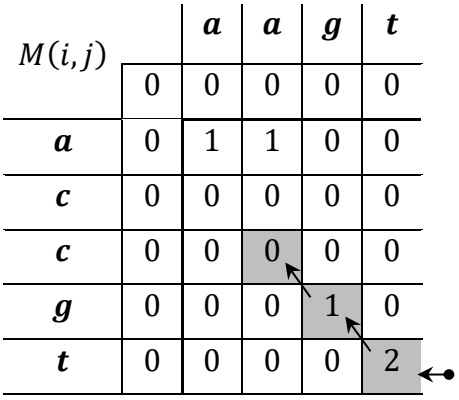
Figura 13 Inicialização da matriz no algoritmo de Smith-Waterman



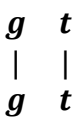
No algoritmo de Smith-Waterman após a computação da matriz de escores, o traceback inicia a partir da posição com máximo escore e segue até encontrar um escore zero.

A matriz de escores resultante pode ser vista na Figura 14(a). Nota-se também que o traceback iniciou a partir da posição de maior escore $= 2$. O resultado do alinhamento local, com limiar de escore $\theta \geq 2$ (por exemplo), é ilustrado na Figura 14(b), em que foi encontrada uma única subsequência de alta similaridade em ocorrem dois acertos $(gt - gt)$.

Figura 14 Resultado do alinhamento local para a query $accgt$ e a sequência $aagt$



(a)



(b)

2.3 Alinhamento Heurístico Local de DNA

O custo computacional dos algoritmos de alinhamento ótimos é alto. Por causa desse custo computacional, os algoritmos de computação dinâmica são muito custosos para o problema de busca em grandes bancos de sequências sem o uso de supercomputadores ou de hardware de propósito especial (Altschu, et al. 1990).

No problema de busca em bancos de dados de DNA, normalmente se usa alinhamentos locais visto que esses bancos armazenam DNAs inteiros de indivíduos e as consultas geralmente representam regiões codificantes do DNA, chamados de genes (Altschu, et al. 1990).

O algoritmo BLAST (Basic local Alignment Search Tool) (Altschu, et al. 1990) é um dos mais utilizados na área da bioinformática para a busca por sequências em uma base de dados. Ele utiliza uma heurística para acelerar a busca por regiões locais de alta similaridade entre sequências.

O BLAST retorna uma lista de pares de segmentos com alta similaridade. A ideia básica em que o BLAST foi construído é que em bons alinhamentos locais existem bons alinhamentos sem gaps (buracos ou espaços). Um alinhamento sem buracos nada mais é do que a redução das combinações, reduzindo o alfabeto do alinhamento para apenas o alfabeto dos nucleotídeos $\Sigma = \{a, c, t, g\}$, o que resulta em dois resultados possíveis na comparação: acerto ou troca. Essa operação é feita apenas comparando nucleotídeo a nucleotídeo em sequências pequenas, o que pode ser feito em tempo linear $O(n)$. Após a fase de se encontrar as microrregiões iguais, processa-se um crescimento dessas regiões para se encontrar os alinhamentos locais. Esse crescimento, agora é feito utilizando-se a possibilidade de espaços. Esta é a razão pela qual o BLAST é mais rápido do que os algoritmos ótimos (Setubal e Meidanis 1997).

Com o algoritmo do BLAST consegue-se aumentar o desempenho por se diminuir o espaço de busca ou número de comparações que são feitas. No caso dos algoritmos ótimos cada nucleotídeo é comparado com todos da outra sequência (complexidade $O(n^2)$, o que não acontece no BLAST.

No BLAST uma pequena "palavra" de tamanho w é usada para criar os alinhamentos "semente". Tomando como exemplo a partir da query, aqui chamada de *p-word,* $accgt$ todas as palavras de tamanho $w = 3$ obtidas da query são mostradas abaixo:

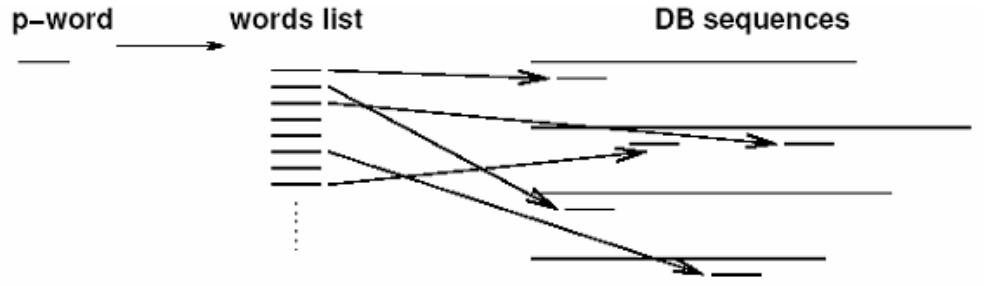
$$\begin{array}{ccccc} \boldsymbol{a} & \boldsymbol{c} & \boldsymbol{c} & \boldsymbol{g} & \boldsymbol{t} \\ \boldsymbol{a} & \boldsymbol{c} & \boldsymbol{c} & & \\ & \boldsymbol{c} & \boldsymbol{c} & \boldsymbol{g} & \\ & & \boldsymbol{c} & \boldsymbol{g} & \boldsymbol{t} \end{array}$$

O algoritmo então utiliza essas palavras de tamanho, no exemplo $w = 3$, para a

lista de palavras pré-selecionadas. Na etapa seguinte, como mostra a Figura 16 utiliza-se uma medida de escore para controlar o crescimento das regiões, a fim de se evitar a deterioração do alinhamento. Quando o escore do alinhamento cai abaixo do limiar determinado T, o crescimento precisa parar. A Figura 2.10 ilustra o passo 3 do algoritmo.

No passo 2, utiliza-se o operador igualdade para comparar cada palavra da lista é comparada, nucleotídeo a nucleotídeo, com as sequências do banco. Nessa fase se buscam acertos exatos das *p-words* dentro do banco.

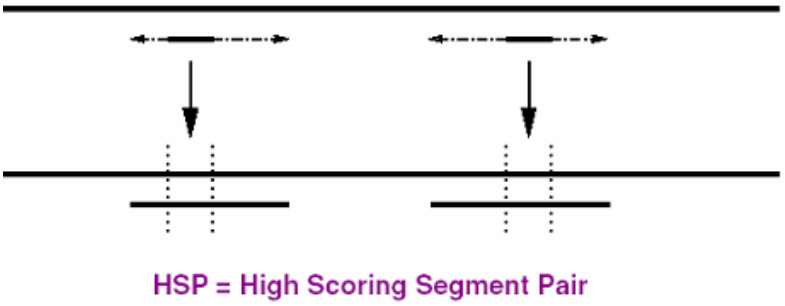
Figura 15 Passo 2 do algoritmo BLAST (Bordoli 2003)



No passo 3, as regiões que resultaram em igualdade com o banco, na fase 2, são as regiões de crescimento da próxima fase. O crescimento ocorre para ambos os lados, a partir da região de igualdade.

Utiliza-se uma medida de escore para controlar o crescimento das regiões, a fim de se evitar a deterioração do alinhamento. Quando o escore do alinhamento cai abaixo do limiar determinado T, o crescimento precisa parar. A Figura 16 ilustra o passo 3 do algoritmo.

Figura 16 Passo 3 do algoritmo: extensão dos alinhamentos no algoritmo BLAST (Bordoli 2003)

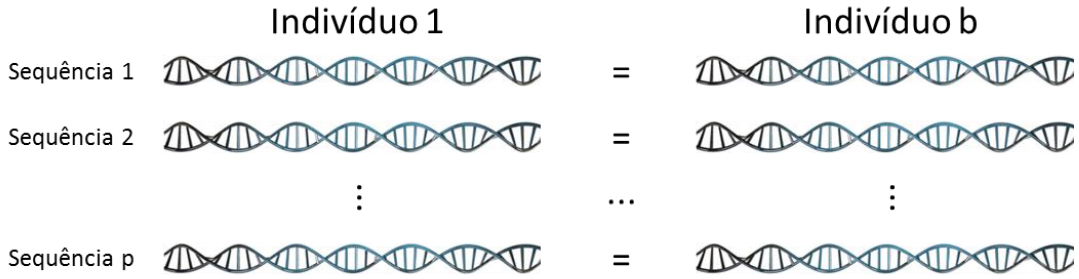


Os segmentos com maiores escores HSP (High Scoring Segment Pair) são retornados como o resultado do algoritmo.

2.4 Aplicação ao escopo forense

Em relação ao problema da aplicação forense do banco nacional de criminosos, cada sequência de DNA precisa ser comparada com seu par correspondente entre todos os indivíduos do banco de DNA. A Figura 17 ilustra esse esquema de comparação

Figura 17 Esquema da comparação entre sequências de indivíduos no banco forense



A comparação entre duas sequências de DNA é feita através do alinhamento entre elas. Para o problema forense, objetivo desta Tese, o algoritmo que se enquadra é o alinhamento global, visto que ele representa uma medida de similaridade global entre duas sequências de DNA.

3

Graphics Processing Unit (GPU)

Neste capítulo são apresentados detalhes da arquitetura das GPUs e do seu uso em aplicações visando a aceleração na execução de problemas computacionais.

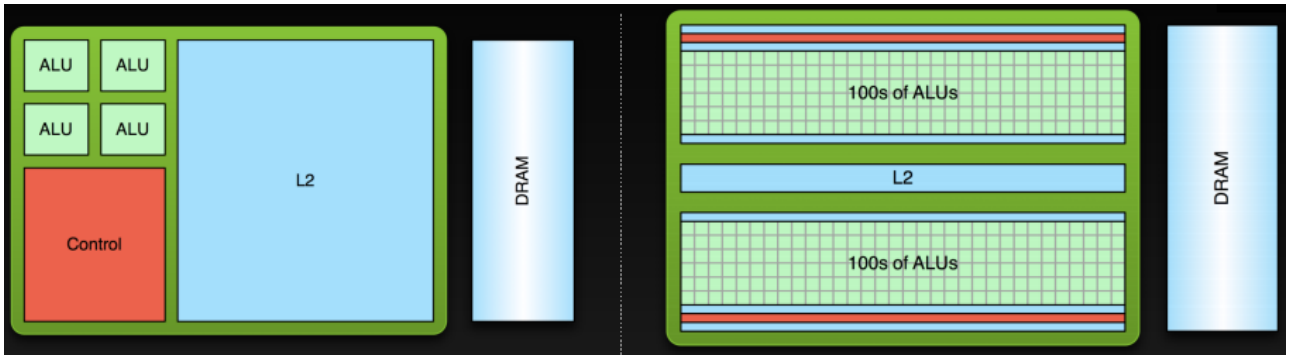
3.1 Arquitetura das GPUs

Enquanto a arquitetura dos processadores de propósito geral evoluiu explorando, ao longo de muitos anos, o paralelismo no nível de instruções (ILP). Depois que essa estratégia não foi mais suficiente para entregar o ganho de desempenho nos parâmetros da lei de Moore, entrou-se na era dos multi-cores. Com a arquitetura multi-core o paralelismo de threads real pode ser atingido com execução de simultâneas tarefas em núcleos diferentes.

Do lado das GPUs a estratégia foi o paralelismo vetorial através da arquitetura chamada de SIMD (Single Instruction Multiple Data). Por esse paradigma, uma única instrução é executada por milhares de unidades de execução ou núcleos. Os núcleos das GPUs foram criados para serem bem mais simples do que os núcleos das CPUs e para se conseguir fabricar em silício um grande número dessas unidades a um custo compatível com aplicações de computação pessoal.

A Figura 18 ilustra o contraste entre as duas abordagens. No lado esquerdo, a arquitetura das CPUs, e no lado direito a das GPUs.

Figura 18 Arquitetura das CPUs (esquerda) vs a das GPUs (direita) (Clark s.d.)



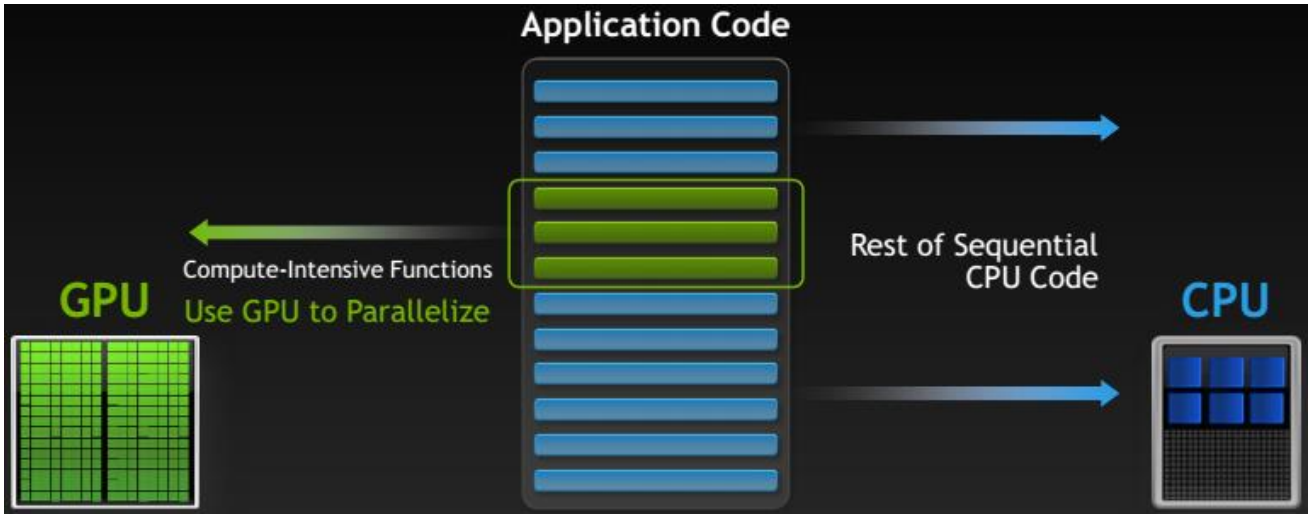
Ainda na Figura 18, pode-se ver as unidades de execução da CPU (representadas pelas ALUs) e uma grande unidade de controle que permite otimizações na execução de código como a execução fora de ordem, controle e predição de desvios e execução especulativa. As memórias caches também são mais robustas nas CPUs para melhor aproveitar o princípio da localidade e diminuir acessos desnecessários à memória principal.

Já do lado da GPU, as unidades de controle são bem mais simples, assim como as unidades de execução visando a alta vazão na computação de dados o que resulta em mais área de silício dedicada ao processamento propriamente dito.

Um limitante importante às duas arquiteturas é o consumo de potência. Nas CPUs, em que as frequências máximas giram em torno de 3,5 a 4 GHz, a potência dissipada é reduzida desligando-se os núcleos ociosos e reduzindo a frequência sempre que possível. No caso das GPUs, o consumo de potência cresce bastante pelo grande número de núcleos. Por isso, a frequência máxima de operação dos núcleos da GPU gira em torno de 1GHz.

Dessa forma, um código inerentemente sequencial pode se valer do paralelismo de instruções da CPU, enquanto uma multiplicação de matrizes, por exemplo, pode ser acelerada utilizando uma GPU. A Figura 19 apresenta um diagrama que ilustra a aceleração de um trecho de código acelerado em GPU.

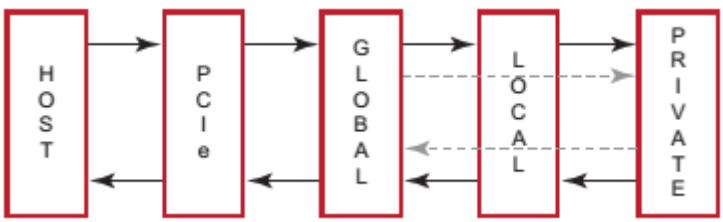
Figura 19 Acelerando aplicações com GPUs (Clark s.d.)



Assim, a GPU é utilizada como um coprocessador na plataforma incluindo uma CPU como HOST. Em determinado momento do código, um trecho de código especificado para se valer do paralelismo da GPU é invocado a partir do Host, após as transferências de memória via barramento PCI de forma explícita. Ao final do processamento da GPU, também explicitamente, ocorre a transferência para a memória principal do sistema Host dos resultados da computação.

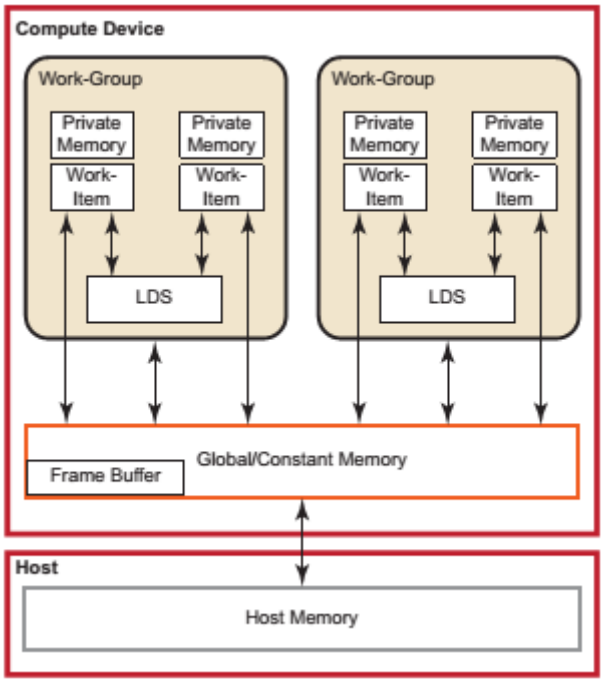
A Figura 20 mostra as transferências de dados entre as unidades de memória do Host até a GPU. A memória global é a memória mais abundante e de acesso mais lento, a memória local é compartilhada entre as threads dentro de um mesmo grupo de threads, e a memória privada da thread é a mais rápida e em menor quantidade. Ainda existem, em pequena quantidade, os registradores associados às threads.

Figura 20 Fluxo de dados na GPU (AMD APP SDK OpenCL User Guide 2015)



Como mostra a Figura 21, um grupo de threads forma o que se chama de Work-Group e uma thread é o mesmo que um work-item. A estrutura chamada LDS é a memória local compartilhada entre os work-items, que é uma ordem de grandeza mais rápida que a memória global.

Figura 21 Hierarquia de memória da GPU (AMD APP SDK OpenCL User Guide 2015)



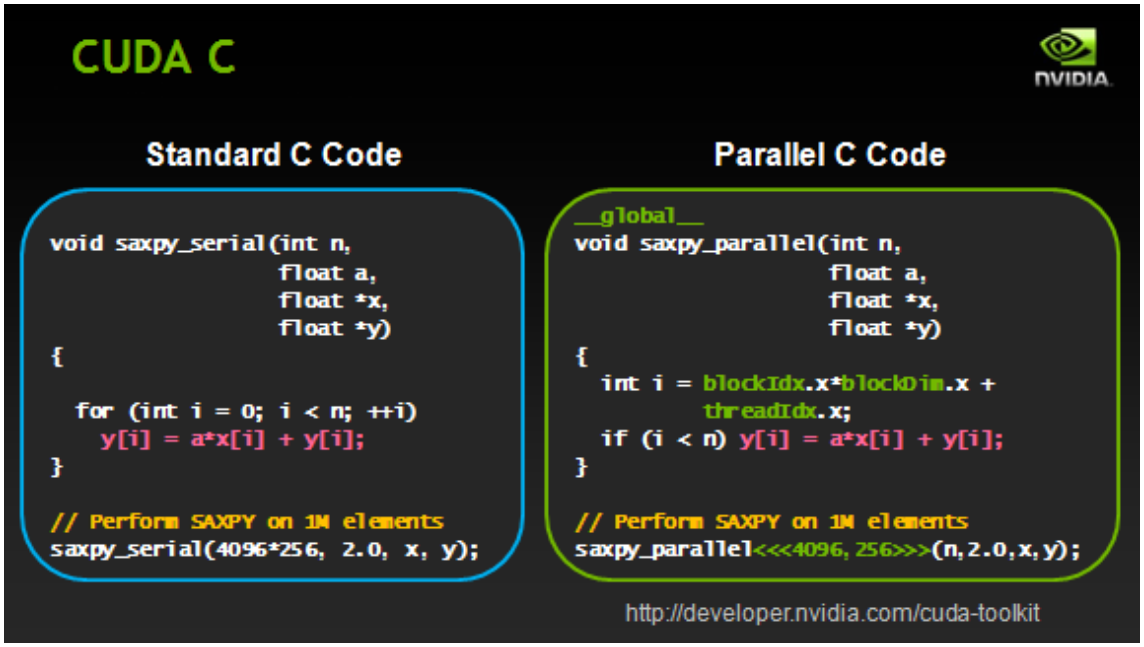
3.2 Programação em GPUs

A Nvidia apresenta em 2006 uma plataforma que permitiu o desenvolvimento de aplicações em alto nível para utilização da plataforma das GPUs para computação, chamada de Cuda (Compute Unified Device Architecture). Antes disso, os desenvolvedores precisavam descrever suas aplicações em assembly da GPU.

A Figura 22 exibe um exemplo de uma aplicação de multiplicação do vetor x pelo escalar a, a soma do resultado com o vetor y e o resultado final sendo armazenado em y. Do lado esquerdo da Figura 22 a aplicação é descrita em linguagem C e do lado direito em um kernel de CUDA. Dentro do kernel as estruturas blockIdx, blockDim e threadIdx armazenam a informação da

distribuição da carga nos núcleos da GPU, de forma que se pode, nesse caso, realizar em cada thread o endereçamento na memória global de uma única posição dos vetores, a computação e a escrita do resultado de forma coerente.

Figura 22 Adição de vetores em C e em CUDA



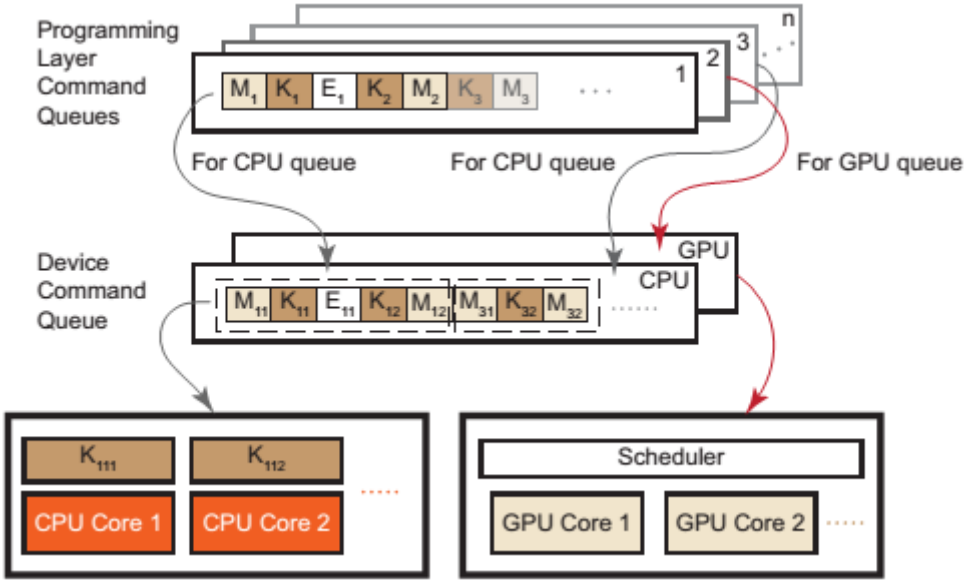
Na invocação do kernel (utilizando o operador <<< >>>) são passadas as informações da dimensão do bloco (que pode ser 3d, 2d ou 1d) e do número de threads por bloco.

Cuda tornou-se padrão como linguagem para desenvolvimento para GPUs, mas sempre se limitou às GPUs da Nvidia. Até que no ano de 2008 foi lançada a linguagem Opencl (Open Computing Language) com o intuito mais amplo de ser uma linguagem para arquiteturas heterogêneas, contendo CPUs, GPUs e outros aceleradores.

Em OpenCL o lançamento dos kernels é feito através de filas de execução de tarefas para uma determinada unidade de computação (GPU, CPU, etc.). As filas servem também para o escalonamento de transferências de leitura e escrita da memória do acelerador e também para a configuração de parâmetros do kernel. A Figura 23 mostra essa dinâmica.

A linguagem também possui uma sequência de comandos de inicialização e consulta aos dispositivos para se descobrir, em tempo de execução, por exemplo a quantidade de memória disponível, a quantidade de elementos de computação, etc. Isso serve para o gerenciamento do contexto que pode ser heterogêneo, permitindo assim, o desenvolvimento de aplicações que se adaptem aos recursos disponíveis.

Figura 23 Esquema do lançamento de kernels em OpenCL



4

Field Progamable Gate Array (FPGA)

Nas próximas seções são descritos conceitos referentes aos FPGAs, o conceito de HDL, o fluxo de projeto para os mesmos e sua aplicação em sistemas de computação de alto desempenho.

4.1 FPGA

O FPGA foi criado pela Xilinx, e teve o seu lançamento em 1985 trazendo a motivação de ser um hardware configurável, ou seja, sendo programado de acordo com as aplicações e em tempo de execução (Rocha 2010).

Desde então, a utilização desses dispositivos em diversas áreas de aplicação tem crescido e se consolidado (Gokhale e Graham 2005). Os FPGAs são utilizados como forma de validação rápida de sistemas digitais em projetos visando a implementação em ASICs (*Aplication Specifc Integrated Circuits*).

Entretanto, em alguns casos, as implementações em FPGA apresentam ganhos de desempenho em relação aos seus pares em software para processadores de propósito geral (Gokhale e Graham 2005) e os FPGAs podem ser utilizados como plataforma alvo.

Em outras tantas aplicações na área de processamento de imagens e

processamento matricial 2D e 3D, a arquitetura customizada das GPUs para o processamento vetorial SIMD (Single Instruction Multiple Data), com hierarquia de memória otimizada e com largura de banda de memória que atinge[1] os 336 GB/s ganham larga vantagem no acesso à memória em comparação com os FPGAs.

No mundo real, mesmo nas aplicações em que as GPUs batem os FPGAs também pode-se buscar nichos favoráveis aos últimos. Um fator corrente é o uso da métrica desempenho/watt que pode beneficiar os FPGAs pelo seu menor consumo de potência em relação às GPUs.

De fato, o quesito potência tem sido um importante empecilho no crescimento do desempenho das GPUs que limitam as frequências de operação por volta de 1GHz a fim de evitar o aumento da dissipação de potência em energia térmica, enquanto as frequências máximas de operação dos FPGAs são tipicamente por volta de 300 a 400MHz.

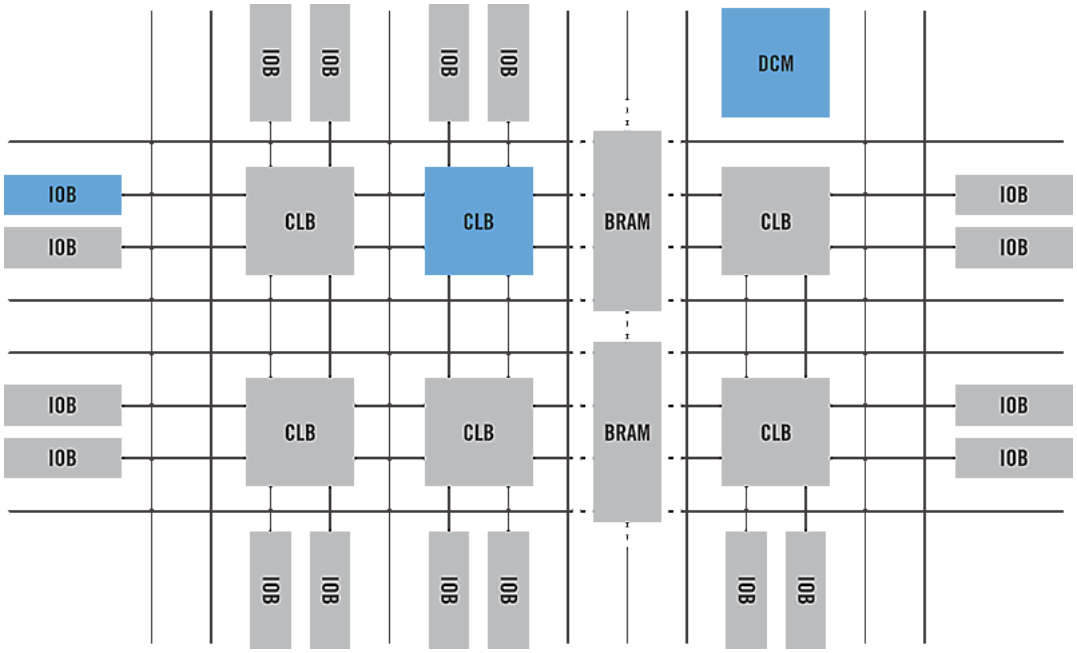
No longo prazo, a métrica desempenho/watt pode também gerar um ganho monetário favorável aos FPGAs que possa justificar o maior custo de implantação e de projeto dos mesmos.

A arquitetura interna do FPGA segue a organização de uma matriz bi-dimensional de circuitos lógicos distribuídos no chip e interconectados por trilhas ou barramentos.

No detalhe da Figura 24 pode-se ver os CLB (*Configuration Logical Blocks*) que são os blocos operacionais do FPGA. As com vias verticais e horizontais que interconectam todos os recursos. Os IOB (*Input Output Block*) que ficam na periferia realizando a comunicação através dos pinos do FPGA. Os blocos de BRAM (Block RAM) que são blocos de memória RAM espalhados pelo FPGA para facilitar o acesso às unidades de armazenamento interno.

[1] http://www.nvidia.com/gtx-700-graphics-cards/gtx-780ti/

Figura 24 Estrutura interna do FPGA (Xilinx s.d.)



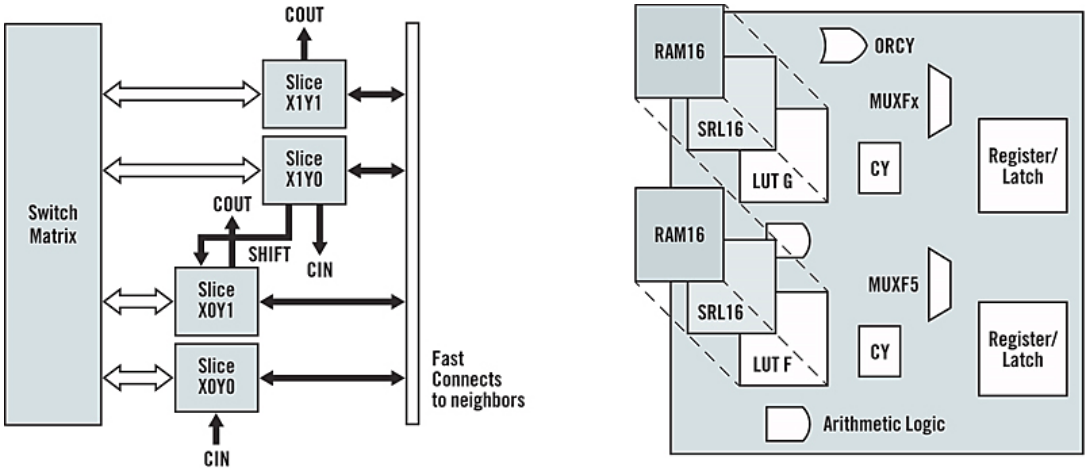
A funcionalidade implementada por esses circuitos é configurada via software pelo engenheiro de hardware. A programação vai determinar quais blocos serão utilizados, qual sua funcionalidade e quais vias serão ativadas para interconectar os blocos.

A Figura 25 representa a estrutura interna do CLB da arquitetura Xilinx. Ele é composto por uma matriz de conexões (*Switch Matrix*) e por *Slices* que é mostrado em detalhe à direita na figura. O Slice contém os blocos lógicos e de armazenamento (MUX, pequenas RAMs, *flip-flops, registradores*, *look-up tables*, etc).

A concepção do FPGA e das estruturas aqui apresentadas demonstram que o mesmo foi planejado para possuir um hardware com flexibilidade, reusável e programável.

Para programar esses dispositivos são utilizadas as linguagens de descrição de hardware (HDL). Essas linguagens apresentam estruturas específicas que facilitam o projeto dos circuitos pretendidos.

Figura 25 Estrutura interna do CLB (Xilinx s.d.)



Nas suas bibliotecas padrão estão definidas estruturas incomuns às linguagens de propósito geral como por exemplo registradores, fios, tipos de dados orientados ao nível de bits, eventos baseados na subida ou descida do sinal de clock, etc.

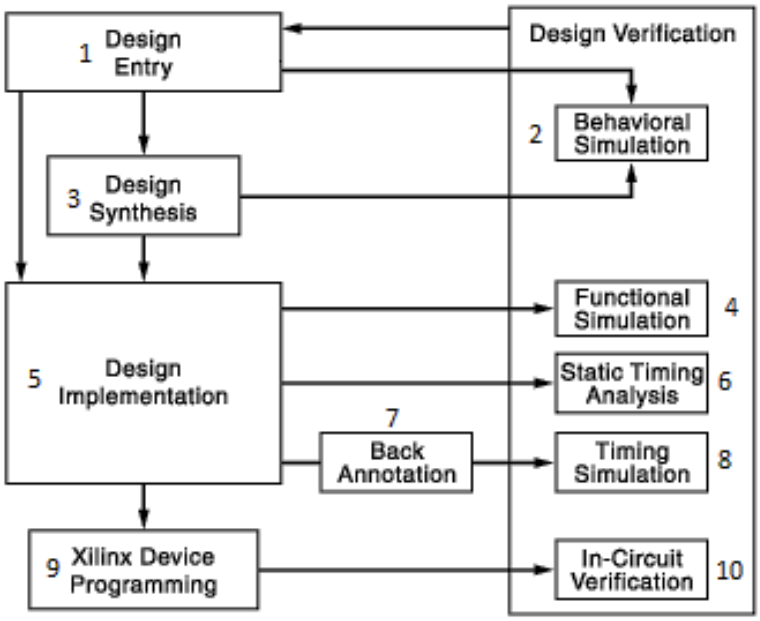
O projeto do circuito começa com a concepção do projeto e arquitetura do sistema. Normalmente um sistema complexo é dividido em sub-módulos que compõem o módulo de mais alto nível. A arquitetura de um módulo passa pela escolha do caminho de dados e da máquina de estados que controla os sinais.

