Pós-Graduação em Ciência da Computação

“DESENVOLVIMENTO DE UMA PLATAFORMA RECONFIGURÁVEL PARA MODELAGEM 2D, EM SÍSMICA, UTILIZANDO FPGAS”

Por

Rodrigo Camarotti Ferreira da Rocha

Dissertação de Mestrado

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

RECIFE, AGOSTO/2010

UNIVERSIDADE FEDERAL DE PERNAMBUCO

CENTRO DE INFORMÁTICA

PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

RODRIGO CAMAROTTI FERREIRA DA ROCHA

"DESENVOLVIMENTO DE UMA PLATAFORMA RECONFIGURÁVEL PARA MODELAGEM 2D, EM SÍSMICA, UTILIZANDO FPGAS"

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 MESTRE EM CIÊNCIA DA COMPUTAÇÃO.

ORIENTADOR(A): MANOEL EUSÉBIO DE LIMA

RECIFE, AGOSTO/2010

Catalogação na fonte
Bibliotecária Jane Souto Maior, CRB4-571

Rocha, Rodrigo Camarotti Ferreira da
Desenvolvimento de uma plataforma reconfigurável para modelagem 2D, em sísmica, utilizando FPGAS / Rodrigo Camarotti Ferreira da Rocha - Recife: O Autor, 2011.
x, 123 folhas : il., fig., tab.

Orientador: Manoel Eusébio de Lima.
Dissertação (mestrado) - Universidade Federal de Pernambuco. CIn, Ciência da Computação, 2011.

Inclui bibliografia.

1. Engenharia da computação. 2. Arquitetura de computador. 3. FPGA. 4. Computação de alto desempenho. I. Lima, Manoel Eusébio de (orientador). II. Título.

621.39 CDD (22. ed.) MEI2011 – 190

Dissertação de Mestrado apresentada por **Rodrigo Camarotti Ferreira da Rocha** à Pós-Graduação em Ciência da Computação do Centro de Informática da Universidade Federal de Pernambuco, sob o título "**Desenvolvimento de uma Plataforma Reconfigurável para Modelagem 2D, em Sísmica, Utilizando FPGAS**", orientada pelo **Prof. Manoel Eusebio de Lima** e aprovada pela Banca Examinadora formada pelos professores:

Prof. Abel Guilhermino da Silva Filho
Centro de Informática / UFPE

Prof. Paulo Sérgio Brandão do Nascimento
Instituto Federal de Pernambuco

Prof. Manoel Eusebio de Lima
Centro de Informática / UFPE

Visto e permitida a impressão.
Recife, 3 de setembro de 2010.

Prof. Nelson Souto Rosa
Coordenador da Pós-Graduação em Ciência da Computação do
Centro de Informática da Universidade Federal de Pernambuco.

Resumo

A exploração sísmica é uma técnica exploratória, que tem como objetivos localizar depósitos de minerais, hidrocarbonetos (ex: petróleo e gás natural), e sítios arqueológicos, capturando informações geológicas do ambiente que comporta estes elementos. A maioria das companhias de petróleo apoia-se na interpretação sísmica para definir os lugares de exploração dos poços.

Os métodos sísmicos utilizados na exploração sísmica têm como objetivo gerar uma imagem do terreno que se tem interesse. Esses métodos geralmente requerem sistemas com alto poder computacional, devido à quantidade massiva de dados e de computações necessária para a resolução dos mesmos.

A Migração Reversa no Tempo (Reverse Time Migration - RTM) é um método sísmico que resolve a equação de onda assumindo que seus campos de pressão podem se propagar da fonte de ondas sísmicas para os hidrofones, estágio denominado de modelagem, e dos hidrofones para a fonte de ondas sísmicas, o que é definido como migração. O RTM consegue gerar boas imagens em terrenos bastante complexos, porém seu custo computacional é bastante elevado. Sua utilização vem aumentando nos últimos anos, devido a grande melhora no desempenho das CPUs e o surgimento de ambientes de computação paralela, como clusters, GPU, e FPGA.

Esta dissertação irá explorar a possibilidade de utilização de FPGAs para realizar a aceleração do problema de modelagem sísmica em 2D, primeiro passo computacional do método RTM. Para realizar essa exploração foi desenvolvida uma plataforma reconfigurável baseada em FPGA que utiliza uma plataforma da GiDEL, denominada PROCe-III. O sistema a ser apresentado como proposta adota um modelo co-design, tendo a unidade de software representada por uma CPU e, um FPGA, representando o componente de hardware, como um coprocessador.

Palavras-chave: FPGA, computação de alto desempenho, exploração sísmica, RTM.

Abstract

The seismic exploration is an exploratory technique that aims to locate mineral deposits, hydrocarbons (e.g. oil and natural gas), and archaeological sites, capturing geological information of the environment that includes these elements. Most oil companies are based on seismic interpretation to define the places of exploration wells.

Seismic methods used in seismic exploration are intended to generate an image of the terrain that has an interest. These methods typically require systems with high computational power, due to the massive amount of data and computations required to solve them.

The Reverse Time Migration (Reverse Time Migration - RTM) is a seismic method that solves the wave equation assuming that their pressure fields can propagate from the source of seismic waves to the geophones, called the modeling stage, and from the geophones to source of seismic waves, which is defined as migration. The RTM can produce good images in very complex terrain, but its computational cost is quite high. Its use has increased in recent years due to great improvement in the performance of CPUs and the emergence of parallel computing environments such as clusters, GPU and FPGA.

This dissertation will explore the possibility of using FPGAs to perform the acceleration of the problem of 2D seismic modeling, the first step of the RTM method. To perform this operation, a reconfigurable FPGA-based platform was developed using a platform called PROCe-III from GiDEL. The system to be submitted as a proposal adopts a co-design model, with the software unit represented by a CPU and an FPGA, representing the hardware component, such as a coprocessor.

Keywords: FPGA, high-performance computing, seismic exploration, RTM

Sumário

Lista de Figuras

Lista de Tabelas

Capítulo

1

1 Introdução

Este capítulo relata as principais motivações que levaram ao desenvolvimento desta dissertação, qual seja, uma plataforma reconfigurável para problemas de modelagem-2D, em sísmica. Nesse capítulo são apresentadas as principais características do problema a ser resolvido, assim como, possíveis sistemas que visam solucioná-lo. Neste capitulo também é apresentada em detalhes a estruturada desta dissertação.

1.1 Contexto e Motivação

O setor de energia nos dias de hoje, em qualquer nação, é um dos fatores estratégicos mais importantes para seu desenvolvimento. Esta energia pode ser identificada de várias formas, tais como: energia de fonte hidráulica, eólica, marés, além daquelas fontes oriundas de material Fóssil, como carvão e petróleo.

O Brasil, um dos países mais ricos do mundo, membro do seleto grupo G20 [71] e do BRIC [72], possui hoje em seu portfólio, com uma de suas principais fontes energética, a extração e refinamento do petróleo. Este hidrocarboneto encontra-se concentrado em vários pontos do continente e costa brasileira, em poços subterrâneos, com diferentes profundidades, desde centenas a milhares de metros, em terra, ou abaixo do nível do mar. Empresas brasileiras como a Petrobrás, exploram hoje com sucesso poços a quilômetros de profundidade, como o projeto pré-sal [73], um desafio de bilhões de dólares em investimento.

Observa-se também que a exploração de riquezas com o petróleo em regiões de difícil acesso, exige um alto investimento em equipamentos e técnicas de exploração, para avaliar e definir potencial e posições ideais para perfuração de poços e por conseguinte, extração de petróleo.

A exploração sísmica, tema principal desta dissertação, é uma das técnicas exploratórias, que tem como objetivos localizar depósitos de minerais, hidrocarbonetos (ex: petróleo e gás natural), e sítios arqueológicos, capturando informações geológicas do ambiente que comporta estes elementos. Este tipo de exploração fornece dados que, quando são utilizados em conjunto com dados geofísicos, geológicos e dados obtidos através de poços, podem fornecer informações sobre a estrutura e distribuição das rochas da região em análise.

Essencialmente a exploração sísmica consiste na geração de ondas sísmicas e na medição do tempo requerido para que estas ondas viajem da fonte de emissão de ondas até um conjunto de hidrofones, estes geralmente colocados na superfície do terreno em análise. Hidrofones são equipamentos que captam informações de ondas geofísicas.

Através da medição do tempo de viagem e da velocidade das ondas sísmicas, o caminho percorrido por essas ondas pode ser reconstruído, resultando em dados que são utilizados para verificar a disposição das rochas no terreno e a existência de possíveis reservas de óleo.

A maioria das companhias de petróleo apoia-se na interpretação sísmica para definir os lugares de exploração dos poços. Apesar dos métodos sísmicos serem indiretos, já que eles não indicam onde está o petróleo, mas sim, como é a estrutura geológica do terreno, a probabilidade de localização de um poço com sucesso é bem maior quando tais métodos são utilizados. Por isso, a utilização dos mesmos é de grande importância para essas empresas.

Atualmente, a maioria dos métodos sísmicos utilizados pelas empresas de extração de gás e óleo utilizam métodos sísmicos que podem ser divididos em dois conjuntos. O primeiro são os métodos que resolvem diretamente a equação de propagação de ondas, e utilizam aproximações matemáticas que assumem que os campos de pressão da onda se propagam apenas em uma direção: da fonte de ondas sísmicas para o hidrofones. O segundo conjunto são os métodos que não resolvem diretamente a equação de onda, como o método de Kirchhoff [6], que é, atualmente, um dos métodos mais utilizados pelas empresas petrolíferas [74]. No entanto, a imagem resultante destes dois conjuntos de métodos fica bastante comprometida em terrenos muito complexos, com grandes inclinações e acidentados.

Em geral, para a utilização destes métodos, como o método de Kirchhoff [6], nessas áreas complexas, por exemplo, torna-se necessário a introdução de várias restrições e compromissos para a melhora da qualidade das imagens geradas. Mesmo assim, em terrenos que possuem formações com inclinações superiores a 70°, essas técnicas não conseguem mostrar em suas imagens geradas as estruturas que estão abaixo dessas inclinações [1].

A Migração Reversa no Tempo (Reverse Time Migration - RTM) [1], por outro lado, é um método matemático que resolve a equação de onda assumindo que seus campos de pressão podem se propagar da fonte de ondas sísmicas para os hidrofones, estágio denominado de modelagem, e dos hidrofones para a fonte de ondas sísmicas, o que é definido como migração. O RTM, método de modelagem utilizado neste projeto, consegue gerar boas imagens em terrenos bastante complexos, porém seu custo computacional é muito mais elevado, já que este método realiza dois passos computacionais massivo em dados. Neste fluxo de processamento, no segundo estágio, a migração, várias comparações são realizadas entre os resultados do primeiro e do segundo passo. Ao término do processo, uma soma é realizada em

seus resultados. Apesar desse método não ser novo, sua utilização tem aumentado nos últimos anos, devido a grande melhora no desempenho das *CPUs* e o surgimento de ambientes de computação paralela, como sistemas baseados em clusters, com centenas de milhares de núcleos de processamento e unidades de armazenamentos de dados. Os principais problemas que tornam o RTM extremamente custoso são a grande quantidade de dados necessários para a computação de um único ponto e a computação para tratamento de bordas do modelo, já que a modelagem, diferentemente da propagação das ondas no terreno físico, possui reflexões artificiais na borda do modelo. Um outro problema, não menos relevante, é a quantidade de dados que deve ser armazenado para a computação do RTM, principalmente nos modelos 3D, aonde essa quantidade de dados chega a ordem de terabytes [2].

Pelo exposto acima, pode-se sugerir que otimizações, tais como representação algorítmica, técnicas de compactação, etc., para diminuir o número total de operações e reduzir a colocação de dados em disco rígido poderia resolver grande parte do problema de exploração sísmica. No entanto, apesar da redução do número de operações ser importante, na maioria dos sistemas que utilizam RTM, o grande custo ainda é o tempo necessário para a transferência de uma grande massa de dados armazenados em dispositivos de memória, para a unidade de processamento. Esse custo está em geral associado a aspectos como latência e largura de banda da memória. Latência diz respeito ao tempo de acesso e busca de dados da memória, enquanto que, largura de banda, representa o número de palavras que se pode transferir por unidade de tempo entre a memória e a unidade de processamento.

Um outro aspecto importante em computação científica, como o método RTM, é a representação de seus dados, inteiros, ponto fixo, ou ponto flutuante, dupla ou simples precisão, etc. Particularmente em sísmica, os dados a serem processados são em geral representados por números ponto flutuante com precisão simples (32 bits). Assim, a medida de desempenho mais utilizada para realizar comparações entre os diversos sistemas computacionais neste tipo de representação é dada em Megaflops ou Gigaflops. Este tipo de representação, embora necessária, exige um grande armazenamento de dados em estruturas não regulares de acesso, dificultando desempenho.

Devido às características citadas acima, o RTM é considerada uma aplicação que necessita de um sistema com um alto poder computacional, ou alto desempenho computacional (*high-performance computing - HPC*). Ou seja, para se processar de forma

eficiente o algoritmo *RTM* são necessárias plataformas computacionais com arquiteturas voltadas para processamento de problemas de alto desempenho.

1.1.1 Arquiteturas para processamento de alto desempenho

A rápida melhora de desempenho que os computadores digitais vêm sofrendo nos últimos nos, tornaram as *CPU multi-core* [49], de propósito geral, arquiteturas flexíveis, de baixo custo, e com grande potencial computacional, candidatas na resolução de aplicações cientificas como a modelagem em sísmica, processamento de imagens médicas, meteorologia, finanças, etc. No entanto, mesmo com velocidades nominais bastante elevadas, com arquiteturas *multi-core*, estes dispositivos não conseguem atingir, em média, um bom desempenho em operações com aritmética ponto flutuante, ou quando os algoritmos envolvidos apresentam uma alta complexidade computacional, ou ainda quando uma grande massa de dados precisa ser acessada durante o processamento (acesso a memória). No entanto, técnicas de otimizações com o particionamento do problema em módulos e distribuição de tarefas em *clusters*, ainda é uma das opções mais utilizadas por grandes corporações [26].

Outra possível arquitetura que poderia ser utilizada para a resolução de métodos como o *RTM* seria a utilização de dispositivos eletrônicos customizados, especialmente desenvolvidos para o problema, denominados de *ASIC* (*Application-Specific Integrated Circuits*) [50]. Diferentemente de *CPUs* convencionais, os *ASICs* são projetados com suas funcionalidades específicas, voltadas para a execução do algoritmo em desenvolvimento. Esta customização possibilita melhora de desempenho, que pode variar de 100 a 1000 ou mais vezes daquelas atingidas pelas *CPUs*, com menos área em silício e uma melhor eficiência no uso de energia. No entanto, esses sistemas não são flexíveis, requerem um longo período de desenvolvimento, além de um alto custo de fabricação. O custo do projeto pode ser amortizado quando uma grande quantidade de *chips* são implementados. Qualquer ajuste no algoritmo implicaria em um novo projeto e, por conseguinte, a confecção de um novo dispositivo.

Uma outra arquitetura bastante interessante que vem sendo utilizada, como plataforma de alto desempenho, na resolução de métodos científicos como *RTM* é a *GPU (Graphics Processing Unit)*[51] [52]. Estes dispositivos são verdadeiras máquinas paralelas com dezenas de núcleos de processamento interligados, possuem implementação de aritmética ponto

flutuante, além de uma grande largura de banda de memória, algumas vezes maior que a largura encontrada nas *CPUs* atuais. A maioria das GPUs é conectada a placas mãe através do slot *PCI-Express*.

Uma *GPU* é geralmente considerada difícil de ser programada, embora novas ferramentas como o CUDA [75], tenham sido lançadas com o intuito de tornar seu ambiente de programação mais amigável. Olhando simplesmente para a largura de banda da memória e para o possível poder computacional oferecido, as *GPUs* parecem ser o melhor sistema para resolver um método de modelagem como o RTM. No entanto, algumas limitações no modelo de programação e na hierarquia de memória não tornam essas expectativas tão reais. Em relação ao modelo de programação, para se atingir o máximo desempenho nestas arquiteturas, deve-se ter o maior número de *threads* possíveis, o que torna algumas otimizações inviáveis, além de tornar o sistema de cache complexo de ser manipulado.

Em relação à hierarquia de memória, o tamanho das memórias mais velozes restringe a ordem das aproximações que podem ser aplicadas à equação de onda, fazendo com que apenas ordens mais baixas possam ser utilizadas. O tamanho de modelos reais também se torna um problema, já que geralmente esses não cabem em toda a memória das *GPU*, fazendo com que o modelo tenha que ser dividido. Este particionamento pode criar outros gargalos, como o de comunicação *GPU* - memória principal, através de barramentos como o *PCI-Express*.

Uma outra arquitetura que vem sendo utilizada na execução de problemas que requerem alto desempenho computacional são os dispositivos lógicos reconfiguráveis, os *Field programmable gate arrays (FPGAs)* [76]. Em geral neste tipo de abordagem, os *FPGA* são tratados como coprocessadores reconfiguráveis, interligados a *CPUs*. Assim, em um ambiente híbrido, uma arquitetura *co-design*, onde os elementos de *software* seriam representados por *CPUs* de propósito geral e os de *hardware* por *FPGAs*, poder-se-ia alcançar grandes ganhos de desempenho e, ao mesmo tempo, um bom nível de flexibilidade, maior que aquele encontrado nas soluções puras, em *software*.

Inicialmente propostas nos anos 90, as plataformas reconfiguráveis têm sido amplamente utilizadas em aplicações de processamento digital de sinais em aritmética de ponto fixo, alcançado ganhos de desempenhos quando comparadas com os computadores genéricos [53] [54] e até GPUs [55] [56].

Recentemente, com o crescimento continuo da densidade destes dispositivos, e a inclusão de novos módulos aritméticos para *DSP* e mais bancos de memória, os projetistas começaram a considerar o potencial existente destes dispositivos na implementação de aplicações de alto desempenho também em aritmética de ponto flutuante. Agora, unidades aritméticas para operação ponto fixo e ponto flutuante, com precisão simples ou dupla poderiam ser implementadas facilmente. Alguns exemplos de pesquisas sendo realizadas nessa área incluem a migração sísmica [3], dinâmica de fluidos [4] e dinâmicas moleculares [5].