A máquina de estados é similar a um autômato, com estado inicial, uma função de transição e conjunto de estados. O estado futuro da máquina é decidido conforme os valores das entradas, do estado presente e dos valores dos registradores internos. O caminho de dados, como o próprio nome já diz, é o conjunto de circuitos combinacionais, registradores e memória interligados por fios, também chamados de sinais.

A Figura 26 mostra o fluxo de desenvolvimento de um projeto em FPGA iniciando pela descrição em linguagem HDL e terminando com a depuração intra-chip. No número 2 da referida figura é realizada a simulação comportamental. Nessa fase utiliza-se uma ferramenta CAD (*Computer Aided Design*) que interpreta a descrição do circuito em HDL e simula seu comportamento como descrito na especificação da linguagem. Nessa fase

procura-se saber se a descrição em HDL realiza funcionalmente a tarefa especificada para o circuito.

Figura 26 Fluxo de projeto em FPGA[2]



Após a fase 2 se inicia a de síntese, que transforma a descrição HDL em circuitos lógicos. Para tanto é necessária a ferramenta de síntese. Essa nova representação do circuito precisa ser validada a fim de se garantir a coerência com a sua descrição comportamental. Essa etapa é feita em 4.

Em 5 são realizadas as tarefas de mapeamento tecnológico, *placement* e roteamento do circuito gerado em 2. Também em 5 também é feita a análise estática de tempo (6), com base nas restrições tecnológicas e de frequência pretendidas pelo projetista, e a anotação dos atrasos de cada componentes e vias (7).

Essas informações de tempo servem de base para a simulação com timing (8) que é a última simulação realizada antes que se carregue o projeto no FPGA. Nessa simulação os atrasos dos componentes reais do FPGA são simulados e o projeto pode ser validado nos quesitos temporais. Após a última simulação o

[2] http://www.xilinx.com/itp/xilinx10/isehelp/ise_c_fpga_design_flow_overview.htm

FPGA pode ser configurado (9) e ainda é possível depurar o funcionamento do circuito dentro do chip em tempo de execução (10).

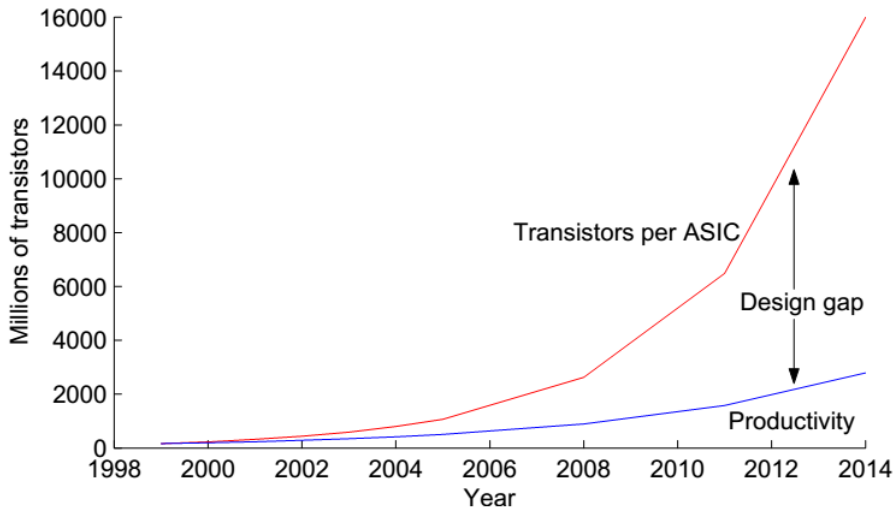
A cada fase de validação pode levar a alterações na descrição HDL e consequentemente a uma nova iteração do fluxo completo.

Claramente, esse fluxo de desenvolvimento demanda um grande esforço que se traduz em custo e tempo de projeto bem mais elevado em comparação aos projetos visando aplicações em GPUs ou mesmo as CPUs.

Atrelado a isso, a tecnologia de miniaturização e integração de transistores em um circuito integrado cresce próximo à Lei de Moore porém a produtividade dos engenheiros ao entregar projetos cada vez mais complexos, cresce muito abaixo disso.

A Figura 27 ilustra essa diferença, ao que é chamado de gap de produtividade no projeto de circuitos integrados. Esse gap torna também os projetos cada vez mais longos e mais difíceis de gerenciar, uma vez que mais projetistas são necessários em um mesmo projeto.

Figura 27 Gap de produtividade[3]



[3] http://cas.ee.ic.ac.uk/people/nps/research.html

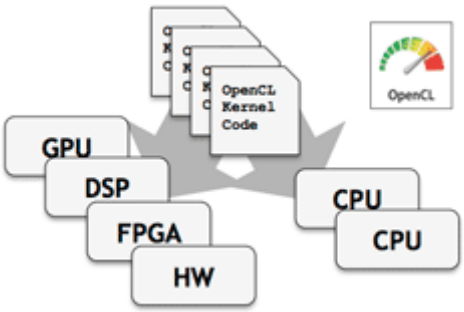
Para suplantar esse gap, diversas tentativas de elevar a abstração da descrição do hardware foram propostas sob a sigla de HLS (High-Level Synthesis) como, por exemplo, o Cadence Cynthesizer e o Synopsys Synphony o primeiro utilizando a linguagem SystemC e o segundo na linguagem C.

Ambos se propõem a sintetizar o código em alto nível e gerar código HDL no nível RTL (Register Transfer Level) e a partir deste ponto as ferramentas de síntese realizam seu trabalho. Porém essas iniciativas não apresentam grande expressividade e são iniciativas individuais de empresas para prover sua solução de alto nível.

Esse cenário mudou com a adoção da linguagem OpenCL (Open Computing Language) para a especificação de hardware, primeiro adotado pela Altera em 2013 e no final de 2014 também pela Xilinx.

O padrão de OpenCL definido pelo consórcio Khronos é formado, dentre outros, pela Intel, AMD, Altera, Xillinx, Apple, ARM, IBM, Nvidia, Samsung, etc. OpenCL foi concebida para sistemas heterogêneos, contendo GPUs, CPUs (single-core, multi-core ou multi-processadas) e até FPGAs, embora, por alguns anos, as plataformas alvo dos desenvolvedores de OpenCL tenham se restringido principalmente a CPUs e GPUs. Mas a possibilidade da mesma descrição em OpenCL poder ser avaliada em diversas plataformas diminui o esforço em se explorar qual a melhor arquitetura para determinado problema. A Figura 28 ilustra esse cenário.

Figura 28 OpenCL é um padrão para diversas arquiteturas[4]



[4] https://www.khronos.org/opencl/

A adoção de OpenCL para FPGAs deve acelerar o tempo de projeto até se obter uma versão inicial prototipada em FPGA. Permite, também, que o projeto customizado da arquitetura em FPGA possa seguir em paralelo, nos casos em que se precise de maior desempenho.

4.2 FPGAs em HPC

O termo HPC (High Performance Computing) designa os sistemas computacionais de processamento massivo, com grande poder computacional para resolução de tarefas específicas. Do inglês HPC significa computação de alto desempenho. Com o crescimento da demanda e da massa de dados processada, essa definição também designa sistemas com enlaces de larga banda de comunicação (Rocha 2010).

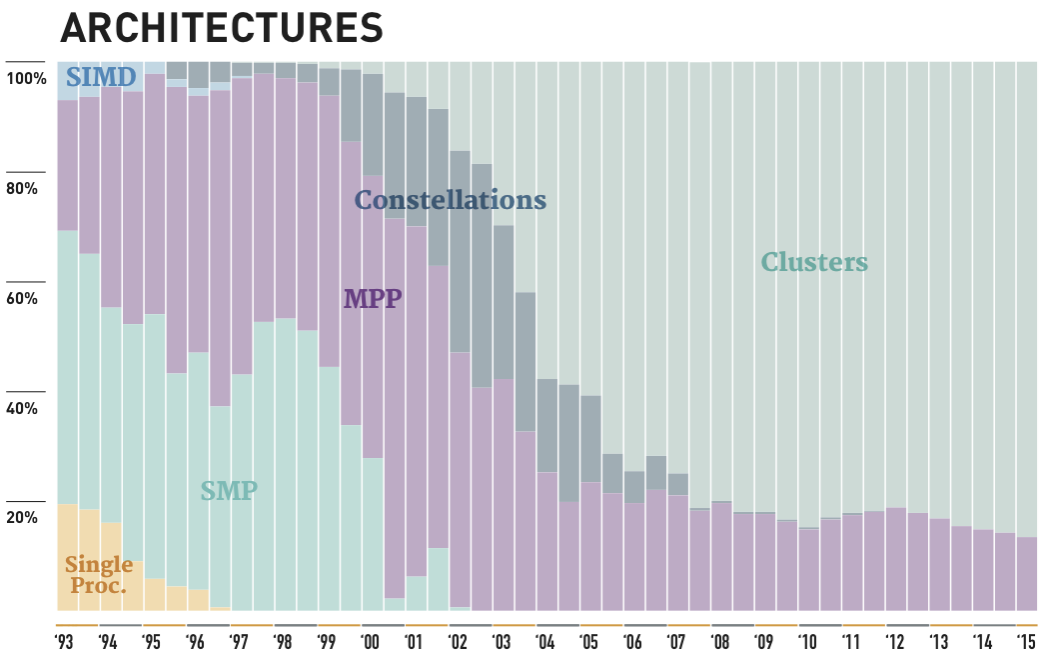
Os sistemas originais de HPC utilizavam as arquiteturas MPP (Massively Parallel Processing) e SMP (Symetric Multiprocessing). Essas arquiteturas apresentam hardware específicos e um custo elevado de infraestrutura (Rocha 2010).

As arquiteturas de MPP e SMP vêm dando espaço a sistemas menos custosos como a arquitetura de Cluster Computing, como pode se ver na Figura 29. Na arquitetura de Cluster as unidades possuem custo mais baixo, apresentam modularidade e possuem desempenho similar às anteriores. Atualmente supercomputadores com essa arquitetura lideram a lista dos computadores mais rápidos do mundo (Rocha 2010).

Os nós de um Cluster são computadores de propósito geral conectados por uma rede ethernet sincronizados por um nó mestre que divide as tarefas e gerencia os recursos (Rocha 2010).

De fato, segundo a organização TOP500, a evolução das arquiteturas de supercomputadores, mostrada na Figura 29, mostra o domínio da arquitetura de cluster sobre a MPP há dez anos.

Figura 29 Arquiteturas dos supercomputadores (top500 2015)



Outro dado relevante é a área de aplicação desses supercomputadores. A partir da Figura 30 conclui-se que o maior nicho de aplicações se encontra na indústria e, em seguida, encontram-se em aplicações de pesquisa e acadêmicas em proporção similar.

A Figura 31 mostra a distribuição do uso de coprocessadores nos supercomputadores. Observa-se a dominância do uso de GPUs tanto da Nvidia quanto da AMD, seguida pela adoção da recente do coprocessador da Intel chamado de Xeon Phi.

Os FPGAs não figuram na lista dos coprocessadores dos grandes supercomputadores. Espera-se que com a adoção pela indústria do padrão de OpenCL para FPGAs essa realidade mude. O código OpenCL descrito para as GPUs pode ser adaptado ou otimizado visando sua utilização nos FPGAs, fazendo com que projetos existentes possam ter códigos reusados.

Figura 30 Tipo de aplicação dos supercomputadores (top500 2015)

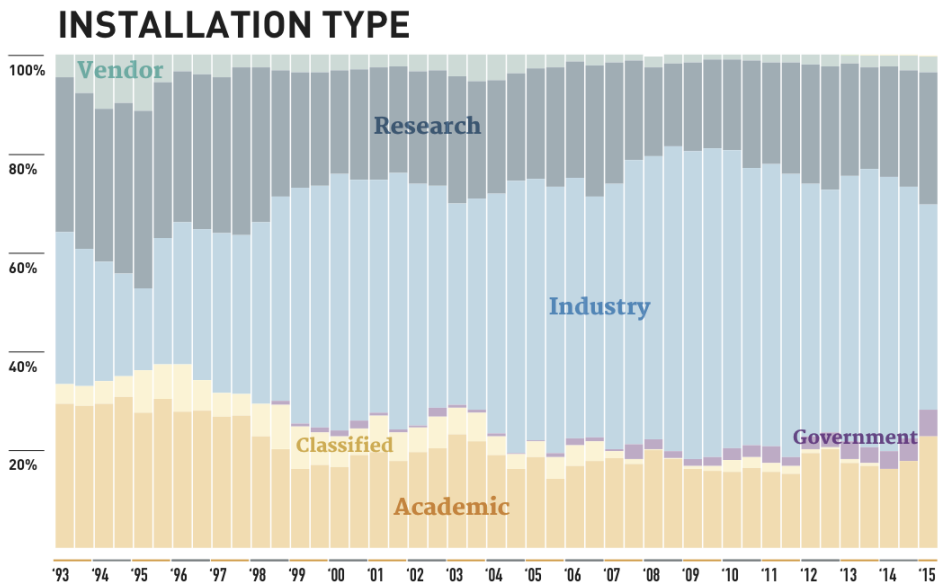
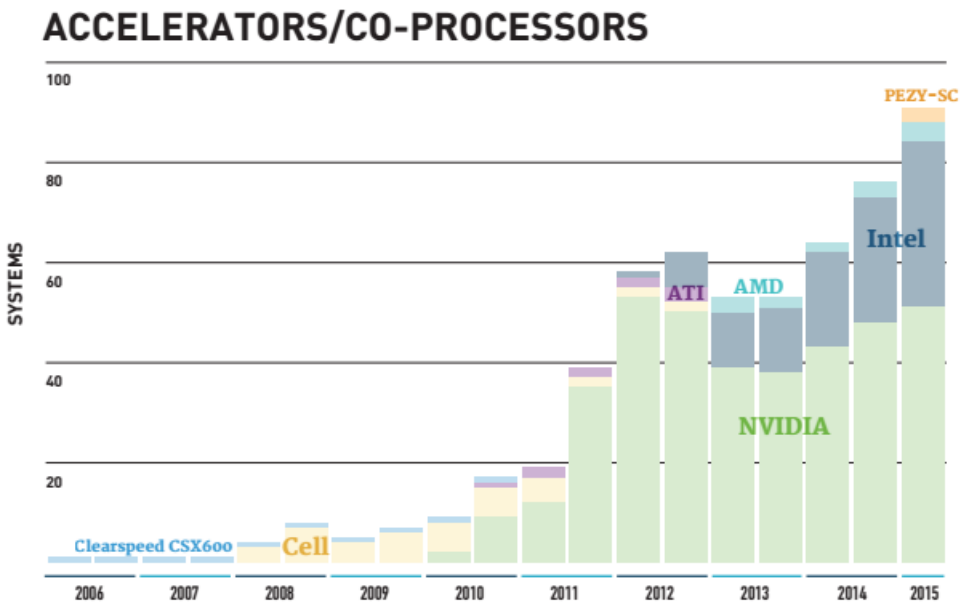


Figura 31 Coprocessadores utilizados nos supercomputadores (top500 2015)



Outro importante fator que beneficia os FPGAs é o custo de se manter um supercomputador com alto consumo de energia pelo processamento e pelo arrefecimento.

Um exemplo é o maior supercomputador atual do mundo, o Tianhe-2 pertencente ao governo chinês, custou 390 milhões de dólares, tem taxa

processamento de 33,86 petaflops/s e já ocupa o primeiro lugar da lista há dois anos. O supercomputador é composto por 16 mil nós, cada um com dois processadores Intel Xeon Ivy Bridge e três coprocessadores Xeon Phi.

O Tianhe-2 consome 17,6 MW (24 MW com resfriamento) resultando numa eficiência energética de 1901,54 Mflops/W, segundo o TOP500. Porém, o custo de manutenção apenas com consumo de energia gira em torno de 65 a 100 mil dólares por dia. Isso quer dizer que em dez anos o supercomputador gastou seu valor de instalação em consumo energético.

Nesse contexto foi concebido em 2009 no CHREC (*Center for High-Performance Reconfigurable Computing*) o maior supercomputador reconfigurável do mundo, o Novo-G. Ele possui hoje 64 FPGAs Altera Stratix-V D8, 192 FPGAs Stratix-IV E530 e 192 FPGAs Stratix-III E260.

Além do quesito potência, existem aplicações em que os FPGAs apresentam inerente fator de ganho, como por exemplo operações com números inteiros, processamento de sinais, comunicações, processamento embarcado de imagens, processamento sísmico, etc.

As soluções mais recentes de FPGAs já incluem suporte nativo a unidades de ponto flutuante, garantindo o desempenho de até 10 Tflops por FPGA.

5

Trabalhos Relacionados

Muitos pesquisadores têm desenvolvido implementações tanto em software, em GPUs, quanto em hardware para acelerar o problema de alinhamentos de DNA (Morony e Moreno 2008).

Neste capítulo são apresentados trabalhos da literatura divididos em trabalhos de implementação em processadores de propósito geral, trabalhos utilizando a arquitetura das GPUs e por fim trabalhos com aceleração em FPGA.

Ao final do capítulo, uma seção é dedicada à análise comparativa entre todos os trabalhos detalhados aqui.

5.1 Processadores de Propósito Geral (GPP)

5.1.1 SWAPHI-LS: Smith-Waterman Algorithm on Xeon Phi coprocessors for Long DNA Sequences (Liu, et al. 2014)

Nesse trabalho, os autores apresentaram o SWAPHI-LS, um algoritmo de software paralelo para explorar o paralelismo do coprocessador *Xeon Phi* (Intel s.d.) com o intuito de acelerar o alinhamento local de cadeias longas de DNA utilizando o algoritmo de Smith-Waterman.

O *Xeon Phi* segue a arquitetura da Intel chamada *Many Integrated Cores* (MIC)

que apresenta configurações com até 61 cores, 244 threads e 16 GB de memória GDDR5 além do barramento de 256 bits e instruções SIMD (*Single Instructions Multiple Data*) de 512 bits.

Os autores implementaram três tipos de abordagens: a abordagem canônica, a abordagem por *tiles* e outra distribuída. As três abordagens compartilham do mesmo princípio de paralelizar o processamento no sentido das anti-diagonais como mostrado na Figura 32.

A abordagem canônica realiza o alinhamento de cada anti-diagonal, por vez, iniciando da mais acima à esquerda até a mais abaixo à direita. Nesse caso, são aplicadas técnicas de vetorização para o grupo de *threads*.

Já a abordagem por *tiles* otimiza o processamento particionando a matriz de escores em regiões retangulares, como pode ser visto na Figura 33. Dessa forma, a computação é realizada em multi-threads entre tiles de uma mesma anti-diagonal, e dentro dos tiles se utiliza o paralelismo SIMD.

Figura 32 Modelo Canônico (Liu, et al. 2014)

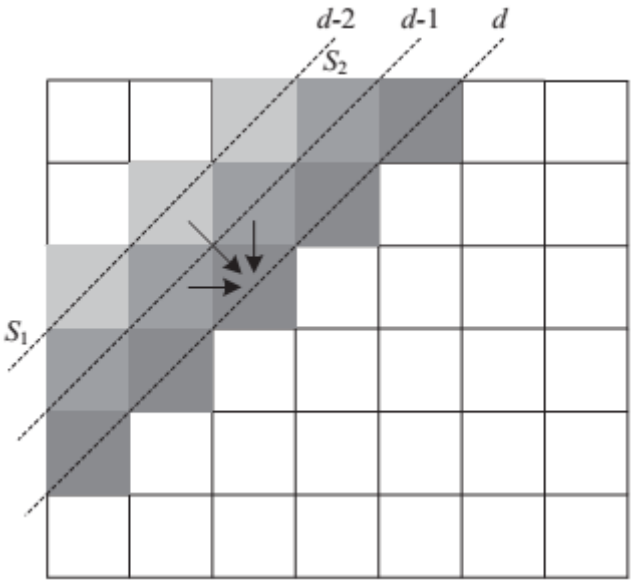


Figura 33 Modelo por Tiles (Liu, et al. 2014)

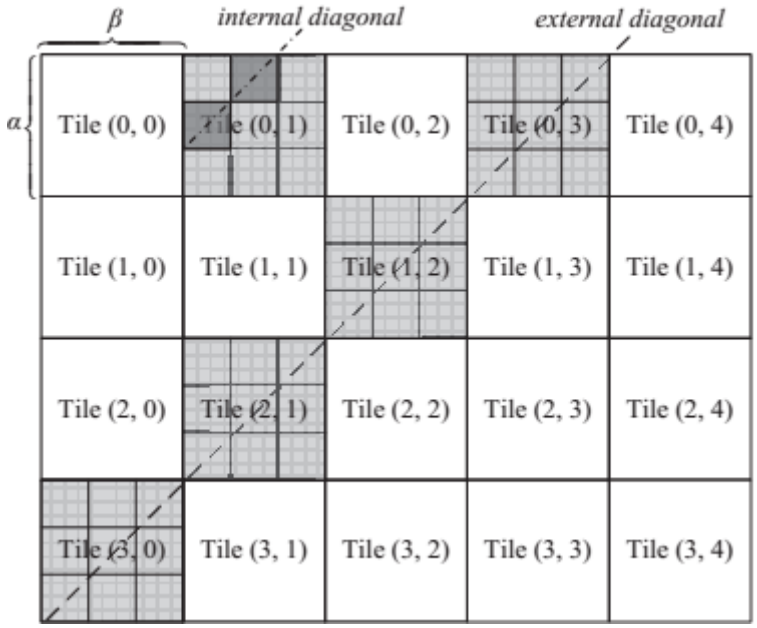
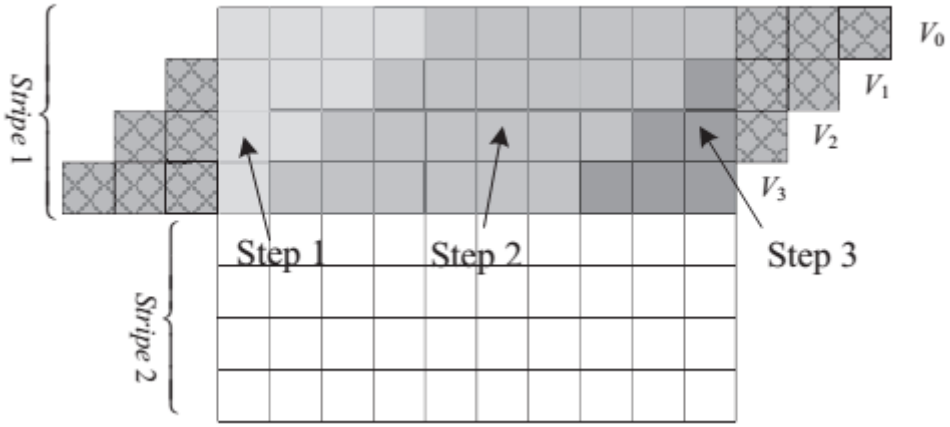


Figura 34 Detalhe do processamento dentro de um tile (Liu, et al. 2014)



Na Figura 34, pode-se ver os detalhes da computação dentro de um tile. O processamento é dividido em stripes ou fatias de processamento. Dentro de um stripe primeiro existem os passos transientes 1 e 3. No passo 2 o processamento segue a diagonal paralelizada no nível de instruções SIMD.

A abordagem distribuída abstrai a abordagem por *tiles*, os distribuindo em uma rede de Xeon Phis. Nessa abordagem divide-se a matriz de escores em grupos de *tiles*, criando blocos maiores. Consequentemente, aplica-se o paralelismo em blocos na mesma anti-diagonal. Cada bloco é escalonado no mesmo Xeon Phi e

é computado usando a abordagem por tiles.

Nesse caso é utilizada a infraestrutura de MPI (*Message Passing Interface*) já que um Xeon Phi pode ser visto como um PC endereçável por um endereço IP. Outra facilidade de implementação é que não se necessita de um *cross-compiler* já que o Xeon Phi segue a arquitetura X86.

A avaliação de desempenho foi feita realizando alinhamentos para seis genomas com tamanho que variam de 4,4 a 50 milhões de nucleotídeos, em um nó com dois processadores Intel E5-2670 de 8 cores cada, com frequência de 2,60 GHz, 64 GB de memória RAM. Cada Xeon Phi possui 60 cores, a uma frequência de 1,05 GHz, com 7,9 GB de memória RAM.

Nesse trabalho foram testadas as versões em tiles e a distribuída. Os autores identificaram que a versão canônica apresentava um desempenho baixo e não realizaram mais testes com essa versão.

A versão em tiles foi avaliada usando um único Xeon Phi e a distribuída utilizou dois e quatro Xeon Phis. O programa atingiu o speedup de 28,63 sobre uma implementação de terceiros em software mono-processado. A versão distribuída com dois Xeon Phis atingiu 50,69 de speedup, já a versão com quatro Xeon Phis atingiu 98,73 de speedup.

O software desenvolvido rodando em um único Xeon Phi não conseguiu superar o desempenho de uma GPU Tesla K20c da Nvidia. Na GPU o algoritmo de Smith-Waterman rodou 1,35x mais rápido que no Xeon Phi. Entretanto, em dois Xeon Phis a performance superou a da GPU por um fator de 1,33 vezes e usando quatro Xeon Phis por um fator de 2,69 vezes.

5.1.2 Characterization of Smith-Waterman sequence database search in X10 (Ji, Liu e Yang 2012)

Nesse trabalho os autores utilizaram a linguagem X10 a fim de caracterizar o desempenho da linguagem usando uma aplicação em bioinformática, o algoritmo de Smith-Waterman.

A linguagem orientada a objeto X10, desenvolvida pela IBM, foi especificamente

projetada para computação paralela. Os criadores da linguagem pretenderam otimizar tanto a produtividade quanto o ganho de desempenho ao se construir uma aplicação paralela. A linguagem define novos construtores para especificar operações paralelas tais como concorrência, distribuição e sincronização.

A característica principal de X10 envolve duas noções de *kernel*: lugar e atividade. Lugares são o resultado de uma partição nos recursos disponíveis feita por X10, que enfatiza a localidade bem como a escalabilidade. Atividades são computações leves e independentemente executadas em um recurso local. Cada atividade pode criar várias atividades locais assíncronas localmente ou remotamente.

Em um programa em X10 a atividade principal lança a função *main* e todas outras atividades constituem uma estrutura de pai-filho num relacionamento em árvore.

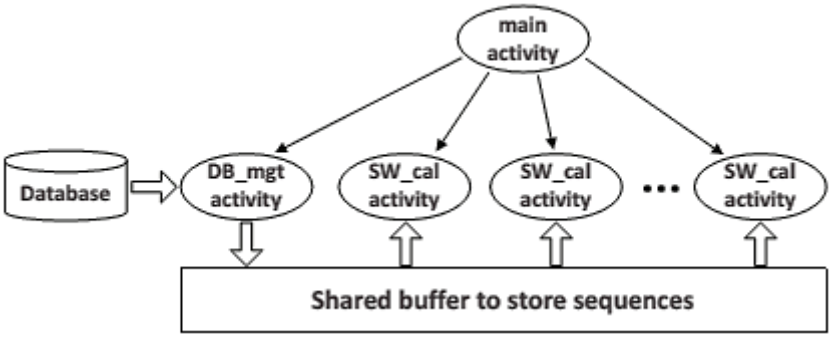
Os autores implementaram uma aplicação em X10 em uma arquitetura multi-core de memória compartilhada. Os resultados dessa implementação foram comparados com um software em C++. Posteriormente foram feitas sugestões tanto para a linguagem quanto para o seu compilador.

De acordo com a arquitetura multi-core, a implementação em X10 consiste em dois níveis: versão em single-core, que roda sequencialmente em um core, e a versão multi-core que roda concorrentemente a versão single-core em múltiplos cores.

Na versão single-core algumas otimizações de código foram feiras, como reduzir o acesso indireto e a utilização de variáveis locais para armazenar resultados intermediários. Porém, não se utilizou instruções SIMD porque a linguagem X10 não tem suporte para instruções vetoriais.

A versão multi-core se baseia no modelo produtor-consumidor, em que o produtor controla a Entrada\Saída e o consumidor realiza a computação. A Figura 35 mostra o esquema da distribuição das tarefas. A atividade principal invoca as atividades filhas, incluindo a de gerenciamento do banco de dados de sequências.

Figura 35 Versão multi-core implementada em X10 (Ji, Liu e Yang 2012)



A implementação foi testada numa plataforma com dois processadores Intel de seis cores rodando a 2,27GHz e com 24GB de memória RAM. Foi também implementada uma versão em C++ com OpenMP para fins de comparação. OpenMP (Open Multi-Processing) é uma API programação multi-threads de memória compartilhada.

Foram usados dois bancos de sequências, um com 32.283 sequências menores que 2.000 e outro com 5.126 sequências de tamanho maior que 4.000. As dez sequências de query tinham tamanhos variáveis de 144 até 5.147.

Os resultados comparativos podem ser vistos na Figura 36. A implementação em X10 multi-core é 2.25 vezes mais lenta que a versão em C++ usando OpenMP.

Mesmo não existindo o suporte em X10 para instruções SIMD, os autores compararam a implementação single-core em C++ com uma implementação de terceiros chamada SWPS3-sse que explora o paralelismo no nível de instruções SIMD. O resultado pode ser visto na Figura 37. As instruções 128-bit SSE de SWPS3-sse resultaram no speedup de 8,7 a 17,7 vezes sobre a versão single-core em C++.

Figura 36 Resultado comparativo entre C++ e X10 (Ji, Liu e Yang 2012)

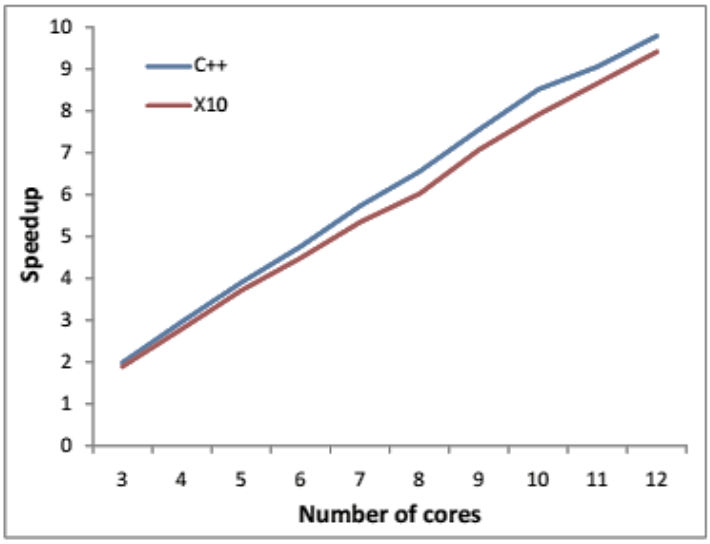
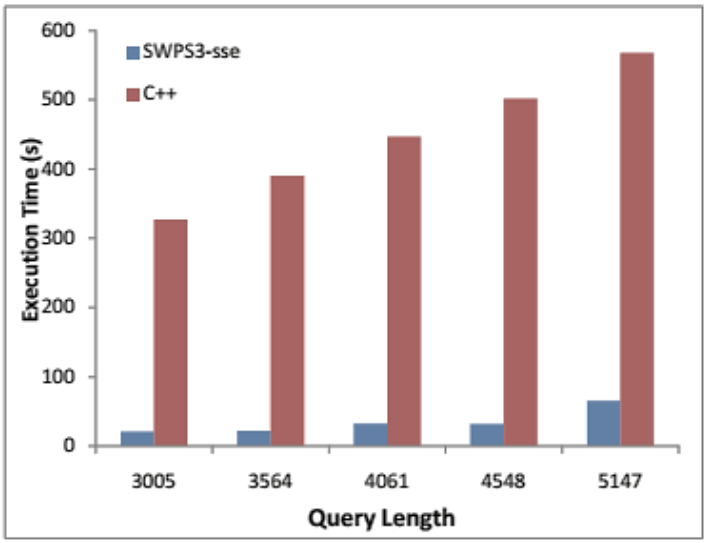


Figura 37 C++ single-core versus SWPS3-sse (Ji, Liu e Yang 2012)



5.2 Unidades Gráficas de Processamento (GPU)

5.2.1 CUDASW++ 3.0: accelerating Smith-Waterman protein database search by coupling CPU and GPU SIMD instructions (Liu, Wirawan e Schmidt 2013)

Nesse trabalho foi proposta a implementação do CUDASW++ na versão 3.0. O CUDASW++ integra o paralelismo das GPUs com o processamento da CPU do Host de forma concorrentemente para a solução do problema do alinhamento

local de proteínas. Esse algoritmo é usado como medida de similaridade de uma query em comparação com as sequências de um banco de proteínas.

Para aumentar o desempenho, foram utilizadas adicionalmente instruções SIMD de vídeo da plataforma de GPU da Nvidia.

A distribuição da carga entre a CPU e a GPU segue a fórmula do quociente R da Equação 4.1. f_G e f_C são as frequências de operação da GPU e da CPU respectivamente. N_G e N_C são o número de núcleos da GPU e da CPU. E por final, C é uma constante empírica. Quando são usadas mais de uma GPU ou CPU, o número de núcleos dela é somado para formar N_G e N_C.

$$R = \frac{N_G f_G}{N_G f_G + N_C f_C / C} \tag{5.1}$$

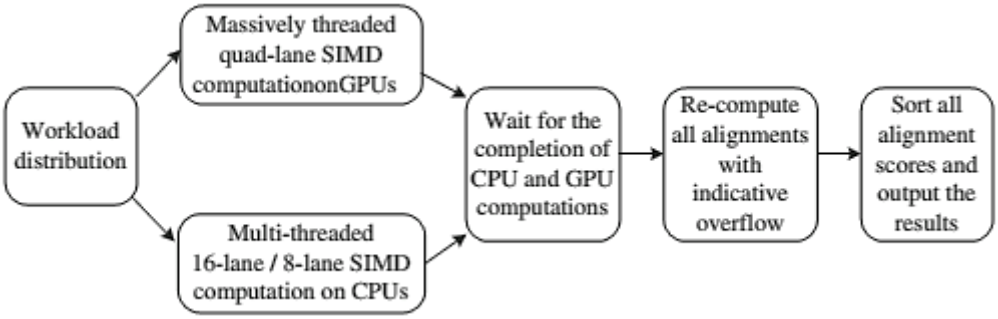
As sequências designadas para a GPU precisam ter o tamanho menor ou igual ao produto da Equação 4.2.

$$N_R = R * tam\_sequência \tag{5.2}$$

As sequências designadas para a CPU são processadas utilizando instruções SIMD do conjunto SSE (Streaming SIMD Extensions), da Intel, utilizando a precisão menor do que a necessária. São usados escores com 8bits. Dessa forma podem ser feitos 16 alinhamentos independentes através das instruções SIMD.

Nas sequências em que houve a sinalização do overflow de representação são marcadas para a posterior recomputação com maior precisão. Essa ideia vem do software SWIPE. O fluxo de operação do sistema é apresentado na Figura 38. Primeiro há a distribuição de carga seguindo as Equações 4.1 e 4.2, depois do processamento ser realizado, caso se indique um overflow em algum alinhamento é necessário sua recomputação e por fim os escores dos alinhamentos são ordenados.

Figura 38 Fluxo de execução do algoritmo em CPU e GPU (Liu, Wirawan e Schmidt 2013)



Na GPU o processamento SIMD foi feito com a implementação em *assembly*. Em Cuda existem tipos vetoriais como por exemplo o Int4 e Float4, que são representações empacotadas de quatro inteiros e quatro variáveis do tipo *float*. A Figura 39 apresenta as instruções de 8 bits sinalizadas no trecho de código que computa quatro alinhamentos independentes.

Figura 39 Trecho de código em assembly da GPU (Liu, Wirawan e Schmidt 2013)

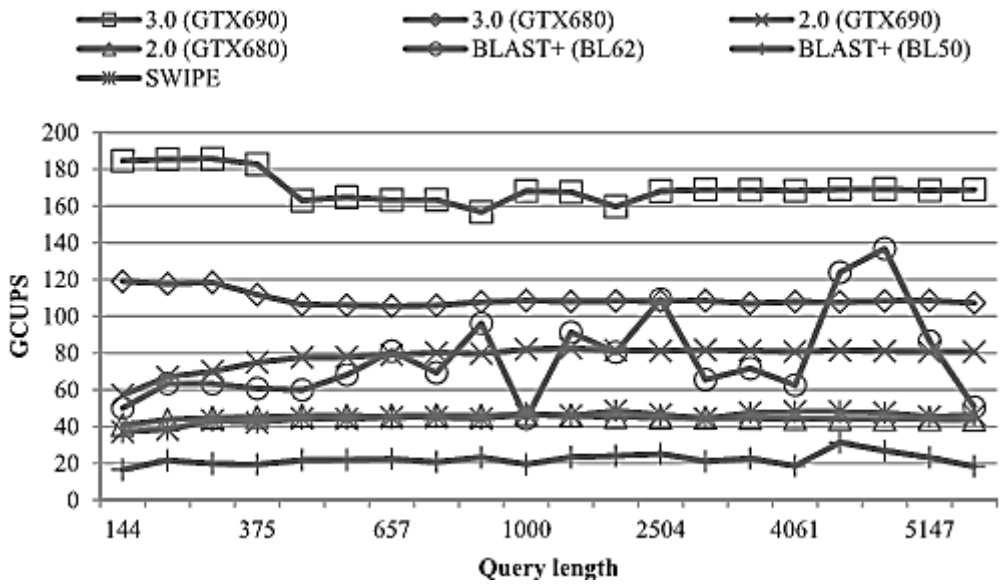
vsub4.s32.s32.s32.sat	$f : f, \beta$	// f = f - β
vsub4.s32.s32.s32.sat	$e : e, \beta$	// e = e - β
vsub4.s32.s32.s32.sat	$h : h, \alpha$	// h = h - α
vmax4.s32.s32.s32	$f : f, h$	// f = max(f, h)
vsub4.s32.s32.s32.sat	$h : h_e, \alpha$	// h = $h_e - \alpha$
vmax4.s32.s32.s32	$e : e, h$	// e = max(e, h)
vadd4.s32.s32.s32.sat	$h : n, M_q$	// h = n + M_q
vmax4.s32.s32.s32	$h : h, f$	// h = max(h, f)
vmax4.s32.s32.s32	$h : h, e$	// h = max(h, e)
vmax4.s32.s32.s32	$h : h, 0$	// h = max(h, 0)
vmax4.s32.s32.s32	$S : S, h$	// S = max(S, h)

O desempenho da solução proposta foi comparada com a versão 2.0 do CUDASW++, e com os softwares SWIPE e o BLAST+. Foram utilizadas 20 proteínas como query com tamanho variando entre 144 até 5,478 e dois bancos de proteínas, o Swiss-Prot com 538.585 sequências de tamanho variado e outro sintetizado com 200.000 de tamanho igual a 3.000.

No banco do Swiss-Prot o resultado do protótipo em uma GPU Nvidia GTX680

resultou num speed-up de 2,9x sobre a versão anterior do CUDASW++, 3,2x de speed-up sobre o software SWIPE e até 7,2x sobre o BLAST+. A Figura 40 ilustra as comparações realizadas utilizando a escala de GCUPS (Giga Cell Updates Per Second) ou bilhões de atualizações de células por segundo.

Figura 40 Gráfico comparativo do desempenho dos algoritmos (Liu, Wirawan e Schmidt 2013)



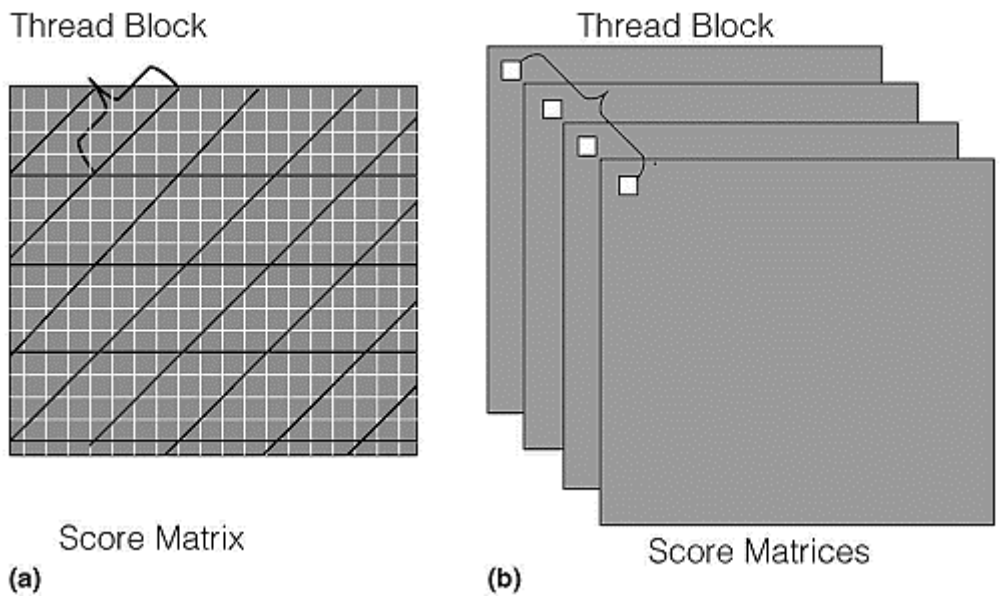
5.2.2 Large-Scale Pairwise Sequence Alignments on a Large-Scale GPU Cluster (Savran, Gao e Bakos 2014)

Nesse artigo é descrita a implementação em GPU de uma aplicação para o alinhamento global de DNA usando o algoritmo de Needleman-Wunsch. A aplicação usa a distribuição em nós de um cluster de GPUs utilizando MPI para atingir a maior vazão de alinhamentos possível. Os nós do cluster são híbridos, contendo CPU mais GPUs como coprocessadores.

O desempenho da aplicação foi medido usando o supercomputador Stampede do Centro de Computação Avançada do Texas, em Austin nos EUA. A execução dos kernels ocorreu em 32 nós contendo uma GPU K20 da NVIDIA cada, totalizando 79.872 núcleos de GPU. O problema completo consistiu em 34 milhões de alinhamentos individuais.

Como mostrado na Figura 41(a), as abordagens da literatura normalmente implementam os algoritmos de Smith-Waterman e Needleman-Wunsch (incluindo a primeira versão do CUDASW++) usando a estratégia que emprega múltiplas threads para computar um único alinhamento, localizando-as na anti-diagonal. Isso é possível uma vez que ao longo da anti-diagonal os dados podem ser computados concorrentemente. Esta estratégia pode ser efetiva para kernels que realizam um único alinhamento por vez, por exemplo, quando o tamanho das sequências é muito grande como quando se alinha um genoma inteiro.

Figura 41 Comparando duas abordagens de paralelização dos alinhamentos (Savran, Gao e Bakos 2014)



Os autores citam duas desvantagens nessa abordagem. Quando é alocada uma thread por ponto na anti-diagonal da matriz, o acesso à memória global da GPU, de forma coalescente, precisa ser reorganizada no sentido da anti-diagonal, ao invés da organização padrão por linha ou coluna. Organizar a matriz dessa forma, adiciona um *overhead* de processamento.

Outra desvantagem citada é a introdução de vários ramos de execução pelos diversos casos particulares. Isso ocasiona a inatividade de várias *threads* e a

degradação do desempenho.

Ambos os casos levam a um maior uso de registradores por *thread* o que limita a ocupação da GPU, ou seja, o número de *threads* que podem ser executadas simultaneamente por cada núcleo da GPU.

Dessa forma os autores propuseram distribuir a carga de trabalho, como visto na Figura 41(b), de maneira que cada *thread* executa um alinhamento independentemente. Visto que a aplicação utiliza cadeias de DNA do tamanho de 400 nucleotídeos, então pode-se processar um grande número de alinhamentos em paralelo.

Uma vez que cada alinhamento é independente, o *host* pode designar para cada GPU, num ambiente de *cluster*, uma carga de trabalho consistindo num subconjunto dos alinhamentos a serem paralelizados e dividi-los através de múltiplas GPUs. Nesse trabalho os autores utilizam MPI para fazer a tarefa da divisão da carga.

Cada nó do *cluster* Stampede contém dois processadores Intel Xeon E5-2680 com oito cores cada com 16 multi-threads, rodando a 2,7-GHz, que podem executar 16 processos MPI.

O resultado da execução de um nó do cluster pode ser visto na Figura 42. Nota-se que o desempenho conseguido por uma GPU se compara com o desempenho conseguido por 2 nós do cluster.

No quesito potência consumida, a GPU NVIDIA GTX 680 dissipa 195 W, enquanto um Intel Xeon E5-4650 CPUs consome 130 W. Esse dado mostra a grande diferença de eficiência energética usando-se a GPU.

O resultado do desempenho para múltiplas GPUs pode ser visto na Figura 43. Foram usadas de 4 até 32 GPUs NVIDIA K20s.

Figura 42 Tempos em segundos para CPUs versus GPU (Savran, Gao e Bakos 2014)

Cluster processes or GPU	Execution Time for 8192 Sequences	Execution Time for 6144 Sequences
32	2479	1394
64	1241	698
128	620	698
256	620	349
512	310	175
1024	155	87
GTX 680	1118	634

Figura 43 Resultados, em minutos, para múltiplas GPUs NVIDIA K20s (Savran, Gao e Bakos 2014)

Number of GPUs	Execution time (minutes)	2^{14} (16K) sequences 1.34×10^8 alignments	2^{15} (32K) sequences 5.37×10^8 alignments	2^{16} (64K) sequences 2.15×10^9 alignments	2^{17} (128K) sequences 8.59×10^9 alignments	2^{17} (256K) sequences 3.44×10^{10} alignments
4 x K20	ideal	15	59	237		
	actual	16	66	266		
	overhead	10%	12%	12%		
8 x K20	ideal		30	119	474	
	actual		32	131	531	
	overhead		8%	11%	12%	
16 x K20	ideal		15	59	237	
	actual		16	64	261	
	overhead		11%	7%	10%	
32 x K20	ideal			30	119	474
	actual			30	126	524
	overhead			1%	7%	10%

Os tempos em CPU não foram informados, para os tamanhos de problema mostrados na Figura 43.

5.3 FPGA

5.3.1 A Review of Hardware Acceleration for Computational Genomics (Aluru e Jammula 2014)

Nesse trabalho são revisadas diversas abordagens para aceleração de tarefas frequentes na área da genômica. Nesse contexto a proposição de um hardware dedicado, na forma de um acelerador ou coprocessador de hardware são propostos para a aceleração de uma tarefa específica.

Os autores informam que o ganho de desempenho e o benefício de custos conseguidos pelos aceleradores estão compensando o esforço adicional em programação necessário para sua construção.

Em muitos domínios de aplicação, tipicamente, a parte dominante do tempo de processamento é gasto em pequenos núcleos de computação o que facilita sua exportação na forma de aceleradores. No domínio da biologia computacional não é diferente, apesar de que programas de análise de sequências biológicas usarem predominantemente operações em inteiros, com frequência elas empregam o uso de aritmética de ponto flutuante.

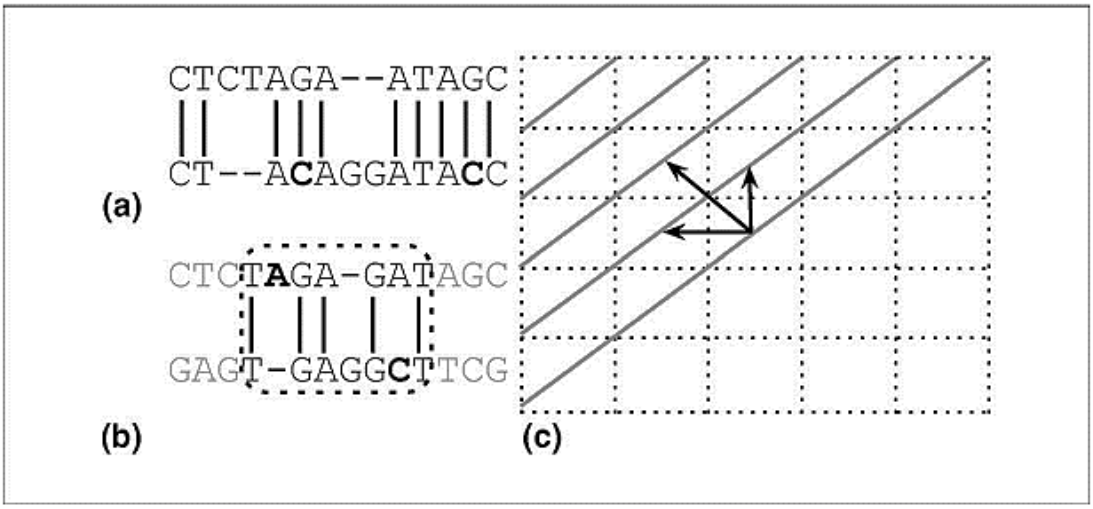
O alinhamento de duas sequências é uma operação fundamental na genômica e suas inúmeras variantes ocorrem frequentemente devido a razões biológicas. Uma vez que a evolução implica em mutações e inserções/deleções das sequências de DNA, alinhamentos próximos de sequências refletem uma história evolucionária próxima.

Na Figura 44, em (a) é mostrado um alinhamento global e em (b) um alinhamento local para a mesma sequência. Em (c) aparece a matriz usada na programação dinâmica com a dependência de dados dos algoritmos de alinhamento formando uma frente de onda.

A forma mais frequente de implementação do problema do alinhamento de duas sequências utiliza o fato da dependência de cada anti-diagonal ser apenas com as duas anteriores e os dados na anti-diagonal podem ser computados em paralelo. Essa abordagem também é chamada de abordagem da frente de onda.

Uma segunda abordagem computa uma linha por vez e lança p *threads* realizando a computação da matriz de escore em tempo $O(m * n/p)$, em que m e n são os tamanhos das sequências.

Figura 44 Dependência de dados do algoritmo (Aluru e Jammula 2014)



Abordagens paralelas para o problema incluem plataformas em FPGA, GPUs, Cell BE da IBM (embora a arquitetura Cell não seja mais usada em sistemas de HPC) e arquiteturas System-On-a-Chip (SoC).

Seguindo as métricas de desempenho, consumo de energia e custo de projeto, as soluções foram comparadas. Os autores indicam um trabalho na literatura (Benkrid, et al. 2012) que ranqueia a plataforma em FPGA como a melhor, segundo os critérios mencionados.

5.3.2 An Efficient Processing Element Architecture for Pairwise Sequence Alignment (Isa, et al. 2014)

Nesse trabalho é proposta uma arquitetura parametrizável e eficiente em área para realizar o alinhamento de sequências biológicas. A arquitetura foi prototipada em uma plataforama FPGA com os elementos de processamento seguindo a distribuição de um array sistólico. Os resultados, em comparação com a implementação em software, indicaram um speedup mínimo de 15x.

A Figura 45 mostra os passos do algoritmo de Smith-Waterman utilizando uma função de gap afim. A função afim de gap afim possibilita penalizar a abertura de um vazio (inserção ou remoção), bem como uma sequência de vazios.

Figura 45 Descrição da inicialização e do passo recursivo do algoritmo de Smith-Waterman (Isa, et al. 2014)

Initialization

$$
\begin{aligned}
&\textit{for} \quad i \leftarrow 0 \textit{ to } M \\
&\quad M(i,0) \leftarrow 0 \\
&\quad I_x(i,0) \leftarrow -\infty \\
&\quad I_y(i,0) \leftarrow -\infty \\
&\textit{for} \quad j \leftarrow 0 \textit{ to } N \\
&\quad M(0,j) \leftarrow 0 \\
&\quad I_x(0,j) \leftarrow -\infty \\
&\quad I_y(0,j) \leftarrow -\infty
\end{aligned}
$$

Recursion

$$
\begin{aligned}
&\textit{for} \quad i \leftarrow 0 \textit{ to } M \\
&\quad \textit{for} \quad j \leftarrow 0 \textit{ to } N \\
&\quad F(i,j) \leftarrow \max \begin{cases} 0, \\ F(i-1,j-1) + s(x_i, y_j) \\ I_x(i-1.j-1) + s(x_i, y_j) \\ I_y(i-1.j-1) + s(x_i, y_j) \end{cases} \\
&\quad I_x(i,j) \leftarrow \max \begin{cases} F(i-1,j) - d \\ I_x(i-1,j) - e \end{cases} \\
&\quad I_y(i,j) \leftarrow \max \begin{cases} F(i,j-1) - d \\ I_y(i,j-1) - e \end{cases}
\end{aligned}
$$

Essa função visa cumprir o paralelo biológico de que a criação de um *gap* pode ser mais desvantajosa biologicamente do que sua extensão. No caso do algoritmo de Smith-Waterman a utilização da função de *gap* afim é necessária para se ter um melhor alinhamento.

A estrutura interna do PE (Elemento de Processamento) proposto pelos autores é ilustrada na Figura 46.

Nessa figura a variável gdw assume a largura de 4 bits, para representar os pesos de abertura e extensão do gap. O PE consiste em três circuitos aritméticos: O maior escore final $F(i,j)$, o maior escore da inserção $I_x(i,j)$ e o maior escore da remoção $I_y(i,j)$.

Já o bloco *PE Best Score* calcula o máximo valor já encontrado, visto que o algoritmo de Smith-Waterman inicia o passo de traceback pelo máximo valor de escore. O resultado do PE é então propagado através da cadeia de PEs no array sistólico.

O desempenho da arquitetura proposta pelos autores foi comparado com o do

software SSEARCH35, além da comparação com as implementações em trabalhos relacionados. O speedup conseguido sobre a implementação em software é mostrado na Figura 47. Nota-se que para o tamanho de cadeia 192 o speedup atingido foi de 380x. O software rodou em um processador Intel Dual Core E6600 com clock a 2.0 GHz.

Os recursos do FPGA utilizados para implementar 195 PEs somam 88 slices do total de 17.280 slices existentes no FPGA Virtex-5 XC5VLX110 a 200MHz.

Figura 46 Estrutura interna do PE (Isa, et al. 2014)

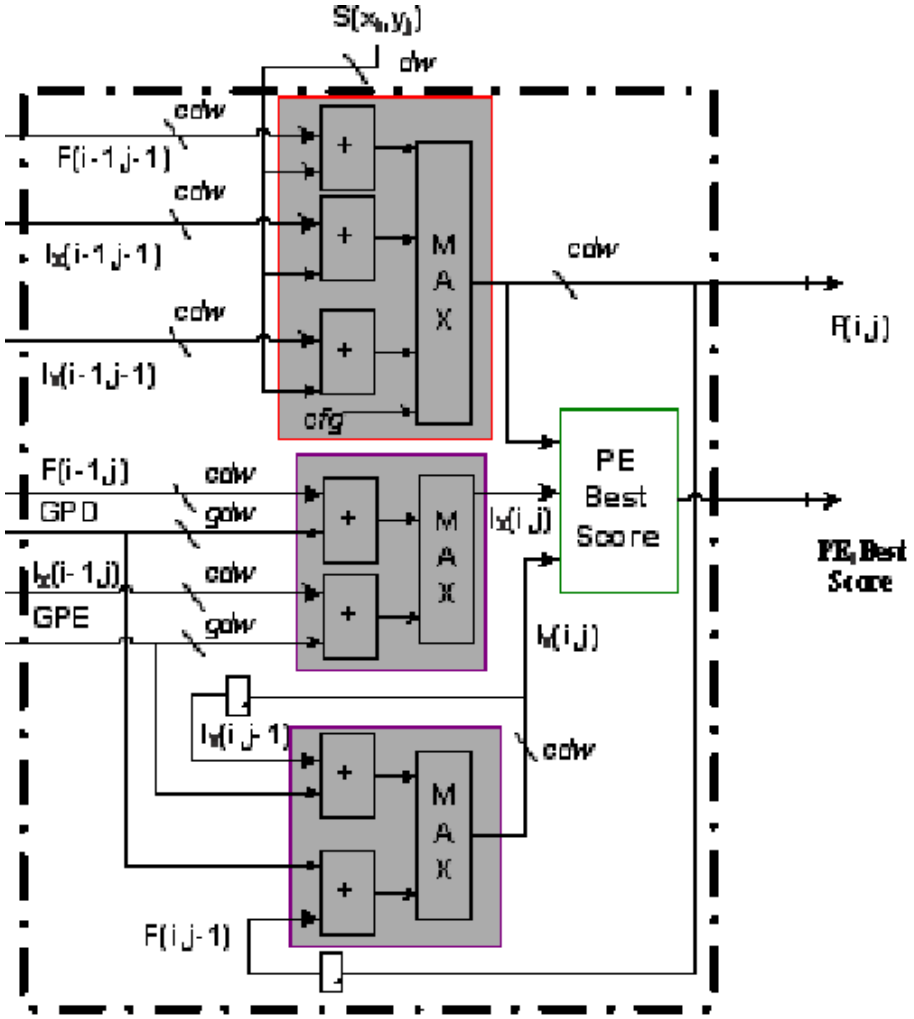


Figura 47 Desempenho comparado com implementação em software (Isa, et al. 2014)

Query Accession Number	Length	FPGA Speed up
P36515	4	15.98
P80709	8	20.55
P83511	16	38.81
O19927	32	60.12
A4T9V0	64	149.38
Q2IJ63	128	276.89
Q13323	160	316.60
Q9H9L7	192	380.31

Na Figura 48 é mostrada a comparação de desempenho da implementação proposta pelos autores frente a seis trabalhos na literatura. A medida de CUPS de pico (Cell Updates Per Second) corresponde ao número de PEs vezes a frequência de operação da arquitetura. A implementação proposta em FPGA atingiu $195\,PEs * 200\,MHz = 39\,GCUPS$. O que foi bem acima dos trabalhos relacionados.

Figura 48 Tabela comparativa com trabalhos na literatura (Isa, et al. 2014)

Reference	Device	Slices/PE	PEs (#)	Freq (MHz)	Peak CUPS (Giga)
Yamaguchi et al. (2002)	XCV2000E	-	144	40	5.8
Oliver et al. (2005)	XCV6000	192	168	45	7.6
Jiang et al. (2007)	EPS1S30	192	80	82	6.6
Benkrid et al. (2009)	XC2V6000	85	168	45.6	7.66
Meng et al. (2010)	XC2V6000	-	119	-	11.1
Yamaguchi et al. (2011)	2V6000-4	58	168	59.3	10.0
Proposed	**XC5VLX110**	**88**	**195**	**200.0**	**39.0**

A medição do desempenho em CUPS não representa uma medida realista, uma vez que, outras variáveis como largura de banda de memória precisam ser avaliadas no cálculo do desempenho do acelerador.

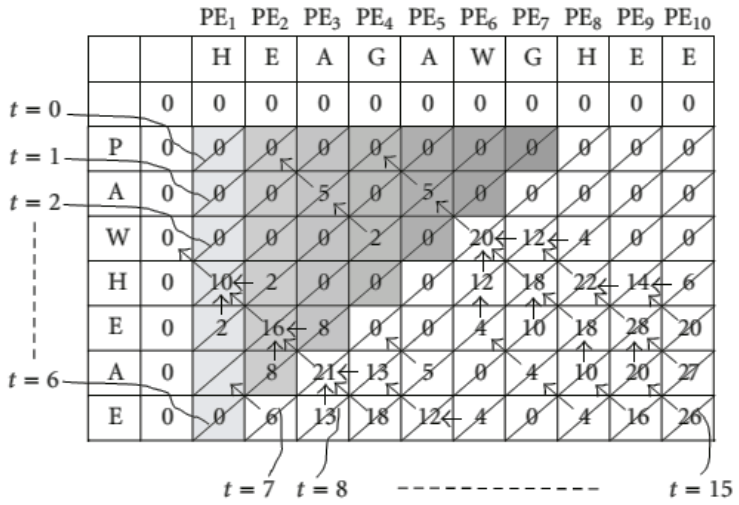
5.3.3 High performance biological pairwise sequence alignment: FPGA versus GPU versus cell BE versus GPP (Benkrid, et al. 2012)

Benkrid et al. trazem nesse trabalho uma análise comparativa do desempenho dos FPGAs para a sua aplicação em computação de alto desempenho. Os FPGAs são comparados com as GPUs, com o IBM Cell BE e com os processadores de propósito geral (GPP) utilizando o estudo de caso da implementação do algoritmo de Smith-Waterman. Os critérios de comparação incluem a velocidade de processamento, consumo de energia, custo e tempo de projeto.

O projeto para FPGA utilizou a linguagem de alto nível Handel-C com a suíte DK5 para compilar o código em Handel-C, com a distribuição dos PEs na arquitetura em FPGA consistindo em um array sistólico de PEs. Sendo assim, para sequências de tamanho n e m, e se for possível instanciar m PEs, a matriz de escores é computada em $n + m - 1$ ciclos.

A Figura 49 mostra o esquema de processamento utilizado no trabalho. Cada anti-diagonal é processada em cada instante de tempo, respeitando a dependência de dados.

Figura 49 Geração da matriz de escores (Benkrid, et al. 2012)



Os autores optaram por não fazer o traceback dentro do FPGA. Eles justificaram essa escolha dizendo que em um grande banco de sequências, alguns poucos alinhamentos resultam em escores altos. Dessa forma não seria necessário realizar a fase de traceback para todos os alinhamentos e essa tarefa seria facilmente processada pelo host em tempo negligenciável.

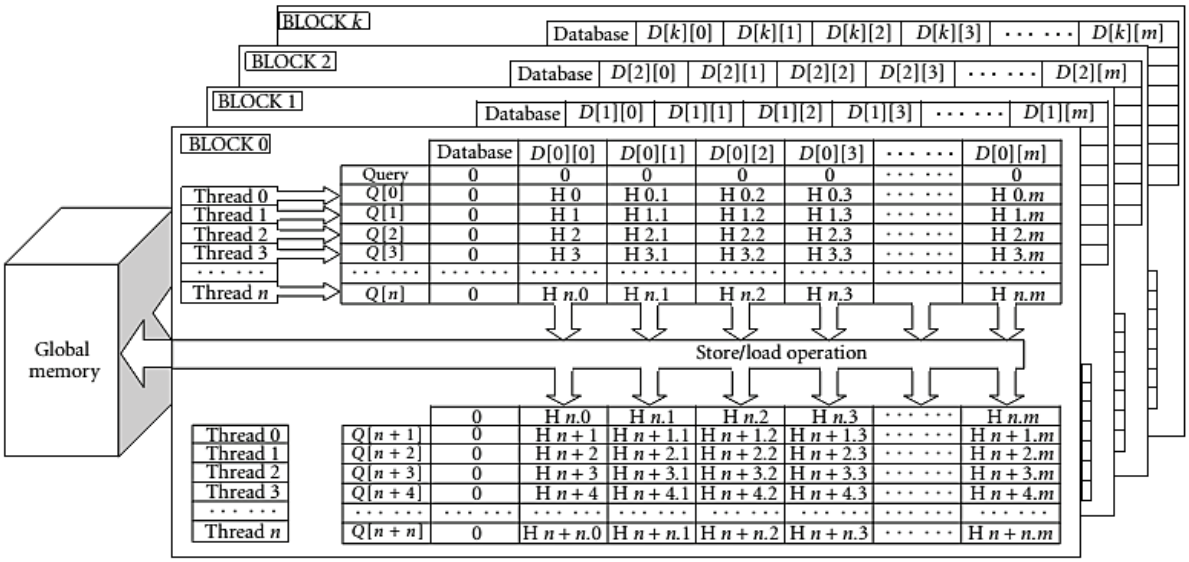
O número de PEs instanciados no FPGA é limitado pelos recursos internos do mesmo. Por exemplo, o número máximo de PEs em um FPGA Xilinx XC2V6000 Virtex-II, na implementação dos autores, foi em torno 250. Certamente isso é muito pouco, visto que o tamanho das sequências reais pode chegar a milhares ou milhões.

A solução foi particionar a execução do algoritmo em passos menores e instanciar um número fixo de PEs e armazenar os resultados intermediários em uma FIFO (First-In-First-Out). O tamanho da FIFO é proporcional ao tamanho da cadeia.

Na implementação em GPU, a estratégia de paralelização é baseada em multi-thread e multi-processamento, nesse caso várias threads são alocadas na computação de um único alinhamento, em paralelo, dentro de um mesmo bloco de *threads* da GPU, enquanto vários blocos são alocados para computar diferentes alinhamentos. A Figura 50 ilustra essa distribuição, em que cada plano representa um bloco de threads e cada thread é responsável pela computação de uma linha da matriz de scores.

Se por acaso, o número de sequências no banco for menor que o número de threads no bloco, um tempo adicional de espera precisa ser adicionado a fim de sincronizá-las. Não importa o quão rápido outras *threads* rodem, elas precisam esperar nos pontos de sincronização. Pode-se ver na Figura 51 que cada thread computa uma linha, porém, a dependência de dados do algoritmo impede que elas rodem fora de ordem. Na Figura 51 cada $\{D[i], D[j]\}$ representa um ponto na matriz de escores.

Figura 50 Distribuição das threads em GPU (Benkrid, et al. 2012)



No caso dos processadores IBM Cell BE, a implementação explora o paralelismo de dados envolvido em se alinhar uma query com um grande banco de dados de sequências. Assim, se distribui diferentes sequências para cada um dos oito SPEs (Synergistic Processing Element) do processador.

Figura 51 Sincronização das threads do bloco na GPU (Benkrid, et al. 2012)

Thread 0	$Q[0]$	$D[0], D[1]$	$D[2], D[3]$	$D[4], D[5]$	... ...	$D[m-1], D[m]$
Thread 1	$Q[1]$	$D[0], D[1]$	$D[2], D[3]$	$D[4], D[5]$	... ...	$D[m-1], D[m]$
Thread 2	$Q[2]$	$D[0], D[1]$	$D[2], D[3]$	$D[4], D[5]$	... ...	$D[m-1], D[m]$
... ...	... ...	... ...	... ...	... ...	... ...	$D[m-1], D[m]$
Thread n	$Q[n]$	$D[0], D[1]$	$D[2], D[3]$	$D[4], D[5]$	... ...	$D[m-1], D[m]$

O PPE (Power Processor Element) fica responsável por gerenciar a E/S, alocar tarefas e escalonar os SPEs. Ele lê as entradas do banco do disco, transmite as sequências para os SPEs, os quais realizam os alinhamentos de forma independente. Esses passos podem ser vistos na Figura 52.

Para o caso de sequências de tamanho 256, os autores observaram que 92,7% do tempo é gasto em computação e apenas 7,3% é gasto em transferências de

mensagens.

Os tempos de execução para as implementações são mostrados na Figura 53. Esses tempos não incluem transferências iniciais da base.