Neste contexto, plataformas reconfiguráveis baseadas em *FPGAs* possuem algumas características que as tornam, potencialmente, um dos melhores sistemas na implementação do método *RTM*. A primeira característica é a possibilidade de utilização de seu paralelismo intrínseco e estruturas *pileline*. *FPGAs* podem realizar processamento em *streams*, no qual vários dados são processados por varias operações diferentes em estágios de *pipelines* diferentes (*multiple data multiple instructions*), diferentemente, por exemplo, das *GPUs* onde, no processamento em *streams*, várias *threads,* idênticas, atuam em diferentes dados (*single instruction mutiple data*). Outra característica importante do *FPGA* é a possibilidade de reuso de dados que este sistema possibilita, fazendo com que, na implementação do *RTM*, um resultado possa ser gerado a cada ciclo de relógio. Esse reuso de dados é possível através dos blocos de memória que existem dentro das *FPGAs*. Outra característica muito interessante é a possibilidade de se poder customizar a formatação de dados, permitindo redução de precisão quando possível, sem comprometimento da qualidade dos resultados, para um determinado problema. Este tipo de abordagem não pode ser utilizada nem nas *CPUs* nem nas *GPUs*. Além dessa diminuição de precisão, um processo de compressão de dados ainda pode ser utilizado, melhorando ainda mais as vantagens citadas acima, principalmente no aumento da largura de banda no acesso a dados na memória do sistema.

Levando em consideração os fatos e argumentos citados anteriormente, esta dissertação propõe a implementação de uma arquitetura híbrida, composta por uma CPU de propósito geral e um dispositivo *FPGA,* que funcionam como coprocessador aritmético reconfigurável. Esta arquitetura foi desenvolvida, a partir de uma plataforma da GiDEL, denominada PROCe-III [34], com o objetivo de resolver o problema de modelagem em sísmica 2D, utilizando o modelo RTM.

A arquitetura foi implementada e estimativas de desempenho foram colhidas através de experimentos feitos na plataforma. Os resultados forma animadores e serão apresentados neste trabalho.

1.2 Estrutura da Dissertação

Esta dissertação encontra-se estruturada da seguinte forma: no capítulo 2 são apresentados os principais conceitos que foram estudados durante o processo de desenvolvimento do projeto da plataforma proposta. No capítulo 3 são apresentados vários trabalhos relacionados ao tema, que foram desenvolvidos abordando essa classe de problemas. No capitulo 4 é apresentada a plataforma PROCe III da GiDEL, que foi utilizada no desenvolvimento da plataforma proposta nesta dissertação. No capítulo 5 é apresentada a Plataforma reconfigurável, baseada em *FPGA,* proposta nessa dissertação, que tem como objetivo implementar a modelagem sísmica 2D. Neste capítulo também são apresentadas comparações de desempenho em relação à *CPUs multi-core*. No capítulo 6 são apresentadas as conclusões e trabalhos futuros.

Capítulo

2

2 Fundamentação Teórica

Neste capitulo são apresentadas as principais teorias que foram estudadas durante o desenvolvimento da plataforma proposta nesta dissertação.

2.1 Processamento de Dados Sísmicos

Os métodos sísmicos, tais como Kirchhoff [6] e Reverse Time Migration - RTM [1], possuem três aplicações principais [6]:

I. Sismologia de engenharia (*Engenering seismology*) - delimitação da geologia do solo próximo à superfície, com até 1 km de profundidade, para estudos de engenharia, e exploração de carvão e minério.

II. Sismologia de Exploração (*Exploration seismology*) - exploração e produção de hidrocarbonetos em profundidades de até 10 km, método sísmico aplicado à exploração e produção de petróleo e gás é conhecido.

III. Sismologia de Tremores de terra (*Earthquake seismology*) - investigação da estrutura da crosta terrestre em profundidades de até 100 km.

A sísmica de reflexão é a técnica de exploração geofísica mais difundida na indústria do petróleo, possuindo um papel chave no processo de localização de petróleo e gás a mais de sessenta anos [7].

O primeiro passo dessa técnica é escolher um local onde será colocada uma fonte de ondas sísmicas, no qual é colocado uma malha de hidrofones. Estes hidrofones, por sua vez, geram sinais elétricos que dependem dos ecos que as ondas sísmicas produzem ao viajar pelo meio em exploração. Junto ao hidrofones existe um sistema que faz o armazenamento e visualização dos sinais gerados pelos mesmos [8].

A ideia por trás dessa técnica é bastante simples: o solo pode ser modelado, de uma forma simplificada, como um meio estratificado com propriedades de material, tais como velocidade de propagação de ondas sísmicas, densidade, anisotropia, etc. Ondas sísmicas impulsivas são injetadas no meio e propagadas dentro do solo. Quando estas ondas encontram uma interface entre camadas de rocha, as ondas são refletidas de volta para a superfície, sendo captadas pelos hidrofones e devidamente armazenadas. O tempo de trânsito e amplitudes dessas reflexões podem ser utilizados para determinar a localização e profundidade das camadas de rocha [7]. A Figura 1 apresenta uma visão geral de como o processo de coleta de dados, através dos hidrofones, é realizada.

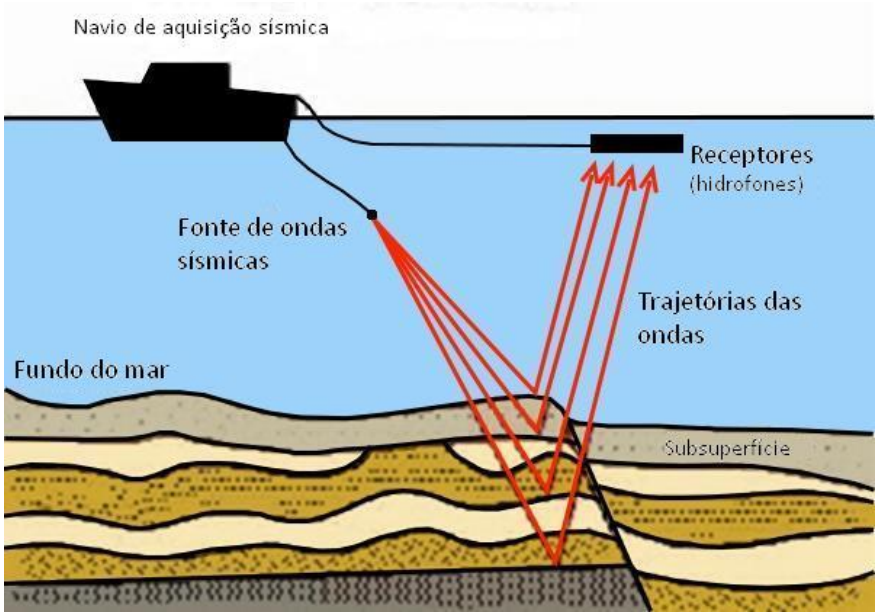


Figura 1 Método de Reflexão Sísmica

O objetivo básico do processamento de dados sísmicos é converter as informações que foram colhidas em campo para uma forma que possa ser utilizada na interpretação geológica. Os dados que inicialmente são gravados em fitas magnéticas (analógicas ou digitais) são transformados, pelo centro de processamento, em uma seção que possui certa semelhança com uma seção da estrutura geológica (seção sísmica). Um dos objetivos do processamento é eliminar ou pelo menos atenuar ruídos na informação que representam sinais que não são, de fato, reflexões primárias. Uma reflexão primaria representa a primeira reflexão que a onda sísmica sofre ao encontrar uma interface reflexora. Estes ruídos tornam as reflexões primárias mais confusas ou obscuras. Na verdade, o objetivo não é reduzir o nível absoluto do ruído, mas sim, aumentar a relação do nível do sinal pelo nível do ruído, chamada de relação sinal – ruído (SNR), onde esse sinal representa a reflexão primaria. Outro objetivo do processamento é apresentar as reflexões, na seção gravada, com a maior resolução e clareza possível, e com as relações geométricas entre elas apropriadas [9].

2.1.1 Modelagem e Migração sísmicas

O processamento de dados sísmicos envolve dois processos principais, a saber: modelagem sísmica e migração sísmica [7].

Esses dois passos, são, de uma forma simplificada, um o inverso do outro [10].

A modelagem sísmica representa o processamento direto em que a propagação da onda é feita a partir da fonte de ondas sísmicas, indo para as interfaces de rochas e depois para os hidrofones. O modelo de interfaces de rocha é obtido por estudo e modelagem de bacias.

A migração representa o processamento inverso à modelagem, agora, a propagação de ondas sísmicas é feita a partir dos hidrofones, indo para as interfaces de rocha até atingir de volta a fonte, sendo usados como fonte de energia em cada hidrofone os sinais capturados no campo.

A Figura 2 mostra um fluxograma simples do processamento de dados sísmicos que envolve os dois processos citados acima. Inicialmente, a modelagem é realizada e dados resultantes desse passo são armazenados. Após a execução da modelagem, a migração é iniciada, onde a cada passo de processamento os dados gerados pela modelagem são corelacionados com os dados que são gerados pela migração. O resultado da migração é uma imagem do terreno de interesse.

Figura 2 Fluxograma do processamento de dados sísmicos.

Um bom resultado do processo, ou melhor, sua corretude de análise do terreno, ocorre quando a fonte do sinal emitido durante a modelagem é alcançada como destino ao final do processo de migração, ou seja, os dados obtidos em campo coincidem com os dados gerados no modelo teórico de bacias.

Matematicamente, a migração sísmica não é um problema que possui uma única solução para as condições de contorno providas pelo conjunto de dados medidos. Por isso, são necessárias de três a cinco iterações para se obter uma imagem migrada do subsolo [6].

Teoricamente, a modelagem e migração sísmicas são processos que estão intimamente ligados. A migração e a modelagem se comportavam como operações inversas, e a partir desse fato, vários métodos de migração utilizando este fato, sendo o mais notável, o método da migração reversa no tempo ou *RTM* [10].

2.1.2 Migração reversa no tempo

Vários métodos de migração sísmica têm sido aplicados nas mais diversas áreas geológicas. O método de Kirchhoff [57] tem sido o algoritmo mais usado na indústria do petróleo, em função do seu baixo custo computacional em relação aos outros métodos existentes. Entretanto, esse método não é satisfatório para áreas do subsolo que possuem estruturas muito complexas, com variações laterais nas velocidades de propagação da onda ou inclinações muito acentuadas nas camadas.

Em casos complexos, o método de migração reversa no tempo, que soluciona diretamente a equação de onda acústica/elástica [7], produz, em geral, resultados extremamente mais precisos. Entretanto, este método possui um custo computacional bastante elevado, tornando obrigatório o uso de estratégias de processamento paralelo. Este é o foco deste trabalho.

O método de migração reversa no tempo resolve por meio de métodos de discretização a equação da propagação de ondas, nas duas direções, da fonte sísmica para os hidrofones e dos hidrofones para a fonte sísmica. O método possui o fluxo apresentado a seguir.

Inicialmente, uma propagação de uma onda de campo de pressão é realizada através de um modelo de velocidade de propagação de onda do terreno, na direção da fonte de ondas sísmicas para os hidrofones. Essa propagação direta é realizada com dados sintéticos, ou seja, a fonte de ondas sísmicas é um modelo matemático de uma fonte real. A cada passo de tempo, dados do campo de pressão são armazenados para depois serem utilizados na condição de imagem, que tem a finalidade de formar a imagem do modelo geológico estudado em cada ponto da malha [78].

O segundo passo da migração reversa é a propagação reversa das ondas dos campos de pressão que foram recebidos no hidrofones, através do mesmo modelo de velocidades do terreno. Os dados que foram captados nos hidrofones, chamados de sismograma, são dados

reais, captados na região de interesse, diferentemente da propagação direta. A cada passo de tempo da propagação reversa, os campos de pressão da propagação direta e da reversa são correlacionados através da aplicação da condição de imagem. Ou seja, o campo de pressão final da propagação direta é correlacionado com o campo de pressão inicial da propagação reversa, e essa correlação é feita a cada passo de tempo da propagação reversa. Os resultados são somados formando a imagem de um tiro, onde esse tiro representa a fonte de ondas sísmicas. Esse processo se repete para vários tiros que são realizados no mesmo terreno. Depois as imagens de cada tiro são somadas gerando na imagem final do terreno. A Figura 3 apresenta o fluxograma explicado acima.

Figura 3 Fluxograma da migração reversa no tempo.

Este método pode ser classificado, de uma forma simplificada, como um problema de condição de contorno associado a uma condição de imagem. Neste caso, a condição de contorno é o registro do campo de pressão feito pelos hidrofones. Existem diversas condições de imagem que podem ser implementadas na migração RTM, tais como, condição de imagem com tempo de excitação, que se baseia na formação da imagem com o campo de ondas refletido quando o tempo de depropagação for igual ao tempo de trânsito, condição de imagem com correlação cruzada entre os campos de ondas incidentes e refletidos, que fornecem a imagem em cada ponto da malha, com o somatório da multiplicação entre estes campos em todos os tempos e, outras condições de imagem.

Como mencionado anteriormente, na migração RTM, utiliza-se a equação da propagação de onda para realizar a propagação direta dos campos de onda gerados na fonte de

ondas sísmicas, e para realizar a propagação reversa dos campos registrados no sismograma. Para o caso 2D, a equação homogênea é a seguinte:

$$\frac{\partial^2 P(x,z,t)}{\partial x^2} + \frac{\partial^2 P(x,z,t)}{\partial z^2} - \frac{1}{c^2}\frac{\partial^2 P(x,z,t)}{\partial t^2} = 0 \qquad (2.1)$$

Onde: $\frac{\partial^2 P(x,z,t)}{\partial x^2}$, $\frac{\partial^2 P(x,z,t)}{\partial z^2}$ e $\frac{\partial^2 P(x,z,t)}{\partial t^2}$ são, respectivamente, as derivadas de segunda ordem em relação ao eixo x, eixo z e ao tempo t

c é a velocidade de propagação da onda acústica no meio.

x e z são as coordenadas horizontal e vertical, respectivamente.

A equação de diferenças finitas faz a aproximação da equação (2.1) de onda acústica. É a resolução da equação de diferenças finitas, para cada ponto do modelo, ao longo dos passos de tempo (ou passos de processamento), que resulta na modelagem em sísmica, em duas dimensões, para o método de migração reversa no tempo (RTM). Essa equação é apresentada abaixo.

$$\mathrm{C_{i,j}} = 2 * \mathrm{B_{i,j}} - \mathrm{A_{i,j}} - \left\{\mathrm{Vel_{i,j}^2} * \mathrm{fat} * \left[16 * \left(\mathrm{B_{i,j+1}} + \mathrm{B_{i,j-1}} + \mathrm{B_{i+1,j}} + \mathrm{B_{i-1,j}}\right) - 1 * \right.\right.$$
$$\mathrm{Bi,j+2+Bi,j-2+Bi+2,j+Bi-2,j-60*Bi,j} \quad (2.2)$$

Onde $A_{i,j}$, $B_{i,j}$ e $C_{i,j}$ representam, respectivamente, o elemento localizado na posição i,j das matrizes, que representam o campo de onda, no tempo n-1, n e n+1.

$Vel_{i,j}$ representa a velocidade da propagação da onda na posição i,j da matriz que representa o modelo do terreno.

fat representa uma constante que é calculada em fator de alguns parâmetros escolhidos na execução do RTM, como o espaçamento entre os pontos da malha, a frequência de corte da onda sísmica que será propagada no modelo, etc.

A Figura 4 mostra os dados de cada uma das matrizes que são necessários para computar o ponto $C_{i,j}$ da matriz do campo de onda no tempo n+1, onde essa matriz é chamada de matriz do campo de pressão futuro, representada pela letra C, enquanto a matriz que possui

os elementos , que representa o campo de onda do tempo n-1, é chamada de matriz do campo de pressão anterior, representada pela letra A, e a matriz que possui os elementos , que representa o campo de onda do tempo n, é chamada de matriz do campo de pressão atual, representada pela letra B. Os valores dessas matrizes são expressos em número ponto flutuante padrão IEEE 754 (32 bits).

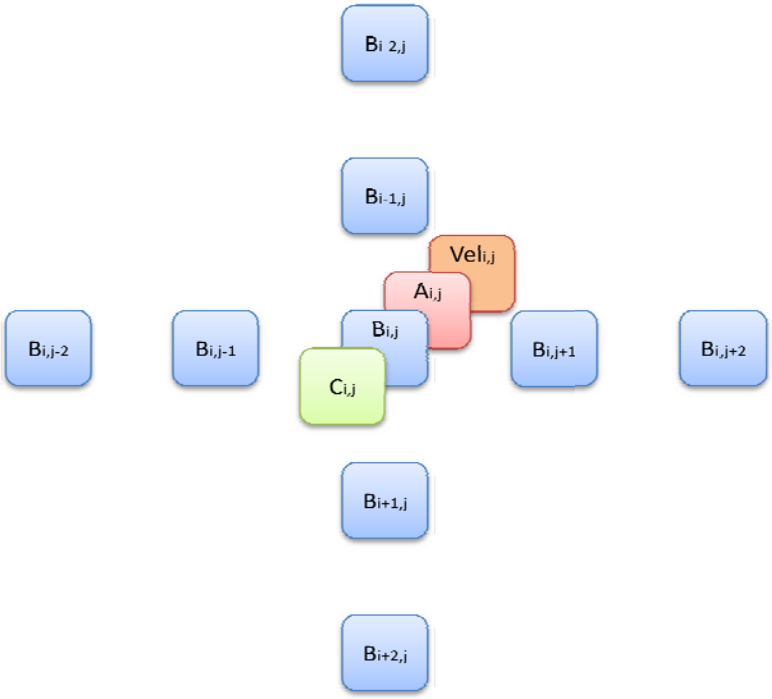


Figura 4 Dados necessários para computação da equação (2.2).

O pseudocódigo mostrado na Figura 5 mostra o funcionamento do método RTM. Pode-se ver que o método é formado por três loops, onde o mais externo representa o loop temporal, no qual os passos de tempo (ou passo de processamento) vão acontecendo, e os dois loops mais internos, que representam o deslocamento espacial da janela, mostrada na Figura 4, pelas matrizes A, B e Vel, gerando a matriz C.

loop no tempo {

$B_{i,j} = B_{i,j} + f(t)$

loop direção x{

loop direção y{

$C_{i,j} = 2 * B_{i,j} - A_{i,j}$

$-\{Vel_{i,j}^2 * fat * [16 * (B_{i,j+1} + B_{i,j-1} + B_{i+1,j} + B_{i-1,j})$

$-1 * (B_{i,j+2} + B_{i,j-2} + B_{i+2,j} + B_{i-2,j}) - 60 * B_{i,j}]\}$

}

}

$A_{i,j} = B_{i,j}$

$B_{i,j} = C_{i,j}$

}

Figura 5 Pseudocódigo do método RTM.

Nota-se que a primeira ação que ocorre no loop temporal é a inserção de um valor da onda sísmica na matriz de pressão atual. Após esse passo, a matriz de pressão futura é gerada nos loops espaciais, e então as matrizes do campo de onda são atualizadas. A matriz de pressão atual se torna a matriz de pressão anterior, e a matriz de pressão futura se torna a matriz de pressão atual. Após essa atualização um novo passo de tempo é realizado. Esse processo se repete por certo número de passos de tempo.

2.1.3 Sismograma

No processo de aquisição sísmica, as ondas ao retornarem da subsuperfície são capturadas por receptores que transformam a vibração do solo, sinal sísmico, em sinal elétrico. Esses sinais são armazenados e formarão um sismograma.

O sismograma pode ser definido como uma matriz em que as colunas são os índices dos receptores, as linhas, os índices dos instantes de tempo em que o sinal foi capturado. Em cada posição dessa matriz é armazenada a amplitude da onda capturada.

A Tabela 1 mostra um exemplo de um sismograma obtido a partir de uma fonte, 4 receptores (rec1, rec2, rec3, rec4) e 6 instantes de tempo (tt1, tt2, tt3, tt4, tt5, tt6). As amplitudes das ondas capturadas em cada receptor *recj* no instante de tempo *tti* formam o conteúdo deste sismograma.

Tabela 1 Exemplo de um sismograma contendo as amplitudes da onda lidas nos receptores em cada instante de tempo.

		Receptor			
		rec_1	rec_2	rec_3	rec_4
Tempo de Trânsito	tt_1	0.06	0	0	0
	tt_2	0.08	0.05	0	0
	tt_3	0.07	0.07	0.04	0
	tt_4	0	0.04	0.06	0.04
	tt_5	0	0	0.04	0.05
	tt_6	0	0	0	0.03

A Figura 6 mostra o sismograma da Tabela 1 representado graficamente. O eixo X corresponde ao afastamento (medida de distância entre a fonte e cada receptor). Esse eixo, neste exemplo, é iniciado com o valor zero e termina com o valor da distância entre a fonte e o último receptor, pois os receptores foram colocados apenas de um dos lados da fonte. Já, o eixo Y corresponde ao tempo das leituras dos receptores. É iniciado com um tempo de referência, neste exemplo o valor é zero, e termina com o tempo máximo de aquisição de dados.

Cada linha vertical da Figura 6 corresponde às leituras feitas por um receptor, que no sismograma pode ser relacionado a uma coluna da matriz. Esse conjunto de dados é denominado de traço sísmico.

Nesta mesma figura pode ser identificado um evento sísmico que corresponde às amplitudes das ondas lidas pelos receptores, sendo que essas ondas foram refletidas numa mesma interface, trazendo informações de uma mesma camada de rocha em tempos diferentes. Para selecionar um evento sísmico com os afastamentos, deve ser traçada uma curva aproximada a uma semi-hipérbole que se ajuste a amplitude máxima de cada traço sísmico [69].

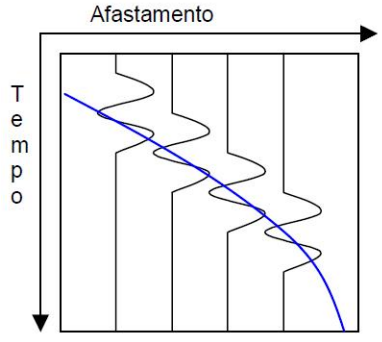


Figura 6 Representação gráfica de um sismograma composto por 4 traços sísmicos e 1evento sísmico. O evento corresponde à curva semelhante a uma semi-hipérbole representada pela linha azul.

Um passo importante no método RTM é a comparação do sismograma gerado na modelagem, que representa o sismograma sintético, com o sismograma adquirido em campo, ou seja, o sismograma que foi gerado na localidade de onde se deseja obter uma imagem do terreno. Desta forma, pode-se verificar onde as estimativas iniciais dos parâmetros do terreno estavam equivocadas e ajustá-las de forma adequada. Esse processo é repetido até que a diferença entre duas iterações consecutivas fique suficientemente pequena [7].

2.2 *FPGA* e sua utilização em sistemas *HPC*

2.2.1 FPGA

Como discutido na seção 1.1.1, existem várias arquiteturas de hardware, na computação convencional, para a execução eficiente de um determinado algoritmo. Dependendo dos recursos e do tamanho da massa dados, opções como *clusters* de *CPUs*, *GPUs* e mesmo *ASICs* têm sido sugeridos.

Mais recentemente, a computação reconfigurável vem surgindo como uma possível opção para preencher gaps entre hardware e software, alcançando desempenho mais alto que o do *software* e mantendo uma flexibilidade bem maior que a do *hardware*. A partir dos dispositivos lógicos reconfiguráveis, incluindo os *FPGAs*, esta tecnologia tem conseguido excelentes resultados em várias aplicações [58].

O *FPGA* foi criado pela Xilinx[1], e teve como sua principal característica ser um dispositivo passível de ser configurado por um *bitstream* gerado a partir de um código desenvolvido pelo usuário, ou seja, ele poderia ser programado de acordo com as aplicações dos usuários.

Estes dispositivos são formados, em geral, por um *array* bi-dimensional de elementos computacionais cuja funcionalidade é determinada por um *stream* programável de *bits*. Esses elementos computacionais, conhecidos como blocos lógicos, podem ser conectados através de matrizes de conexão, que também são programáveis. Assim, circuitos digitais grandes ou pequenos podem ser facilmente mapeados em um único dispositivo reconfigurável através da programação lógica das funções do circuito em seus blocos lógicos e matrizes de interconexão.

[1] www.xilinx.com

Devido a esta flexibilidade no desenvolvimento de novos projetos, na simplicidade das alterações lógicas, velocidade de trabalho, muitas vezes iguais àquelas a serem utilizadas no produto final, redução do consumo de potência e proteção da propriedade intelectual, os *FPGAs* tem sido amplamente utilizados no processamento de informações digitais, principalmente na prototipação rápida de circuitos digitais. A Figura 7 apresenta a estrutura interna de um FPGA convencional.

Um *FPGA* é constituído basicamente de uma matriz de blocos lógicos (*CLB, configuration logical blocks*), com fios de roteamento de sinais e matrizes de interconexão (*Switch Matrix*), localizadas entre as linhas e colunas dos *CLB's*. No perímetro do dispositivo estão localizados blocos de entrada e saída (*IOB, input output block*) usados para conectar outros componentes externos ao FPGA. Abaixo temos a Figura 7 mostrando a disposição dos blocos citados acima no *FPGA*.

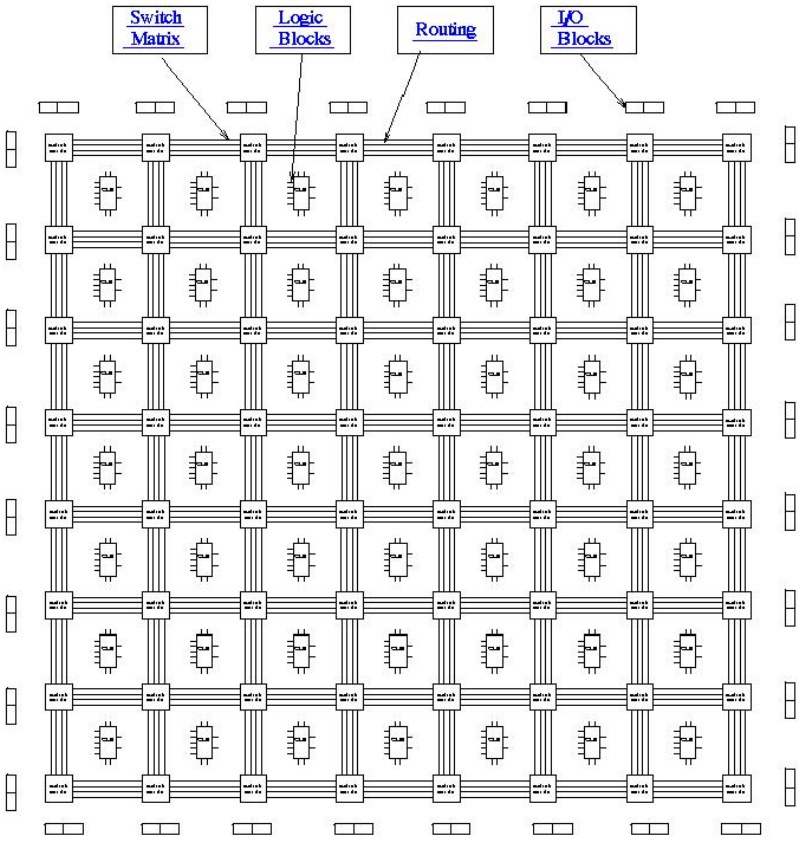


Figura 7 Estrutura do FPGA.

Os *CLB's* são, em geral, formados por um conjunto de *flip-flops (*2 a 4), e blocos lógicos do tipo *lookup-table* de 4 ou mais variáveis, formando uma lógica combinacional. Os circuitos combinacionais apresentam as saídas como função das variáveis de entrada. A Figura 8 representa a estrutura interna de um *CLB*.

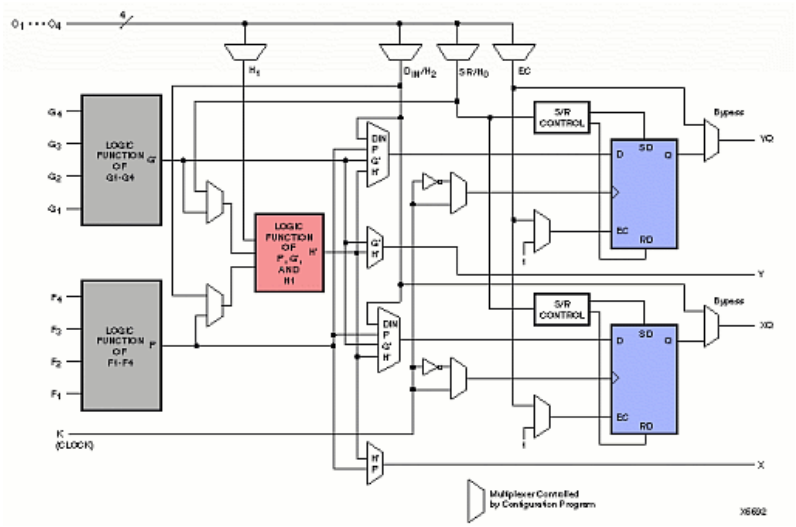


Figura 8 Estrutura do CLB.

No interior de cada bloco lógico do *FPGA*, existem vários modos possíveis para implementação de funções lógicas, em função do tipo de bloco lógico disponível para sua implementação (portas AND, OR, Mux, Look-up-tables). Um dos mais utilizados por fabricantes de *FPGA* são as *LUTs* (*Look-Up Tables*). Esse tipo de bloco lógico contém células de armazenamento que são utilizadas para implementar pequenas funções lógicas e onde cada célula é capaz de armazenar um único valor lógico: zero ou um. Cada *LUT* é essencialmente uma memória pré-programada que fornece uma saída lógica em função do vetor das variáveis de entrada. As *LUTs* possuem geralmente quatro ou cinco entradas, o que permite endereçar de 16 a 32 células. Assim, quando um circuito lógico é implementado em um *FPGA*, os blocos lógicos são programados para realizar as funções necessárias, e os canais de roteamento são configurados de forma a realizar a interconexão necessária entre estes blocos lógicos.

As células de armazenamento dos *LUTs* de um *FPGA* são voláteis por usarem tecnologia de memória *RAM*, o que implica na perda do conteúdo armazenado no caso de falta de suprimento de energia elétrica. Dessa forma, o *FPGA* deve ser programado toda vez que for energizado. Geralmente, utiliza-se uma pequena memória *FLASH EEPROM* (*Electrically Erasable Programmable Read Only Memory*), cuja função é carregar automaticamente as células de armazenamento, toda vez que o *FPGA* for energizado.

As matrizes de chaveamento (*Switch Matrix*) são matrizes que permitem as interconexões entre os *CLB's*. Em cada matriz *n x n*, poderá haver *m x n* pontos de interconexão (onde *m* é menor ou igual a *n*), os quais podem ser utilizados para na definição do roteamento de sinais pelas linhas horizontais e verticais durante a programação do circuito. É dessa forma

que são feitas às configurações das ligações dos *CLB's*, definindo assim a lógica do circuito. Abaixo, na Figura 9, vemos a estrutura de uma *Switch Matrix*.

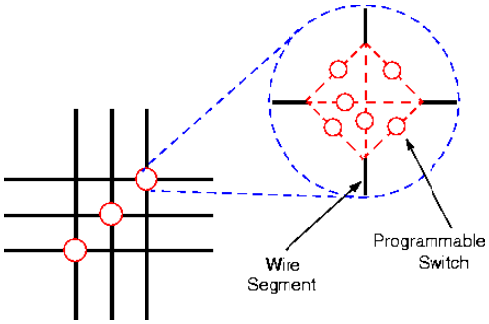


Figura 9 Estrutura de uma Switch Matrix.

Os blocos de I/O (*IOB's)*, como dito acima, são os componentes do *FPGA* responsáveis pelo interfaceamento dos sinais internos do circuito implementado no FPGA com os componentes externos ao FPGA, como memórias e interfaces. Através dos *IOB's* o *FPGA* pode trocar informações com um sistema externo. Estes blocos são basicamente compostos por *buffers*, que funcionam como um pino bidirecional (entrada e saída) do *FPGA*.

2.2.2 Computação de alto desempenho (*HPC*)

Desde sua invenção quase quatro décadas atrás, o microprocessador evoluiu em varias ordens de grandeza o seu desempenho. O desempenho destes dispositivos vem aumentando em conformidade com a de Moore [12] de forma exponencial, o que tornou estes componentes extremamente velozes e largamente utilizados em todo o mundo.

Cada novo aumento de desempenho dos processadores fazia com que novas aplicações com demandas cada vez maiores fossem criadas, fazendo com que o próximo pacote de melhorias fosse sempre esperado com grande expectativa pelos desenvolvedores de software. Em quase toda a história dos processadores, as aplicações sempre aumentavam suas complexidades para tirarem proveito das melhorias dos processadores. Assim, os processadores ditaram o desempenho máximo do *software*. Porém, aplicações de alto desempenho computacional, como em computação cientifica, estão demandando uma capacidade de processamento que um único processador não pode suportar de forma adequada, criando assim um *gap* tecnológico entre a demanda das aplicações e o desempenho desses processadores, como é mostrado na Figura 10.

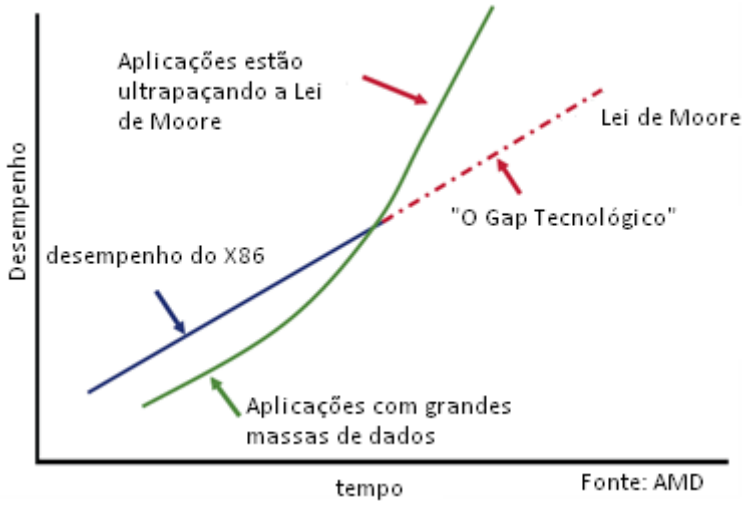


Figura 10 Gap Tecnológico.

O termo *HPC* foi originalmente usado para descrever os primeiros supercomputadores que surgiram para suprir a demanda por um maior poder de computação. Porém, com o passar dos anos, o termo *HPC* evoluiu para englobar um conceito maior. Agora o termo *HPC* inclui também sistemas com uma capacidade de computação maior, sistemas com uma grande taxa de transferência de dados, e sistemas com a habilidade de lidar com computação distribuída.

As arquiteturas de sistemas *HPC* também evoluiram com o passar dos anos. Originalmente os sistemas *HPC* eram supercomputadores que utilizavam arquiteturas com *Massively Parallel Processing* (*MPP*) [59] e *Symmetric Multiprocessing* (*SMP*) [60] [61]. Tais sistemas requeriam processadores específicos, caminhos de dados proprietários, gerando assim sistemas excessivamente caros.

Atualmente, essas arquiteturas vêm perdendo popularidade e os sistemas que vêm sendo adotados são os chamados *clusters computing*, que são sistemas bem mais baratos e que possuem uma capacidade de processamento tão boa quanto os *mainframes* e supercomputadores. Sistemas *cluster computing* estão entre os 10 sistemas mais rápidos do mundo, segundo o top500[2]. A Figura 11 [13] mostra a evolução da arquitetura dos sistemas *HPC*.

[2] http://www.top500.org

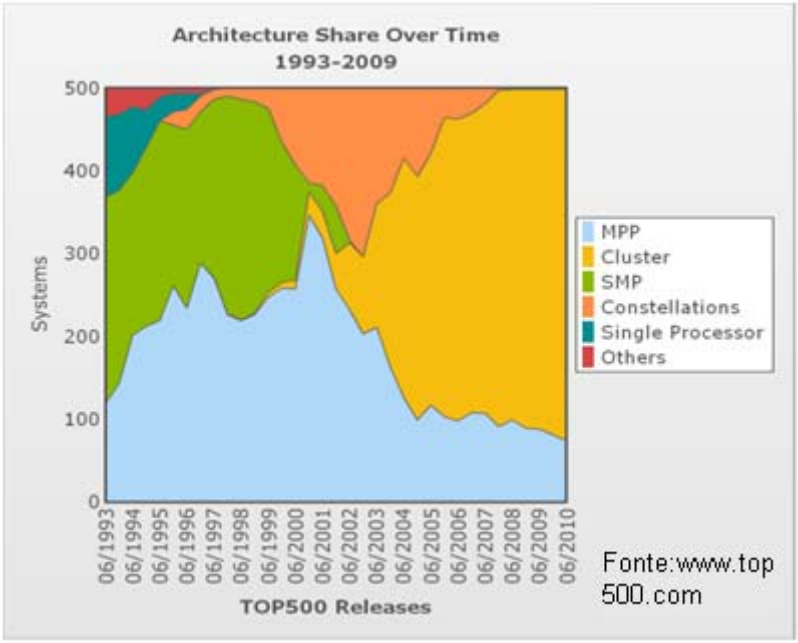


Figura 11 Evolução da arquitetura dos sistemas HPC.