Cada PE no FPGA consome em torno de 110 slices o que representa algo em torno de 500 PEs em um FPGA Xilinx Virtex-4 LX160-11 e o clock do FPGA atingiu 80 MHz

Figura 52 Fluxograma da distribuição de tarefas no Cell BE (Benkrid, et al. 2012)

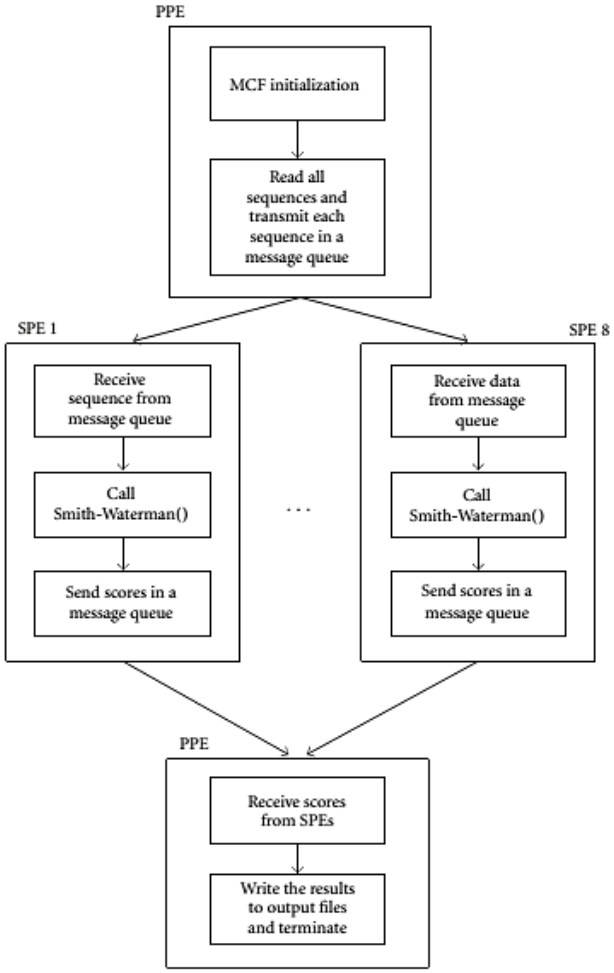


Figura 53 Tempos de execução para todas implementações (Benkrid, et al. 2012)

Query length	FPGA time (sec)	GPU time (sec)	Cell BE time (sec)	GPP time (sec)
4	1.5	4.1	0.5	24
8	1.6	4.1	1.0	30
16	1.6	4.3	1.3	43
32	1.6	4.7	1.4	62
64	1.6	6.7	2.5	115
128	1.6	12.8	5.1	210
256	1.9	30.0	9.4	424
512	4.5	76	17.2	779
768	6.7	136.2	22.2	1356
1024	8.9	172.8	31.8	1817

A Figura 54 traz o dado do speedup conseguido com as implementações sobre a implementação em software. Esse dado mostra que a solução em FPGA é duas ordens de grandeza mais rápida do que a solução em CPU, com as implementações em Cell BE e GPU vindo em segundo e terceira posições respectivamente.

Figura 54 Speedup das implementações sobre a CPU (GPP) (Benkrid, et al. 2012)

Platform	GCUPS	Speed-up
FPGA	19.4	228 : 1
GPU	1.2	14 : 1
Cell BE	3.84	45 : 1
GPP	0.085	1 : 1

Já a Figura 55 elenca o tempo de desenvolvimento, em dias, empregado em cada projeto. Observa-se o esforço bem maior no projeto em FPGA.

Levando-se em conta o custo de desenvolvimento (com preço de US$20/hora), tido como o salário médio de um estudante recém graduado, na época, o custo dos projetos foi calculado e é mostrado na Figura 56. Incluiu-se no custo a compra dos equipamentos e custo de desenvolvimento. No custo de equipamentos do FPGA, GPU e Cell BE incluiu-se o preço do PC *host*.

Figura 55 Tempo de desenvolvimento das soluçoes (Benkrid, et al. 2012)

Platform	Development time in days
FPGA	300
GPU	45
Cell BE	90
GPP	1

Como se pode ver o projeto em FPGA foi 50 vezes mais caro que o projeto em CPU.

Figura 56 Custo dos projetos (Benkrid, et al. 2012)

Platform	Purchase cost ($)	Development cost ($)	Overall cost ($)	Normalized overall cost
FPGA	10,000	48,000	58,000	50
GPU	1450	7,200	8,650	8
Cell BE	8,000	14,400	22,400	19
GPP	1000	160	1160	1

Figura 57 Desempenho por dolar gasto (Benkrid, et al. 2012)

Platform	Performance (MCUPS) per $ spent	Normalized performance per $ spent
FPGA	0.34	4.6
GPU	0.14	1.9
Cell BE	0.17	2.3
GPP	0.07	1

Na Figura 57 é, então, mostrado o índice desempenho por dólar gasto no projeto, obtido dividindo-se o desempenho em GCUPS (Giga Cell Updates per Second) da Figura 54 pelo custo da Figura 56.

Os resultados mostram a liderança da plataforma em FPGA como a mais eficiente para a aplicação.

Já na Figura 58 é mostrado o consumo de potência das plataformas. Nesse cenário, a solução em FPGA é três ordens de grandeza mais eficiente que a solução em CPU.

Fato consolidado na Figura 59, em que constata-se que a medida de MCUPS

por watt beneficia fortemente a plataforma em FPGA.

Figura 58 Eficiência energética das plataformas (Benkrid, et al. 2012)

Platform	Power (watt)	Energy (joule)	Normalized energy consumption
FPGA (clocked at 80 MHz)	39	73	0.0017
GPU	56	1682	0.04
Cell BE	140	1317	0.03
GPP	100	42400	1

Figura 59 Desempenho por Watt (Benkrid, et al. 2012)

Platform	Performance (MCUPS) per watt	Normalized performance per watt
FPGA	508	584
GPU	22	25
Cell BE	27	31
GPP	0.87	1

Já na comparação do desempenho por dólar gasto, a GPU assume a liderança seguida pela solução com CPU, como mostra a
Figura 60. Embora no longo prazo o menor gasto de energia do FPGA possa levar a um menor custo de manutenção da solução.

Figura 60 Desempenho por dólar e desempenho por watt (Benkrid, et al. 2012)

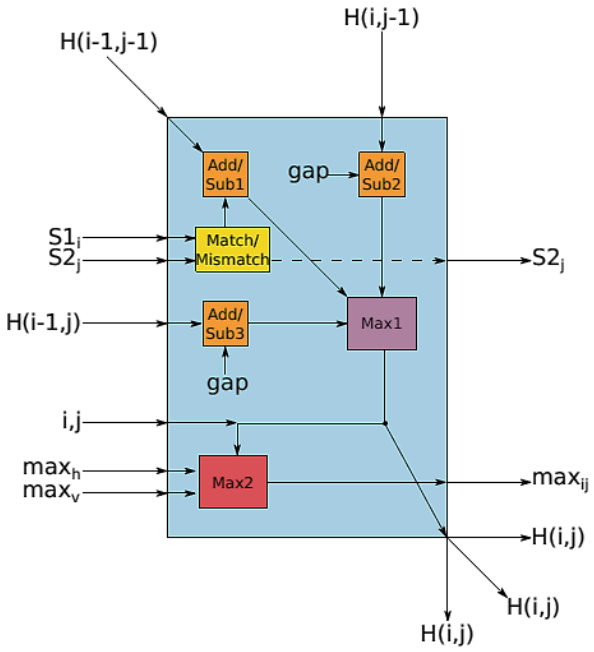
Platform	Performance (MCUPS) per $	Performance (MCUPS) per watt
FPGA	0.34	508
GPU	1.27	196
Cell BE	0.17	27
GPP	1.18	13.7

5.3.4 Hardware Implementation of the Smith-Waterman Algorithm using a Systolic Architecture (Marmolejo-Tejada, et al. 2014)

Os autores trazem nesse trabalho o projeto de um acelerador de hardware para alinhamentos locais de DNA. A Figura 61 mostra a arquitetura do PE utilizado. Três somadores e três comparadores foram necessários para computar a equação do algoritmo de Smith-Waterman. Na Figura 62 é mostrado a disposição dos PEs como um array sistólico.

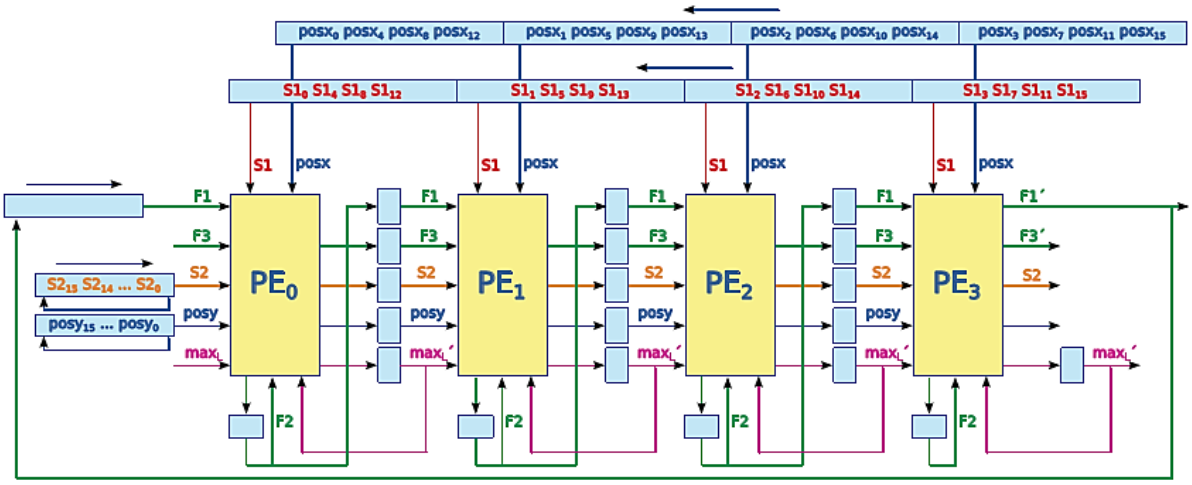
Quando toda a matriz de escores está computada, a fase de traceback é iniciada pelo maior valor de escore encontrado. Nenhuma informação sobre a implementação da fase de traceback é dada.

Figura 61 Arquitetura interna do PE (Marmolejo-Tejada, et al. 2014)



O desempenho da implementação em FPGA foi comparado com uma aplicação em Python, rodando numa CPU Intel Core i7-3770. O tamanho máximo de cadeia suportado pela implementação no FPGA Stratix IV EP4SGX230KF40C2 da Altera foi de 256 nucleotídeos.

Figura 62 PEs dispostos num array sistólico (Marmolejo-Tejada, et al. 2014)



A Figura 63 mostra a utilização de recursos do FPGA, bem como a frequência atingida. Para a configuração com cadeia de tamanho 256, foram instanciados 64 PEs.

Foi reportado apenas o tempo teórico de execução e nenhuma menção à plataforma de prototipação em FPGA foi feita. A Equação 4.2 mostra o cálculo do desempenho para um único alinhamento de tamanho 256, em que n é o tamanho da cadeia e q o número de PEs. Os autores informam que sua implementação em software levou 316,6 ms para processar um único alinhamento. Estima-se que esse tempo encontra-se superdimensionado pela implementação em Python, ou o overhead de computação para realizar um alinhamento está sobrepujando o tempo de computação para o caso de um alinhamento apenas. Assim, o speedup de 5973,6 também estaria superdimensionado.

Figura 63 Recursos utilizados do FPGA e frequência atingida (Marmolejo-Tejada, et al. 2014)

NTs	16	64	256	512
PEs (n)	4	16	64	128
Combinational ALUTs	1033	5748	54097	195808
Dedicated registers	1211	11139	142957	550127
Maximum frequency (MHz)	104.05	77.27	57.94	

$$\frac{\left(\frac{n^2}{q}+8n\right)}{f_{max}} = \frac{\left(\frac{256^2}{64}+8*256\right)}{57{,}94} = 53\mu s \quad (5.3)$$

5.4 Análise Comparativa

Na Tabela 1 se agrupam os atributos quantitativos e qualitativos analisados na comparação entre os trabalhos da literatura relacionados. Os números a que as colunas se referem são descritos abaixo:

1. Algoritmo Local (Smith-Waterman) ou Global (Needleman-Wunsch).
2. Linguagem utilizada.
3. Plataforma utilizada.
4. Comparações de desempenho com outras plataformas.
5. Speedup da implementação em relação à versão em software.
6. Speedup da implementação em relação a GPU.
7. Fmax – frequência de operação máxima.

O trabalho de Aluru (Aluru e Jammula 2014) não foi incluído no quadro comparativo porque é muito superficial quanto às implementações e não apresenta dados suficientes de performance.

O speedup conseguido por Marmolejo-Tejada (Marmolejo-Tejada, et al. 2014) não pode ser considerado porque avalia apenas um alinhamento e compara com uma aplicação em Python.

Tabela 1 Quadro comparativo dos trabalhos relacionados

	Liu	Ji	Liu	Savran	Isa	Benkrid	Marmolejo-Tejada
Algoritmo	Local	Local	Local	Global	Local	Local	Local
Linguagem	C++	X10	Cuda, C++	Cuda	HDL	Handel-C	HDL
Acelerador	Xeon Phi	CPU	GPU+ CPU	GPU	FPGA	FPGA	FPGA
Comparações	CPU, GPU	CPU	CPU, GPU	CPU	CPU	CPU, GPU, Cell	CPU
Speed-up sobre SW	28,6x	0,44x	3,2x	2,21x	380,31x	228X	***
Speed-up sobre GPU	0,74x	-	2,9x	-	-	16,3x	-
Fmax	1,05 GHz	2,27 GHz	3,5 GHz 1,06 GHz	1,1 GHz	200 MHz	80 MHz	58 MHz

Dos trabalhos relacionados em GPP pode-se concluir que além de tentativas em relação a novas linguagens como o trabalho de Ji (Ji, Liu e Yang 2012), o ponto forte é a busca de aceleração em SIMD, bem como a utilização de multiprocessadores na plataforma Xeon Phi. Para a arquitetura dos GPP este trabalho propõe três versões: a versão canônica, a versão em multi-threads e a versão SIMD. Todas apresentam otimizações não encontradas nos trabalhos relacionados, seja a utilização do vetor de escores, seja a utilização dos direcionais, a compressão dos direcionais em 2 bits, bem como o desenrolamento de loops. A versão SIMD é inspirada na proposta de Liu (Liu, Wirawan e Schmidt 2013), e aqui se propõe a linearização da matriz de escores e a simplificação dos buffers utilizados, além do uso da matriz de direcionais.

Para a arquitetura das GPUs, alinhado com o trabalho de Savran (Savran, Gao e Bakos 2014), as threads da GPU foram lançadas para se realizar alinhamentos completos e sem sincronização entre as threads. Aqui a linguagem OpenCL é utilizada para implementar as versões em GPU. Contudo, como a linguagem em OpenCL é heterogênea na sua arquitetura alvo, os mesmos kernels puderam

ser avaliados na plataforma dos GPPs. Assim nomeou-se essas versões em implementações em OpenCL no Capítulo 7.

Como otimizações propostas nas implementações em OpenCL, pode-se citar a compressão dos direcionais que possibilita a internalização dos direcionais nas threads em OpenCL, a utilização do desenrolamento dos laços. Outro ponto de melhoria é a extensão da ideia implementada na versão SIMD de GPP utilizando os Vector Types (tipos vetoriais) de OpenCL. São elas: OpenCL Canônico, OpenCL Unroll, OpenCL Vector Types. A versão OpenCL Vector Types se divide em duas para avaliar o desempenho com tamanho de tipos vetorias com 128 e 256 bits. Ao todo, oito versões são apresentadas para em OpenCL, quatro para GPP e quatro para GPU.

Para a plataforma dos FPGAs, os principais gargalos das implementações dos trabalhos relacionados é a banda de memória utilizada e a memória interna consumida. Neste trabalho propõe-se diversas modificações na dinâmica de processamento que beneficiam a implementação em FPGA como a total internalização dos direcionais e a ordem de leitura dos nucleotídeos para diminuir a banda de memória consumida. A arquitetura em FPGA detalhada é apresentada no Capítulo 8. São também discutidos pontos chave como a banda consumida, a utilização da memória interna e a escalabilidade da implementação.

Outro ponto de otimização em relação aos trabalhos relacionados é que todas as implementações nas três plataformas foram otimizadas para realizar múltiplos alinhamentos em detrimento do alinhamento único, que é o foco da maioria das implementações da literatura. Esse ponto é crucial para o problema forense que pode apresentar a busca em grandes bancos de DNA. Nesse desafio, os recursos da plataforma precisam ser compartilhados entre diversos alinhamentos independentes e executados em paralelo.

6

Implementações em GPP

Nessa seção são apresentadas as implementações em software utilizadas para comparação do desempenho, em relação à arquitetura de hardware proposta. Essas implementações foram otimizadas ao máximo para se obter um critério justo de comparação.

6.1 Versão Canônica

Esta versão usa um único processo de software e nenhum tipo de paralelismo além do paralelismo de instruções extraído pelo compilador. Entretanto ela foi otimizada para obter o melhor desempenho possível, uma vez que ela apresenta também a matriz de direcionais e o vetor de escores. Cada alinhamento é realizado como dois laços FOR aninhados, similarmente ao que é mostrado na Figura 9**Erro! Fonte de referência não encontrada.** (Pág. **Erro! Indicador não definido.**), seguidos pela operação de traceback. Essa versão serve como base de comparação para todas as outras versões, tanto em software como em hardware.

A Figura 64(a) mostra a matriz de escores resultante para o alinhamento dado como exemplo no Capítulo 2. No passo de traceback do algoritmo de NW, então, percorre-se a matriz a partir da diagonal inferior direita até a diagonal

superior esquerda sempre em busca do mínimo local.

Convenientemente, utilizando-se a codificação abaixo para um alinhamento entre as sequências $A = (a_1, a_2, \cdots, a_{\mathrm{n}})$ e $B = (b_1, b_2, \cdots, b_{\mathrm{m}})$. Ao invés de se armazenar os escores propriamente ditos, apenas é necessário armazenar os direcionais, que podem ser armazenados em apenas um byte. A

Figura 64(b) mostra a matriz de direcionais $D(i,j)$ correspondente à matriz escores $M(i,j)$.

$$direcionais(a_i, b_i) = \begin{cases} 0 & a_i = b_i\ (acerto) \\ 1 & a_i \neq b_i\ (troca) \\ 2 & b_i = -\ (inserção) \\ 3 & a_i = -\ (remoção) \end{cases}$$

Figura 64 Matriz de escores $M(i,j)$ e a sua correspondente Matriz de direcionais $D(i,j)$

$M(i,j)$

		a	**a**	**g**	**t**
	0	−1	−2	−3	−4
a	−1	1	0	−1	−2
c	−2	0	0	−1	−2
c	−3	−1	−1	−1	−2
g	−4	−2	−2	0	−1
t	−5	−3	−3	−1	1

(a)

$D(i,j)$

		a	**a**	**g**	**t**
a		0	0	3	3
c		2	1	3	3
c		2	2	1	3
g		2	2	0	3
t		2	2	2	0

(b)

O traceback é, então, realizado a partir da matriz de direcionais e a matriz de escores não é mais necessária. Esse fato leva à próxima otimização que é o uso de um vetor de escores.

O vetor de escores corresponde à uma linha (ou coluna) da matriz de escores necessária para satisfazer a dependência de dados para o cálculo da próxima linha.

Figura 65 Calculo da matriz de direcionais utilizando um vetor de escores

$V(i)$		$\boldsymbol{a}$	$\boldsymbol{a}$	$\boldsymbol{g}$	$\boldsymbol{t}$
	0	-1	-2	-3	-4
$\boldsymbol{a}$					
$\boldsymbol{c}$					
$\boldsymbol{c}$					
$\boldsymbol{g}$					
$\boldsymbol{t}$					

(a)

$V(i)$		$\boldsymbol{a}$	$\boldsymbol{a}$	$\boldsymbol{g}$	$\boldsymbol{t}$
$\boldsymbol{a}$	-1	1	0	-1	-2
$\boldsymbol{c}$					
$\boldsymbol{c}$					
$\boldsymbol{g}$					
$\boldsymbol{t}$					

(b)

A Figura 65(a) e (b) mostram a dinâmica da utilização do vetor de escores. O uso do vetor dos escores diminui a complexidade de espaço de $O(mn)$ para $O(n)$, enquanto o uso dos direcionais não reduz a complexidade de espaço, mas diminui no mínimo pela metade o uso de memória. Isso ocorre porque os escores precisam de, no mínimo, 2 Bytes para serem armazenados, enquanto os direcionais podem ser guardados com 1 Byte.

6.2 Versão Multi-Threads

Esta versão explora o paralelismo no nível de threads. Os trabalhos das *threads* foram divididos para que cada *thread* computasse um alinhamento completo. Essa escolha procurou evitar que a perda de desempenho por sincronização entre *threads* prejudicasse o ganho pretendido com a paralelização.

As *threads* foram iniciadas no começo da aplicação e cada uma tem um espaço de n alinhamentos para processar independentemente. Dessa maneira, não há *overhead* de múltiplos lançamentos das *threads*.

A versão em threads explora o paralelismo de processos usando a infraestrutura de OpenMP para lançar threads cuja operação corresponde à versão canônica. OpenMP gerencia e lança automaticamente o melhor número

de threads visando o desempenho. Nesse caso, como não há dependência de dados nem sincronização entre threads, a execução delas pode se dar fora de ordem. A Figura 66 exibe o trecho de código com o lançamento das threads. Observa-se a estrutura omp parallel que gerencia automaticamente as threads. A função omp_get_thread_num retorna a identificação da thread necessária para a separação da carga de trabalho entre as threads. Por isso as threads recebem o atributo alignment_by_threads que é o número de alinhamentos realizados por thread para que elas possam se localizar no seu espaço de trabalho no vetor score_vector.

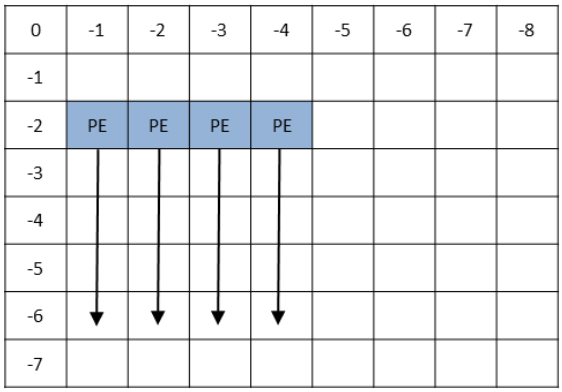
Figura 66 Trecho de código do lançamento das Threads

```
int i;
int* id;

omp_set_num_threads(num_threads);

#pragma omp parallel
{
      id      = new int[1];
      id[0]   = omp_get_thread_num();
      needlemanwunsch_threads((void*)id, tam_sequence, score_vector,
                                         alignment_by_threads,
                                         alignment_by_individuals);
}
```

Além do paralelismo multithread, nessa versão introduziu-se a otimização de se desenrolar o laço da computação dos escores. Como mostra a Figura 67, o laço da computação foi desenrolado 4 vezes. Assim o laço computa quatro pontos por vez. Esses pontos correspondem na Figura 67 às posições com o nome PE (Processing Element) e o PE mais à direita armazena os seus resultados no escore vector. Um PE representa uma unidade de computação ativa. Uma thread processa um alinhamento completo e 4 PEs por vez.

A partir do tamanho de 4 para o desenrolar do laço, o processamento apresentou piora no desempenho. Então esse foi o tamanho escolhido.

Figura 67 Desenrolamento do loop do processamento dos escores



Partindo da ideia de desenrolar o laço quatro vezes, observa-se que existem quatro direcionais sendo gerados ao mesmo tempo. Dessa forma na versão em threads, foi introduzida a melhoria de se comprimir mais ainda a representação dos direcionais, de forma que quatro direcionais caibam em um único byte.

Na linguagem C, essa compressão pode ser feita com o operador de campo de bit além do uso de uma *Union*. Essa melhoria otimiza a utilização, acesso e localidade da memória, uma vez que a matriz de direcionais é reduzida para um quarto do tamanho usado na versão canônica. A Figura 68 mostra detalhes de implementação dessa compressão.

Figura 68 Uma estrutura e uma Union na linguagem C usadas para comprimir os direcionais

```
struct bit {
    unsigned char nc1 :2,
                  nc2 :2,
                  nc3 :2,
                  nc4 :2;};
```

```
typedef union MyUnion {
          unsigned char byte;
          struct bit bits;
} byte_Union;
```

Ao final do processamento dos escores, então o traceback é realizado através da matriz de direcionais comprimida.

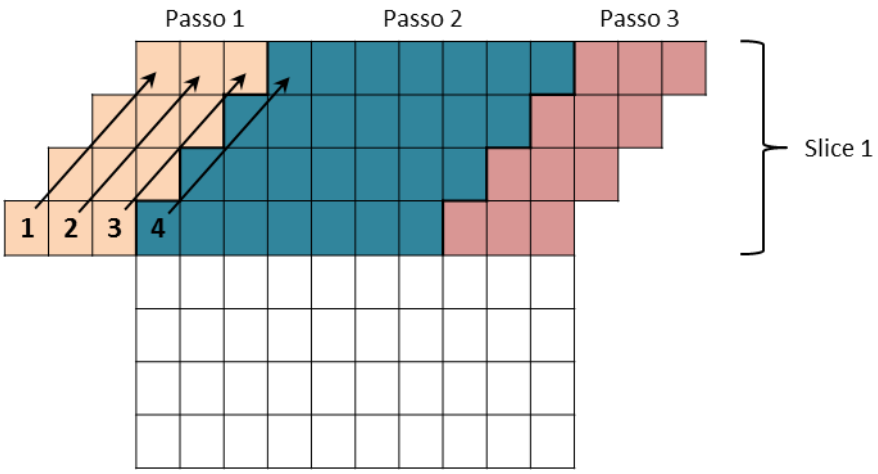
6.3 Versão SIMD

A versão SIMD (Single Instruction Multiple Data) utiliza instruções intrínsecas de 128-bits dos processadores da Intel. Essa versão implementa uma versão melhorada do que foi proposto por Liu (Liu, et al. 2014), como sugere a Figura 34.

Ao invés de vários buffers da implementação de Liu (Liu, et al. 2014), a matriz de escores foi linearizada seguindo as diagonais para melhorar a localidade dos dados em memória. As linhas da matriz são particionadas numa divisão de slices determinada pelo tamanho da instrução SIMD existente no processador. Nesse caso, foi utilizado a altura 8 linhas, que correspondem a 128 bits da instrução SIMD dividido por 16 bits da representação do escore. Dessa forma 8 posições são computadas em paralelo. A Figura 69 mostra essa divisão e o sentido da linearização.

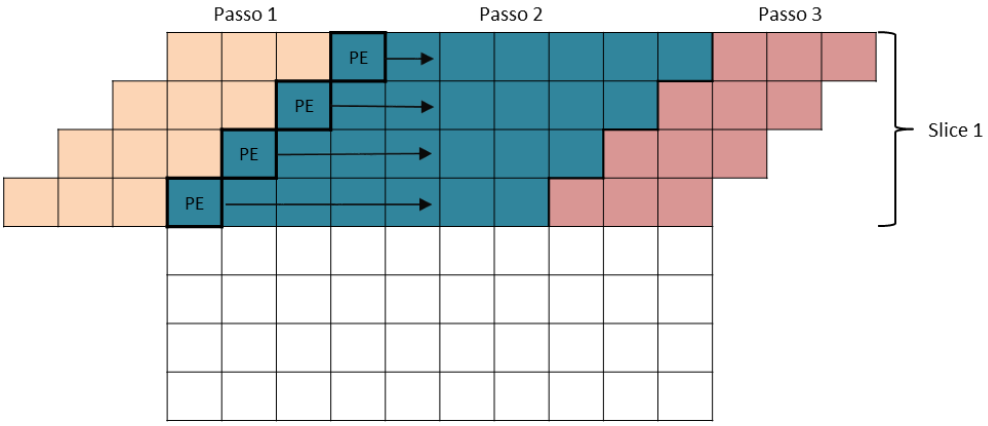
A linearização da matriz de escores diminui o número de faltas de memória cache em detrimento do acesso a diagonais em uma matriz 2D. Os passos transientes 1 e 3, mostrados na Figura 69, são computados em laços separados a fim de se minimizar as estruturas condicionais na implementação do passo estacionário 2.

Figura 69 Linearização da matrix de escores no sentido das diagonais



A Figura 70 mostra a disposição das posições da matriz de escores sendo computadas ao mesmo tempo, representadas na figura como PEs (Elementos de Processamento). Embora a Figura 70 mostre a matriz na forma bidimensional, na verdade ela está linearizada no sentido das anti-diagonais.

Figura 70 Sentido da computação dos escores



Foi utilizado o tipo de dados com 128 bits da Intel chamado de *__m128i*, e para se acessar os 8 inteiros de 16 bits dentro do tipo *__m128i* utilizou-se a union de C mostrada na Figura 71.

Figura 71 Acessando os 8 inteiros de 16 bits no tipo de 128 bits *__m128i*

```
typedef union  union_128{
       __m128i dado_128;
       short m128i_i16[8];
}m128i__;
```

Na Tabela 2 são mostradas as funções vetoriais intrínsecas necessárias para a implementação do algoritmo de NW, em seguida a extensão do conjunto de instruções que a implementa e o pseudocódigo de sua operação.

As funções intrínsecas são mnemônicos para instruções do processador e todas as instruções usadas têm latência de 1 ciclo. Por exemplo a função _mm_sub_epi16 é implementada pela instrução "psubw xmm, xmm".

Tabela 2 Tabela do conjunto de funções intrínsecas utilizado, o conjunto de instruções a que a função pertence e o pseudocódigo de sua operação

Função Intrínseca	Conjunto de instruções	Operação
__m128i _mm_set_epi16 (**short e7, short e6, short e5, short e4, short e3, short e2, short e1, short e0**)	SSE2	dst[15:0] := e0 dst[31:16] := e1 dst[47:32] := e2 dst[63:48] := e3 dst[79:64] := e4 dst[95:80] := e5 dst[111:96] := e6 dst[127:112] := e7

__m128i _mm_set1_epi16 (**short a**)	SSE2	FOR j := 0 to 7
		  i := j*16
		  dst[i+15:i] := a[15:0]
		ENDFOR

__m128i _mm_sub_epi16 (**__m128i a, __m128i b**)	SSE2	FOR j := 0 to 7
		  i := j*16
		  dst[i+15:i] := a[i+15:i] - b[i+15:i]
		ENDFOR

__m128i _mm_add_epi16 (**__m128i a, __m128i b**)	SSE2	FOR j := 0 to 7
		  i := j*16
		  dst[i+15:i] := a[i+15:i] + b[i+15:i]
		ENDFOR

__m128i _mm_max_epi16 (**__m128i a, __m128i b**)	SSE2	FOR j := 0 to 7
		  i := j*16
		  IF a[i+15:i] > b[i+15:i]
		  dst[i+15:i] := a[i+15:i]
		  ELSE
		  dst[i+15:i] := b[i+15:i]
		  FI
		ENDFOR

__m128i _mm_slli_epi16 (**__m128i a, int imm8**)	SSE2	FOR j := 0 to 7
		  i := j*16
		  IF imm8[7:0] > 15
		  dst[i+15:i] := 0
		  ELSE
		  dst[i+15:i] := ZeroExtend(a[i+15:i] <<imm8[7:0])

		FI ENDFOR
__m128i _mm_srli_si128 **(__m128i a,** int **imm8)**	SSE2	tmp := imm8[7:0] dst[127:0] := a[127:0] >> (tmp*8)

__m128i _mm_mask_abs_epi16 **(__m128i src, __mmask8 k, __m128i a)**	AVX-512	FOR j := 0 to 7
		i := j*16
		IF k[j]
		dst[i+15:i] := ABS(a[i+15:i])
		ELSE
		dst[i+15:i] := src[i+15:i]
		FI
		ENDFOR

Como o processador utilizado não possuía o conjunto de instruções AVX-512 (disponível nos processadores da família Xeon lançados em 2015) a função _mm_mask_abs_epi16 precisou ser substituída pelo laço do trecho de código da Figura 72.

Figura 72 Acessando os 8 inteiros de 16 bits no tipo de 128 bits *__m128i*

```
for (int i = 0; i < slice_length; i++){
    cust_diag.m128i_i16[i] = result_cmp.m128i_i16[i] ? CUSTO_MATCH : -CUSTO_MATCH;
}
```

Claramente, o desempenho obtido deve ser penalizado pelo uso do laço para acessar individualmente cada inteiro de 16 bits e depois realizar a computação.

A extensão da implementação para se aumentar o tamanho do slice e, por conseguinte, melhorar o desempenho da aplicação depende de se trocar as funções intrínsecas utilizadas. Esse fato somado à impossibilidade de migração do código para um processador mais antigo ou até mesmo para outro fabricante de CPU exemplifica a baixa portabilidade do código utilizando instruções SIMD.

Uma vez que a versão SIMD contém otimizações de paralelismo em nível de instrução, também se pode explorar o paralelismo em threads. Então, novamente, a infraestrutura de OpenMP foi utilizada para lançar threads. Assim a chamada versão SIMD contém esses dois tipos de paralelismo inclusos.

7

Implementações em OpenCL

Visando utilizar a característica de OpenCL de ser heterogêneo apenas um kernel de OpenCL foi implementado para rodar tanto em uma plataforma com CPUs quanto na plataforma com GPUs.