Sistemas baseados em *clusters* são formados por vários computadores ligados entre si. Por conexões em rede e que trabalham juntos para resolver uma determinada tarefa. Devido as suas interconexões, esses computadores podem ser visto como um supercomputador. Cada um dos computadores desses sistemas são, geralmente, construídos com *hardware* utilizado pela maioria dos consumidores comuns, como plataforma baseadas nos processadores *AMD*[3] ou *Intel*[4], com a interconexão entre os vários computadores (nós) com uma grande largura de banda e baixa latência, como as conexões *Infiniband*[5] e *MyriNet*[6]. O sistema operacional de tais plataformas é geralmente o *Linux*[7] devido à grande variedade de software *open-sou*rce disponíveis, diminuindo o custo dos mesmos.

Devido às vantagens associadas ao baixo preço, uso de máquinas (*PCs*) comerciais, sistemas operacionais *open-source*, criação de ambientes de desenvolvimento de códigos, entre outras características, é esperado que a utilização de *clusters*, principalmente deste tipo, continue crescendo. É esperado que não apenas sistemas do tipo *clusters* continuem crescendo, mas todos os sistemas *HPC*, já que aplicações que necessitam de tais sistemas vêm aumentando ao longo dos anos.

[3] http://www.amd.com/br/Pages/AMDHomePage.aspx
[4] http://www.intel.com/?pt_BR_01
[5] www.infinibandta.org/
[6] www.myri.com/myrinet/overview/
[7] http://www.linux.org/

Aplicações de *HPC* estão distribuídas em varias áreas, como mostra a Figura 12. Elas vão desde o modelamento de elementos finitos, bastante utilizado na indústria automobilística, na área de testes, passando por dinâmica de fluidos, processamento de imagem, aplicações em sísmicas, modelamento molecular, aplicações financeiras, etc. Todas essas aplicações têm um conjunto de características em comum, todas representam problemas de alta complexidade, possuem um grande conjunto de dados e levam muito tempo para serem computadas.

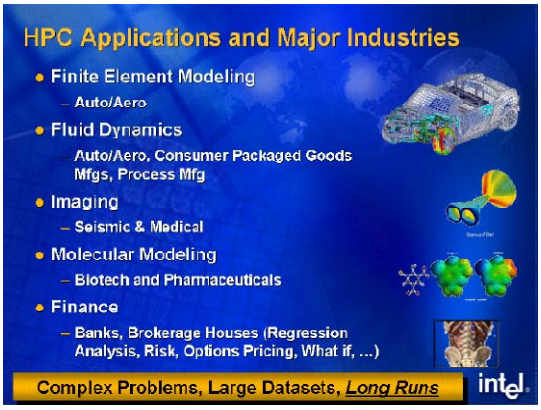


Figura 12 Aplicações que requisitam sistemas HPC.

2.2.3 Utilização de *FPGAs* em sistemas *HPC*

Aplicações que necessitam de um alto poder de computação e de uma grande quantidade de dados vêm ganhando grande enfoque nos últimos anos com o rápido aumento de desempenho dos computadores de propósito geral e com o surgimento de sistemas de computação paralela mais eficientes. Apesar da frequência nominal das *CPUs* de propósito geral ser bem elevada, a utilização media da velocidade computacional é bem reduzida quando comparada com o desempenho de pico obtido pelas *CPU's*. A principal razão dessa ineficiência é a grande quantidade de transistores nessas *CPUs* utilizados na implementação de sua lógica de controle ou utilizados na provisão de um fluxo de dados flexível. Este tipo de arquitetura não é adequada na execução de vários problemas científicos ou de engenharia que possuem seus domínios computacionais em regiões geométricas 2D e 3D, tornando seu tempo de resolução impraticável [7].

Problemas em *HPC* podem envolver diferentes tipos de estruturas computacionais e precisões numéricas. Em computação científica, por exemplo, as operações podem ser realizadas com números no formato de ponto flutuante, com dupla ou simples precisão,

adotando padrões como o IEEE 754 [27]. O desempenho, neste caso, pode ser medido em unidades de operações de ponto flutuante por segundo, como Megaflops ou Gigaflops.

Aplicações que requerem sistemas *HPC* geralmente usam de grandes quantidades de dados e necessitam de grande poder computacional. Essas aplicações geralmente possuem uma relação entre o numero de operações, em ponto flutuante, e o numero de acesso à memória relativamente baixa, requerendo usualmente um espaço de memória considerável para armazenar dados intermediários. Este comportamento tende a gerar padrões de endereçamento indiretos e irregulares, resultando num baixo desempenho no uso de estruturas de *cache* em computadores de propósito geral.

Existe, portanto, uma grande diferença entre o desempenho teórico de pico e a real velocidade de execução de um programa em uma *CPU* de propósito geral.

Implementações puramente em *software*, como por exemplo, um *cluster* de PC com alto grau de paralelismo [14], ou uma implementação com otimizações de utilização de memória ou disco [15], ou uma implementação com reordenação de instruções [16], buscam acelerar a execução de aplicações *HPC*. Entretanto, poucas são as aplicações, como multiplicações de matrizes densas ou a Transformada Rápida de Fourier, que conseguem alcançar de 80 a 90 por cento do desempenho de uma *CPU*. A maior parte das aplicações alcançam índices em torno de 20 a 30 por cento, com algumas aplicações realísticas chegando a 10 ou 5 por cento de desempenho [17].

Uma outra abordagem para computação de alto desempenho em aplicações especificas é a utilização de sistemas customizados. Neste caso, cada dispositivo, é construído para a execução de algoritmos ou aplicações especificas, obtendo assim um melhor desempenho, e menor requisito de energia, quando comparado com *CPUs* convencionais. Esses dispositivos são chamados de circuitos integrados de aplicações especificas (*Application-Specific Integrated Circuit – ASIC*). Do ponto de vista tecnológico, o desenvolvimento de um *ASIC* não é um problema, porém, na prática, tal desenvolvimento encontra várias barreiras, como por exemplo, o custo NRE [77]. Um exemplo de sucesso de um sistema que utiliza um *ASIC* é o GRAPE, projeto gerenciado por um grupo de cientistas da computação e astrofísicos na Universidade de Tokyo [18] [19].

Quando a demanda por desempenho nas aplicações começou a aumentar, uma das alternativas estudadas foi a utilização de coprocessadores junto aos processadores para a realização de certas tarefas especificas. A utilização de tais coprocessadores começou com alguns processamentos especializados de E/S, DMA e coprocessadores aritméticos, estendendo-se para o uso de aceleradores gráficos e sistema de criptografia (*encryption engines*). No auxilio a computação científica e processamento de sinais logo sugiram coprocessadores aritméticos, como as *Floating-point Units (FPUs*), para lidar com operações diversas em ponto flutuante. Unidades especiais para lidar com processamento digital de sinais, os *Digital Signal Processors* (*DSP*)s também foram lançados como coprocessadores, desenvolvidos para lidar com algoritmos matemáticos mais complexos

Observamos, no entanto, que soluções como aquelas apresentadas no sistema GRAPE, ou o uso de coprocessadores como aqueles descritos acima, não são resposta para o *gap* computacional ora existente, já que elas apenas resolvem parte do problema. Atualmente, esse *gap* tecnológico requer um conjunto de atributos mais diversificado do que os atributos que os coprocessadores ou *ASICs* provêm hoje. Além disso, sistemas atuais precisam evitar implementações baseadas em software, como a dos *DSPs,* pois essas implementações sofrem as mesmas limitações dos processadores convencionais.

Uma opção para este novo paradigma computacional seria uso de plataformas baseados em componentes de hardware, provendo três habilidades fundamentais, quais sejam: a plataforma deve prover uma aceleração de *hardware* específica (*hardware specific acceleration*) para alguns processos chaves na aplicação; o coprocessador ou a plataforma precisa oferecer a oportunidade de uso de *pipeline* e estruturas paralelas, para que eles possam continuar oferecendo um bom desempenho, mesmo se a demanda das aplicações aumentar; e o coprocessador ou a plataforma precisa prover uma grande largura de banda (*high-bandwidth*) e uma baixa latência nas interfaces que se ligam com o processador principal e com a memória.

Um dos dispositivos que atende bem a todas essas necessidades e é uma alternativa para que não se tenha que projetar um coprocessador ou uma plataforma para cada uma das varias aplicações possíveis, é o *FPGA*. Este dispositivo pode se configurado de acordo com as necessidades da aplicação do usuário. Assim, para cada aplicação nova, necessita-se apenas de uma nova reconfiguração do dispositivo. Atualmente os *FPGAs* oferecem um desempenho excelente em aplicações onde o processamento é intensivamente paralelo, *loops* e problemas

típicos de *dataflow*. Eles podem suportar estruturas de *pipeline* com profundidades variáveis e provém um alto potencial em computação paralela, permitindo que funções altamente complexas sejam executadas em até um ciclo de relógio. Sua característica de reprogramação garante que ele possa ser direcionado para atingir as especificações requeridas por uma aplicação sem ter o custo ou o gasto excessivo de tempo no desenvolvimento de um coprocessador convencional ou *ASIC*. Caso o *FPGA* suporte reconfiguração dinâmica, ele pode ainda prover um coprocessamento altamente customizado (customização *on-the-fly*) para uma grande variedade de aplicações em um único *chip* [70].

Aplicações de *HPC* começaram a utilizar dispositivos *FPGAs* a partir da década de 90. A Tabela 2 lista alguns institutos/companhias que projetaram e desenvolveram supercomputadores de alto desempenho baseados em *FPGAs* para aplicações especificas, com o objetivo de substituir os computadores de propósito geral.

Tabela 2 Supercomputadores reconfiguráveis baseados em FPGA.

Systems	Manufacturer	Year	Number of FPGAs	Applications	Hardware Costs
BEE-2 [20]	UC Berkeley	2005	5 per Blade 200 per Rack	Radio astronomy	$ 500K
Starbridge [21]	NASA	2005	11 per system	Aeronautics & astronautics	$ 150K

De fato, uma maneira interessante de se tirar proveito de arquiteturas concorrentes é utilizar-se do que há de melhor em cada uma delas, desenvolvendo plataformas híbridas. Por exemplo, adotando um modelo *hardware/software co-design*, tirar vantagens do grande poder computacional de dispositivos como os *FPGAs*, associados a componentes como *CPUs* convencionais e GPUs. Essa arquitetura é interessante, pois ela suprime a ineficiência que os FPGA possuem na realização de tarefas, como operações de ramificação, condição e *loops* com tamanhos variáveis, etc., onde componentes de *software* são mais eficientes. Por outro lado, ela permite que operações aritméticas complexas possam ser resolvidas com alto desempenho em coprocessadores baseados em *FPGAs*. Assim, nesta versão, a execução de componentes típicos de *software* e de *hardware* pode ser acelerada através de *CPUs* e *FPGAs* respectivamente, com o melhor dos dois tipos de tecnologia.

A interface de conexão dos componentes de *hardware e software,* em geral, é realizada através de três modelos: através de uma interface padrão como o PCI ou VME [22] [23]; como

um coprocessador acoplado diretamente a *CPU* [24], ou como um nó de processamento heterogêneo conectado ao sistema e barramento de memória do computador [25].

De uma maneira geral, quanto mais próximo o *FPGA* estiver da *CPU* e quanto mais poderosa for a interface entre eles, mais aceleração/desempenho pode ser conseguida, devido à baixa latência na transferência de dados, maior largura de banda e menor *overhead* de controle. A Figura 13 mostra essas três formas de se acoplar um FPGA a uma CPU.

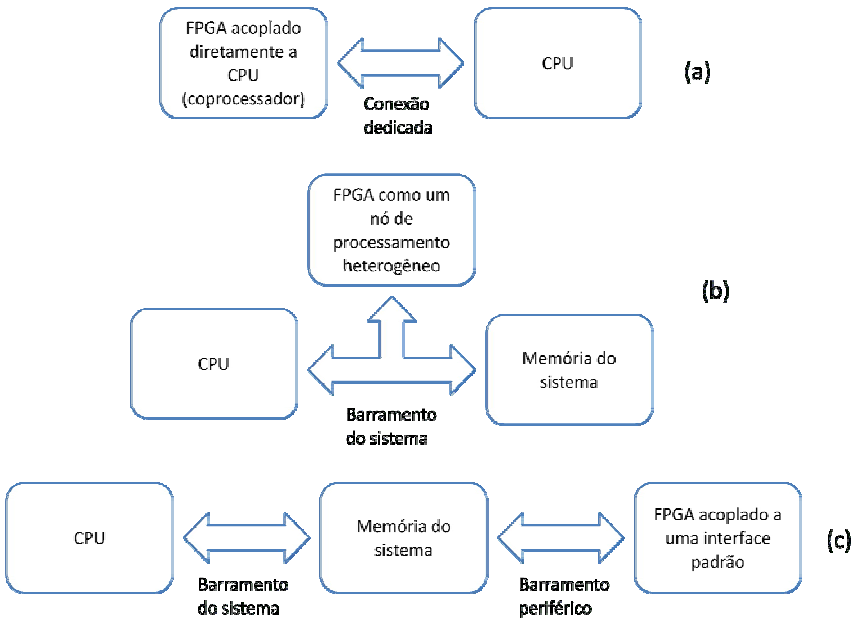


Figura 13 Formas de acoplar o FPGA com uma CPU.

2.3 Conclusões

Neste capítulo foram apresentadas as principais teorias estudadas para o desenvolvimento da plataforma proposta nesta dissertação.

Inicialmente foi apresentada a sísmica de reflexão que é a técnica de exploração geofísica mais utilizada pelos métodos sísmicos que são utilizados pelas empresas. A partir dessa explicação foram citados alguns dos principais métodos sísmicos utilizados atualmente, como o método Kirchhoff [6] e o Reverse time Migration - RTM [1], sendo esse último o método sísmico utilizado na plataforma proposta neste trabalho.

Uma introdução aos *Field programmable gate arrays (FPGAs)* [76] também foi realizada neste capitulo. Nessa seção foi apresentada a arquitetura básica do FPGA, mostrando os seus principais componentes internos.

Uma seção de computação de alto desempenho (*HPC*) mostrou as principais características dos sistemas *HPC*.

Finalmente, uma seção apresenta a utilização de *FPGAs* em sistemas *HPCs*. Nesta seção são apresentados os problemas enfrentados pelos sistemas baseados em *CPUs*, como por exemplo, a grande quantidade de dados, o acesso irregular a estes dados na memória e o baixo desempenho médio. O *FPGA* é sugerido como uma possível alternativa para resolver os problemas inerentes as *CPU's* convencionais.

Capítulo

3

3 Trabalhos Relacionados

Este capítulo tem como objetivo apresentar trabalhos relacionados à utilização de *FPGA*s em métodos de modelagem em sísmica e em outras aplicações ligadas a computação cientifica que utilizam o algoritmo de diferenças finitas.

3.1 Optimized high-order finite difference wave equations modeling on reconfigurable computing platform. Chuan He, Guan Qin, Mi Lu, Wei Zhao, 2007

No trabalho apresentado por Chuan He, Guan Qin, Mi Lu e Wei Zhao [26] propõe-se uma solução baseada em computação reconfigurável para acelerar a execução de problemas de modelagem em sísmica acústica ou elástica. O trabalho aprensentado no artigo pode ser divididos em duas partes: a primeira corresponde as otimizações que foram implementadas no método de diferenças finitas, e a segunda diz respeito a arquitetura desenvolvida para o FPGA. A seguir será mostrada as otimizações implementadas no método.

A equação de onda acústica mais simples no espaço 2D é mostrada abaixo:

$$\frac{\partial^2 P(x,z,t)}{\partial t^2} - v^2(x,z)\Delta P(x,z,t) = f(x,z,t), \quad (3.1)$$

Onde P é a pressão, v é a velocidade acústica, e $\Delta \equiv \frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial z^2}$ representa o operador Laplaciano.

A equação (3.1) é usualmente discretizada de forma que os operadores diferenciais espaciais de segunda ordem são aproximados pelo estêncil central do método de diferenças finitas de segunda ordem como mostrado a seguir:

$$P_{i,k}^{n+1} = 2 \cdot P_{i,k}^{n} - P_{i,k}^{n-1} + (dt)^2 \cdot v_{i,k}^2 \cdot \Delta^{(2)} P_{i,k}^{n} + f_{i,k}^{n} + O(dt^2), \quad (3.2)$$

onde

$$\Delta^{(2)} P_{i,k}^{n} = \left(\frac{P_{i+1,k}^{n} - 2 \cdot P_{i,k}^{n} + P_{i-1,k}^{n}}{(dx)^2}\right) + \left(\frac{P_{i,k+1}^{n} - 2 \cdot P_{i,k}^{n} + P_{i,k-1}^{n}}{(dz)^2}\right) + O(dx^2) + O(dz^2), \quad (3.3)$$

representa a aproximação de segunda ordem do operador Laplaciano.

Pela expansão de Taylor, o operador Laplaciano pode ser aproximado sistematicamente para a (2m)-ésima ordem por um estêncil central com (2m+1) pontos ao longo de cada eixo como mostrado a seguir:

$$\left.\frac{\partial^2 P}{\partial x^2}\right|_{i,k}^{n} = \left(\frac{\alpha_0^m \cdot P_{i,k}^{n} + \sum_{r=1}^{m} \alpha_r^m \cdot (P_{i+r,k}^{n} + P_{i-r,k}^{n})}{(dx)^2}\right) + O((dx)^{2m}), \quad (3.4)$$

onde

$$\alpha_0^m = -2 \cdot \sum_{r=1}^{m} (-1)^{r-1} \frac{2m!^2}{r!(m-r)!(m+r)!} \tag{3.5}$$

e

$$\alpha_r^m = (-1)^{r-1} \frac{2m!^2}{r!(m-r)!(m+r)!} \tag{3.6}$$

são escolhidos para maximizar a ordem dos termos não cancelados da expansão de Taylor.

A Figura 14 [26] mostra o estêncil 2D dos esquemas de diferenças finitas com aproximações de segunda (2,2) e quarta ordem (2,4). Todos os pontos do *grid* que estão envolvidos no cálculo estão marcados com subescritos na figura.

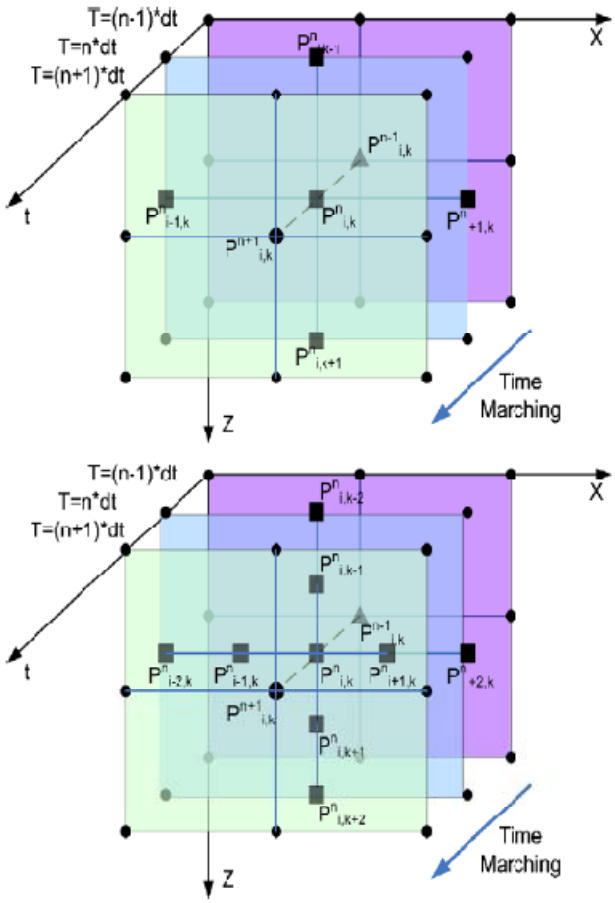


Figura 14 Estênceis 2D para os esquema de diferenças finitas (2,2), em cima, e (2,4), em baixo.

Apesar da computação da equação (3.4) ser bem mais complexa que a da equação (3.3), esquemas de diferenças finitas com ordem mais alta possuem termos truncados de alta ordem que não são cancelados, o que proporciona erros de aproximações menores.

Devido às aproximações de mais alta ordem possuírem um erro menor para modelos de grande tamanho e com um grande número de passos de processamento, os autores do artigo procuraram utilizar tais aproximações, já que, segundo os mesmos, plataformas de computação reconfiguráveis baseadas em *FPGA* não sofreriam com o número de computações que são necessárias para realizar essas aproximações, pois várias unidades aritméticas de ponto flutuante poderiam ser integradas em um único FPGA.

A Figura 15 [26] mostra o diagrama de blocos e o fluxo de dados do módulo de computação que faz a aproximação de quarta ordem do operador Laplaciano.

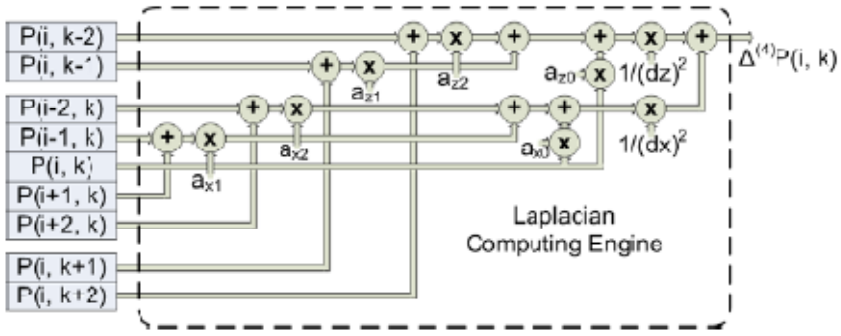


Figura 15 Operador Laplaciano para o esquema de diferenças finitas de quarta ordem espacial (2,4)

Para implementar a equação (3.4) em sua plataforma de computação reconfigurável os autores aperfeiçoaram o algoritmo numérico para não utilizar o padrão IEEE-754 [27] de ponto flutuante, reduzindo, assim, os recursos de *hardware* necessários e o número de estágios de *pipeline*. Os autores observaram, examinando a **Erro! Fonte de referência não encontrada.**, que os esquemas de diferenças finitas de alta ordem possuem uma precisão excessiva para ondas acústicas que possuem um baixo número de onda. Essa precisão vai piorando de forma bastante rápida para ondas com um alto número de onda. O número de onda representa a quantidade de ondas por unidade de distância, ou seja, o número de vezes que uma onda atinge a mesma fase em uma determinada distância de propagação. Para compensar essa desvantagem, os autores desenvolveram um método que requer a menor ordem de precisão nominal de forma a manter seu esquema de diferenças finitas consistente numericamente com a equação original.

Todos os outros coeficientes na equação (3.4) foram escolhidos visando à minimização do quadrado do erro da velocidade de fase da equação de onda discretizada, para as ondas acústicas que possuírem o número de onda que será utilizado no modelo. Este ajuste leva a um

critério de erro denominado *Least Square* no domínio da frequência [62]. Além deste critério, os autores desenvolveram uma outra otimização, o de utilizar um número menor de bits para representar esses coeficientes escolhidos.

Estes ajustes levaram a elaboração de um algoritmo de otimizações de coeficientes com precisão finita para o método de diferenças finitas. Esse algoritmo é uma abordagem heurística simples para atingir um resultado subótimo. Nele, os dois maiores coeficientes no meio do estêncil do método de diferenças finitas são representados em um formato de ponto fixo com dezoito bits de precisão, já que seus valores são essenciais para o desempenho da dispersão numérica. Os outros coeficientes são representados no formato de ponto flutuante com apenas seis bits na mantissa, fazendo com que as multiplicações levem apenas um ciclo de relógio para serem realizadas pelos multiplicadores encontrados no *FPGA*.

Este esquema otimizado de diferenças finitas, além de utilizar menos recursos de *hardware* e possuir uma menor latência de *pipeline*, possui outra melhoria: os recursos de *hardware* necessários para as multiplicações dos coeficientes diminuíram pela metade, o que possibilitou os autores modificarem a ordem das operações de adição e multiplicação, como mostra a Figura 16.

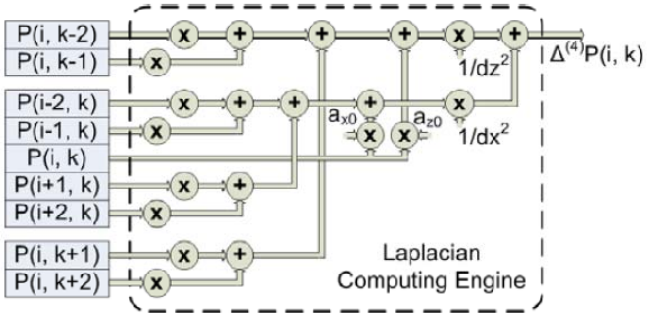


Figura 16 Operador Laplaciano de quarta ordem para o esquema otimizado.

Comparando as Figura 15 e Figura 16, nota-se, na primeira, que um grande número de recursos de interconexão global são utilizados para rotear os operadores dos adicionadores do primeiro nível, o que leva a um projeto com velocidade mais baixa, enquanto que, na segunda, os caminhos de dados estão localizados, possibilitando a utilização de um relógio mais elevado no núcleo de computação, melhorando, assim, seu desempenho.

Para mostrar a efetividade do seu algoritmo, os autores, desenvolveram um experimento simples. Foi escolhido um estêncil de nove pontos que, de acordo com a equação

(3.4), cinco coeficientes desconhecidos devem ser calculados. O número de onda normalizado das ondas acústicas foi restringido para menos de 0.5. A Tabela 3 mostra os valores desses coeficientes para o esquema de máxima ordem, o esquema otimizado com precisão completa, ou seja, são utilizados os números em ponto flutuante precisão simples, e o esquema otimizado com precisão finita, onde os números tem precisão fixa, possuindo um numero de bits que está de acordo com o algoritmo desenvolvido. A Figura 17 [26] mostra a relação de dispersão para esses três esquemas e para a equação de onda ideal. A Figura 18 [26] apresenta os erros numéricos causados pelas aproximações dos métodos de diferenças finitas.

Tabela 3 Coeficientes para os diferentes esquemas de diferenças finitas para o estêncil com nove pontos.

Coefficients	FD schemes		
	Maximum eighth-order	Optimal eighth-order	Finite-accuracy eighth-order (Hex.)
α_0^4	2.8472223	2.9555101	2.9552002(403d2200)
α_1^4	3.2000000	3.3781638	−3.3778076(c0582e00)
α_2^4	0.4000000	0.4969828	0.4970703(3efe8000)
α_3^4	0.0507937	0.0828245	−0.0830078(bdaa0000)
α_4^4	0.0035714	0.0084953	0.0085449(3c0c0000)

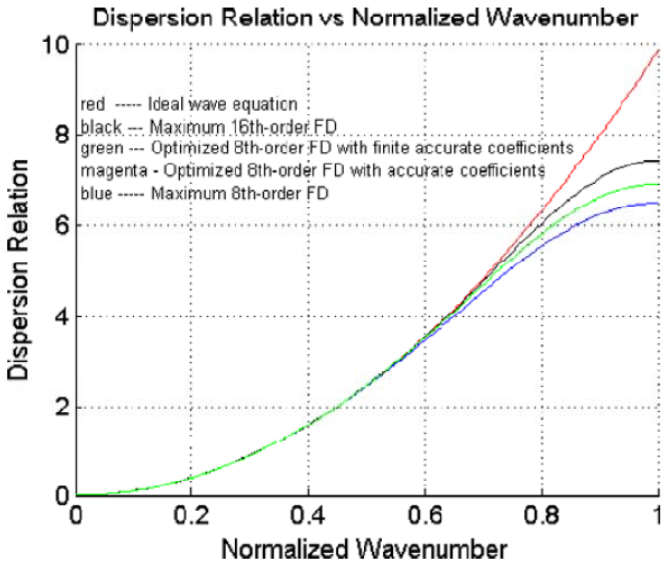


Figura 17 Comparação entre as relações de dispersão entre as diferentes aproximações do método de diferenças finitas.

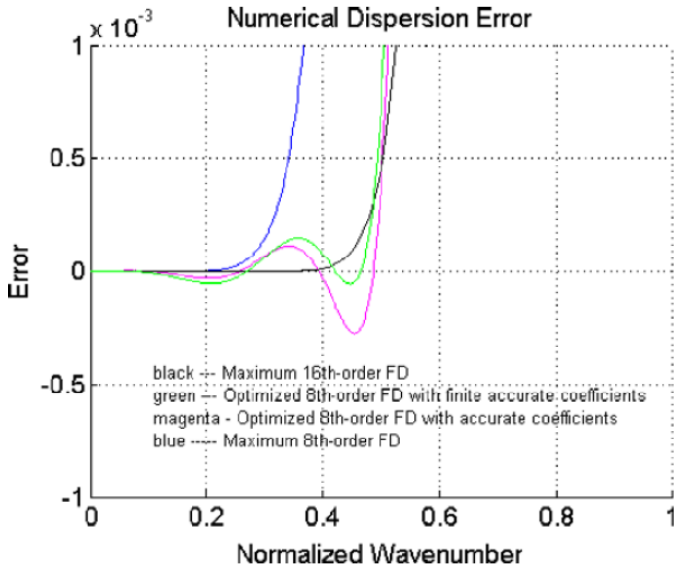


Figura 18 Erros de dispersão causados pelas aproximações do método de diferenças finitas comparados com a equação de onda original.

Pode-se observar pelos resultados que o esquema de diferenças finitas otimizado possibilita a utilização de uma banda de trabalho mais larga que os esquemas de máxima ordem, com o custo de negligenciar a aproximação do erro, devido o erro apresentado nesses esquemas possuir altos valores quando o numero de onda normalizado se aproxima de 0,5. O esquema de precisão finita otimizado também possui um desempenho bastante similar ao esquema de precisão completa otimizado, porém com uma representação dos coeficientes bem mais simples.

A arquitetura desenvolvida pelos autores teve como um dos objetivos aliviar o gargalo que existe na troca de dados entre o *FPGA* e a memória externa. Eles desenvolveram um sistema de bufferização de dados baseado em uma janela que se desloca pela matriz de entrada. Esse sistema, que utiliza os blocos de *RAM* localizados no *FPGA*, fica entre o núcleo de computação e a memória externa da placa.

Este sistema foi implementado a partir do método de diferenças finitas de quarta ordem espacial (2,4). As equações para esse esquema, não levando em conta o termo da fonte sísmica, são:

$$\Delta^4 P_{i,k}^n = \left(\frac{\partial^2 P}{\partial x^2} + \frac{\partial^2 P}{\partial z^2}\right)\Bigg|_{i,k}^n$$

$$= \frac{-\frac{1}{12}\cdot P_{i+2,k}^n + \frac{4}{3}\cdot P_{i+1,k}^n - \frac{5}{2}\cdot P_{i,k}^n + \frac{4}{3}\cdot P_{i-1,k}^n - \frac{1}{12}\cdot P_{i-2,k}^n}{(dx)^2}$$

$$+ \frac{-\frac{1}{12}\cdot P_{i,k+2}^n + \frac{4}{3}\cdot P_{i,k+1}^n - \frac{5}{2}\cdot P_{i,k}^n + \frac{4}{3}\cdot P_{i,k-1}^n - \frac{1}{12}\cdot P_{i,k-2}^n}{(dz)^2}$$

(3.7)

$$P_{i,k}^n = 2\cdot P_{i,k}^n - P_{i,k}^{n-1} + (dt)^2 \cdot v_{i,k}^2 \cdot \Delta^{(4)} P_{i,k}^n, \quad (3.8)$$

Suponha uma matriz com I linhas e J colunas no espaço 2D do problema. Como poucas otimizações, relacionadas ao acesso à memória e a computação, podem ser realizadas na equação (3.8), os autores mostraram a ideia de seu sistema computando a equação (3.7) ponto a ponta a cada linha da matriz. Essa ideia pode ser imaginada como uma janela que se desloca lateralmente pela matriz como mostra a Figura 19.

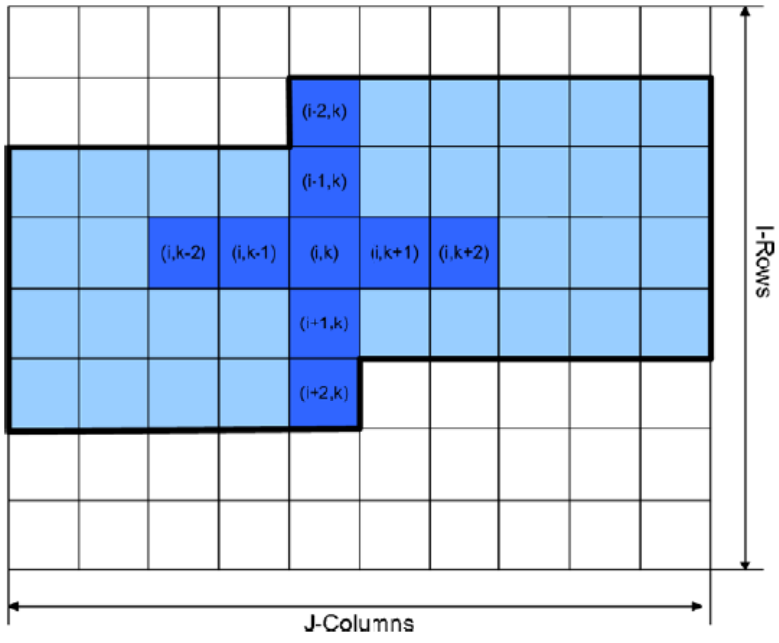


Figura 19 Janela se deslocando no espaço 2D.

Esse sistema *bufferiza* (4*J+1) pontos contínuos da matriz, que são os pontos em cinza na matriz mostrada na Figura 19, além de possuir nove pontos ativos dentro da janela, os pontos em cinza mais escuro na Figura 19. Estes últimos são os valores passados,

simultaneamente, para o núcleo de computação através de caminhos de dados internos. Conectando a porta de entrada dessa janela com a memória externa, apenas uma leitura externa é necessária para mover essa janela um ponto para a direita na matriz.

A Figura 20 mostra o diagrama de blocos e o caminho de dados da estrutura de *bufferização* e janela de deslocamento dentro do *FPGA*. Esta arquitetura utiliza quatro *FIFOs* em cascata para realizar essa *bufferização*, onde cada uma dessas *FIFOs* possui a capacidade de armazenar uma linha inteira de matriz. Os pontos da matriz de entrada são enviados para o *FPGA* através de uma porta que está ligada a entrada da primeira *FIFO*. Os pontos são descartados da estrutura na saída da última FIFO. É necessário atrasar a computação do ponto $P_{i,k}^{n+1}$ até que o ponto $P_{i+2,k}^{n}$ entre na estrutura de bufferização, fazendo com que todos os operandos necessários para a computação do ponto $P_{i,k}^{n+1}$ estejam disponíveis para serem enviados para o núcleo de computação. Circuitos auxiliares como *cache* entre os módulos de memória da placa e o engenho computacional, foram também desenvolvidos para tornar transparente o acesso irregular dos dados e o comportamento periódico de *refreshing* que acontece nas memórias SDRAM. Analisando seu comportamento, observa-se que o sistema necessita de leituras do valor $P_{i,k}^{n-1}$, do parâmetro $v_{i,k}$ e do valor $P_{i+2,k}^{n}$ e da escrita do resultado da computação, o ponto $P_{i,k}^{n+1}$, para computar um ponto da matriz em um passo de processamento. Esse procedimento requer quatro acessos a memória, gerando uma relação de operações em ponto flutuante por acesso a memória de 5.25 para o sistema utilizando o método de diferenças finitas de quarta ordem espacial (2,4).