7.1 Versões em OpenCL Canônica e com desenrolamento de Loop

Assim como foi sugerido por Savran (Savran, Gao e Bakos 2014) e mostrado na Figura 41(b) (Pág. 66), aqui também uma única thread de OpenCL fica responsável por um alinhamento completo. Assim, os três passos do algoritmo de NW são feitos todos dentro do kernel. Dessa maneira diminui-se a comunicação com a memória global e, por conseguinte, se melhora o desempenho do algoritmo. Como mostra o autor, uma prévia implementação utilizando vários *kernels* em um mesmo alinhamento obteve desempenho muito abaixo da versão independente. Assim, lançar muitos alinhamentos independentes em paralelo é melhor do que vários *kernels* em um mesmo alinhamento. Esse fato se deve ao overhead de comunicação entre threads e a existência de threads ociosas. Outro fato que coopera para a perda de desempenho é o compartilhamento de recursos, visto que a memória compartilhada entre threads é uma dezena de vezes mais lenta que a memória

privativa da thread.

Na implementação em OpenCL também se usou o vetor de escores, similar ao feito nas versões em GPP. A característica linear do vetor de escore, do tamanho da sequência original, foi essencial para permitir o acesso à memória global de forma coalescente.

A primeira versão em OpenCL é chamada de versão OpenCL Canônica que se apresenta similar a versão canônica de GPP apresentada na Seção 6.1. Nessa versão, assim como na canônica de GPP, também apresenta o vetor de escores e matriz de direcionais com 8 bits por posição. Os direcionais são armazenados na memória global da thread.

Na extensão da versão OpenCL Canônica segue a versão OpenCL Unroll que se assemelha à versão multi-threads de GPP da Seção 6.2. Nessa versão, como na multi-threads, o loop interno foi desenrolado 4 vezes e os direcionais foram comprimidos para 2 bits.

Dessa forma, a matriz de direcionais comprimida tem 4 direcionais por Byte. Não se pode utilizar a estratégia proposta na Figura 68 de compressão dos direcionais porque a linguagem OpenCL na versão 1.2 não dá suporte ao operador de campo de bits. Então precisou-se utilizar operações de shift e máscaras de bits para se trabalhar com a representação comprimida.

Essa compressão é necessária para que os direcionais possam ser armazenados na escassa memória interna thread, da ordem de kilobytes. A memória interna da thread só não é menor do que o número de registradores e, por exemplo, não comporta a matriz de direcionais e o vetor de escores ao mesmo tempo. Portanto, foi escolhida a matriz de direcionais porque ela precisa ser acessada randomicamente na operação de traceback. Esse acesso não coalescente prejudicaria o desempenho global da aplicação.

Uma vez que os direcionais estão armazenados dentro da thread, a operação de traceback pode ser realizada sem que nenhuma transferência desnecessária entre o host e o device seja feita.

No caso da memória global, eventualmente, a quantidade de memória

disponível no device pode não ser suficiente para armazenar todo o problema. Então, é necessário que se faça diversas iterações de envio de dados, processamento e recebimento de resultados por exemplo para computar uma única consulta a um grande banco de dados.

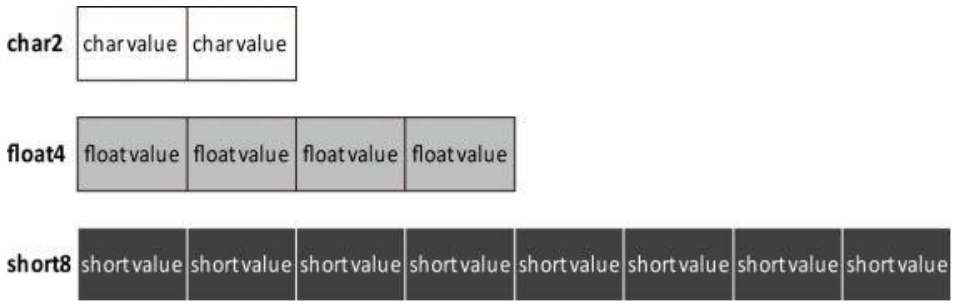
Em OpenCL, pôde-se explorar quatro configurações para se colher os tempos de execução:

- OpenCL CPU Canônica – *kernels* rodando em uma configuração dual CPU com direcionais na memória global.
- OpenCL CPU Unroll – *kernels* rodando em uma configuração dual CPU com direcionais comprimidos, armazenados internamente e com desenrolamento de laço.
- OpenCL GPU Canônica – *kernels* rodando em uma única GPU com direcionais na memória global.
- OpenCL GPU Unroll – *kernels* rodando em uma única GPU com direcionais comprimidos, armazenados internamente e com desenrolamento de laço.

7.2 Versão OpenCL utilizando Tipos Vetoriais

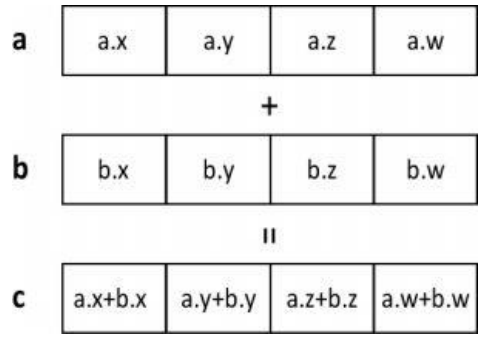
A API de OpenCL na versão 1.0 adiciona o suporte a tipos vetoriais. Três exemplos de tipos vetoriais podem ser vistos na Figura 73, onde os tipos char2 (duas variáveis do tipo char), float4 (quatro variáveis do tipo float) e short8 (oito variáveis do tipo short) e sua estrutura empacotada são destacadas.

Figura 73 Estrutura dos tipos vetoriais em OpenCL (The OpenCL Programming Book s.d.)



Os operadores matemáticos em OpenCL foram sobrecarregados para suportar os tipos vetoriais. Dessa forma com o operador de adição " + " pode-se somar dois tipos vetoriais de n posições. Por exemplo, na Figura 74 é mostrado resultado da operação de adição com duas variáveis vetoriais com 4 posições. No detalhe de cada variável, pode ser visto também que os operadores '.xyzw' são utilizados para acessar individualmente valores do vetor empacotado.

Figura 74 Operação de adição de um tipo vetoriais em OpenCL com quatro posições (The OpenCL Programming Book s.d.)



Utilizando os tipos vetoriais de OpenCL, a segunda versão implementada em OpenCL, chamada de **OpenCL Vector Types** foi inspirada na organização e linearização dos dados encontrados na Versão SIMD e mostrados na Figura 69 (Pág. 91).

Adicionalmente, os direcionais também são comprimidos para caberem dentro do kernel, o que faz com que a versão OpenCL Vector Types uma fusão da versão SIMD da Seção 6.3 com a versão em OpenCL com direcionais comprimidos. A Figura 75 mostra, do lado esquerdo, as máscaras utilizadas para acessar os direcionais com 2 bits, e no lado direito o acesso aos direcionais com o operador de OpenCL ".s1" para acessar, por exemplo o componente 1.

Assim os direcionais puderam ser armazenados na memória privativa da thread, evitando acesso não coalescente à memória global, melhorando a localidade da memória de direcionais e melhorando o desempenho pelo acesso à memória de maior throughput.

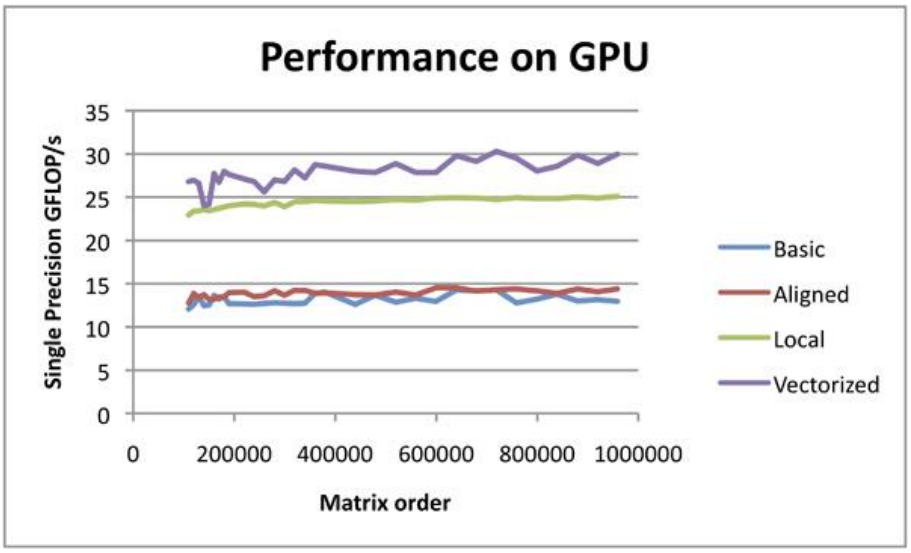
Figura 75 Compressão dos direcionais na versão Opencl VectorTypes

```
#define MASK_NCL1 0x0003     packed_directions = MASK_NCL1 & (directionShort8.s8 << 0)
#define MASK_NCL2 0x000C                       | MASK_NCL2 & (directionShort8.s7 << 2)
#define MASK_NCL3 0x0030                       | MASK_NCL3 & (directionShort8.s6 << 4)
#define MASK_NCL4 0x00C0                       | MASK_NCL4 & (directionShort8.s5 << 6)
#define MASK_NCL5 0x0300                       | MASK_NCL5 & (directionShort8.s4 << 8)
#define MASK_NCL6 0x0C00                       | MASK_NCL6 & (directionShort8.s3 << 10)
#define MASK_NCL7 0x3000                       | MASK_NCL7 & (directionShort8.s2 << 12)
#define MASK_NCL8 0xC000                       | MASK_NCL8 & (directionShort8.s1 << 14);
```

O resultado esperado do uso dos tipos vetoriais é que o paralelismo de instrução seja explorado ao extremo. No exemplo, já mostrado na Figura 75, a representação dos escores é feita com o tipo short8 com largura de 128 bits (8 inteiros do tipo short com 16 bits). Assim o acesso à memória global de um short8 é feita de forma coalescente com a leitura de 128 bits por vez. Essa estratégia de leitura deve aumentar a banda de leitura da memória global, uma vez que as GPUs, por exemplo, em uma leitura sempre trazem blocos múltiplos de 128. De forma semelhante, essa estratégia na CPU se beneficia das transferências em rajada da memória principal.

Um exemplo de uma série de otimizações no desenvolvimento de aplicações em OpenCL pode ser visto na Figura 76, em que se mostra as versões básica, alinhada, com uso de memória local (compartilhada do grupo de threads) e vetorizada no cálculo do produto de matrizes esparsas.

Figura 76 Sequências de otimizações no produto de matrizes esparsas (Catanzaro s.d.).



Da Figura 76 observa-se o ganho obtido ao se vetorizar a aplicação se utilizando dos tipos vetoriais de OpenCL, no exemplo, em uma GPU da ATI/AMD.

Outro ponto importante é a portabilidade trazida pela implementação em OpenCL do paralelismo SIMD. Claramente, se a CPU em que se estiver rodando a aplicação OpenCL Vector Types não suportar o conjunto de instruções necessário, o desempenho ficará aquém do desejado mas rodará sem problemas. O que não ocorre com a versão SIMD, se o processador não der suporte, ocorre uma falha de segmentação e o programa finaliza.

Outro ponto de melhoria da manutenibilidade do código com a possibilidade de se estender a dimensão do paralelismo SIMD de 128 para 256 ou 512 com a evolução do suporte de hardware.

Assim as duas configurações a seguir podem ser testadas a fim de se obter a melhor configuração.

- OpenCL VectorTypes CPU – *kernels* com tipos vetoriais de 256 bits rodando em CPU.
- OpenCL VectorTypes GPU – *kernels* com tipos vetoriais de 256 bits rodando em uma GPU.

8

Implementação em FPGA

Este capítulo descreve em detalhes a plataforma de hardware utilizada, a arquitetura proposta e sua implementação em FPGA, e também contempla aspectos de desempenho esperado e escalabilidade da solução.

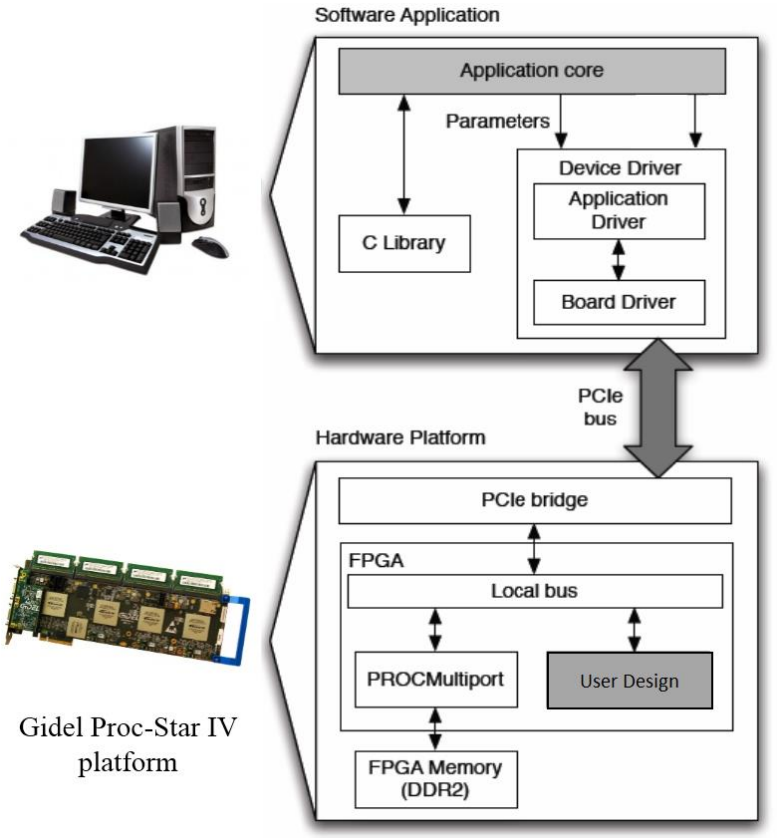
8.1 Plataforma de FPGA Utilizada

Para prototipação da arquitetura em FPGA foi utilizada uma placa Stratix IV da Gidel (Gidel s.d.). A placa dispõe de quatro FPGAS. O ambiente de desenvolvimento da Gidel provê infraestrutura de hardware como acesso à memória externa ao FPGA através do IP Core chamado Multiport. Através do Multiport pode-se ler e escrever na memória por meio de portas através de um protocolo similar ao de uma FIFO. Na Figura 77 pode-se ver o esquema do Multiport. Todo gerenciamento da comunicação com a memória é realizado através desse módulo de controle de acesso à memória. Apenas um Multiport pode ser utilizado por memória física. Também há a possibilidade de se criar diversas portas por Multiport, cujo escalonamento é transparente ao usuário.

Na Figura 77 é mostrada a placa utilizada, cada FPGA está conectado a três bancos de memória DDR2, sendo dois bancos no padrão SODIM e um soldado na placa com capacidade de 512 MB.

Na Figura 77, também se mostra o suporte à comunicação entre o PC host e a placa. Assim as transferências de dados da memória do host e as memórias da placa são gerenciadas via software. O Envio de dados, sinais de controle e parâmetros entre o projeto do usuário em FPGA e a aplicação do usuário também é provida. Essa flexibilidade permite a utilização de parâmetros escolhidos em tempo de execução.

Figura 77 Plataforma ProcStar IV, hardware e software (Gidel s.d.)



8.2 Arquitetura proposta

Neste capítulo é proposta a arquitetura paralela de uma acelerador de hardware, sua implementação em linguagem HDL e sua prototipação em uma plataforma com FPGA.

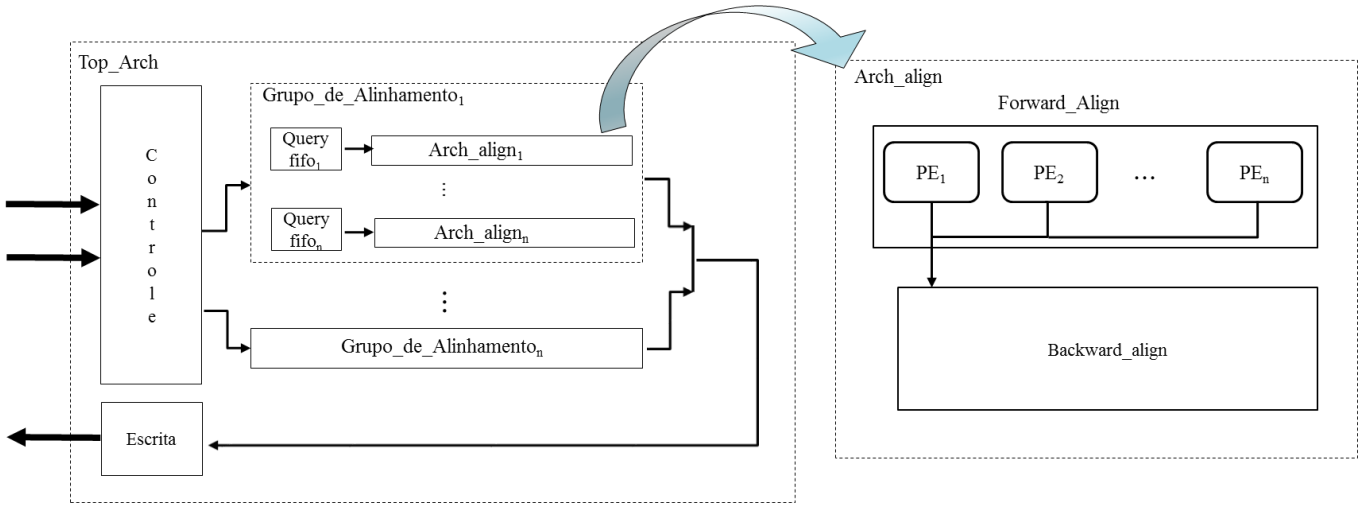
A arquitetura segue uma estrutura regular em que diversos alinhamentos podem ser agrupados formando conjuntos de alinhamentos. E esses grupos podem também podem se instanciados diversas vezes, tantos quantos

couberem no FPGA escolhido.

A Figura 78 apresenta o diagrama geral da arquitetura. O controle é responsável pela interface com a memória, alimentando os módulos internos com os nucleotídeos e interfaceando com o software do host.

Na Figura 78 pode-se ver a instanciação de diversos módulos chamados de Grupo_de_Alinhamento que correspondem a um indivíduo do banco de sequências. Cada indivíduo do banco contém várias sequências que o representam, assim cada Grupo_de_Alinhamento é um conjunto de Arch_align que são os alinhamentos de DNA propriamente ditos.

Figura 78 Visão top level da arquitetura proposta



Seguindo na abordagem top-down, cada Arch_align consiste em uma computação dos escores e no traceback, realizando as 3 fases do algoritmo de NW. Os vários elementos de processamento (PEs) dentro dos Arch_align são os módulos aritméticos que realizam as computações dos escores, segundo as equações do Capítulo 2.

A seguir são discutidos detalhes da arquitetura na proposição das soluções para resolver as fases de computação de escore e *traceback.* Em seguida, são abordados os critérios de escalabilidade, organização da memória e por fim o desempenho esperado pela arquitetura proposta.

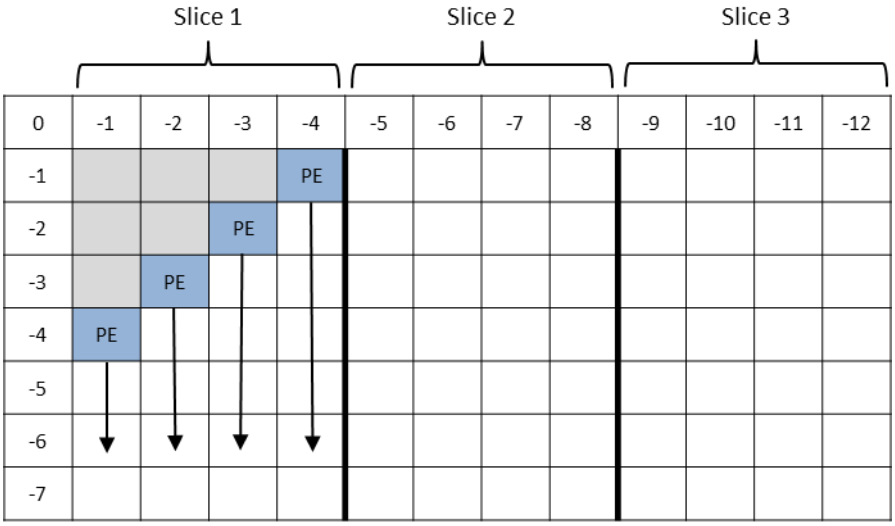
8.3 Computação dos escores

Como mostrado no Capítulo 2, o passo 2 do algoritmo de Needleman-Wunsch consiste na computação dos escores. Na arquitetura proposta, os escores são processados pelos PEs (*Processing Elements*) segundo a Equação 2.1. A Figura 79 mostra a disposição espacial dos PEs em função da dependência de dados do algoritmo. Cada PE se encontra defasado em um ciclo de *clock* em relação ao anterior na configuração de systolic array, em sincronia com os trabalhos relacionados de FPGA (Marmolejo-Tejada, et al. 2014), (Benkrid, et al. 2012) e (Isa, et al. 2014).

A Figura 79 ilustra o processo da geração dos escores. Nesse exemplo existem quatro PEs (Células destacadas em azul) que processam, em paralelo, quatro colunas, no sentido mostrado pelas setas. A quantidade de PEs forma a largura da fatia de colunas a ser processada por vez também chamada de *slice*. Ao fim do *slice* 1, o processamento recomeça da mesma forma no *slice* 2 e assim sucessivamente até o final da matriz.

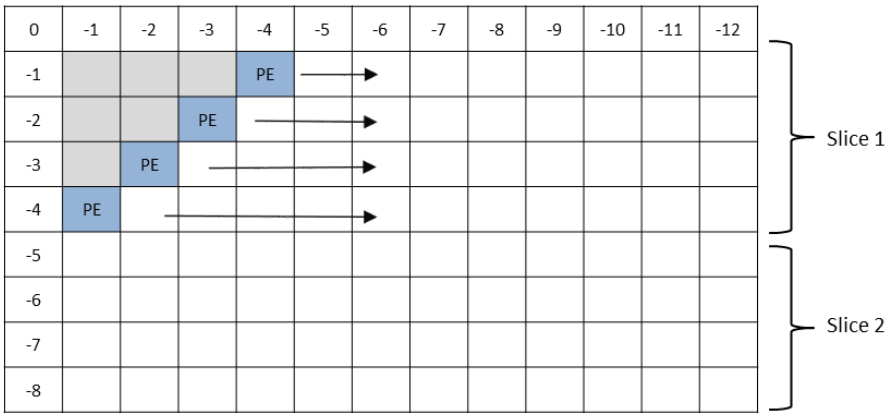
Essa maneira de processar os dados, garante que apenas um nucleotídeo da query (sequência na vertical) é lido da memória por ciclo de clock. O primeiro PE recebe da memória o nucleotídeo da query e o propaga para o segundo PE e assim sucessivamente. Da cadeia do banco de sequência, na horizontal, apenas um nucleotídeo é lido por PE durante o tempo em que se processa o slice inteiro (tamanho igual ao tamanho na query). Como existem apenas quatro nucleotídeos (a = adenina, c = citosina, t = timina, g = guanina), números arbitrários de 0 a 3 foram utilizados para representá-los, totalizando apenas 2 bits por nucleotídeo.

Figura 79 Processamento dos escores



Alternativamente, a Figura 80 mostra o sentido inverso do processamento. Dessa forma, é lido, por ciclo, um nucleotídeo do banco (sequência na horizontal), que é repassado aos PEs vizinhos. A leitura da query (sequência na vertical) é feita de 4 nucleotídeos por slice. A cadeia do banco de sequências não tem vantagem de ser armazenada internamente e o armazenamento da query quase não tem efeito no consumo de banda.

Figura 80 Sentido alternativo de processamento



Assim, o melhor sentido de se processar a computação dos escores é o sentido da Figura 79 que permite a bufferização da query e consome apenas 4 nucleotídeos do banco durante todos os ciclos de processamento de um slice.

Portanto, introduziu-se uma FIFO (First-In-First-Out) para armazenar a query assim que ela é lida da memória externa. No final do primeiro slice a query está completamente armazenada. A partir do segundo slice retroalimenta-se a FIFO e não se lê mais a query da memória externa para uma dada consulta.

A forma como os PEs são encadeados segue uma arquitetura de *array* sistólico. A quantidade de PEs instanciados é parametrizada pelo usuário, e quantos mais PEs maior o paralelismo e menor será o tempo de computação dos escores. A quantidade de lógica presente no FPGA alvo é o limitante da quantidade de PEs pretendida.

Contudo a medida em que se aumenta o número de PEs, aumenta-se também os ciclos inúteis de processamento ou ciclos de espera. Na Figura 81 são mostradas três configurações com quantidades crescentes de PEs. As células em cinza acima da linha dos PEs representam os ciclos passados antes que todos os PEs estejam computando juntos. No primeiro caso, os PEs em azul, dos 4 PEs o primeiro esteve inativo durante 3 ciclos, o segundo ficou dois ciclos inativo e o terceiro ficou um ciclo inativo, totalizando 6 ciclos de inatividade no início do slice e no fim dele. Esse processo se repete em todo slice. No segundo caso, com 7 PEs em laranja, a quantidade de ciclos inativos cresce bastante até chegar no caso extremo com os PEs em verde. Nesse caso, o primeiro e o último PE só são utilizados uma vez em todo processamento.

Figura 81 Variação da quantidade de PEs

0	-1	-2	-3	-4	-5	-6	-7	-8	-9	-10	-11	-12
-1				PE			PE					PE
-2			PE			PE					PE	
-3		PE			PE					PE		
-4	PE			PE					PE			
-5			PE					PE				
-6		PE					PE					
-7	PE					PE						
-8					PE							
-9				PE								
-10			PE									
-11		PE										
-12	PE											

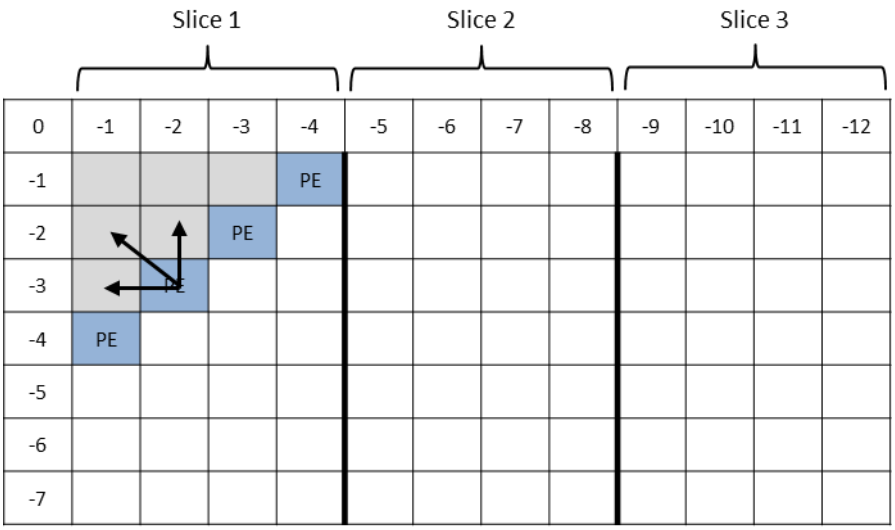
Vale ressaltar que com o aumento do número de PEs também mais banda de memória é demandada. Os recursos de lógica e banda utilizados com PEs inativos poderiam ser empregados no alinhamento de outras sequências em paralelo. Isso permite o ganho global de desempenho, em detrimento do ganho individual do processamento de um único alinhamento.

Como forma de otimizar o uso de memória interna do FPGA (diminuindo drasticamente o acesso externo à memória externa) não se utilizou a convencional matriz de escores e sim apenas um vetor. Essa otimização diminui o uso de memória interna de quadrático para linear. O vetor de escore armazena o escore resultado do último PE do *slice*, de forma transitória, para permitir o processamento do próximo *slice*.

Nessa abordagem, o resultado armazenado para a fase de *traceback* é uma matriz de direcionais. O direcional guarda a meta-informação do momento da geração do escore. Existem 4 possibilidades: acerto, troca, inserção ou remoção, os quais indicam a origem de cada escore, podendo ter sido originado de uma diagonal esquerda, da esquerda ou acima como mostra a Figura 82. No caso de um acerto ou troca, o sentido é a diagonal, e internamente o PE

distingue entre eles. Essa distinção é importante na computação do escore final do alinhamento.

Figura 82 Direções possíveis.



Dessa maneira, é necessário armazenar, por alinhamento, uma matriz de 2 bits por célula (acerto, troca, inserção ou remoção). Isso representa uma redução de uso de memória de 8 vezes, comparado com o armazenamento de uma matriz de escores de 16 bits.

8.4 Traceback

A fase seguinte à computação dos escores, que corresponde ao passo 3 do algoritmo de Needleman-Wunsch, é a fase de *traceback*. Como mencionado na Seção 8.3, a computação dos escores segue a estratégia da divisão da matriz por *slices*. Também se mencionou a utilização da matriz de direcionais a ser utilizada na fase de *traceback*.

Na Figura 83(a) é mostrada a matriz de direcionais na forma convencional bidimensional. Lembrando que o processamento dos direcionais divide a matriz em slices. A cada ciclo de clock cada PE produz 2 bits de direcionais, dessa forma os dados produzidos por todos PEs têm a forma da Figura 83(b), cuja

largura é dada por $BitsDirecionais * NrPEs$.

Essa divisão por slices facilita armazenamento dentro do FPGA da matriz de direcionais, visto que ela tem a estrutura de uma memória RAM (*Ramdom Acess Memory*).

A matriz de direcionais é, então, armazenada em uma estrutura chamada Directions_buffer, o qual possui as seguintes características:

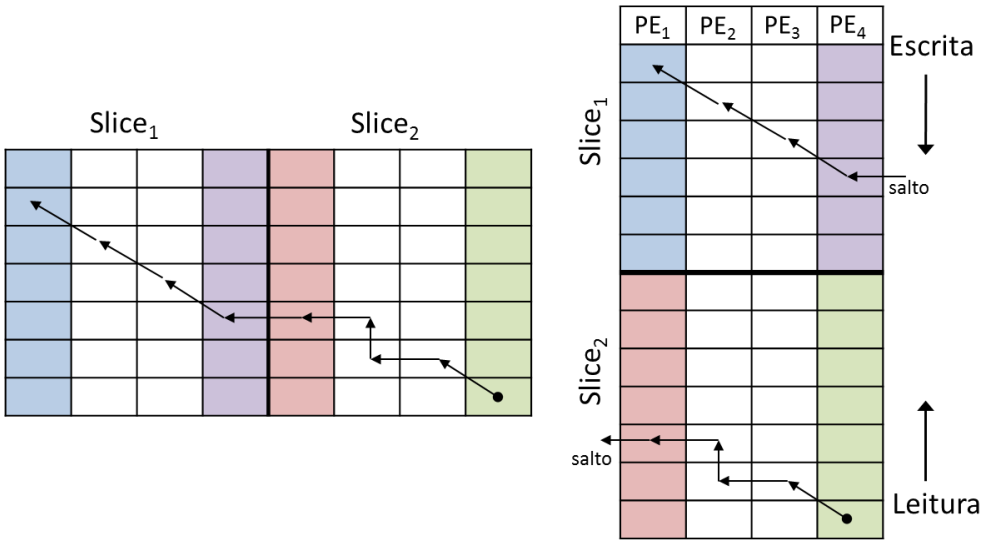
- Orientado a *slices* de processamento – cuja largura é igual a quantidade de PEs utilizada.
- Escritas de cima para baixo – seguindo o sentido da geração dos direcionais.
- Leituras de baixo para cima – seguindo o sentido do *traceback*.
- Realizar saltos – para permitir continuidade na ligação entre *slices*.

Essa estrutura é similar à de uma Pilha (LIFO – *Last-In-First-Out*) adicionada da característica de realizar saltos para simular a interface contínua entre *slices* que haveria numa matriz. Um exemplo ilustrativo pode ser visto na Figura 83 onde é apresentada a operação de *traceback* feita numa matriz convencional e no Directions_Buffer.

As escritas na estrutura do Directions_Buffer é feita de cima para baixo seguindo o processamento feito pelos PEs. No final do processamento de todos os escores do alinhamento, seguindo o último passo do algoritmo de NW, então o traceback ocorre a partir do último escore gerado do último PE.

Ao final desse processamento, a memória RAM utilizada para armazenar os direcionais está totalmente preenchida. Desse ponto, então, se processa o traceback começando pelo último endereço gerado.

Figura 83 Matriz de direcionais convencional e Directions_Buffer



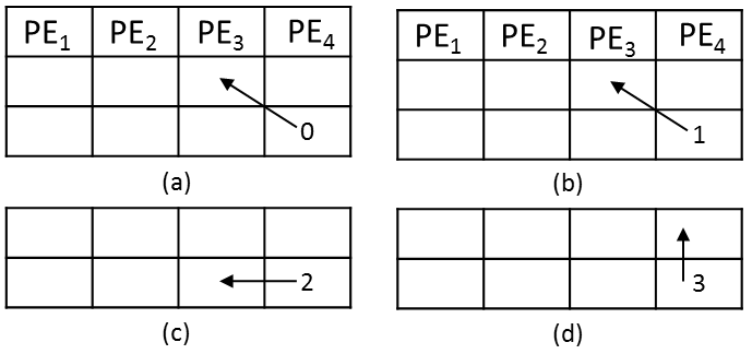
Os movimentos nessa RAM são feitos segundo o resultado armazenado na posição corrente. A cada leitura de um endereço na RAM apenas a saída de um único PE é usada por vez para se realizar o movimento de traceback.

Retomando a notação anteriormente utilizada na Seção 6.1, os direcionais podem ser representados pela convenção abaixo.

$$direcionais(a_i, b_i) = \begin{cases} 0 & a_i = b_i\ (acerto) \\ 1 & a_i \neq b_i\ (troca) \\ 2 & b_i = -\ (inserção) \\ 3 & a_i = -\ (remoção) \end{cases}$$

A Figura 84 mostra os movimentos necessários no Directions_Buffer. Um acerto e uma troca na Figura 84 (a) e (b) representam movimentos na diagonal. Então é preciso uma leitura da RAM no endereço acima do corrente e o deslocamento à esquerda do ponteiro de leitura para o PE à esquerda

Figura 84 Movimentos no Directions_Buffer



Na Figura 84 (c) é mostrado o movimento quando ocorre uma inserção. Não é necessário ler nenhum dado adicional da RAM, apenas descola-se o ponteiro de leitura para o terceiro PE. Na Figura 84 (d), uma remoção ocorre, então precisa-se ler o endereço da RAM acima sem modificar o ponteiro de leitura da posição do quarto PE.

Quando se atinge a posição de leitura no primeiro PE e se faz um movimento à esquerda, precisam-se descartar todos os endereços restantes do slice corrente e continuar o traceback na mesma linha do slice anterior. Na RAM do Directions_Buffer essa operação é implementada com um salto do tamanho do número de linhas de um slice para cima. Essa operação é vista na Figura 83(b).

8.5 Acesso e Organização da Memória Externa

A abordagem utilizada neste trabalho otimizou o acesso à memória externa, principalmente porque a sequência de query é lida apenas uma vez da memória externa e armazenada em uma FIFO interna ao FPGA. A cadeia armazenada na FIFO é reusada todo *slice* de processamento e também para os demais alinhamentos com os indivíduos restantes no banco.

Outra grande redução de acesso à memória externa é o armazenamento interno dos direcionais no Directions_Buffer, fazendo com que nenhuma outra informação precise ser lida concorrentemente ao processamento.

A organização da memória externa precisa fornecer os dados na orientação do

sentido da geração de escores. Valendo-se do fato que a maneira mais eficiente de acesso à memória é por meio de leituras de endereços sequenciais, a orientação da memória da query é mostrada na Figura 85. Cada Q_{ij} representa um nucleotídeo da memória da query.

As p sequências de query (p = Num_align_Individuos é o número de alinhamentos por indivíduo e m = tamanho das cadeias) estão dispostas na vertical. Assim quando uma palavra de $2 * p$ bits é lida da memória, um nucleotídeo por sequência é lido para todas as p sequências de tamanho m.

Figura 85 Organização de memória da Query

$$\begin{matrix} Q_{1,1} & Q_{1,2} & \cdots & Q_{1,p} \\ Q_{2,1} & Q_{2,2} & \cdots & Q_{2,p} \\ \vdots & \vdots & \ddots & \vdots \\ Q_{m,1} & Q_{m,2} & \cdots & Q_{m,p} \end{matrix}$$

Essa organização de memória permitiu o acesso aos dados para:

- Múltiplos Elementos de processamento (PEs) por alinhamento
- Múltiplos alinhamentos por indivíduo
- Múltiplos indivíduos sendo processados em paralelo.

A matriz da Figura 86 está destacada como submatrizes na Figura 87. Cada sub-matriz nxk, correspondente à primeira sequência de DNA de tamanho m de um indivíduo que foi dividida em $n = m/k$ slices, sendo processados por k PEs.

Figura 86 Sub-matriz da memória do banco de sequências

$$\begin{bmatrix} S_{1,1} & S_{1,2} & \cdots & S_{1,k} \\ S_{2,1} & S_{2,2} & \cdots & S_{2,k} \\ \vdots & \vdots & \ddots & \vdots \\ S_{n,1} & S_{n,2} & \cdots & S_{n,k} \end{bmatrix}$$

A estrutura do processamento em *slices* e o número de PEs escolhido definem quantos dados são necessários a cada *slice*. Assim para cada alinhamento são necessários tantos nucleotídeos quantos PEs foram instanciados.

Seguindo na dimensão das colunas, na Figura 87 a sub-matriz que representa o conjunto de alinhamentos que representam um indivíduo engloba p sub-matrizes da mostrada na Figura 86.

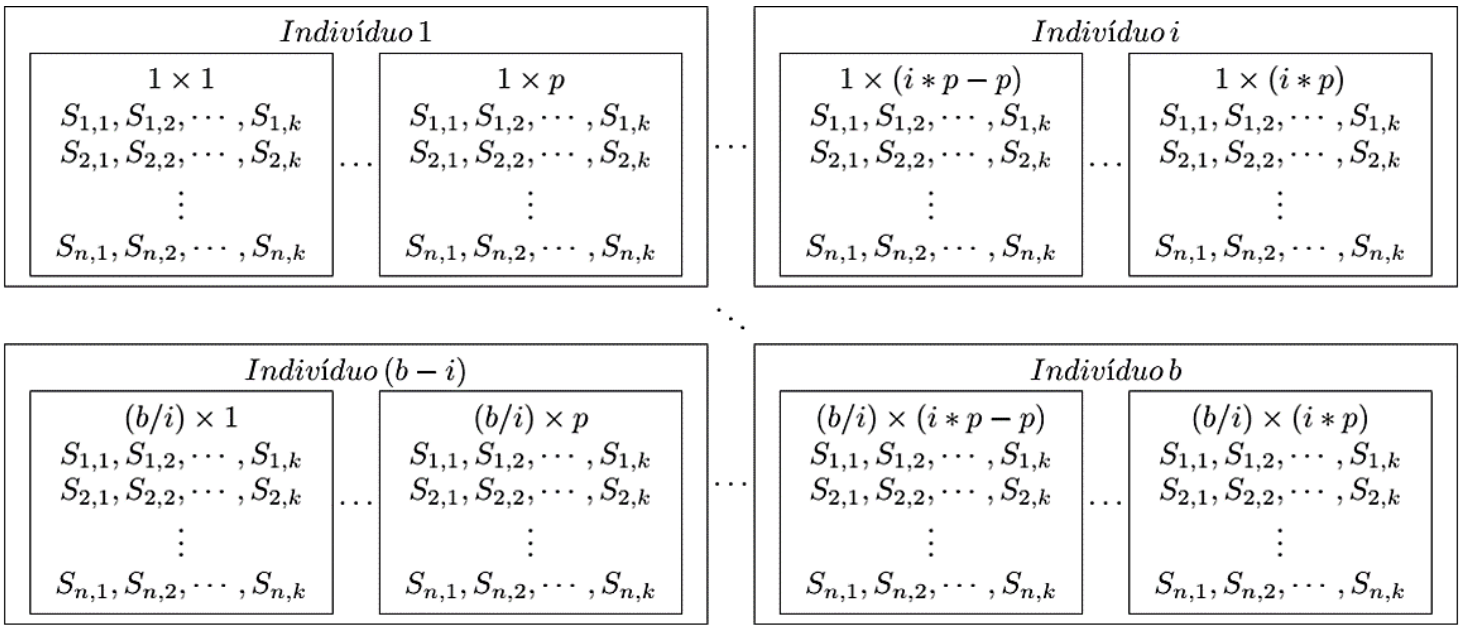
Para o processamento de i indivíduos em paralelo, é preciso se concatenar $i * p$ sub-matrizes da Figura 86, completando os dados necessários para uma iteração da arquitetura. Assim, são lidos por vez uma palavra de $k * p * i$ nucleotídeos com 2 bits e o banco inteiro com b indivíduos tem $(b/i) * n$ linhas

de $k * p * i * 2$ bits.

A medida que o número de indivíduos do banco cresce, torna-se necessário fazer o particionamento da matriz completa em mais de uma memória RAM externa ao FPGA. Para tanto, particiona-se na vertical, a matriz completa, distribuindo o armazenamento em diversos bancos de memória, preservando a organização dos dados e sua forma de acesso.

Seguindo essa organização, o banco de sequências de DNA indivíduos precisa ser reestruturado antes do processamento. Essa reorganização pode ser feita à medida em que os indivíduos são inseridos no banco de forma off-line e incremental, sem prejuízo ao desempenho no momento de uma consulta.

Figura 87 Organização da memória do banco de sequências



8.6 Módulos de hardware Implementados

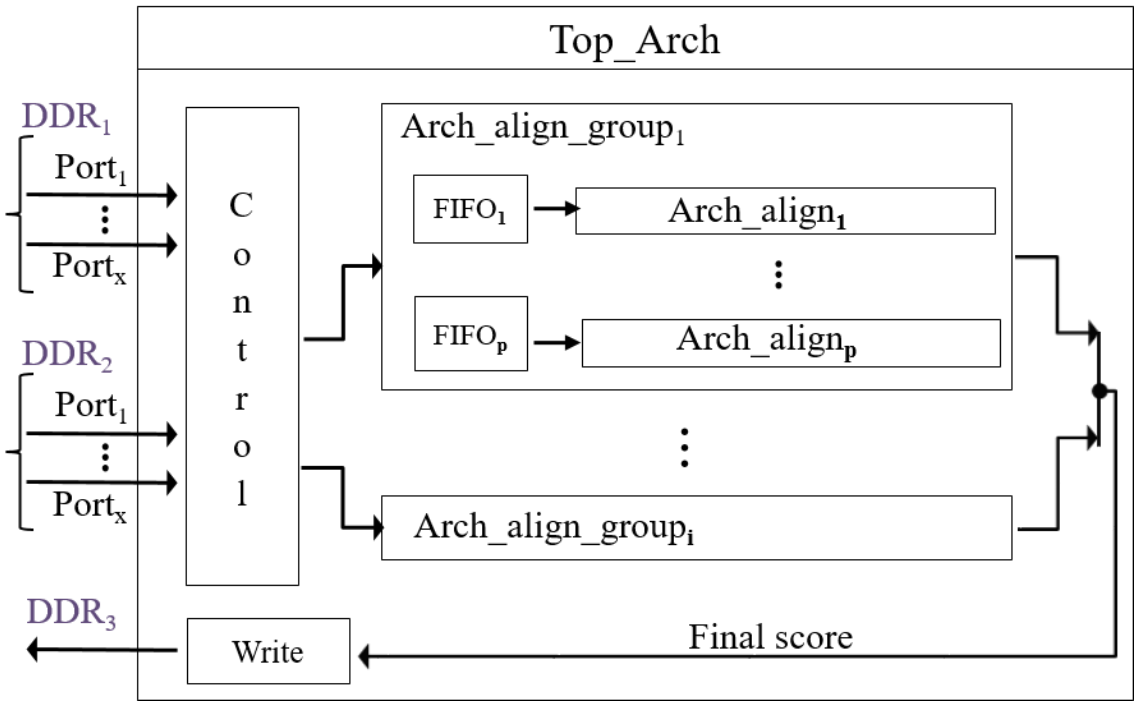
Os módulos cuja arquitetura foi apresentada no Capítulo 8.2 foram implementados utilizando a linguagem de descrição de hardware SystemVerilog (subconjunto sintetizável). Quando necessárias, as máquinas de estados dos módulos foram implementadas utilizando o padrão Moore (Sutherland, Davidmann e Flake 2006) pelo menor caminho crítico de lógica combinacional gerado.

O número de instâncias de diversos módulos são parâmetros. Logo utilizou-se o recurso de SystemVerilog *generate for* para se instanciar uma quantidade variável de módulos.

8.6.1 Módulo Top_Arch

O Top_arch é o módulo que está em mais alto nível na hierarquia da arquitetura. Ele é responsável por instanciar os módulos Arch_Align_Group, o Control_Unit e o Write_Mem. O módulo Write_Mem espera os resultados dos módulos Arch_Align_Groups e os escreve na memória. O esquema simplificado da arquitetura interna do Top_arch pode ser visto na Figura 88.

Figura 88 Arquitetura interna do modulo Top_Arch



8.6.2 Módulo Arch_Align_Group

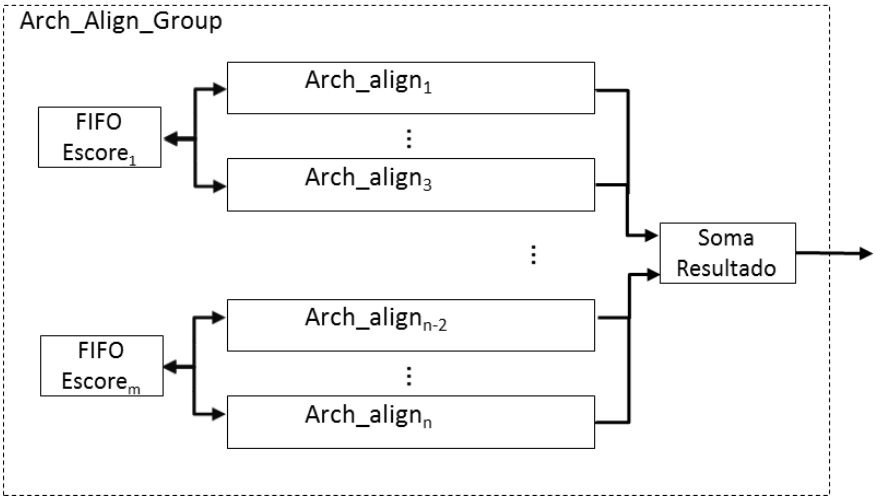
O Arch_align_Group é o módulo que agrupa vários Arch_aligns. No estudo de caso utilizado neste trabalho, o número de Arch_Aligns utilizados por grupo é igual a 15.

A Figura 89 mostra as entradas e saídas do módulo Arch_Align_Group. Ele recebe os nucleotídeos do banco e da query, sinais de controle para a FIFO interna e o sinal de Start. Como saída, há a propagação do sinal de Valid do último PE, o contador de *slices* interno e o escore resultante da soma das saídas de todos os alinhamentos do grupo.

Na Figura 90 pode ser vista a arquitetura interna do módulo Arch_align_Group. Por motivo da otimização dos recursos de memória disponíveis no FPGA, os direcionais de saída de três módulos Arch_Aligns foram agrupados em uma mesma FIFO de escore. Nessa implementação, foram utilizados 12 bits de largura para representar os escores, implicando em FIFOS de escore com 36 bits de largura.

Figura 89 Entradas e saídas do Arch_Align_Group

Figura 90 Arquitetura interna do módulo Arch_Align_Group



8.6.3 Módulo Arch_Align

O módulo Arch_Align instancia um módulo Forward_Align e um módulo Backward_Align, representando dessa forma um alinhamento completo.

Como entrada ele recebe os nucleotídeos de S1 e S2, o valor de escore à esquerda que pode vir da FIFO de escores ou dos contadores iniciais, o sinal de Start, e o valor a ser forçado na saída de cada PEs no momento do reset.

Como saída, é propagado o escore do último PE e seu sinal de Valid_LastPE. O escore do último PE segue para armazenamento externo ao módulo na FIFO de escores como pode ser visto na Figura 91. O sinal de End_Process sinaliza o fim da fase de traceback e o fim do alinhamento, atribuindo ao sinal de traceback_score_result o valor de escore obtido pelo módulo Backward_Align, como pode ser visto também na Figura 92. O sinal Contador_slices é propagado

para servir de controle do fim do alinhamento.

Figura 91 Entradas e saídas do Arch_Align

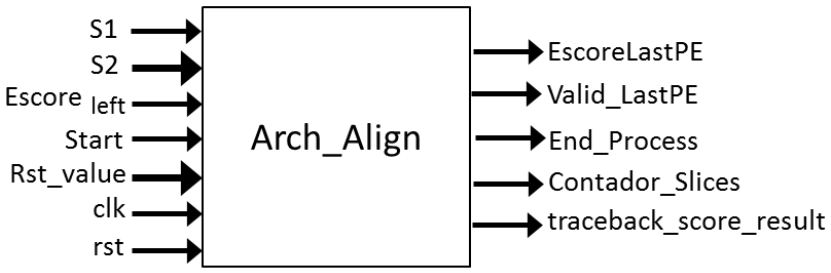
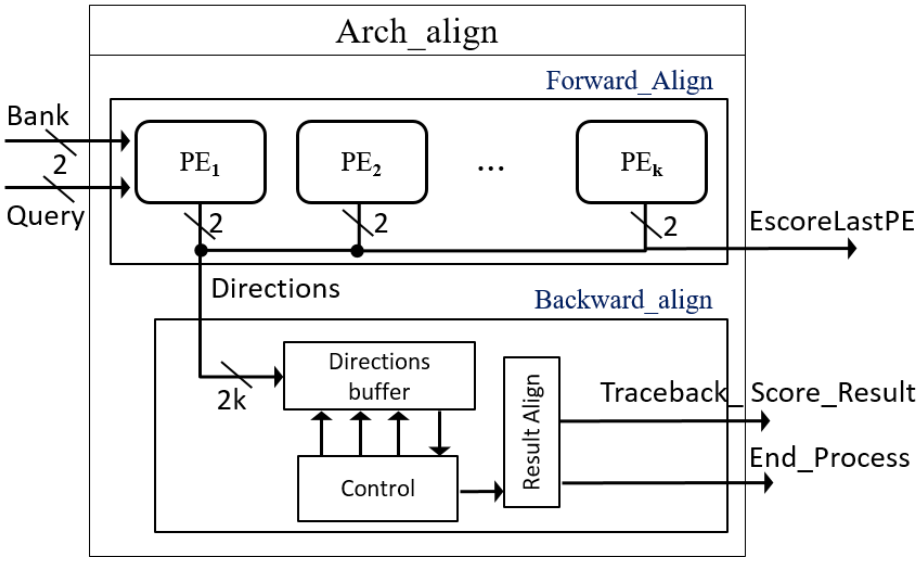


Figura 92 Arquitetura interna do modulo Arch_Align



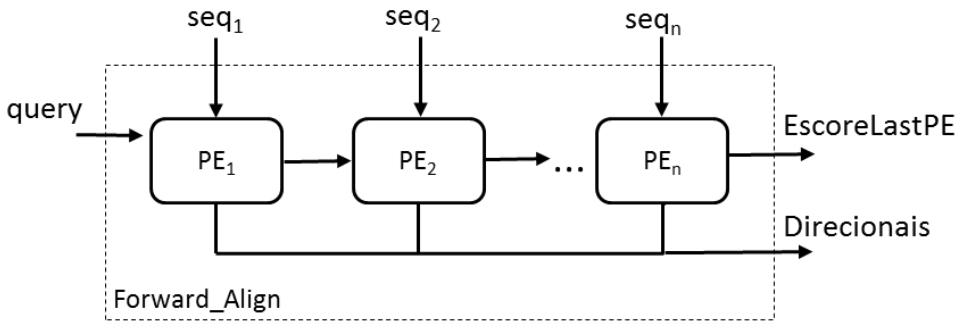
Na Figura 92 pode-se ver a estrutura interna do módulo Arch_align, com a instância dos módulos Forward_Align e Backward_align.

8.6.4 Módulo Forward_Align

O módulo Forward_Align agrupa os múltiplos PEs para computar os escores e os direcionais.

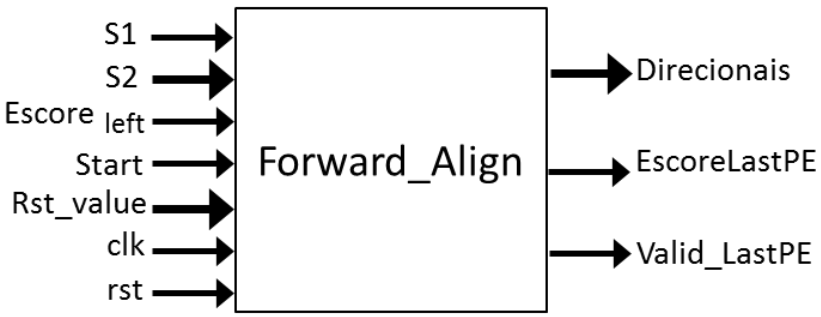
Na Figura 93 pode ser vista a arquitetura interna do módulo. A query é lida e propagada de PE para PE. Cada PE recebe também seu respectivo nucleotídeo da sequência do banco.

Figura 93 Arquitetura interna do módulo Forward_Align



Na Figura 94 podem ser vistas as entradas e saídas do módulo Forward_Align. Em que o sinal S1 representa a *query* e o sinal S2 os nucleotídeos do banco. O Escore_Left é o valor do escore à esquerda que pode vir da FIFO de escore ou do valor da inicialização da matriz. Ainda como entradas existem o sinal Start e o Rst_value. Como saídas, há os direcionais e o escore gerado pelo último PE e seu sinal de valid.

Figura 94 Entradas e saídas do Forward_Align



8.6.5 Módulo Elemento de Processamento (PE)

O Elemento de Processamento é o módulo aritmético da arquitetura. Ele computa as Equações 2.1 e 2.2.

$$\boldsymbol{M}(i,j) = max\begin{cases}\boldsymbol{M}(i-1,j-1) + \boldsymbol{S}(x_i, y_i)\\ \boldsymbol{M}(i-1,j) + g_1\\ \boldsymbol{M}(i,j-1) + g_2\end{cases} \quad (2.1)$$

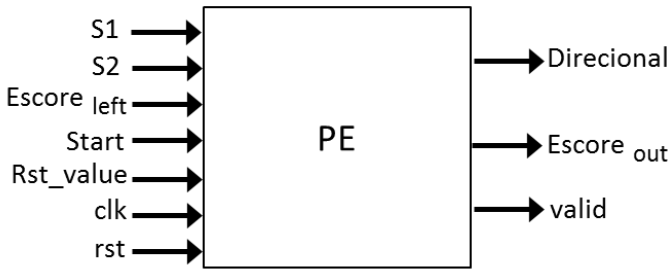
$$\boldsymbol{S}(x_i, y_i) = \begin{cases} \delta, & Se\ x_i = y_i \\ -\delta, & Se\ x_i \neq y_i \end{cases} \quad (2.2)$$

O PE foi inteiramente projetado como um pipeline de 6 estágios. Ele tem como entradas as cadeias S1 e S2 (query e sequência original, respectivamente); o escore gerado pelo PE a sua esquerda; um sinal de Start que indica que os dados da entrada do PE são válidos, essa é uma proteção contra a falta de dados de entrada que, quando inativa, segura todos estágios do pipeline evitando a dessincronização; o sinal Rst_value que é o valor de reset assumido pelo PE na subida do sinal de reset. Uma falta de dados momentânea pode ocorrer no acesso às memórias externas cujas leituras são gerenciadas via escalonamento round robin.

Como saídas o módulo apresenta o direcional, o escore e o valid. O escore de saída pode estar ligado a outro PE à direita ou à FIFO de escore. O direcional segue para armazenamento no Directions_Buffer. O sinal de valid indica que os valores de saída são válidos e podem ser lidos. A Figura 95 ilustra as entradas e saídas do PE.

Como parâmetros do PE existem a largura em bits dos escores e a largura em bits dos nucleotídeos.

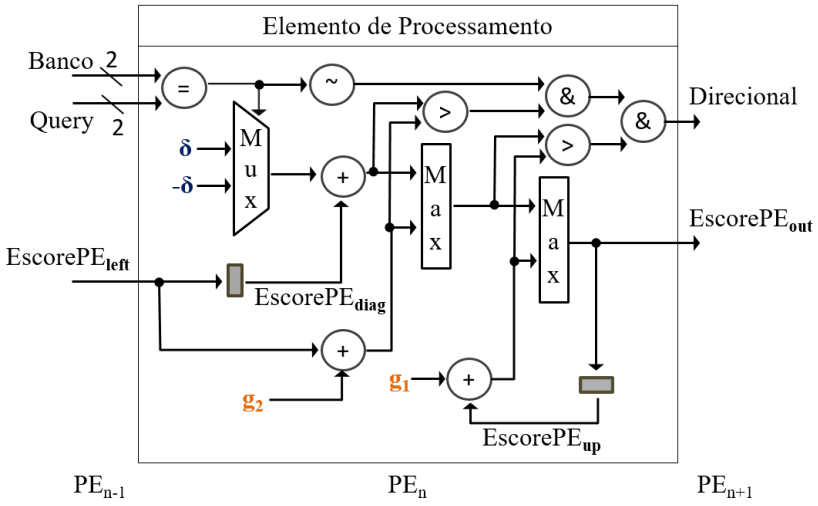
Figura 95 Entradas e saídas do PE



Na Figura 96 é mostrada a arquitetura simplificada do PE. Cada PE se conecta à esquerda e à direita com outros PEs na forma da arquitetura sistólica. Primeiro se realiza a comparação entre os nucleotídeos S1 e S2, a fim de se determinar a igualdade entre eles, segundo a Equação 2.2. Esse valor do peso é então

somado com o $EscorePE_{diag}$, que corresponde ao valor do $EscorePE_{left}$ defasado no tempo por um registrador. O resultado é então somado ao resultado da soma de $EscorePE_{left}$ com o pesos de gap g_2 depois o resultado da comparação fornece o maior valor entre eles. O maior valor entre as duas primeiras somas é comparado com a soma do $EscorePE_{up}$ com o peso de gap g_1. O $EscorePE_{up}$ é o resultado anterior do PE. Ele é obtido através da defasagem temporal do $EscorePE_{out}$. Por fim $EscorePE_{out}$ é o resultado da última comparação que fornece o maior valor entre as três somas e o sinal direcional indica qual dos escores gerou esse valor.

Figura 96 Arquitetura interna do PE



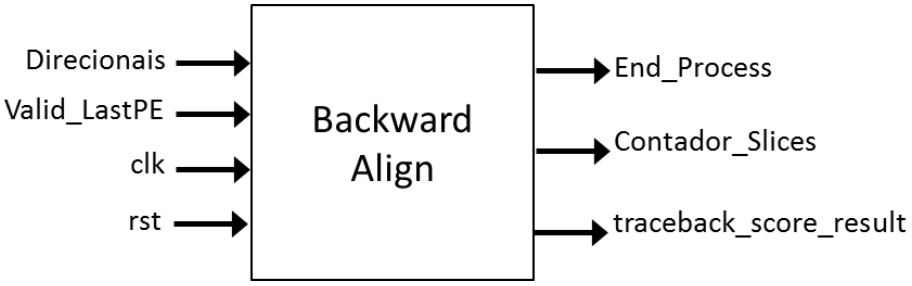
8.6.6 Módulo Backward_Align

O módulo Backward_Align realiza a fase de *traceback*. Este módulo recebe os direcionais da fase de geração dos escores gerados pelos PEs e os armazena no módulo Directions_Buffer. Com o fim da geração dos escores, o módulo Backward_Align inicia a leitura dos direcionais armazenados e gera o alinhamento resultante.

Na Figura 97 se pode ver as entradas e saídas desse módulo. Os sinais de direcionais e o Valid_LastPE recebidos do módulo Forward_Align como entradas. Como saídas, a sinalização de final do alinhamento, a propagação do

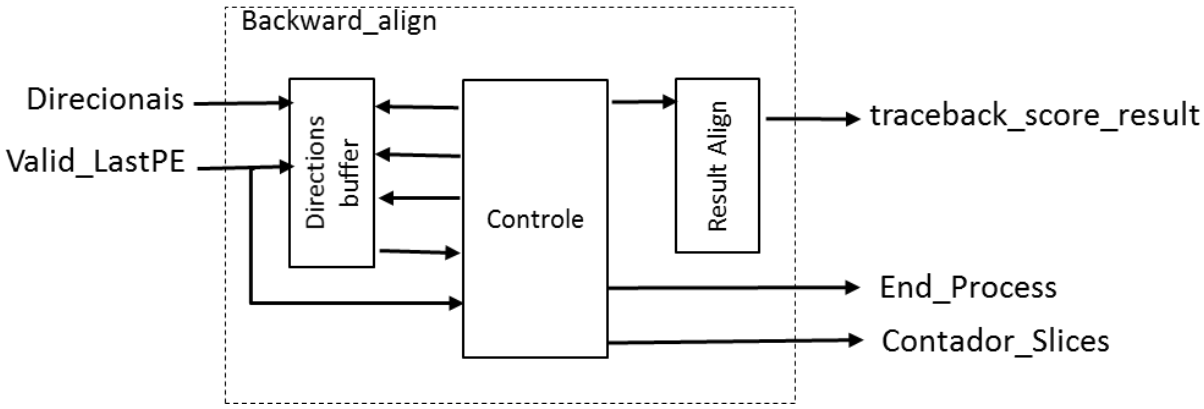
contador de *slices* e o resultado do alinhamento.

Figura 97 Entradas e saídas do Backward_Align



A Figura 98 ilustra a estrutura interna do módulo, representando o módulo Directions_Buffer e a unidade de controle interna responsável pelo traceback. Também se vê o módulo aritmético Result_Align que realiza a contabilização dos acertos, das trocas, das inserções e remoções do alinhamento para produzir um escore final.

Figura 98 Arquitetura interna do modulo Backward_Align



8.6.7 Módulo Directions_Buffer

O Directions_Buffer é um módulo de armazenamento que recebe os direcionais gerados pelos PEs e permite a sua leitura pelo módulo Backward_Align no momento do *traceback*.

Na Figura 99 se pode ver a estrutura interna do módulo Directions_Buffer. Esse módulo é formado por uma pequena unidade de controle que gerencia as

escritas, leituras e saltos (*jumps*) e de um ou mais módulos de memória RAM internas no FPGA além de uma FIFO. A largura da memória é parametrizada pela quantidade de PES, assim quando forem usados 4 PEs, a largura da RAM será de 8 bits. Já seu tamanho é parametrizado pela Equação 8.1.

$$\left(\frac{Tam\_Sequencia}{Num\_PEs}\right) * Tam\_Sequencia \quad (8.1)$$

A memória RAM interna tem uma latência de leitura de 2 ciclos de *clock*, assim, foi colocada uma pequena FIFO *show ahead* (o dado a ser lido está disponível na cabeça da FIFO sem atraso) que mascara a latência da memória. O controle realiza o *prefetching* dos dados da RAM e os deixa disponíveis na porta da FIFO. Essa funcionalidade simplifica o controle do módulo Backward_Align.

Na Figura 100 são mostradas as entradas e saídas do módulo Directions_Buffer. A saber, as entradas são o dado de entrada, os sinais de read, write e jump; a quantidade de posições a serem puladas em Jump_Position. Como saída existe o dado e a sinalização de que a FIFO interna está vazia.

Figura 99 Arquitetura interna do modulo Directions_Buffer

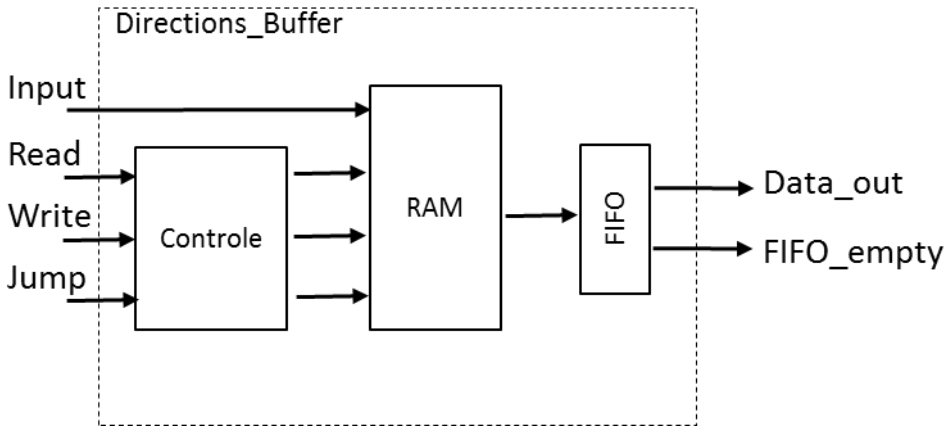
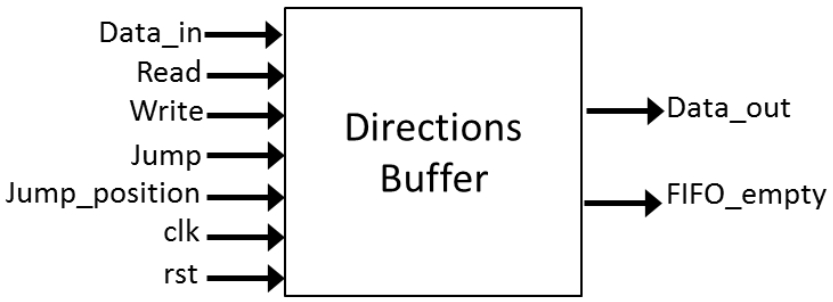


Figura 100 Entradas e saídas do Directions_Buffer



8.6.8 Módulo Control_unit

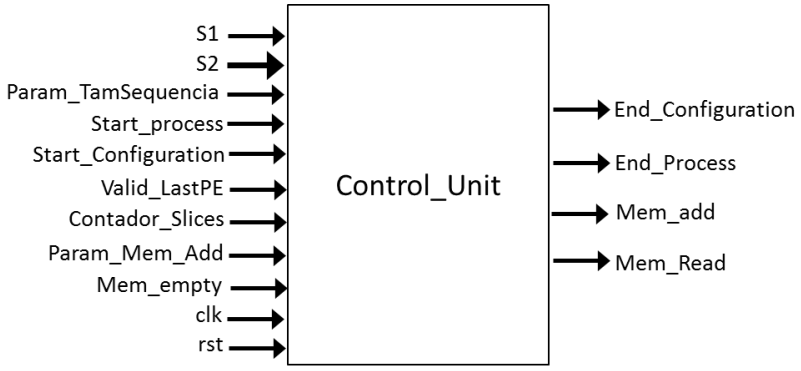
O módulo Control_Unit contém duas unidades de controle com 18 e 17 estados. Esse módulo é responsável pelo controle do acesso de leitura da memória, como a inicialização das portas, escrita dos endereços e também por identificar possíveis falta de dados na memória através dos sinais de Mem_Empty.