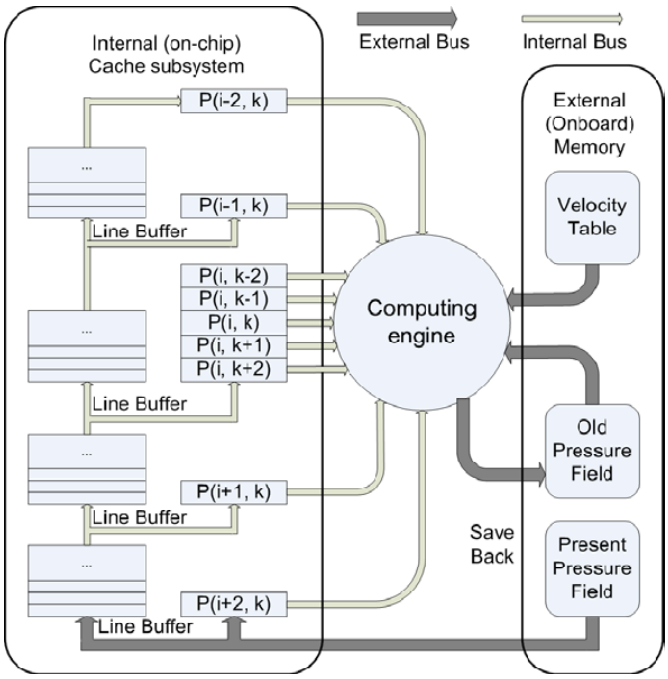


Figura 20 Diagrama de blocos do sistema de bufferização para o esquema (2,4) no espaço 2D.

O interessante dessa arquitetura é que diferentemente da solução em *software* onde os requisitos de memória crescem linearmente com a ordem do problema, na arquitetura dos autores a quantidade de acessos se mantém constante em quatro acessos, para computar um ponto da matriz em um passo de processamento. O custo adicional nessa arquitetura é a quantidade de *FIFOs* que são necessárias para a estrutura de *bufferização*, que utiliza blocos de *RAM* do *FPGA*, e o número adicional de unidades aritméticas de soma ou multiplicação.

A plataforma utilizada no trabalho dos autores é a placa de avaliação *Xilinx ML401 Virtex-4* [28]. Essa placa possui um *Virtex-4 XC4LX25* [29] que contém 24,192 células lógicas, 48 slices de *DSP* e 72 blocos de *SRAM* com 18-KB cada. A placa também possui módulos *DDR-SDRAM* com 64MB e 32 bits de interface com o *FPGA*, e uma *ZBT-SRAM* com 9MB e 32 bits de interface com *FPGA*. Os autores utilizaram uma plataforma *Intel* P4 3.0 GHz com 1GB de memória para comparação com a plataforma proposta. O código, desenvolvido em C, foi compilado no sistema operacional Linux utilizando o Intel C++ v8.1 com otimização para velocidade(-O3 –tpp7 e -xK).

O exemplo utilizado para comparação de desempenho entre a plataforma desenvolvida pelos autores e uma *CPU* foi uma modelagem sísmica-2D em um meio com velocidade constante, uma matriz de entrada com 1000 linhas e 1000 colunas e 6000 passos de processamento. Eles utilizaram esse exemplo e variaram a ordem dos esquemas de diferenças

finitas para avaliar o aumento de velocidade no tempo de computação (*Speedup*) da sua implementação em *hardware*.

Esse exemplo possui um milhão de pontos, sendo necessárias quatro milhões de palavras da memória *DDR-SDRAM* para armazenar todos os dados. Esses dados são organizados como quatro volumes de dados, um para a matriz de pressão anterior, um para a matriz de pressão atual, um para a matriz de pressão futura e um para a matriz de velocidades. A frequência de operação da memória *DDR-SDRAM* é de 100MHz e o do núcleo de computação 50MHz.

Os resultados da simulação, para diferentes ordens de esquema de diferenças finitas, podem ser visto na Tabela 4 [26]. Segundo os autores, os resultados mostrados são satisfatórios, já que a placa utilizada possui poucos recursos. O mais interessante desses resultados é que o desempenho da plataforma desenvolvida no artigo continua crescendo à medida que a ordem do esquema de diferenças finitas vai aumentando, já que a vazão de resultados se mantém quase que constante para a plataforma, enquanto que no PC a vazão vai decaindo.

Tabela 4 Comparação de desempenho.

FD schemes	Software computational throughput (Million Grid/s)	Hardware implementations		
		Computational throughput (Grid/s)	Speedup	Resource utilizations (RAM blocks/ DSP slices/ Logic slices)
(2, 2)	33.27	49.71	1.49	16/10/4885
(2, 4)	27.53	49.63	1.80	20/22/7212
(2, 8)	19.90	49.50	2.49	28/30/9732
(4, 4)	15.10	48.38	3.20	28/30/9818

3.1.1 Conclusões

A arquitetura proposta neste trabalho é bastante semelhante à desenvolvida nessa dissertação. Uma diferença é que, no artigo, são *bufferizadas* as linhas da matriz de entrada, enquanto que na arquitetura mostrada nessa dissertação são *bufferizadas* as colunas da matriz.

A ordem utilizada nos esquemas de diferenças finitas também é diferente, já que nesta dissertação é utilizada apenas a quarta ordem. As otimizações implementadas, no artigo, no

método de diferenças finitas parecem bastante promissoras, porém foram mostrados resultados apenas para ondas acústicas com valores de número de onda em até 0.5.

Outro possível problema que a solução mostrada neste trabalho pode apresentar e que não foi citado é o erro causado pela diminuição da precisão. Caso o número de passos de processamento seja muito grande, com valores entre 10000 e 15000, esse erro pode aumentar, gerando resultados que não são aceitáveis.

3.2 FPGA-Based Acceleration of the 3D Finite-Difference Time-Domain Method. James P. Durbano, Fernando E. Ortiz, John R. Humphrey, Petersen F. Curt, Dennis W. Prather.2004

Em [30] os autores James P. Durbano, Fernando E. Ortiz, John R. Humphrey, Petersen F. Curt e Dennis W. Prather apresentam o desenvolvimento de um acelerador *full-3D* para o método de diferenças finitas, no domínio do tempo (*Finite-Difference Time-Domain - FDTD*), implementado em uma plataforma *FPGA*. Esse acelerador provê aumento de velocidade de mais de três ordens de grandeza quando comparado com soluções em um único *PC* e ultrapassa a vazão de dados de saída (*throughput)* de clusters de *PCs*, como será demonstrado a seguir.

Os autores apresentam a implementação de um algoritmo computacional eletromagnético que pode ser utilizado para uma vasta gama de problemas, como, por exemplo, em um oscilador *"coupled-ring"*. O algoritmo deve levar em conta as dependências tanto temporais como espaciais que existem nas equações de Maxwell, que governam a propagação eletromagnética. Este método substitui as derivadas espaciais encontradas nas equações de Maxwell por aproximações de diferenças-centrais, reduzindo as equações a operações simples como adições, subtrações, e multiplicações.

O acelerador em *hardware* possui três componentes principais: a interface *Host-PC*, a hierarquia de memória, e o caminho de dados computacionais. A seguir será descrito cada um desses componentes:

1. A interface *Host-PC* consiste de um aplicação em Java, e uma placa *PCI* que comporta o *FPGA*. O *software* controla o envio de dados para o *FPGA*, como: o tamanho da malha, o

número de passos que serão executados e uma *lookup table* que contém valores da fonte de ondas eletromagnéticas. O acelerador *FDTD* processa os dados e periodicamente retorna os resultados para o *host PC* para eles serem pós-processados ou visualizados.

1.1. Um dos aspectos interessantes desta plataforma, é que o controlador do barramento *PCI* é um *ASIC*, o que libera toda a lógica do *FPGA* para implementação do algoritmo.

1.2. Um outro aspecto operacional importante é que o tráfego de dados, entre a placa onde está localizado o *FPGA* e a *CPU*, utiliza canais de *DMA*, permitindo operações do tipo *burst*. Como essas transferências são realizadas independentemente do caminho de dados computacionais, a latência de comunicação não atrasa as computações.

2. O armazenamento de dados e a largura do barramento da memória do método *FDTD* são bastante altos, com problemas representativos possuindo de 20 – 200 milhões de nós, requerendo algo em torno de 1 – 10 GB de *RAM*. Estes fatores exigem uma grande quantidade de memória de alta velocidade, memória principal e cache. Esse sistema é composto de *SRAM*, *DRAM* e blocos de *RAM* internos do *FPGA* (*BRAM*).

   2.1. A plataforma contém quatro *DDR SDRAM DIMMs* independentes. Canais *DMA*, via método *burst*, são utilizados para transferência de dados entre *FPGA* e memória externa. Um mecanismo de buffer interno a *FPGA* (*BRAM*) também foi utilizado para guardar dados temporários.

   2.2. Nesta plataforma os blocos de *RAM* são utilizados com *caches dual-ported*, o que possibilita maximizar a comunicação entre os blocos de *RAM* e o caminho de dados computacionais, mantendo o *pipeline* sempre cheio.

3. O caminho de dados da arquitetura pode ser visualizada através da Figura 21.

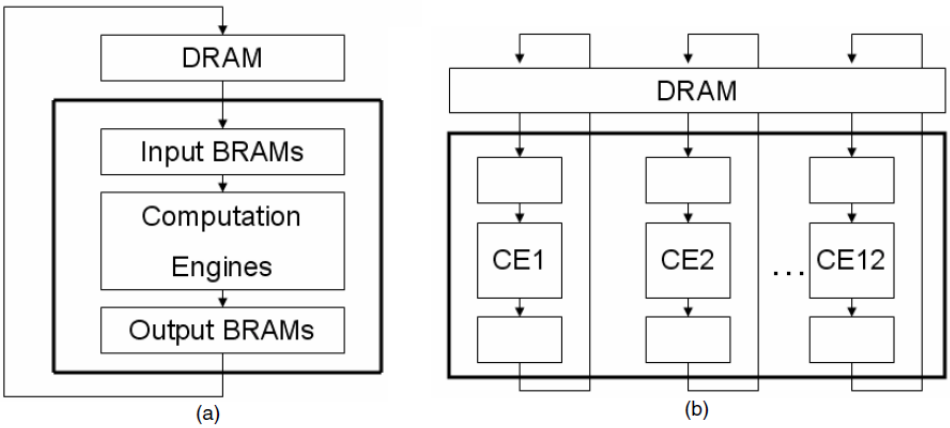


Figura 21 (a) Essa figura mostra, em alto nível, como é o fluxo de dados entre a DRAM e o FPGA. (b) Essa figura mostra o mesmo fluxo dos dados, porem com mais detalhes nos sistemas de cache e engenho de computação.

Devido a precisão exigida na resolução do problema, sua necessidade de aritmética de ponto flutuante com precisão simples de 32 bits no padrão IEEE 754 foi utilizada neste projeto.

Em simulações eletromagnéticas deve haver uma fonte de excitação, a qual é responsável pela geração de ondas eletromagnéticas a serem analisada. A geração é customizada e implementada em circuitos, através de *look-up tables* e técnicas de interpolação. Essa tabela é parcialmente armazenada na *SRAM* e nos blocos de *RAM* do *FPGA*.

Os Núcleos de computação implementam as equações de atualização do campo, as quais representam a forma discretizada da equação de Maxwell. Essas equações são responsáveis pela propagação dos campos eletromagnéticos e são o coração da implementação baseada em *hardware* do método *FDTD*. Na arquitetura proposta, doze equações são executadas em paralelo, o que permite que doze campos sejam atualizados simultaneamente.

A placa utilizada pelos autores suporta até 16 GB de *DDR SDRAM* e 36 MB de *DDR SRAM*. Essa placa possui uma *Xilinx Virtex II 8000* [32], e um controlador externo *PCI PLX 9656* [31] que suporta o barramento *PCI 64/66*. Esses componentes proveem uma largura de memória de pico de 9.7 GB/s. A Figura 22 mostra o diagrama de blocos da placa e também mostra fisicamente a placa.

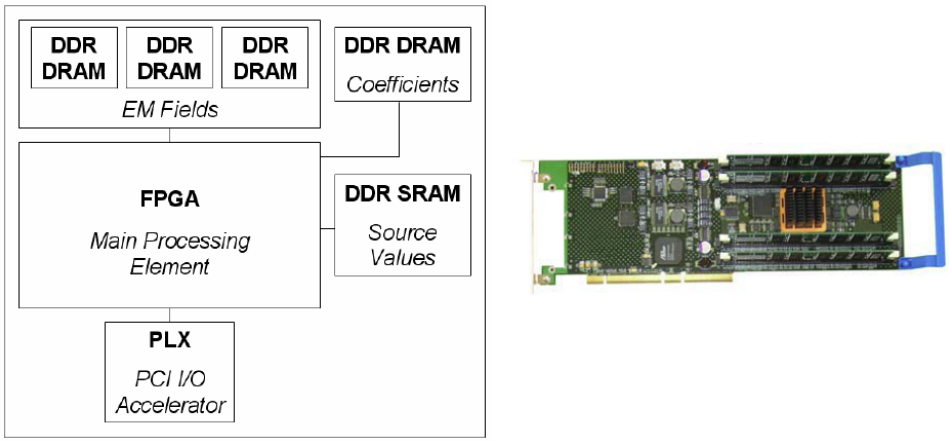


Figura 22 Diagrama de blocos da placa (esquerda) e a placa utilizada (direita).

É esperado que esse sistema atualize aproximadamente 30 milhões de nós por segundo 30 Mnps, constante, independente do tamanho do problema, enquanto que uma *Workstation* da *Dell* com 3.0 GHz, 2 GB PC 3200 *DDR SDRAM*, por exemplo, pode atualizar até 4.3 Mnps, no pico. Em média este número cai para algo em torno de 1.3 Mnps ou ainda menos, caso o problema seja muito grande para caber em sua memória e necessite de utilização do disco

rígido. A Figura 23 [30] mostra uma comparação de desempenho em relação ao tamanho do problema.

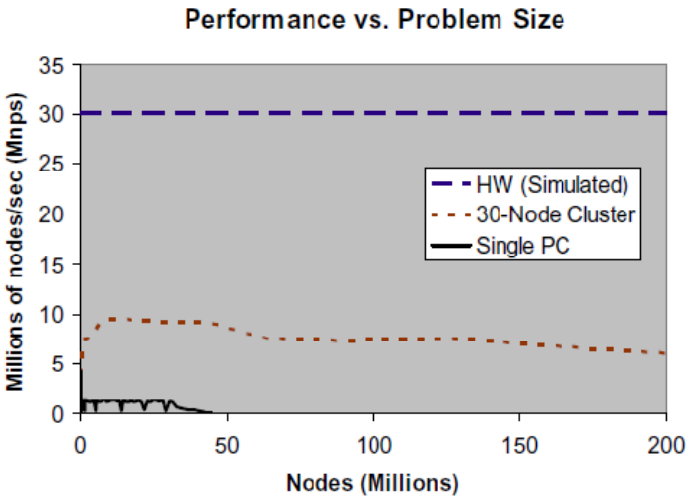


Figura 23 Comparação de desempenho entre PC, cluster de 30 nós e a arquitetura proposta.

O acelerador pode suportar problemas com até 256 milhões de nós, enquanto a *Workstation* pode suportar problemas com até 44 milhões de nós. Embora o custo do acelerador seja bem superior a de um *single-PC* seu desempenho é 23 vezes maior. Quando comparado um *cluster* de 90 *PCs*, apesar de terem uma vazão de dados (*throughput*) aproximadamente iguais, o preço do acelerador é bem menor, além de ocupar uma área bem menor e consumir bem menos potência.

3.2.1 Conclusões

A arquitetura apresentada nesse artigo possui a mesma estratégia de utilização de memórias da placa e de memórias internas do FPGA que é utilizada na plataforma proposta nesta dissertação. Como citado anteriormente, na arquitetura proposta neste artigo, doze equações são executadas em paralelo. Porém não está explícito quantos dados são necessários para computar essas doze equações, nem como eles estão organizados na memória.

Uma possível melhoria seria a utilização de uma representação customizada para os dados, como é feito em [33], melhorando a utilização dos recursos de *hardware* e a transferência dos dados, já que utilizar a representação de ponto flutuante seguindo o padrão IEEE 754 é bastante custosa.

Um ponto negativo da arquitetura desse artigo é que ela ainda não foi validada em *hardware*, pois os autores estavam encontrando alguns problemas de "timing" para realizar os testes no FPGA. Assim, os resultados apresentados foram obtidos a partir de simulações funcionais.

3.3 Accelerating 3D convolution using streaming architectures on FPGAs. H. Fu, R. G. Clapp, O. Mencer and O. Pell .October 2009.

Em [63] os autores H. Fu, R. G. Clapp, O. Mencer e O. Pell investigam a capacidade que o *FPGA* possui na resolução de problemas de convolução 3D. Para realizar tal objetivo, foram exploradas algumas variações na arquitetura proposta no trabalho, como: utilização de estênceis diferentes; utilização de vários operadores executando em paralelo no FPGA; execução de vários passos de processamento de forma simultânea; e customização da precisão utilizada nos números.

3.3.1 Arquitetura em *Stream* para computação da convolução

A aplicação que foi escolhida para se chegar ao objetivo citado anteriormente foi a Migração Reversa no Tempo. O tamanho do problema foi 512x512x512, utilizando ordens espaciais variando da quarta a sexta, e utilizando a segunda ordem no tempo.

A placa de aceleração utilizada neste trabalho foi a *MAX2* da *Maxeler*[8]. Essa placa possui um *FPGA Virtex-5 LX330T* [64], 12 GB de memória, e uma interface *PCI-Express* x16, que é utilizada para comunicação com o *host PC*.

A arquitetura em *stream* implementada em *FPGA* computa um resultado a cada ciclo de relógio. Para conseguir esse desempenho foram utilizadas estruturas de memória que fazem a *bufferização* de dados. A Figura 24 mostra como é realizada a computação em *stream* dos dados em um exemplo em duas dimensões. Para a computação do elemento (3,3) mostrado na Figura 24, as seis linhas da matriz que vão do elemento (0,3) até o elemento (6,3) são *bufferizadas*, assim para a computação do próximo elemento (3,4) apenas um leitura é necessária, já que todos os outros dados já se encontram na estrutura de *bufferização.*

[8] http://www.maxeler.com

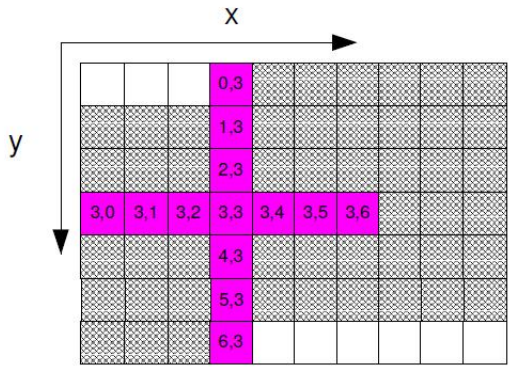


Figura 24 Computação dos dados em stream da convolução 2D.