O módulo Control_Unit também implementa o protocolo com o software do host. O software envia os parâmetros de processamento, transfere os dados para o hardware e aciona o sinal Start_Configuration, via barramento PCI-express. O componente de hardware inicia as portas, e aciona o sinal End_Configuration para o software. O software, então, pode pedir uma consulta ao banco de dados acionando o sinal de Start_Process, que causa a inicialização do processamento da consulta acionando o sinal End_process avisando ao software que o resultado está disponível para leitura.

Durante o processamento da busca no banco o Control_Unit monitora o andamento do processamento através dos sinais de Valid_LastPE e do Contador_Slices. Esses sinais são mostrados na Figura 101.

Na Figura 102 é mostrado a máquina de estados do módulo Control_Unit que interfaceia com a memória e com o software no host. O estado inicial **wait_start_config**, aguarda o comando de start_configuration do software, que indica ao hardware que o banco foi enviado ao FPGA e as portas do multiport do banco podem ser iniciadas.

Figura 101 Entradas e saídas do Control_Unit



Isso é feito no estado **start_multiport_b** e após há dois estados **wait_multiport_empty_up** e **wait_multiport_empty_down**, que esperam o sinal de almost_empty (quase cheio) subir e descer, respectivamente. Dessa forma se garante que as portas do multiport foram iniciadas corretamente e estão com os primeiros dados a serem lidos. Então, no estado **wait_end_configuration** envia-se ao software o sinal de end_configuration que indica o fim da configuração. Esse sinal fica ativo por 128 ciclos, a fim de garantir a visualização pela aplicação no host.

Após isso se aguarda o sinal de start_process no estado **wait_start_process**, que indica a solicitação de uma consulta ao banco e que a query se encontra na memória da placa. Segue-se com a inicialização da porta de query e dois estados de espera em **start_multiport_q**, **wait_multiport_q_empty_up** e **wait_multiport_q empty_down**, respectivamente.

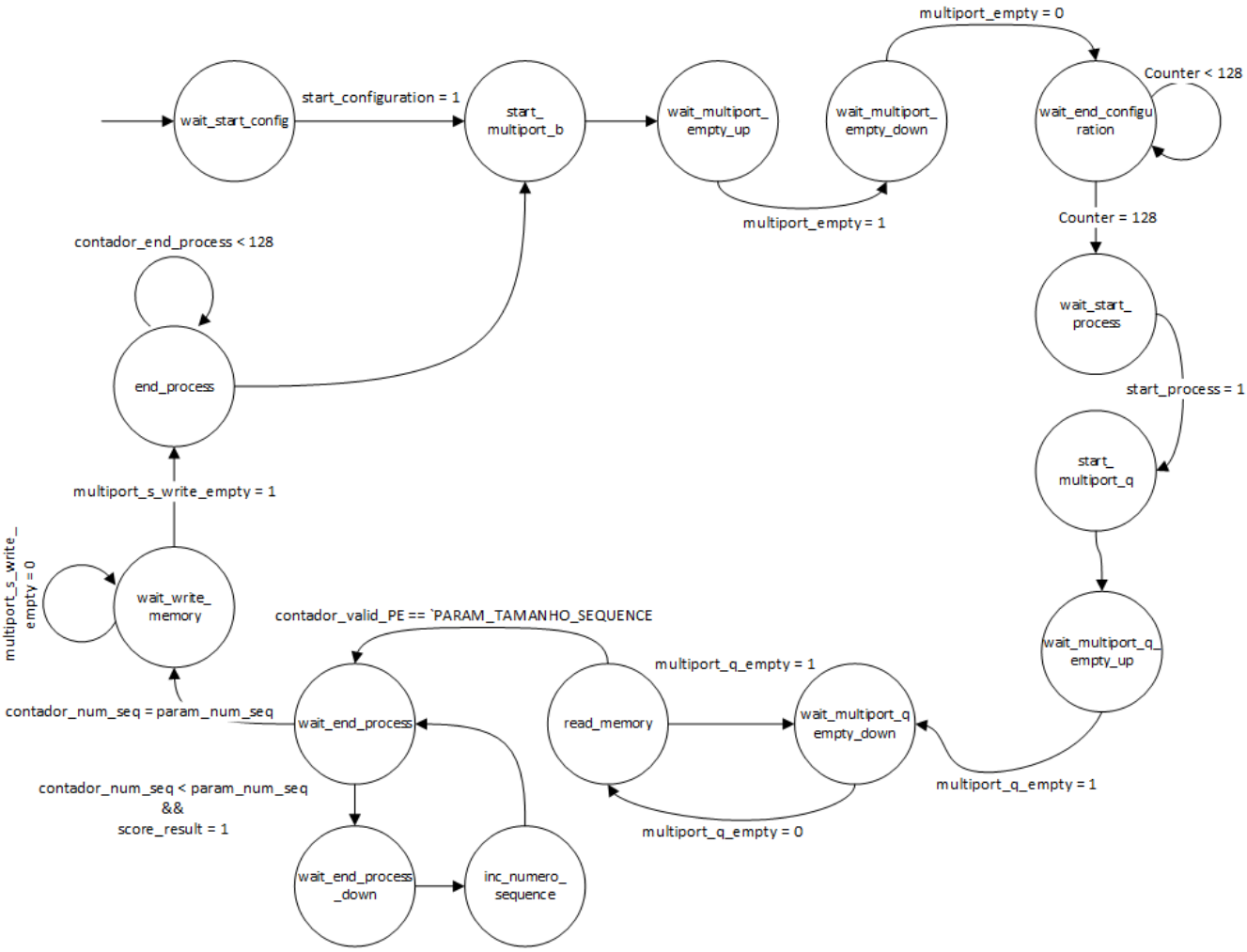
No estado **read_memory** todas as portas são lidas e é dado o início da computação. Assim que o contador atinge o tamanho da cadeia, passa-se para o estado **wait_end_process** que checa se todo o banco foi comparado, senão é incrementado o contador de sequências a cada saída de escore final, incrementa-se o contador de sequências em **inc_numero_sequence**.

Com todas as sequências do banco concluídas, espera-se até que a porta de escrita esteja vazia em **wait_write_memory**, o que indica a escrita de todos os dados na memória. No estado **end_process** sinaliza-se para o software que o processamento foi finalizado, mantendo-se o sinal por 128 ciclos.

Após isso, a máquina retorna o ponteiro de leitura para o primeiro indivíduo do banco deixando o hardware pronto para uma nova consulta com a subida do sinal de start_process.

Porém, caso haja uma atualização no banco ou em quaisquer parâmetros, pode-se enviar um novo start_configuration.

Figura 102 Máquina de estados do Control_Unit de interface com o host e memória



8.7 Desempenho esperado

O primeiro critério de desempenho da arquitetura analisado é a utilização da memória. Esse é um limitante crítico para qualquer sistema de alto desempenho. Na arquitetura proposta, apenas um nucleotídeo por PE é lido durante todo *slice* do processamento.

A banda de memória externa ao FPGA necessária por alinhamento segue a fórmula:

$$BandaMemExt = (BitsN * numPEs * Freq.)/(8bits * TamCadeia)$$

Uma vez que apenas são lidos da memória os nucleotídeos do banco são lidos 1 por PE (os nucleotídeos da query estão buferizados na FIFO de query). Essa leitura ocorre uma vez a cada slice de processamento, por isso dividiu-se o produto pelo tamanho da cadeia na vertical.

Numa instância da arquitetura proposta com dezesseis PEs por alinhamento, com cadeia de tamanho igual a 240 nucleotídeos e com frequência de operação de 280MHz, a largura de banda consumida em um *slice* de processamento é de $(2bits * 16PEs * 280MHz)/(8bits * 240) = 4{,}66\ MB/s$ por alinhamento. Uma memória DDR2 800 (PC2-6400) tem largura de banda ideal de $6400\ MB/s$ e reais $70\% * 6400 = 4480\ MB/s$. Assim, com uma memória apenas se poderia alimentar 961 alinhamentos.

Em oposição, caso fossem também escritos os direcionais na memória externa, a banda necessária para um alinhamento seria de $(16PEs * 2bits * 280MHz)\ /\ 8bits + 4{,}66\ MB/s = 1124.66\ MB/s$. O que representa a possibilidade de se realizar apenas 3 alinhamentos.

Logo a decisão de armazenar os direcionais internamente e realizar o *traceback* dentro do FPGA é a melhor escolha para otimizar o desempenho.

Dessa forma, o requisito de largura de banda não é um limitante para o desempenho da arquitetura. Fazendo com que o gargalo de desempenho sejam a frequência atingida pela implementação em FPGA e os recursos internos de lógica e memória.

Esse cenário é bastante vantajoso para uma implementação em FPGA, visto que a largura de banda disponível dessas plataformas é frequentemente uma ordem de grandeza menor em comparação com as plataformas de GPUs.

Para o requisito uso de memória interna, a arquitetura prevê o armazenamento

duas FIFOs da query e de escore para cada alinhamento. Além do armazenamento do Directions_buffer. Isso leva à Equação 8.2 abaixo:

$$\boldsymbol{UsoMemoriaIdeal} = \mathrm{k} * \mathrm{NrIndiv} * (n * (bitsN + bitsE) + n^2 * bitsD) \quad (8.2)$$

Em que k é o número de alinhamentos por indivíduo, NrIndiv é o número de indivíduos processados em paralelo, n é o tamanho de cadeia, bitsN é o número de bits ocupado por um nucleotídeo, bitsS é a largura em bits do escore, bitsD é a largura em bits dos direcionais. Para a instância de valores dados na Tabela 3, que totaliza 150 alinhamentos em paralelo, a utilização de memória resulta idealmente em $20.164.800$ bits, que corresponde a $83{,}6\%$ da memória de um FPGA Stratix IV 530 da Altera.

Tabela 3 Parâmetros da arquitetura

NrAlimPorIndiv = 15	bits_nucleo = 2
Nr_indiv = 10	bits_score = 12
tam_cadeia = 240	bits_direc = 2
NumPEs = 16 (PEs por alinhamento)	

Porém, nesse caso, o uso da memória interna do FPGA não pode ser vista como uma variável contínua, mas sim discreta, com um passo de incremento igual ao tamanho da memória instanciada em cada módulo **Directions_Buffer** descrito na Seção 8.6.7. A memória interna Directions_Buffer tem quantidade de posições necessárias definida pelo fator $\mathrm{PosicoesMem} = \lceil \mathrm{TamCadeia}^2 / NumPEs \rceil$ e largura $BitsDirec * NumPEs$. Contudo a quantidade de posições precisa ser um número da base 2, chamado $TamMemRam$. Assim a variável UsoMemoriaReal é descrita pela Equação 8.3

$$\mathrm{k} * \mathrm{NrIndiv} * (n * (bitsN + bitsE) + TamMemRam * bitsD * NumPEs) \quad (8.3)$$

Seguindo essa métrica, o uso de memória interna no FPGA na instância da Tabela 3 consome, na verdade, 95% do disponível no FPGA por causa da fragmentação interna dos blocos de RAM instanciados. Esse número fica fora do implementável, visto que dificilmente o algoritmo de alocação dos blocos do FPGA conseguirá essa utilização de memória.

Como observou-se na Equação 7.2 a utilização de memória interna é independente do número de PEs utilizada, e a quantidade de PEs depende apenas dos recursos de lógica disponíveis.

Esses dados, somados à utilização de lógica mostrada posteriormente, revelam a factibilidade de implementação da arquitetura escalável proposta em um FPGA comercial.

Os resultados de uso de memória interna, limitando em 10 o número de indivíduos que cabem no FPGA mencionado, assume o papel do recurso mais crítico para a implementação, suplantando o uso de lógica e de banda da memória externa.

8.8 Escalabilidade

A escalabilidade da arquitetura é um fator primordial para permitir a implementação da mesma em múltiplas plataformas com diferentes recursos de lógica digital, memória interna e largura de banda. Adicionalmente, diversas aplicações demandam diferentes requisitos. Por exemplo, o alinhamento de cadeias muito longas de DNA seria beneficiado por um número grande (tipicamente centenas ou milhares) de PEs por alinhamento. E quanto maior a quantidade de PEs, maior o desempenho, até que se atinja o limite de recursos do FPGA. A partir daí a adição de outros FPGAs encadeados poderia ser utilizada para um alinhamento único.

Antagonicamente, no cenário previsto no estudo de caso do banco forense,

apresentado na Seção 1.2, grandes quantidades de PEs em um único alinhamento significaria a subutilização de recursos do FPGA. Isto ocorre devido ao tamanho das cadeias para o referido problema girar em torno de 200 a 300 nucleotídeos. E grande parte dos PEs estaria a maior parte do tempo ociosa. Outro ponto importante é a separabilidade do problema, em que o processamento de cada indivíduo pode se dar independentemente, como também os alinhamentos de um indivíduo podem ser feitos em paralelo.

Assim propõe-se a utilização de poucas dezenas de PEs por indivíduo e o processamento em paralelo de diversos alinhamentos. E como no estudo de caso, cada indivíduo é identificado por 15 sequências, agrupamentos de alinhamentos se fazem necessários. Logo, ao invés de se computar alinhamentos isolados, alguns indivíduos (quantos couberem no FPGA) podem ser computados em paralelo, a cada iteração.

A implementação seguiu a parametrização definida na Tabela 4. Esses parâmetros são definidos em tempo de síntese, para a configuração desejada. Internamente são instanciados e ajustados os módulos utilizando-se as estruturas **Generate For** de SystemVerilog. Uma mudança nesses parâmetros precisa de uma nova síntese do código HDL.

Tabela 4 Parâmetros da Implementação em Hardware

Parâmetro	Descrição
CUSTO_MA_P CUSTO_MA_N CUSTO_IN CUSTO_RE	Custos de acerto, troca, inserção e remoção.
CUSTO_TRACEBACK_MATCH CUSTO_TRACEBACK_MISSMATCH CUSTO_TRACEBACK_GAP CUSTO_TRACEBACK_GAP_R	Custos utilizados no cálculo do escore final do alinhamento.
SCORE_WIDTH DIRECTION_WIDTH	Larguras em bits da representação interna dos escores, direções e nucleotídeos.

NUCLEOTIDEO_WIDTH	
TAM_SEQ_MAX NR_SLICES_MAX	Tamanho de sequência máximo utilizado. Tamanhos menores são admitidos em tempo de execução. O número de slices é computado a partir do tamanho da sequência máxima e o número de PEs por alinhamento.
NUM_PES	Número de PEs por alinhamento
TAM_DIRECTION_MEMORY	Tamanho da memória instanciada no Directions_Buffer
NUMERO_ALIGN_GROUP	Número de alinhamentos por indivíduo.
NUM_INDIVIDUOS	Número de indivíduos rodando em paralelo.

Por exemplo no módulo Forward_Align são instanciados tantos PEs quanto foram definidos em NUM_PES. A Tabela 5 exemplifica o bloco Generate sendo usado para se instanciar os PEs no módulo Forward_Align.

Essa parametrização do código HDL permite que a implementação seja utilizada para outros problemas diferentes do problema forense com diversas sequências relacionadas à um indivíduo e de tamanhos pequenos.

Tabela 5 Trecho de código da instância dos PEs

```
genvar Iter;
        generate for(Iter = 1; Iter < (`NUM_PES-1); Iter += 1) begin:PE_new
                PE pe_i(.clk(clk),
                        .i_reset(w_reset_pe[Iter-1]),
                        .i_M_left(w_score_out_pe[(`SCORE_WIDTH*Iter)-1:`SCORE_WIDTH*(Iter-1)]),
                        .i_s1(w_s1_pe[(`NUCLEOTIDEO_WIDTH*Iter)-1:`NUCLEOTIDEO_WIDTH*(Iter-1)]),
                        .i_s2(r_s2_reg[(`NUCLEOTIDEO_WIDTH*(Iter+1))-1:`NUCLEOTIDEO_WIDTH*Iter]),
                        .i_up_reset_value(r_contador_pe[(param_tam_contador*(Iter+1))-1:
                                                                param_tam_contador*Iter]),
                        .i_start_value(w_valid_pe[Iter-1]),
                        .o_s1(w_s1_pe[(`NUCLEOTIDEO_WIDTH*(Iter+1))-1:`NUCLEOTIDEO_WIDTH*Iter]),
                        .o_reset(w_reset_pe[Iter]),
                        .o_end_value(w_valid_pe[Iter]),
                        .o_dir0(o_dir0_pe[Iter]),
                        .o_dir1(o_dir1_pe[Iter]),
                        .o_score_out(w_score_out_pe[(`SCORE_WIDTH*(Iter+1))-1:`SCORE_WIDTH*Iter]));
        end
endgenerate
```

Um exemplo de parâmetros opostos aos utilizados aqui nesse trabalho seria a utilização da arquitetura proposta em FPGA para um único alinhamento. Nesse

caso os parâmetros poderiam ser como os definidos na Tabela 6. O número de alinhamentos por indivíduo seria 1, assim como o número de indivíduos em paralelo. Os parâmetros de largura de nucleotídeo, direcionais e escore se mantem os mesmos.

Tabela 6 Parâmetros para um único alinhamento

NUMERO_ALIGN_GROUP = 1	NUCLEOTIDEO_WIDTH = 2
NUM_INDIVIDUOS = 1	SCORE_WIDTH = 12
TAM_SEQ_MAX = 3.000	DIRECTION_WIDTH = 2
NUM_PES = 1024 (PEs por alinhamento)	

O tamanho da cadeia máxima seria a maior cadeia em que os direcionais podem caber dentro do FPGA e o número de PEs nesse único alinhamento pode ser o máximo, lembrando-se que esse número precisa ser escolhido com cuidado para diminuir a fragmentação interna de memória RAM na memória do DirectionsBuffer. Nesse caso, diversas memórias precisam se instanciadas para se estender o range de endereços sem que se aumente o desperdício visto que o tamanho da memória é uma potência de 2.

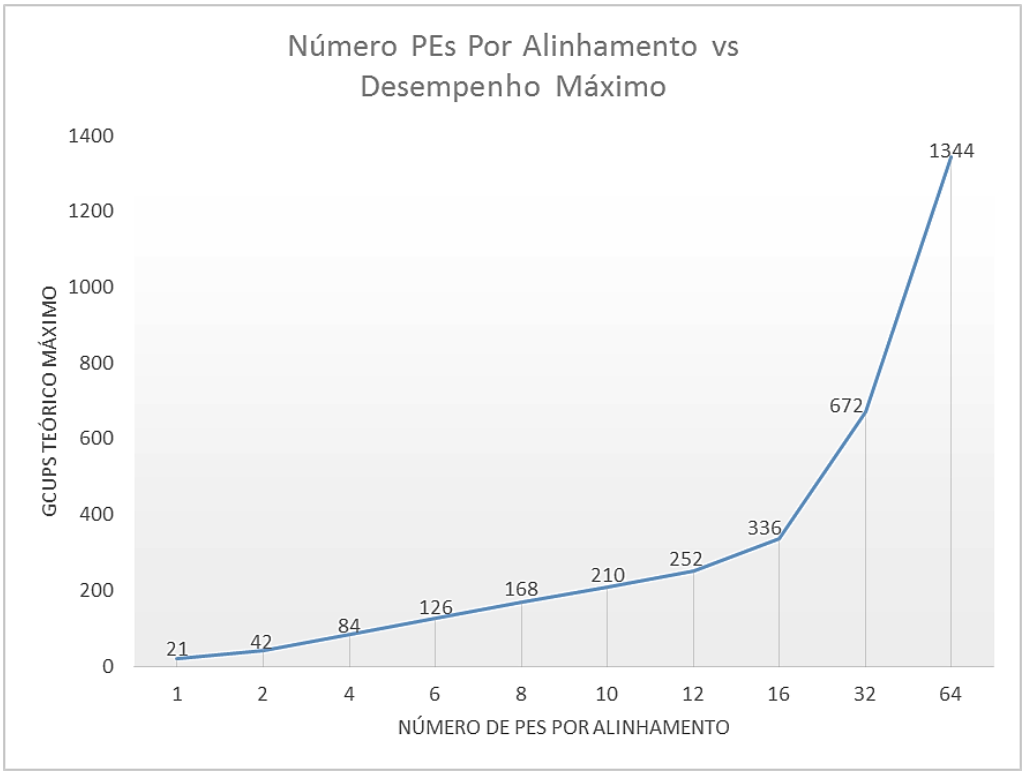
Com 1024 PEs o uso de memória ideal fica em 85% e de memória real fica em 86,6% para o FPGA Stratix IV 530 da Altera que tem $20.164.800$bits de memória interna. Assim, o tamanho máximo de alinhamento fica em torno do número 3mil nucleotídeos para o referido FPGA.

Portanto qualquer configuração intermediária, com maiores tamanhos de cadeia da utilizada no problema do banco de DNA forense, podem ser instanciadas com o compromisso da redução do número de indivíduos em paralelo ou com a troca por um FPGA com maior memória interna.

A Figura 103 mostra a curva do desempenho teórico máximo, medido em

GCUPS, para uma instância do problema com 5 indivíduos, 15 alinhamentos por indivíduo e frequência de operação de 280MHz.

Figura 103 Evolução do Desempenho (GCUPS) em função do número de PEs por alinhamento



A função é obtida da Equação 7.4 abaixo com $Fmax$ em MHz.

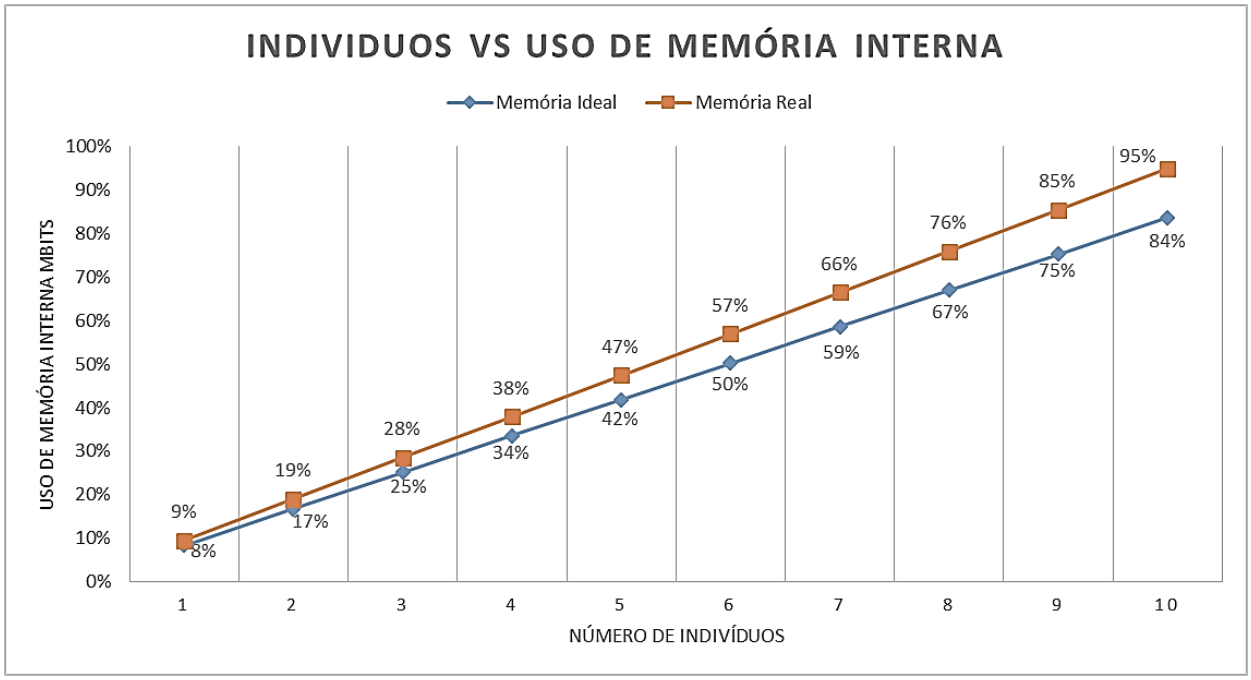
$$GCUPS_{Máximo} = (NRIndiv * AlignPorIndiv * NrPEsPorAlign * Fmax)/10^3 \quad (7.4)$$

Claramente essa medida apenas representa o desempenho máximo teórico (ou de pico), uma vez que os PEs não estão a todo tempo operando e também existe as penalidades de transferência da memória externa.

Para a mesma instância do problema, 15 alinhamentos por indivíduo, frequência de operação de 280MHz (lembrando que o número de PEs por alinhamento não interfere no uso de memória interna), observa-se, na Figura 104, a evolução do consumo de memória interna ao se aumentar o número de

indivíduos sendo feitos em paralelo. As curvas representam o consumo ideal de memória e o realmente instanciado pela arquitetura. As curvas da Figura 104 são descritas nas Equações 7.2 e 7.3. Observa-se o distanciamento do uso real e do ideal de memória com o aumento do número dos indivíduos dentro do FPGA.

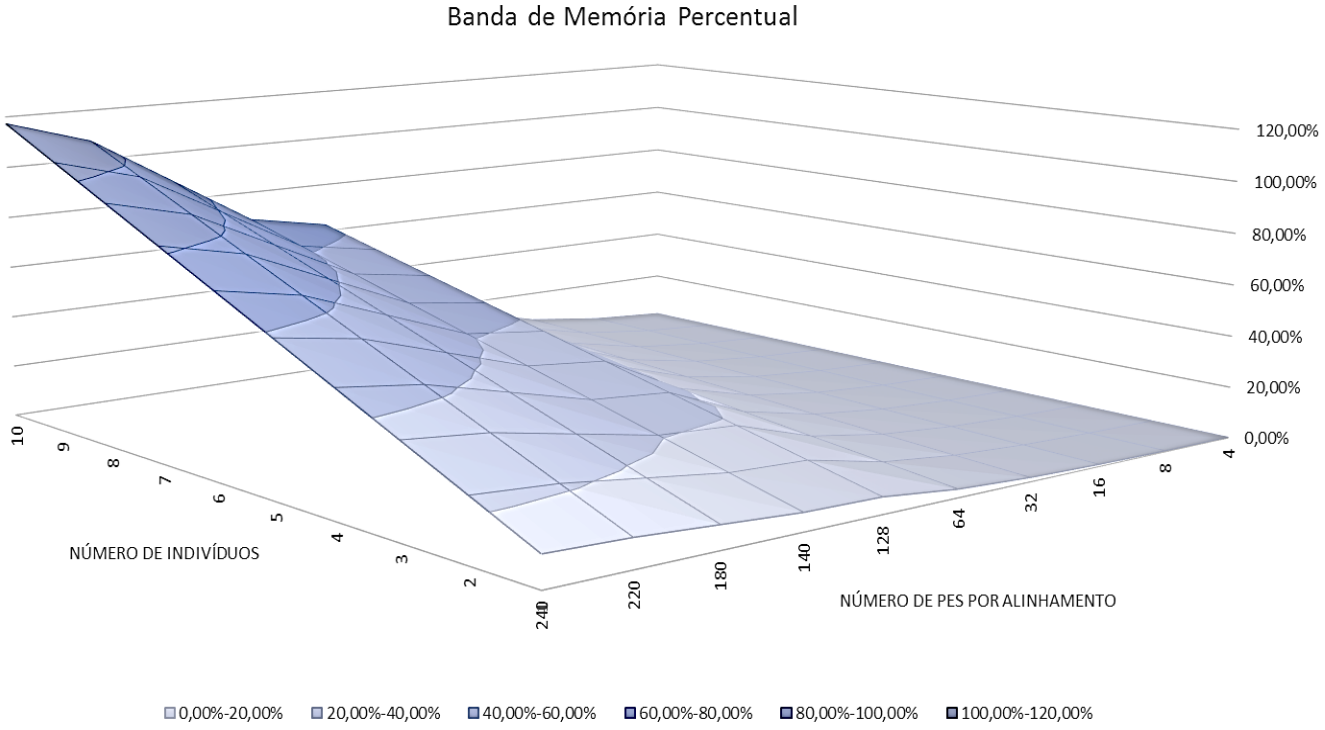
Figura 104 Gráfico do uso de memória interna em função do número de indivíduos



No gráfico da Figura 105 mostra-se a evolução do uso de banda da memória externa em função do número de PEs por alinhamento e do número de indivíduos (dada pela Equação 7.5), para 15 alinhamentos por indivíduo, frequência de operação de 280MHz, 2 bits por nucleotídeo e tamanho da cadeia de 240 nucleotídeos.

$$BandaMemExt\% = \frac{NrIndiv * NrAlignPorIndiv * PEsAlign * Fmax * BitsNucleo}{(8bits * TamCadeia) * \text{BandaPorFPGA}} \quad (7.5)$$

Figura 105 Gráfico do uso de banda de memória externa em função do número de indivíduos e do número de PEs por alinhamento



A plataforma da Gidel utilizada, apresenta 3 memórias DDR2 por FPGA, porém na distribuição dos dados a memória menor (apenas 512MB) foi deixada para as escritas do resultado e armazenamento da query. Assim, a banda de memória por FPGA corresponde ao fornecido por duas memórias, que pelo cálculo de banda real da DDR já mencionado é igual a $70\% * 6400 = 4480\, MB/s$ por DDR e $2 * 4480 = 8960 MB/s$.

Dessa forma, do gráfico da Figura 105, no caso extremo, com um PE por ponto da matriz (240 PEs ao todo) a banda consumida ultrapassa o disponível na plataforma de FPGA acima de 9 indivíduos. Claramente esse número de PEs ($15 * 9 * 240 = 32400$) não cabe na lógica do FPGA. Esse fato indica, assim, como segundo recurso limitante, depois da memória interna, a banda de memória externa. Esse fato indica a boa utilização da banda na arquitetura proposta. Faltando avaliar a projeção do uso de lógica, que será apresentada no Capítulo 9.

8.9 Prototipação em FPGA

A implementação dos módulos de hardware em HDL (*Hardware Description Language*) foi seguida do seu teste unitário e a posterior integração. Foi utilizada a ferramenta de simulação *Questasim* da *Mentor Graphics* para simular testes com geração randômica usando a linguagem *SystemVerilog*.

A fase de implementação seguiu os requisitos de funcionalidade e da maior frequência possível. Assim, quando a implementação e validação do módulo estava completa a síntese do mesmo era realizada visando uma frequência acima da pretendida. Foi estabelecida a meta de 300MHz a ser atingida na plataforma final integrada. Dessa forma cada módulo isolado precisava obter uma frequência de operação acima dos 300MHz. Visto que ao se integrar os módulos a tendência é de queda na frequência.

O primeiro módulo implementado foi o PE e em sequência seguiu-se a hierarquia da arquitetura no sentido bottom-up (de baixo para cima). Esta estratégia foi seguida até a obtenção do mínimo necessário para se simular a arquitetura. A saber, um único Arch_Align_Group com 15 Arch_aligns. Dessa forma se pôde validar a arquitetura integrada com a emulação da comunicação com a infraestrutura de acesso à memória provida na plataforma de hardware utilizada.

Para evitar uma queda na frequência de operação em FPGA, não se estressou ao máximo os recursos de lógica na implementação em hardware. Assim, o protótipo desenvolvido instancia apenas 5 indivíduos em paralelo e 16 PEs por alinhamento foram utilizados. Totalizando assim, $5\ x\ 15\ =\ 75$ alinhamentos e $75\ x\ 16\ =\ 1200$ PEs no protótipo.

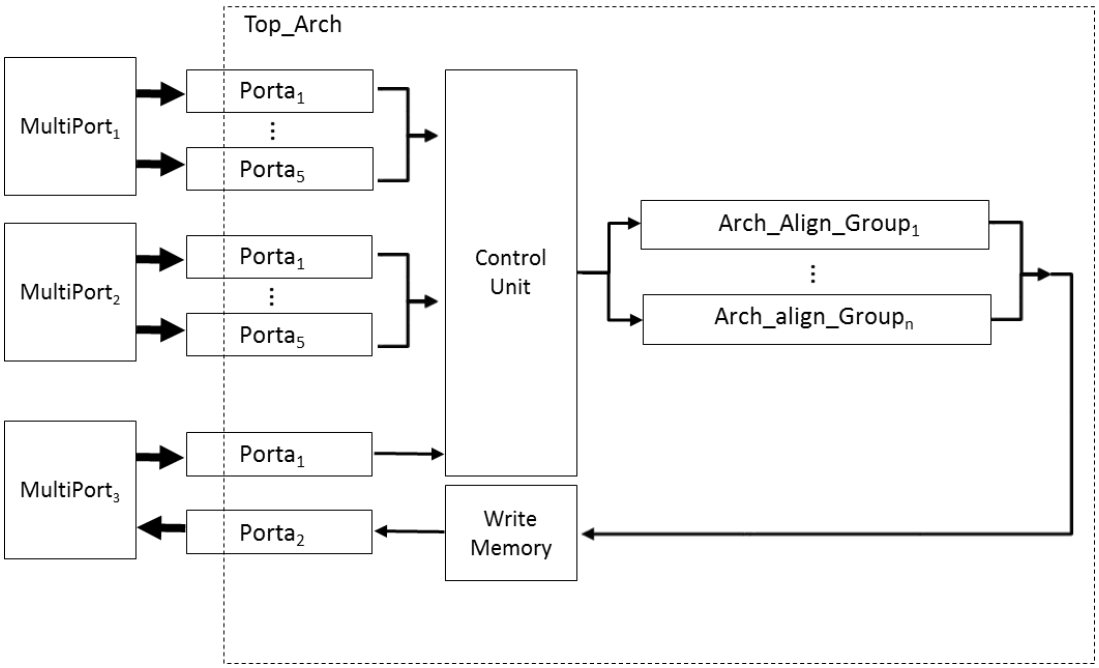
A Equação 7.6 mostra o cálculo do número de bits necessários na interface com o *Multiport*. A plataforma da *Gidel* tem um limite máximo de 256 bits por porta do *Multiport*. Assim dez portas de 256 bits precisaram ser utilizadas para a leitura das sequências do banco.

$$5\ individuos\ x\ 15\ AligsPorIindiv\ x\ 16\ PEs * 2\ BitsPorNucleotideo\ =\ 2.400 \qquad (7.6)$$

Foram necessárias, ainda, uma porta para a query de 32 bits e uma porta de 16 bits para a escrita para o resultado.