Para uma linha com tamanho 512, esta estratégia requer o armazenamento de 512x6 dados da matriz. Para o caso da convolução 3D com um estêncil com 7 pontos e uma matriz com tamanho 512x512x512, o número de elementos atinge o valor de 512x512x6. Como essa quantidade de dados não caberia na FPGA utilizada neste trabalho, os autores dividiram a matriz em matrizes menores e realizaram a convolução em cada uma dessas matrizes menores separadamente.

3.3.2 Exploração de opções de projeto

Na arquitetura desenvolvida, foi utilizado um estêncil 'estrela' com 7 pontos, em cada eixo, para realizar a computação do algoritmo de diferenças finitas de sexta ordem. O outro estêncil utilizado nas explorações realizadas na arquitetura foi o estêncil 'cubo' 3-por-3-por-3, que realiza a computação do algoritmo de diferenças finitas de sexta ordem. Estes dois estênceis podem ser vistos na Figura 25.

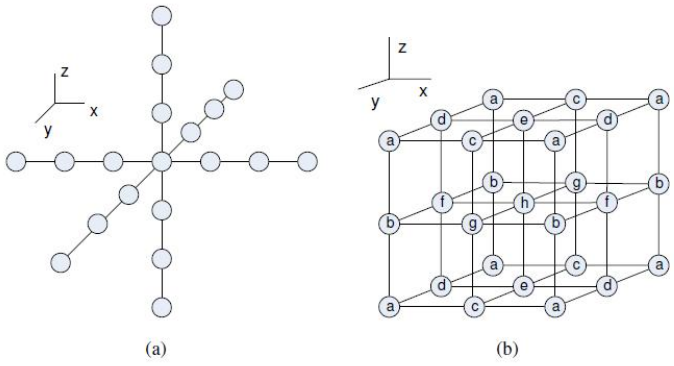


Figura 25 Estêncil 'estrela' (a) e o estêncil 'cubo' (b).

A diferença do custo de recursos para esses dois estênceis pode ser visto na parte superior da Tabela 5. Os autores realizaram otimizações na computação desses estênceis,

explorando a simetria que existe nos coeficientes dos mesmos, como pode ser visto nos coeficientes que possuem a mesma letra no estêncil ‘cubo’ mostrado na Figura 25. Assim, várias computações aritméticas puderam ser reutilizadas. O resultado foi uma redução dos recursos utilizados pelos dois estênceis, que é mostrado na parte inferior da Tabela 5.

Tabela 5 Custo dos recursos para os estênceis 'estrela' e 'cubo' para uma matriz com tamanho 120x120x120.

Normal Stencils		‘star’	‘cube’
FPGA resource costs	#slices	5618	7072
	#BRAMs	87	30
	#DSP48Es	50	60
Optimized Stencils		‘star’	‘cube’
FPGA resource cost	#slices	5207	6256
	#BRAMs	87	30
	#DSP48Es	32	18

Outra exploração desenvolvida neste artigo foi utilizar a máxima quantidade de estênceis concorrentes que podem ser colocados no *FPGA*. Os autores constataram que esse aumento do número de estênceis possui um limite do ponto de vista do desempenho, pois a utilização demasiada de estênceis faria com que a memória não conseguisse fornecer todos os dados necessários para a computação.

Assim, foi desenvolvida uma ferramenta em *software*, utilizando resultados experimentais, que modela o custo e o desempenho de várias arquiteturas. A Figura 26 mostra o desempenho estimado para o processamento de uma convolução 3D com tamanho 512x512x512 utilizando diferentes números de núcleos de computação em *FPGA*. Esse desempenho foi atingido com as arquiteturas possuindo uma frequência de trabalho de 125 MHz, e comparado com uma implementação em *software* sendo executada em um *Intel Xeon* 2.0 GHz. Devido à restrição do numero de slices lógicos do *FPGA*, o número máximo de estênceis concorrentes do tipo ‘cubo’ foram seis, e oito para o tipo ‘estrela’.

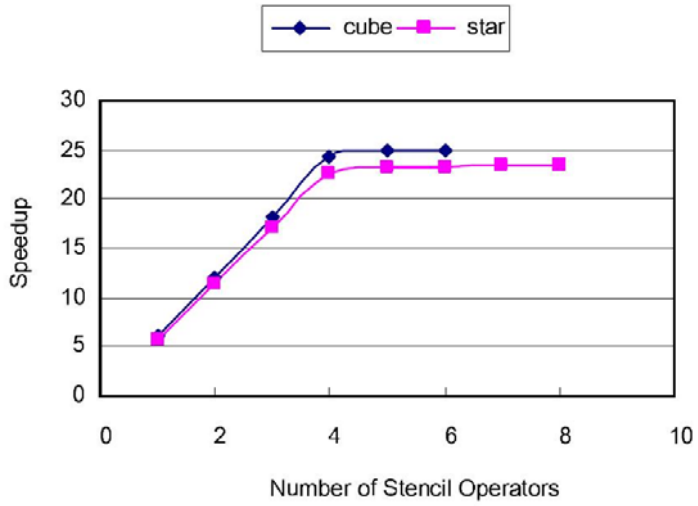


Figura 26 Speedup para o processamento de uma convolução 3D 512x512x512 com vários operadores de estênceis.

Na Figura 26 pode notar que o *speedup* alcançado pela arquitetura ficou estável quando o numero de estênceis concorrentes chegou a quatro. Isso aconteceu devido à memória não conseguir fornecer a quantidade de dados necessaria para um número de estênceis maior que quatro. Assim, para a utilização de uma quantidade de estênceis maior que quatro, alguma estratégia de diminuição de velocidade de processamento teve de ser adotada, como, por exemplo, a diminuição da frequência de operação de tais estênceis, resultando no mesmo *speedup* atingido na utilização de quatro estênceis.

Outra exploração realizada neste artigo foi processar múltiplos passos de processamento de forma simultânea, ao invés de utilizar vários núcleos de processamento concorrentes. A Figura 27 mostra a estrutura básica de um circuito que processa três passos de processamento simultâneos. Na Figura 27 pode ser visto que a saída de uma unidade de computação em um passo de processamento é a entrada de uma unidade de computação no passo de processamento seguinte. O exemplo mostrado na figura utiliza um estêncil do tipo ‘cubo’ 3-por-3-por-3.

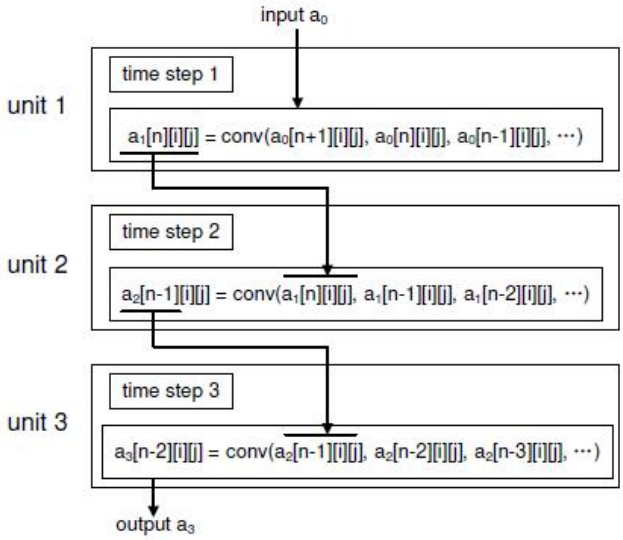


Figura 27 Estrutura de um circuito básico utilizado para processar multiplos passos de processamento (a_i representa o campo de pressao no passo de processamento *i*).

A vantagem de processar múltiplos passos de processamento, quando comparado com de utilizar vários núcleos de processamento concorrentes, é que o desempenho não fica restrito a largura de banda da memória, já que cada passo de processamento está pegando dados do passo anterior, com exceção do primeiro.

Uma desvantagem da utilização de múltiplos passos de processamento é a quantidade de dados que precisam ser armazenados, já que cada passo de processamento precisa armazenar os resultados do passo de processamento anterior. Por isso, os autores tiveram que quebrar ainda mais suas matrizes de entrada, aumentando, assim, o custo da computação dos dados sobrepostos, e aumentando o número de *streams* que devem ser realizadas.

A Figura 28 mostra os desempenhos estimados das arquiteturas em *FPGA* que processam múltiplos passos de processamento. Na Figura 28 são mostrados o desempenho do estêncil ‘estrela’, e do estêncil ‘cubo’ de segunda e quarta ordem. O estêncil ‘cubo’ possui um melhor desempenho, pois sua utilização de *BRAM* é menor que a do estêncil ‘estrela’. Devido à restrição no número de *slices* lógicos do *FPGA*, foram utilizados oito passos de processamento simultâneos para o estêncil ‘estrela’, seis e cinco passos para o estêncil ‘cubo’ de segunda e quarta ordem, respectivamente.

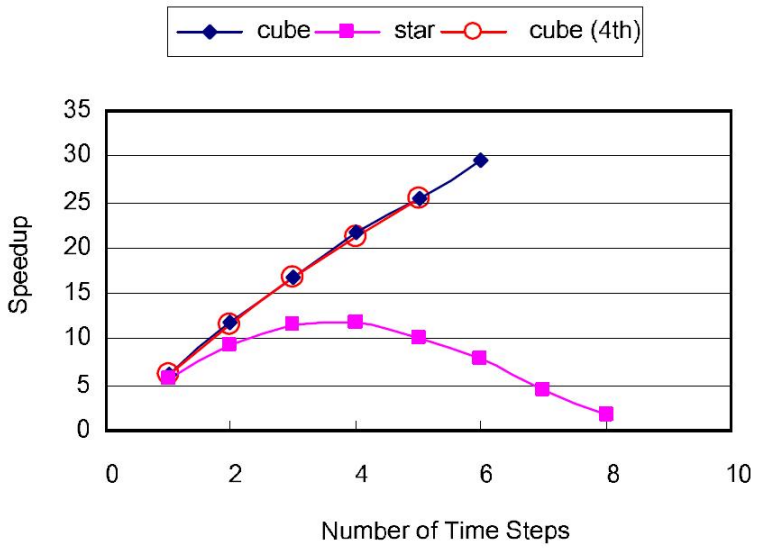


Figura 28 Speedup da computação de múltiplos passos de processamento simultâneos.

Na Figura 28 pode notar a grande queda de desempenho que o estêncil estrela sofre quando o numero de passos de processamento simultâneos passa de quatro. Isso ocorre devido a grande quantidade de dados que deve ser armazenada temporariamente dentro do FPGA para que a computação possa ser realizada. No caso de estêncil estrela 6 linhas inteiras da matriz devem ser armazenadas dentro do FPGA entre cada um dos passos de processamento, enquanto que no estêncil cubo apenas duas linhas são necessárias.

A última exploração realizada no trabalho foi modificar a precisão dos números utilizados pela arquitetura. A Figura 29 mostra o desempenho atingido utilizando números ponto flutuante com precisão reduzida. Com uma precisão de 16 bits ponto flutuante, o *speedup* alcançado pela estratégia de múltiplos estênceis concorrentes foi 49x e para a estratégia de múltiplos passos de processamentos foi 46x.

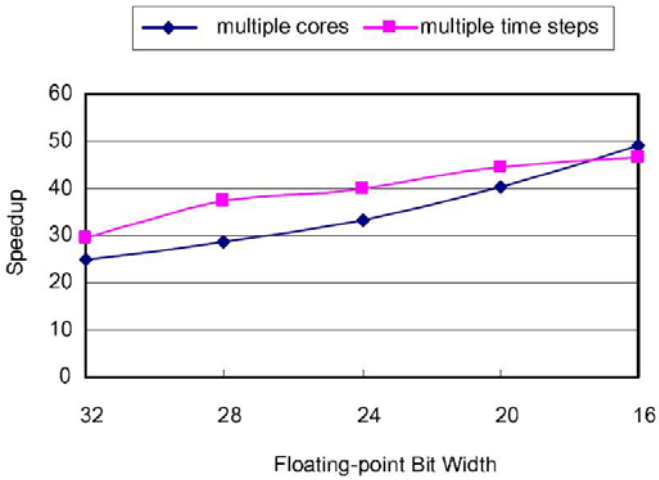


Figura 29 Speedup para precisão reduzida.

3.3.3 Resultados da aceleração

O ‘cubo’ de segunda ordem foi implementado com seis passos de processamento simultâneos, e o ‘cubo’ de quarta ordem com cinco passos de processamento simultâneos. A primeira implementação computou seis passos em 1,383 segundos, enquanto que a segunda computou cinco passos em 1,346 segundos. Comparado com a implementação de segunda ordem em *software* que computa um passo em 6,36 segundos, as duas implementações em *FPGA* obtiveram um *speedup* de 27,5x e 23,5x na primeira e na segunda implementações, respectivamente.

Essas implementações citadas anteriormente foram realizadas em um *FPGA* da placa utilizada em [63]. Porém, essa placa possui dois FPGAs idênticos, assim, segundo os autores, o *speedup* mostrado anteriormente para as duas implementações em FPGA podem ser duplicados, alcançando 55x e 47x. Ainda segundo os autores, caso fosse utilizada a Virtex-6 SX475T [65], seria possível utilizar 13 passos de processamentos simultâneos em um único FPGA, alcançando um *speedup* de 55x. Utilizando dois FPGAs, esse *speedup* chegaria a 110x, quando comparado com uma implementação em uma CPU com um único core.

Essas implementações citadas anteriormente foram realizadas em um *FPGA* da placa utilizada no artigo. Para as duas *FPGAs* existentes na placa seria alcançado o dobro de desempenho (*speedup*), ou seja, 55x e 47x respectivamente. Ainda segundo os autores, caso fosse utilizada a *Virtex-6 SX475T* [65], seria possível utilizar 13 passos de processamentos simultâneos em um único *FPGA*, alcançando um *speedup* de 55x. Para os dois *FPGAs* teríamos um *speedup* de 110x, quando comparado com uma implementação em *software*.

3.3.4 Conclusões

A arquitetura em *stream* proposta neste artigo possui uma grande semelhança com a arquitetura proposta nesta dissertação. Porém, a estrutura de *bufferização* utilizada neste artigo segue a mesma ideia da estrutura mostrada no artigo [26], onde as linhas da matriz do campo de pressão atual são *bufferizadas*, diferentemente da estratégia utilizada nesta dissertação que *bufferiza* as colunas dessa matriz, que possui um menor custo, como citado na seção 3.1.1.

No trabalho apresentado, não foi explorada a utilização de vários núcleos concorrente em conjunto com a utilização do vários passos de processamentos simultâneos. Essa abordagem seria interessante porque aumentaria o reuso dos dados, já que o aumento do numero de núcleos concorrentes aumenta o reuso.

A estratégia de redução de precisão parece bastante promissora na redução de transações que são necessárias com a memória, porém um estudo bastante cuidadoso deve ser realizado, através da execução de vários exemplos com modelos e tamanhos diferentes, e com vários números de passos de processamento. Isto é importante, pois essa redução de precisão pode ocasionar em pequenos erros, que, com o crescimento do numero de passos de processamento, pode levar a geração de imagens que não representem o terreno corretamente.

O *speedup* mostrado através da utilização de dois *FPGA* se mostra bastante promissor, porém, para se ter uma duplicação do desempenho, o tempo da troca de informações, entre os dois *FPGAs* deve se sobrepor ao tempo de processamento. Os autores não mostraram como iriam realizar tal tarefa.

3.4 Conclusões

Neste capítulo foram mostrados alguns trabalhos que utilizam *FPGAs* como coprocessadores da *CPU* com o objetivo de melhorar o desempenho de aplicações cientificas que utilizam o método de diferenças finitas na resolução de seus problemas.

O primeiro trabalho [26] propôs uma solução baseada em computação reconfigurável para acelerar a execução de problemas de modelagem em sísmica acústica ou elástica. Para tal, foram realizadas otimizações na escolha dos coeficientes do método de diferenças finitas, assim como uma precisão modificada foi utilizada. A arquitetura desenvolvida se baseou em FIFOS para alcançar a computação de um dado a cada ciclo de relógio e um bom reuso de dados. Bons ganhos de desempenhos foram alcançados, principalmente nas arquiteturas que utilizaram o método de diferenças finitas com ordens mais altas.

O segundo trabalho [30] propôs o desenvolvimento de um acelerador *full-3D* para o método de diferenças finitas, no domínio do tempo (*Finite-Difference Time-Domain - FDTD*).

Doze núcleos de processamento são utilizados na arquitetura desenvolvida, alcançando um desempenho comparado com um *cluster* de 90 *PCs.*

O terceiro trabalho [63] investiga a capacidade que o *FPGA* possui na resolução de problemas de convolução 3D. Para realizar tal objetivo, foram exploradas algumas variações na arquitetura proposta no trabalho, como: utilização de estênceis diferentes; utilização de vários operadores executando em paralelo no FPGA; execução de vários passos de processamento de forma simultânea; e customização da precisão utilizada nos números. Foram realizadas duas implementações que atingiram um *speedup* de 27,5x e 23,5x.

Capítulo

4

4 Plataforma PROCe-III

Este capítulo tem como objetivo apresentar a plataforma *PROCe III* que foi utilizada no desenvolvimento do protótipo da plataforma desenvolvida nesta dissertação. Essa plataforma consiste da placa onde se encontra o *FPGA*, do *IP-PROCMultiport* utilizado para controle das memórias e do barramento *PCIe*, e do *device driver* que é utilizado pela aplicação para acessar os componentes da placa.

4.1 O Sistema *PROCe III*

A plataforma *PROCe III* é composto pela placa de prototipação, por *IPs*, *device drivers* e pela ferramenta *PROCWizard* [37]. Nessa seção serão mostradas as principais características da placa que foi utilizada no desenvolvimento da plataforma proposta nessa dissertação.

O Sistema *PROCe III* da *GiDEL* [34] fornece uma plataforma baseada em FPGA de alta velocidade com grande taxa de transferência de dados (*throughput*) e uma grande quantidade de memória. Este sistema é voltado para aplicações que requerem um grande poder computacional e possuem uma grande quantidade de dados, como por exemplo, processamento de imagens, sistemas aeroespaciais e militares, prototipação e *debug* de sistemas, etc. A Figura 30 mostra a placa.

Figura 30 Placa PROCe III.