A Figura 106 ilustra a interface dos *Multiports* com o módulo Top_Arch. Cada *multiport* está associado à uma única memória da placa. Dessa forma, os dados do banco foram divididos em duas memórias (*Multiport* 1 e 2) e a terceira memória (a menor de 512 MB) contém apenas a query e os resultados escritos representados pela $Porta_1$ e $Porta_2$ do $Multiport_3$, respectivamente.

Figura 106 Interface da arquitetura com as memórias através dos *Multiports*



9

Resultados Comparativos e Discussões

Os experimentos foram conduzidos utilizando um banco de DNA sintético com 8 milhões de indivíduos. Cada um desses indivíduos é representado por $p = 15$ sequências de DNA com o tamanho de $m = 240$ nucleotídeos. Isso representa um total de 120 milhões de sequências.

Todas as implementações apresentadas consistem nos três passos do algoritmo de NW (inicialização, computação dos escores e na fase de traceback). Adicionalmente, os experimentos só registram os tempos de processamento, então quaisquer penalidades de transferência foram negligenciadas.

A arquitetura proposta em FPGA foi implementada na linguagem HDL SystemVerilog e prototipada na placa de FPGA da Gidel (Gidel n.d.) e conseguiu rodar à frequência de 280MHz. Nos testes 1 e 4 FPGAs foram utilizados. A cada FPGA da placa estão anexadas três memórias DDR2, como descrito na Seção 0.

Uma vez que a plataforma de FPGA apresenta uma interface PCIe, uma aplicação no PC host foi desenvolvida para particionar e enviar o banco de dados para as memórias da placa, seguindo a organização detalhada na Seção 8.5. Um banco de dados com 8 milhões de indivíduos foi o máximo que se conseguiu armazenar mesmo com a representação comprimida dos

nucleotídeos, com 2 bits cada.

Para cobrir o estudo de caso, na implementação em FPGA instanciou-se $i = 5$ indivíduos (5 Arch_align_groups) em paralelo. Isso corresponde em 5 indivíduos * 15 alinhamentos o que resulta em 75 alinhamentos com 240 nucleotídeos sendo processados em paralelo em cada FPGA. O número de PEs por alinhamento usado foi $k = 16$, resultando no total de 1200 PEs por FPGA.

Os resultados da síntese do protótipo em FPGA, em relação aos recursos de hardware utilizados, são mostrados na Tabela 7. Esses resultados já incluem toda infraestrutura inserida pela plataforma da Gidel para comunicação com a memória e com o host. Foi utilizado para síntese a ferramenta Quartus II 14.0.2 da Altera para um FPGA Stratix IV número de série EP4SE530H35C2, que compõe a placa PROCStar IV da Gidel.

Nota-se que o recurso mais utilizado foram os registradores. A utilização da lógica manteve-se baixa e o uso de memória manteve-se próximo da metade disponível no FPGA. Observa-se a possibilidade de aumento da quantidade de Arch_Align_Groups no protótipo, observando-se as projeções mostradas na Seção 8.8. É preciso lembrar que o aumento do consumo dos recursos deve provocar uma diminuição da frequência máxima atingida.

Tabela 7 Utilização dos Recursos de Hardware

ALUTs	Registradores	Memory ALUTs	Bits de Memória	Blocos de DSP
154.740	282.245	3.816	11.410.448	0
(36%)	(66%)	2%	(54%)	(0%)

Observa-se ainda na Tabela 7 que ao se comparar com a projeção do consumo de memória interna da Figura 104, que a projeção se encontra próxima do obtido no relatório de síntese. Uma correção da utilização projetada precisa ser feita, visto que a memória interna, assim como a lógica são, também, utilizados pela infraestrutura da Gidel no acesso da memória externa (FIFOs internas, controladores, escalonamento, etc.) e no acesso à PCIe (controlador).

Da Tabela 7 também se observa que a lógica interna do FPGA é um recurso mais crítico do que a memória interna, sendo os dois seguidos de longe pela banda da memória externa.

A Figura 107 apresenta, em escala logarítmica, um gráfico em barras com as medições de desempenho para todas as configurações em hardware e em software implementadas, além de dois softwares de terceiros que aqui foram utilizados como benchmarks: SWIPE (Rognes 2011) and FASTA (The FASTA program package n.d.). O gráfico usa a métrica GCUPS (Giga Cells Updates per Second – Giga atualizações de células por segundo) que pode ser inferida pela Equação 9.1.

$$\boldsymbol{GCUPS} = \frac{b * m^2 * p}{tempo\_processamento * 10^9} \tag{9.1}$$

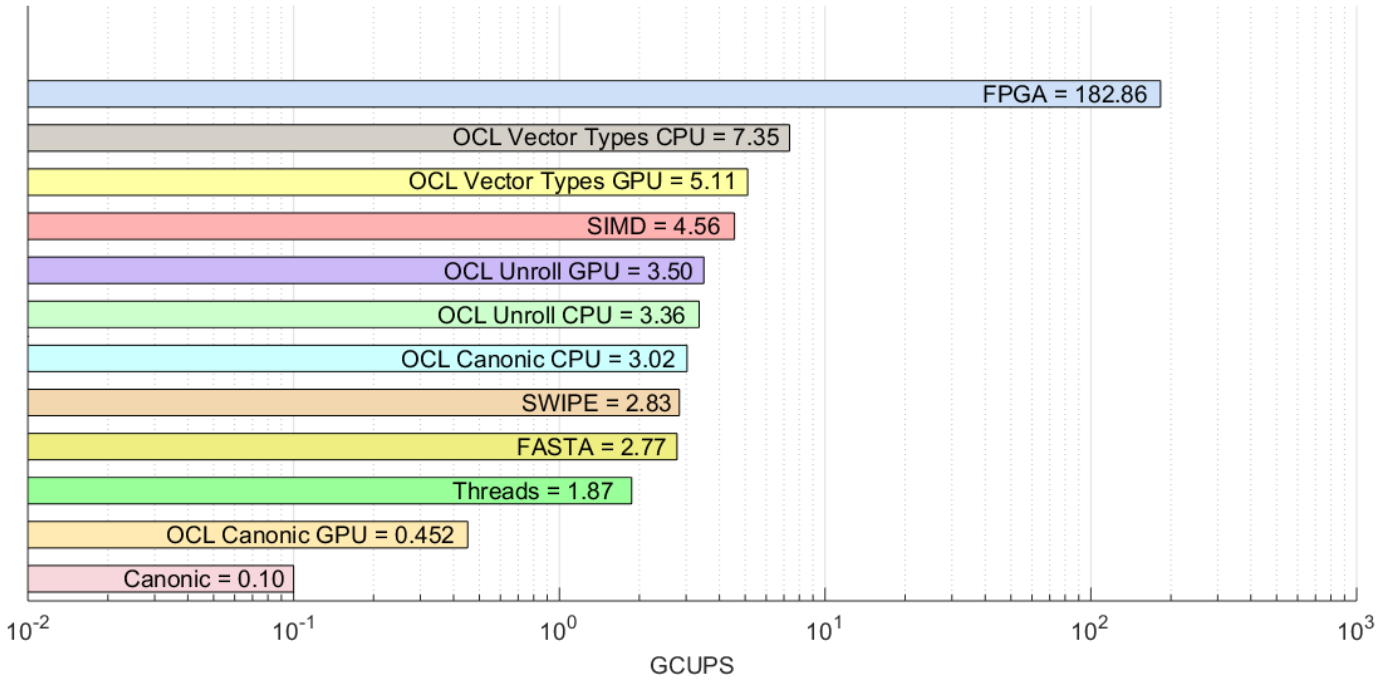
A Equação 9.1 descreve a celeridade na qual todos os b indivíduos no banco de dados são alinhados no evento de uma consulta. Em que cada indivíduo possui uma matriz mxm além de p sequências que o representam. Essa equação cobre somente a computação dos escores, porém, os tempos medidos nos experimentos também incluem o tempo da computação do traceback.

Na Figura 107 pode ser visto que a implementação em um único FPGA atingiu o maior desempenho de 182,86 GCUPS, o que representa a um speed-up de 1885x sobre a versão canônica.

Comparando-se o desempenho do FPGA com a projeção mostrada na Seção 8.8, para essa configuração vê-se que a previsão era de 336 GCUPS. Porém a medida real de desempenho corresponde à 54% da medida de pico. Conclui-se que a métrica GCUPS de pico, utilizada por exemplo em Isa (Isa, et al. 2014), previsivelmente, se mostra bastante imprecisa.

Os speed-ups dos softwares de referência SWIPE e FASTA foram $2{,}83$ e $2{,}77$ vezes, respectivamente. O que corresponde apenas a um desempenho 51% e 48% melhor que a versão Multi-Threads.

Figura 107 Comparação do desempenho médio das configurações em GCUPS



A Versão Multi-threads fica abaixo (62%) da Versão OCL Canonic CPU, para a mesma configuração de CPU. Isso expressa o bom gerenciamento de threads da solução OpenCL, além da extração do paralelismo SIMD obtido pelo compilador de OpenCL. Também se observa que a versão OCL Canonic CPU já ultrapassa o desempenho dos softwares SWIPE e FASTA.

As versões que exploram um certo nível de desenrolamento de laços, chamadas de OCL Unroll, ultrapassam suas respectivas versões canônicas e mais fortemente isso acontece na versão em GPU que deixa de utilizar a memória global e passa a utilizar a memória privada da thread. Na GPU esse ganho foi de 7,74x e na CPU bem menor de 1,11x. No caso da CPU esse ganho se deve ao desenrolar do laço.

A Versão SIMD sobrepõe-se as versões OCL Unroll sendo 30% melhor que a versão OCL Unroll GPU. Essa melhoria pode ser insuficiente para justificar o maior esforço de projeto e mais baixa portabilidade de código da implementação com instruções SIMD.

A versão OCL Vector Types GPU supera a versão SIMD em apenas 12%, porém, a versão OCL Vector Types CPU supera a SIMD em 61%, demonstrando o melhor uso do paralelismo vetorial na mesma configuração de CPU.

Na comparação entre GPU e CPU, utilizando-se como base as versões OCL Vector Types a CPU é 43% mais rápida que a GPU, demonstrando o domínio do paralelismo no nível de instruções da CPU em um problema com característica de frente de onda, utilizando-se maciçamente operações de inteiros. Outro fato que pode explicar o maior desempenho da versão em GPP é sua maior frequência de operação de 2,4GHz contra 1GHz do núcleo da GPU.

É importante ressaltar que o desempenho da GPU não foi penalizado pelas transferências de dado via barramento PCI-e. Apesar do banco de dados ser maior que a memória da GPU, somente o tempo líquido de processamento foi computado.

O desempenho da GPU foi limitado pelo número de *stream processors* e pela frequência de operação do seu núcleo. Esse fato é demonstrado, nos experimentos, visto que a memória global da GPU utilizada consegue armazenar 11,024 indivíduos x 15 sequências, resultando em 165.360 alinhamentos, o que é muito mais do que os 640 *stream processors* dentro da GPU.

A Tabela 8 lista os tempos de processamento e seus correspondentes speed-ups para todas as implementações e para os benchmarks sobre a versão canônica. Esses dados correspondem à média do tempo de uma única consulta repetida 10 vezes ao banco de dados com 8 milhões de indivíduos.

Além das implementações de referência em CPU (SWIPE e FASTA), a Tabela 8 mostra que os resultados apresentados aqui apresentam maiores speed-ups também em comparação com os trabalhos de Benkrid (Benkrid, et al. 2012) e de ISA (Isa, et al. 2014) que são trabalhos de implementação em FPGA e ainda maiores speed-ups do que o trabalho de GPU de Liu (Liu, Wirawan e Schmidt 2013) (que demonstra um ganho 4.5x sobre o SWIPE).

Tabela 8 Tempos de processamento para uma única consulta a um banco de dados com 8 milhões de indivíduos e o speed-up sobre a versão canônica conseguido

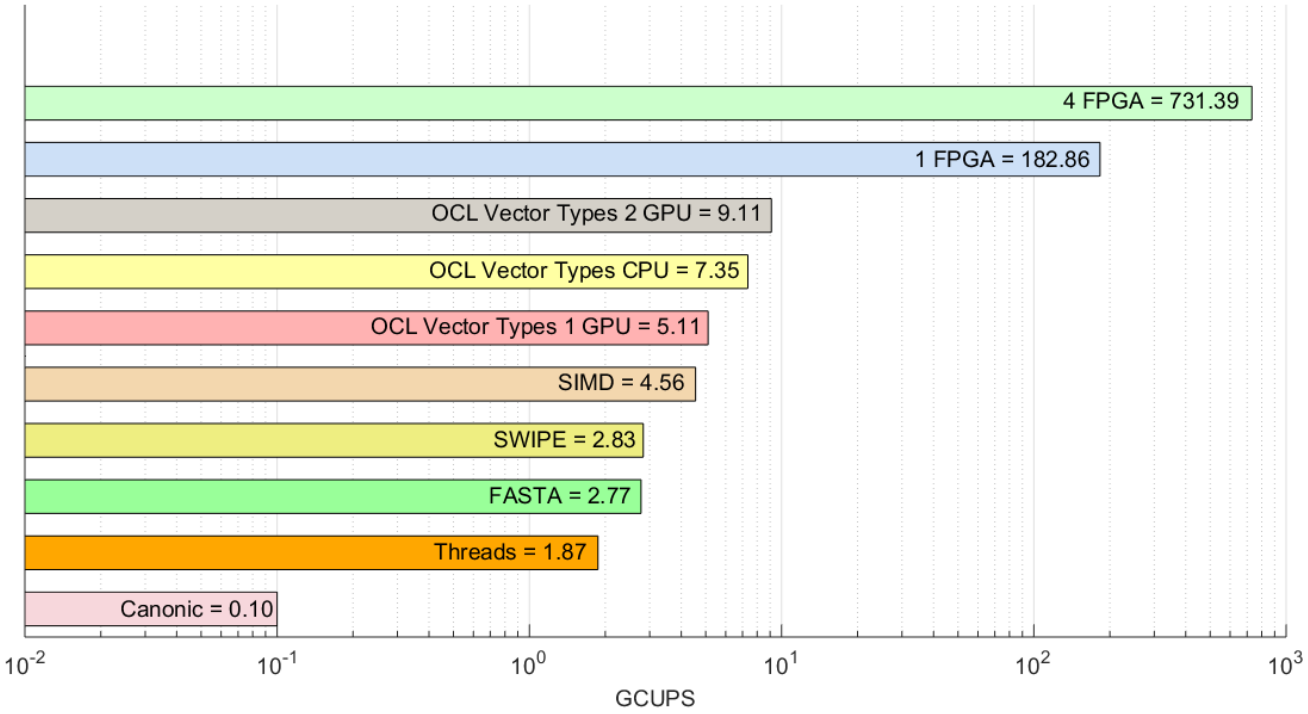
Implementação	Tempo	Speed-up
Canônico	19,8 h	1x
OCL Canonic GPU	4,24 h	4,68x
Threads	1,0 h	19,78x
SWIPE [31]	40,67 min	29,20x
FASTA [32]	41,61 min	28,54x
OCL Canonic CPU	37,60 min	31,30x
OCL Unroll CPU	33,67 min	34,74x
OCL Unroll GPU	32,79 min	36,20x
SIMD	24,38 min	47,24x
OCL Vector Types GPU	22,10 min	52,87x
OCL Vector Types CPU	15,38 min	76,09x
1 FPGA	37,81 seg	1885,02x
Benkrid [24]	–	228x
Isa [25]	–	380x

Para demonstrar a escalabilidade do uso múltiplos aceleradores para acelerar o problema, utilizou-se duas GPUs e os quatro FPGAs da placa da Gidel. Na Figura 108 pode-se ver que Versão OpenCL rodando em 2 GPUs (chamada de OCL 2 GPUs) dobra o desempenho da versão com uma GPU, particionando a quantidade total de indivíduos por dois. Isso mostra a linearidade esperada para uma aplicação com múltiplas GPUs.

No caso do particionamento do problema com aceleradores em FPGA, a Figura 108 também mostra o desempenho de 4 FPGAs responsáveis por rodar cada um 2 milhões de indivíduos. A independência entre alinhamentos de diferentes indivíduos, permitiu que o desempenho do processamento com 4 FPGAs também fosse linear.

Observa-se, também, na Figura 108 que a versão com duas GPUs foi apenas 1,39x mais rápida que a versão OCL Vector Types CPU.

Figura 108 Desempenho médio das configurações em GCUPS acrescentando 2 GPUs e 4 FPGAs



A Tabela 9 mostra os tempos de processamento e speed-ups ressaltando a escalabilidade para 2 GPUs e para 4 FPGAs. Na Tabela 9 observa-se que o desempenho de um único FPGA só é atingido utilizando-se 35 GPUs.

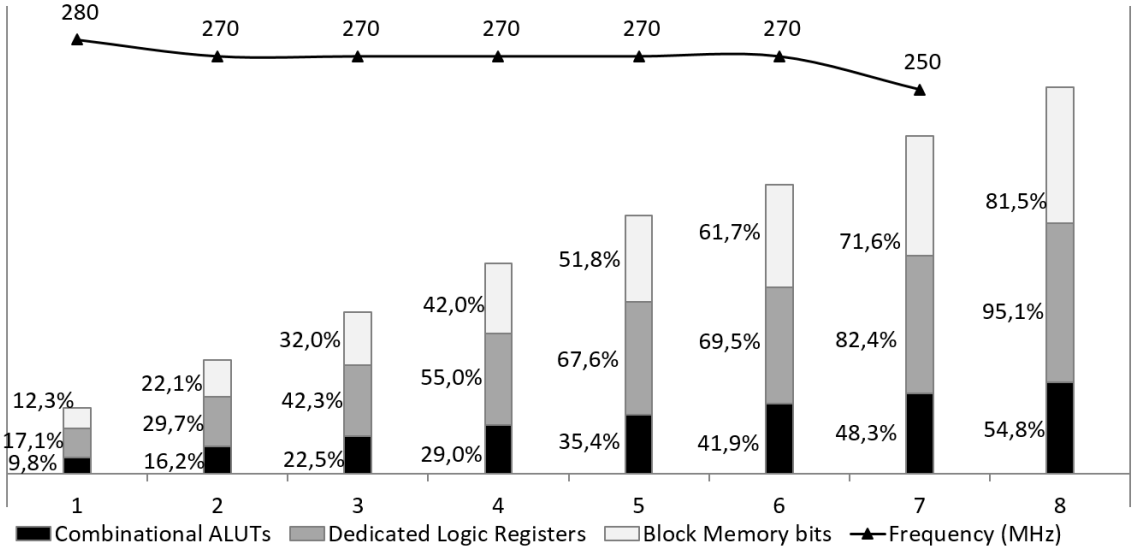
Tabela 9 Escalabilidade para 2 GPUs e para 4 FPGAs

Implementação	Tempo	Speed-up
Canônico	19,8 h	1x
OCL Vector Types 1 GPU	22,10 min	52,87x
OCL Vector Types 2 GPUs	12,22 min	97,17x
1 FPGA	37,81 seg	1885,02x
4 FPGAs	9,45seg	7212,96x

Numa análise posterior, outra configuração, agora com $m = 256$ nucleotídeos, $k = 16$ PEs, $p = 15$ alinhamentos foi utilizada para se verificar a evolução do consumo de recursos e frequência máxima de operação atingida quando se varia a quantidade de indivíduos i em paralelo no FPGA. A Figura 109 mostra esses dados para $i = 1,\ldots,8$. A quantidade máxima de

indivíduos que coube no FPGA foi de $i = 7$ e o recurso crítico foi o uso de registradores. Observa-se também que a frequência máxima de operação ficou em 250MHz, para $i = 7$.

Figura 109 Evolução da Utilização de recursos e da frequência de operação com o crescimento do número de indivíduos



A Tabela 10 mostra o tempo de processamento, a medida de desempenho em GCUPS e o speed-up conseguido, tomando como referência o tempo do software FASTA. Como comparativo, foram utilizados o software SWIPE, e a versão em GPU com melhor desempenho no experimento anterior.

Tabela 10 Tempo de processamento, GCUPS e speed-up

Implementação	Tempo	GCUPs	Speed-up
FASTA [32]	63,4 min	2,07	1x
SWIPE [31]	40,20 min	3,26	52,87x
OCL Vector Types 1 GPU	26,79 min	4,89	97,17x
1 FPGA	**0,557 min**	**235,46**	**113,90x**

Observa-se que o FPGA mantém uma larga vantagem, sendo 113,9x mais rápido que o software FASTA e 48x mais rápido que a GPU.

A Tabela 11 confronta a implementação em FPGA com os trabalhos da literatura em FPGA e em GPU, utilizando a métrica GCUPS, dando destaque tanto à frequência de operação atingida quanto à tecnologia utilizada nos protótipos. Pode-se ver, também, na **Erro! Fonte de referência não encontrada.** que o trabalho aqui proposto apresenta uma grande vantagem em relação aos outros em FPGA, em GPU e em Xeon Phi.

Tabela 11 Comparativo do desempenho com outros trabalhos

Trabalho	Dispositivo (Tecnologia)	F_{max}	GCUPs
Benkrid	Virtex-4 160 (90nm)	80MHz	19,4
ISA	Virtex-5 110 (65nm)	200MHz	39,0 (Pico)
Liu	GPU GTX 680 (28nm)	1GHz	80,0
Liu	Xeon Phi 5110P (22nm)	1GHz	30,1
Proposto	Stratix-IV 530 (40nm)	**250MHz**	**235,46**

9.1 Reprodutibilidade dos Testes

Todas as versões de software usadas neste trabalho, bem como o gerador do banco de dados sintético estão disponíveis como um projeto do Sourceforge (Ferreira n.d.).

O projeto em FPGA foi sintetizado usando o Quartus II 14.0.2 da Altera, visando o FPGA Stratix IV EP4SE530H35C2, que compõe a placa PROCStar IV da Gidel (Gidel n.d.).

As versões canônica, multi-threads e OpenCL CPU rodaram em um servidor Supermicro dual Intel Xeon E5645 (6 cores, 12 threads) 2.4GHz, e com 48GB de RAM DDR3 ECC. Os softwares SWIPE 2.0 e FASTA 36.3 usados como benchmarks também utilizaram o servidor da Supermicro como plataforma.

Duas GPUs AMD Radeon HD7770 (1GB GDDR5, frequência de núcleo: 1 GHz, banda de memória: 72 GB/s, Stream Processors: 640) foram usadas para rodar as versões OpenCL em GPU.

Todas as versões em software foram desenvolvidas na linguagem C e

compiladas usando o GCC 5.0 com a opção -O3 em modo release, no SO Linux, distribuição Ubuntu 15.04 64 bits

.

10

Conclusões

Conclui-se este trabalho com uma arquitetura de hardware de um acelerador implementada em HDL, validada em simulação pós-síntese com timing e prototipada em uma plataforma de FPGA. Foi construído um estudo de caso para verificar o desempenho e corretude da implementação.

A arquitetura proposta visou a flexibilidade, deixando parametrizáveis diversas variáveis e a implementação refletiu essa preocupação, empregando diretivas de geração de código em HDL de SystemVerilog. Essa estrutura permite menor esforço na obtenção de diferentes instâncias da arquitetura.

Inúmeros testes comparativos de desempenho foram realizados. Incluindo modelo CPU canônico, modelo em Threads, modelo em OpenCL em CPU, modelo em OpenCL em GPU AMD e modelo em OpenCL em GPU Nvidia. Para realizar esses testes um banco artificial de sequências foi gerado contendo 8 milhões de indivíduos identificados por 15 sequências cada um e cada sequência com tamanho de 240 nucleotídeos.

Em todos os testes a solução desenvolvida em FPGA apresentou larga margem de liderança, obtendo o speedup médio de 1885,02x para o estudo de caso.

Observa-se que este trabalho atinge o seu objetivo acelerando fortemente a

busca em um banco de DNA composto pelos referidos 8 milhões de indivíduos, na forma do foi proposto para o banco nacional de criminosos do Brasil.

10.1 Publicações

O acelerador de hardware descrito neste trabalho foi objeto de um pedido de patente nacional depositado no INPI (Instituto Nacional de Propriedade Intelectual) sob o número BR 10 2014 022189 1. A parte de hardware deste trabalho obteve resultado favorável na submissão ao ASAP 2016 sob o título "*A Hardware Accelerator for the Alignment of Multiple DNA Sequences in Forensic Identification*".

11

Trabalhos futuros

Como trabalhos futuros, se pode elencar a extensão da solução, proposta neste trabalho, para realizar alinhamentos locais utilizando o algoritmo de Smith-Waterman. Bem como sua adaptação para incluir a opção de processar o alinhamento global e local de proteínas.

Outros algoritmos de alinhamento podem ser explorados como o de alinhamentos múltiplos, ou a implementação de algoritmos heurísticos como o BLAST.

Na ceara da produção de fármacos um problema realmente desafiante é o da predição da estrutura de proteínas a partir das cadeias de aminoácidos simples. Nesse contexto o algoritmo de Needleman-Wunsch pode ser utilizado para a predição da estrutura de segunda ordem para proteínas. Esse é o primeiro passo para a predição da estrutura da molécula de uma proteína.

Referências

(NHGRI), National Human Genome Research Institute. *Bioinformatics: Finding Genes.* s.d. http://www.genome.gov/25020001 (acesso em 01 de 11 de 2015).

Altschu, Stephen F., Warren Gish, Webb Miller, Eugene W. Myers, e David J. Lipman. "Basic Local Alignment Search Tool ." *Journal of Molecular Biology* 215 (1990): 403-410.

Aluru, S., e N. Jammula. "A Review of Hardware Acceleration for Computational Genomics." *Design & Test*, 2014, 1 ed.: 19-30.

"AMD APP SDK OpenCL User Guide." 08 de 2015. http://amd-dev.wpengine.netdna-cdn.com/wordpress/media/2013/12/AMD_OpenCL_Programming_User_Guide2.pdf (acesso em 27 de 01 de 2016).

Benkrid, Khaled, Ali Akoglu, Cheng Ling, Yang Song, Ying Liu, e Xiang Tian. "High performance biological pairwise sequence alignment: FPGA versus GPU versus cell BE versus GPP." *International Journal of Reconfigurable Computing - Special issue on High-Performance Reconfigurable Computing*, Janeiro de 2012.

Bordoli, Lorenza. "Similarity Searches on Sequence Databases." Outubro de 2003. http://www.ch.embnet.org/CoursEMBnet/Basel03/slides/BLAST_FASTA.pdf (acesso em 23 de Fevereiro de 2015).

Catanzaro, Bryan. *OpenCL™ Optimization Case Study: Diagonal Sparse Matrix Vector Multiplication Test.* s.d. http://developer.amd.com/resources/documentation-articles/articles-whitepapers/opencl-optimization-case-study-diagonal-sparse-matrix-vector-multiplication-test/ (acesso em 27 de 10 de 2015).

Clark, Mike. "Introduction to GPU Computing." s.d. http://www.int.washington.edu/PROGRAMS/12-2c/week3/clark_01.pdf (acesso em 27 de 01 de 2016).

CODIS—NDIS Statistics. n.d. https://www.fbi.gov/about-us/lab/biometric-analysis/codis/ndis-statistics (accessed 08 12, 2015).

Conselho Nacional de Justiça. "Novo Diagnóstico de Pessoas Presas no Brasil." junho de 2014. s.conjur.com.br/dl/censo-carcerario.pdf (acesso em 20 de Janeiro de 2015).

Conselho Nacional do Ministério Público. "ENASP." s.d. http://www.cnmp.mp.br/portal/images/stories/Enasp/relatorio_enasp_FINAL.pdf (acesso em 20 de Janeiro de 2015).

—. *Inquerômetro.* s.d. http://inqueritometro.cnmp.gov.br/inqueritometro/home.seam (acesso em 20 de Janeiro de 2015).

Crick, Francis. "Central Dogma of Molecular Biology." *Nature* 227 (1970): 561-563.

DNA. s.d. https://en.wikipedia.org/wiki/DNA (acesso em 01 de 11 de 2015).

Ferreira, Antonyus Pyetro Amaral. *homepage of the BIOHPCIN Sourceforge project.* n.d. http://sourceforge.net/projects/biohpcin/ (accessed 09 02, 2015).

Gidel. *Página do produto: PROCStar IV.* s.d. http://www.gidel.com/PROCStar IV.htm (acesso em 20 de janeiro de 2015).

—. *PROCStar IV product page.* n.d. http://www.gidel.com/PROCStar IV.htm (accessed 01 15, 2015).

Gokhale, M. B., e P. S. Graham. *Reconfigurable computing: Accelerating computation with field-programmable gate arrays.* New York: Springer-Verlag, 2005.

Intel. "Intel Xeon Phi Product Family: Product Brief." s.d. http://www.intel.com/content/www/us/en/high-performance-computing/high-performance-xeon-phi-coprocessor-brief.html (acesso em 18 de Fevereiro de 2015).

Isa, M.N., S.A.Z. Murad, R.C. Ismail, M.I. Ahmad, A.B. Jambek, e M.K. Md Kamil. "An efficient processing element architecture for pairwise sequence alignment." *Electronic Design (ICED), 2014 2nd International Conference on.* Penang, 2014.

Ji, Yingsheng, Li Liu, e Guangwen Yang. "Characterization of Smith-Waterman sequence database search in X10." *Proceedings of the 2012 ACM SIGPLAN X10 Workshop* . Nova York, 2012 .

Liu, Y., A. Wirawan, e B. Schmidt. "CUDASW++ 3.0: accelerating Smith-Waterman protein database search by coupling CPU and GPU SIMD instructions." *BMC Bioinformatics* , 2013.

Liu, Yongchao, Tuan-Tu Tran, F. Lauenroth, e B Schmidt. "SWAPHI-LS: Smith-Waterman Algorithm on Xeon Phi coprocessors for Long DNA Sequences ." *Cluster Computing (CLUSTER), IEEE International Conference on* . Madrid, 2014.

Marmolejo-Tejada, J.M., V. Trujillo-Olaya, C.P. Renteria-Mejia, e J. Velasco-Medina. "Hardware implementation of the Smith-Waterman algorithm using a systolic architecture ." *Circuits and Systems (LASCAS), 2014 IEEE 5th Latin American Symposium on* . Santiago, 2014.

Morony, C. S., e E. D. Moreno. "FPGA-Based Implementation and Performance of the Global and Local Algorithms for the Gens Alignment." *IEEE LATIN AMERICA TRANSACTIONS*, 12 de 2008.

National Human Genome Research Institute (NHGRI). n.d. http://www.genome.gov/sequencingcosts/ (accessed 05 03, 2016).

Needleman, S. B., e C. D. Wunsch. "A general method applicable to the search for similarities in the amino acid sequence of two proteins." *Journal of Molecular Biology*, 1970: 443-453.

Office, UK's Home. "National DNA Database statistics." 02 25, 2015. https://www.gov.uk/government/uploads/system/uploads/attachment_data/file/452250/NDNAD_Website_statistics_Q1_15-16_-_Final_Copy.xls (accessed 08

12, 2015).

Parberry, Ian. “Dynamic Programming.” In: *Problems on Algorithms*. Prentice Hall, 1995.

Rocha, R. C. F. “DESENVOLVIMENTO DE UMA PLATAFORMA RECONFIGURÁVEL PARA MODELAGEM 2D, EM SÍSMICA, UTILIZANDO FPGAS.” *Dissertação de Mestrado.* 2010.

Rognes, Torbjørn. “Faster Smith-Waterman database searches with inter-sequence SIMD parallelisation.” *BMC Bioinformatics*, 2011.

Savran, I., Yang Gao, e J.D. Bakos. “Large-Scale Pairwise Sequence Alignments on a Large-Scale GPU Cluster .” *Design & Test, IEEE* 31, n. 1 (2014): 51-61 .

Setubal, J., e J. Meidanis. *Introduction to Computational Molecular Biology.* Boston: PWS Publishing Company, 1997.

Singh, Gautam B. *Fundamentals of Bioinformatics and Computational Biology.* Springer, 2015.

Smith, Temple F., e Michael S. Waterman. “ Identification of Common Molecular Subsequences.” *Journal of Molecular Biology* 147 (1981): 195-197.

Sutherland, S., S. Davidmann, e P. Flake. *SystemVerilog For Design.* Vol. Segunda edição. Springer, 2006.

Teama, Salwa Hassan. “Introduction To Molecular Biology.” s.d. http://www.pitt.edu/~super7/32011-33001/32771.ppt (acesso em 01 de 11 de 2015).

"The FASTA program package." n.d. http://faculty.virginia.edu/wrpearson/fasta/fasta_guide.pdf (accessed 08 25, 2015).

The OpenCL Programming Book. s.d. http://www.fixstars.com/en/opencl/book/OpenCLProgrammingBook/opencl-c/ (acesso em 19 de 10 de 2015).

top500. *top500.* 2015. http://www.top500.org/lists/2014/11/download/TOP500_201411_Poster.png (accessed março 22, 2015).

WATSON, J. D., e F. H. C. CRICK. “A structure for deoxyribose nucleic acid.” *Nature*, n. 171 (1953).

Xilinx. *What is a FPGA?* s.d. http://www.xilinx.com/fpga/ (acesso em 20 de março de 2015).