A arquitetura da *PROCe III* é baseada na *Stratix III EP3SE80* da *Altera* [35] e suporta frequências de operação de até 300 MHz. Esse *FPGA* possui as seguintes características:

- 80.000 elementos lógicos (LE).
- 6.683 Kbits de memória distribuída em três blocos de *RAM*, denominados de *TriMatrix*, que são utilizados para implementar memórias *dual-port* e *FIFOs*.
- 672 blocos *DSPs* de alta velocidade, que possuem multiplicadores 18x18 bits (aritmética inteira), utilizados para implementar multiplicadores de 9×9, 12×12, 18×18,e 36×36 bits com velocidades de até 550 MHz, funções de multiplicação - acumulação, e filtros *FIR*.
- Oito *PLLs (Phase Locked Loop)*

A arquitetura também possui ainda uma memória *DDR2 SDRAM* de 512 MB com largura de barramento de 64 bits ligada diretamente ao *FPGA*. Dois *slots* adicionais de 200 pinos para memórias *DDR2 DRAM SODIMM* com 64bits de largura de barramento e com capacidade de endereçamento de até 8GB também são disponíveis na placa. Esses três blocos de memória podem ser controlados independentemente, por um controlador desenvolvido pelo usuário, ou pelo *PROCMultiPort* [36] da *GiDEL*. O *PROCMultiport* é um controlador de memória que pode ser gerado automaticamente através da ferramenta *PROCWizard* da *GiDEL*.

Uma interface *PCI Express* de 4x possibilita que a placa seja conectada à maioria dos *PCs* atuais. Esta interface suporta a utilização de até 32 canais de *DMA*, os quais permitem que a placa tenha acesso ao barramento na forma de um dispositivo *master*. O controle desse *DMA* é feito através de um *device driver*, gerado pela ferramenta *PROCWizard*. A Figura 31 mostra o diagrama de blocos da arquitetura.

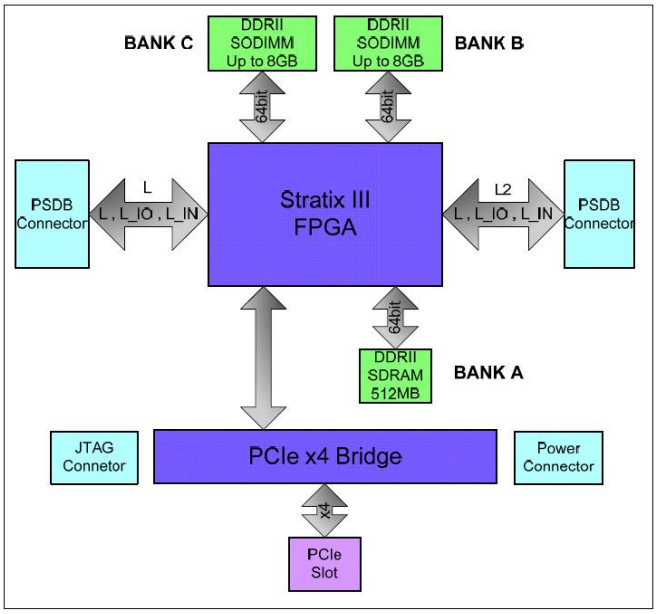


Figura 31 Diagrama de Blocos da arquitetura da PROCe III.

4.2 Infraestrutura de Hardware – Módulo *PROCMultiPort*

O módulo *PROCMultiPort* [36], é um *IP-core* gerado pela ferramenta *PROCWizard*, utilizado para controlar as memórias disponíveis na placa de forma eficiente, mantendo uma interface de comunicação simples. Utilizando os blocos de memória *DRAM*, localizados na placa, e a memória interna do *FPGA*, o *PROCMultiPort* pode implementar a funcionalidade de vários tipos diferentes de memórias mantendo uma interface simples para a aplicação no *FPGA*.

O *PROCMultiPort* consegue transformar uma memória localizada na placa, que possui apenas uma porta de acesso, em uma memória com várias portas de acesso (*multi-port*).Assim, um único bloco de memória pode ser acessado simultaneamente por até 16 portas, onde cada uma dessas portas possui seu próprio domínio de frequência, assim como, sua própria largura de *bits* dos dados. O acesso à memória pode ser feito ainda no modo sequencial ou randômico. No modo sequencial o acesso pode ocorrer na forma de *burst*, ou palavra por palavra, utilizando uma interface semelhante a interfaces de uma *FIFO*. Além disso, a utilização do *PROCMultiPort*, em conjunto com os canais de *DMA* da placa, faz com que o usuário possa iniciar uma transferência de dados no modo *DMA*, liberando a CPU para a realização de outras tarefas, durante a transferência de dados entre a memória do PC e da placa, aumentando o paralelismo de execução.

O modo randômico de acesso à memória, por sua vez, permite que o usuário realize leituras ou escritas randômicas na memória através das portas de acesso. As portas randômicas possuem a prioridade mais alta no acesso a memória quando comparadas com as sequenciais.

A Figura 32 mostra o diagrama de blocos do PROCMultiPort onde o caminho de dados entre a memória e a aplicação pode ser visto.

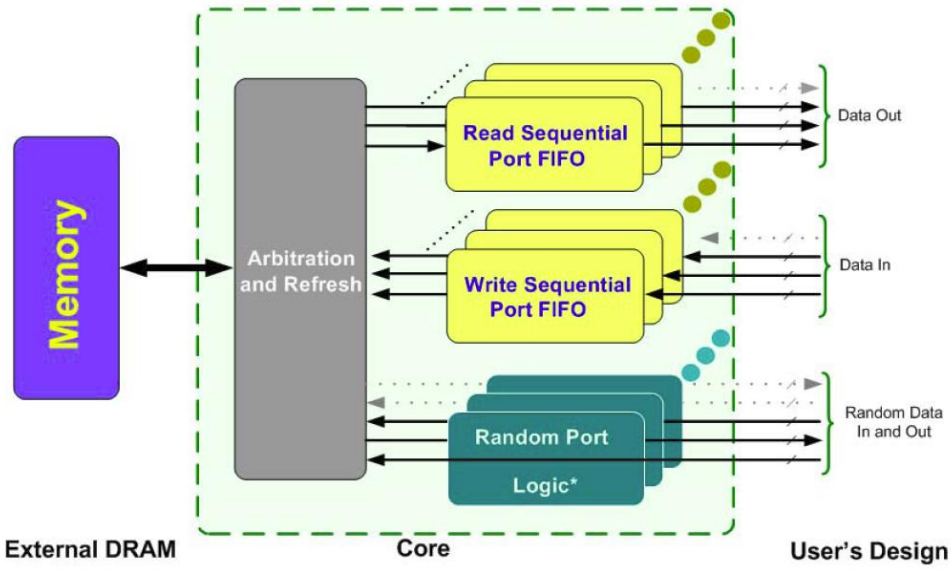


Figura 32 Diagrama de blocos do PROCMultiPort.

O *IP core* do controlador do *PROCMultiPort* possui duas interfaces básicas:

- Interface com a memória, que se conecta diretamente com a memória DDR.

- Interface das portas ou interface com o usuário, que se conecta com o *host-pc* ou com a lógica do usuário, especificando a largura de dados, frequência de operação, tipo de acesso, etc.

4.3 Infraestrutura de *Software*

Nessa seção serão apresentadas as principais características da aplicação *PROCWizard*, ferramenta utilizada na integração entre os módulos de *hardware* e *software*, criação e integração de *IP-cores*. Nessa seção também será mostrado o *device driver* criado pela aplicação *PROCWizard*, responsável pela comunicação entre a aplicação que está sendo executada no *host* e a placa de prototipação.

4.3.1 A PROCWizard

A *PROCWizard* da *GiDEL* [37] é uma aplicação que tem como objetivo integrar componentes de *hardware* e *software* durante o desenvolvimento de um projeto. Trabalhando em conjunto com as placas da GiDEL, essa aplicação permite que o usuário rapidamente desenvolva um projeto que pode ser automaticamente traduzido em código *HDL* e *C++*. O código *C++* se comunica com o código *HDL* através do barramento *PCI*, garantindo uma fácil integração *hardware-software*.

PROCWizard disponibiliza também um ambiente para teste/depuração do projeto do usuário. No modo de *Debug* (depuração) da ferramenta, pode acessar a placa no momento da execução do projeto através de um *browser*, manualmente ou utilizando scripts / macros. O ambiente de co-simulação permite o compartilhamento simultâneo, pelos componentes de hardware e software, da mesma informação e definições do projeto. Dessa forma, o tempo de desenvolvimento do projeto é reduzido, e a confiabilidade e manutenção do projeto se tornam melhores.

As principais características e funcionalidades do *PROCWizard* são:

- ✓ Integração automática entre *software* e *hardware*.
- ✓ Integração automática de IP cores da *GiDEL* no projeto.
- ✓ Geração de código *HDL*.

- ✓ Geração do *Device driver* em *C++.*
- ✓ Geração de documentação.
- ✓ *Debug* de *hardware*.

Dentre as características acima, talvez a mais importante seja a geração automática dos módulos de hardware, em uma linguagem de descrição de *hardware* (*HDL)*. Neste sentido, a ferramente cria um módulo top-level que conecta o sub-módulo responsável pela interface *hardware/software* com todos os submódulos do usuário. Todas as restrições da placa que são necessárias para a geração do *hardware* e o projeto do Quartus [38], contendo essas restrições, e o módulo top-level também são gerados automaticamente. Isso faz com que o desenvolvedor não tenha que definir os detalhes da alocação dos pinos do *FPGA* e protocolo de comunicação do barramento *PCI*. Uma visão geral do sistema é mostrada na Figura 33.

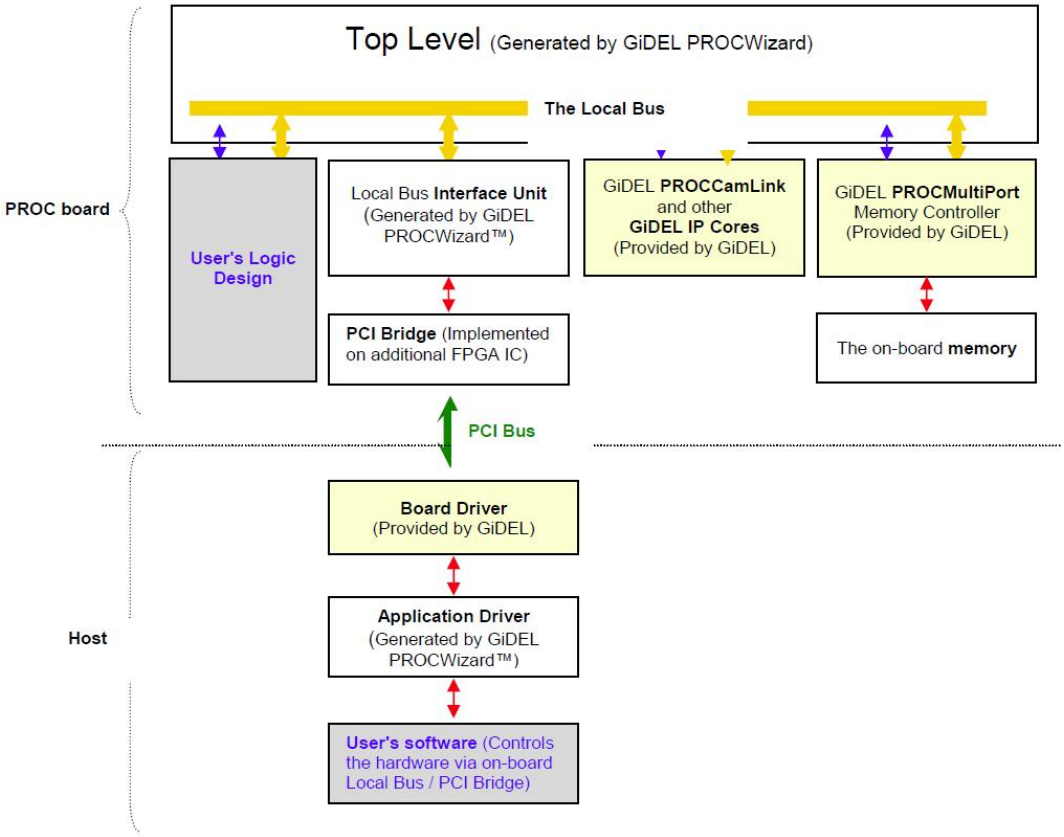


Figura 33 Visão Geral do Sistema gerado na PROCWizard.

4.3.2 O Device driver

O aplicativo *PROCWizard* permite a geração de um *device driver* para acesso aos módulos de *hardware* na placa de prototipação, como uma classe em *C++*. Isto faz com que o mesmo seja visto como um objeto em *C++* na aplicação do usuário. Este *driver* também realiza

a função de inicialização do *hardware*(configuração do *FPGA*), assim como configurações das frequências de trabalho da placa.

4.4 Conclusões

O sistema *PROCe III* apresentado se mostrou um sistema bastante interessante para ser usado por desenvolvedores de *IP-cores* em *FPGA* que necessitam de uma boa quantidade de memória externa e uma comunicação veloz com a *CPU*, através do barramento *PCIe*.

Esse sistema apresenta alguns pontos positivos e negativos. A característica mais interessante do produto da *GiDEL* é o fato de o desenvolvedor poder se concentrar no desenvolvimento do núcleo de seu problema, já que o sistema *PROCe III* disponibiliza várias ferramentas que implementam a comunicação tanto com a memória como com barramento *PCIe*, fazendo com que o desenvolvedor tenha que se preocupar em apenas utilizar essas ferramentas.

Um ponto negativo desse sistema é a quantidade de blocos de memória disponíveis. Caso a placa possuísse mais blocos de memória, mais núcleos de processamento poderiam ser instanciados na arquitetura, fazendo com que sua velocidade de computação aumentasse.

Capítulo

5

5 Plataforma Desenvolvida

Este capítulo tem como objetivo mostrar a plataforma desenvolvida nessa dissertação. A plataforma pode ser dividida em três partes principais: a Aplicação de Controle e Comunicação da Arquitetura para processamento em *stream*; o Núcleo de Processamento que é o módulo responsável por implementar a equação de diferenças finitas; e a Arquitetura para processamento em *stream*,responsável por controlar todo o fluxo de dados.

5.1 Visão geral da plataforma desenvolvida

A Plataforma desenvolvida nessa dissertação segue uma das tendências atuais de desenvolvimento de plataformas de alto desempenho, qual seja, a utilização de dispositivo lógicos programáveis, os *FPGAs*, para aceleração de processamento de dados, com consequente aumento no desempenho de sistemas computacionais. A plataforma proposta utiliza um *FPGA* acoplado a uma *CPU* de propósito geral, através de um barramento rápido *PCIe*. O *FPGA* é utilizado como um coprocessador que implementa a parte de um algoritmo que possui uma computação intensiva de dados. Dessa forma, o código que já está sendo usado para resolver o problema pode ser mantido, substituindo, apenas, a parte que realiza a computação intensiva, por uma chamada de função que, na verdade, estará sendo executada em um componente de hardware (*FPGA*).

Esta plataforma foi desenvolvida com o objetivo de acelerar o processamento de um algoritmo, que lida com modelagem 2D em sísmica, o qual representa o primeiro passo computacional do método sísmico RTM.

O sistema é dividido em duas partições (SW/HW). A primeira diz respeito ao componente de software, um aplicativo de controle e comunicação que é executado na *CPU*. Esse aplicativo tem como objetivo controlar e fornecer todos os dados necessários para que a arquitetura em hardware possa ser executada, no FPGA. A segunda partição, a arquitetura para processamento em *stream,* foi desenvolvida em hardware e tem como objetivo implementar, de uma forma otimizada, o método de diferenças finitas utilizado na modelagem sísmica, o primeiro passo computacional do método *RTM*. A Figura 34 mostra o diagrama de blocos dessa plataforma.

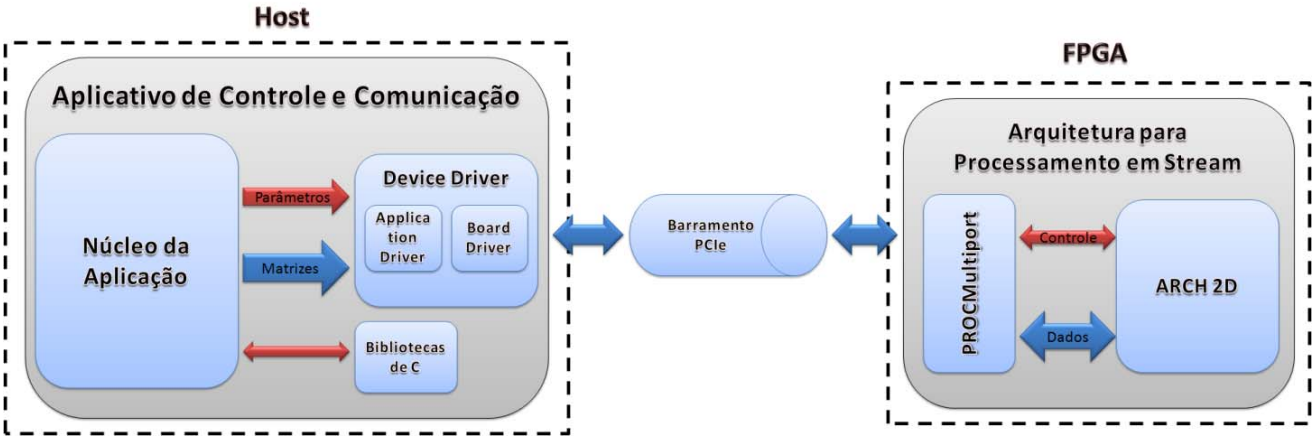


Figura 34 Diagrama de blocos da plataforma desenvolvida.

A plataforma possui o fluxo de processamento mostrado na Figura 35. O primeiro passo do fluxo é a escolha do tamanho do problema e da localização do pulso sísmico pelo usuário. Após essa escolha, o aplicativo de controle e comunicação realiza alguns pré-processamentos, como, por exemplo, o cálculo do vetor de pulso sísmico, a geração das matrizes de entrada, e o calculo e geração de alguns parâmetros que são enviados para a arquitetura em hardware. Em seguida, o aplicativo verifica, através do *device driver,* se a placa está pronta para receber os dados. Caso a placa esteja preparada para receber os dados, o aplicativo os envia para a memória da placa. Em seguida, o aplicativo sinaliza para arquitetura para processamento em *stream* que todos os dados necessários para a computação do problema escolhido pelo usuário já se encontram na memória. A arquitetura, então, realiza o um passo de processamento sobre os dados. Caso todos os passos de processamentos tenham sido realizados, a arquitetura sinaliza para o aplicativo que os resultados estão disponíveis. Caso contrário, o passo seguinte de computação é realizado. Quando todos os passos de processamento são realizados, o aplicativo, então, lê da memória da placa os resultados, e os disponibiliza para o usuário através de um arquivo.

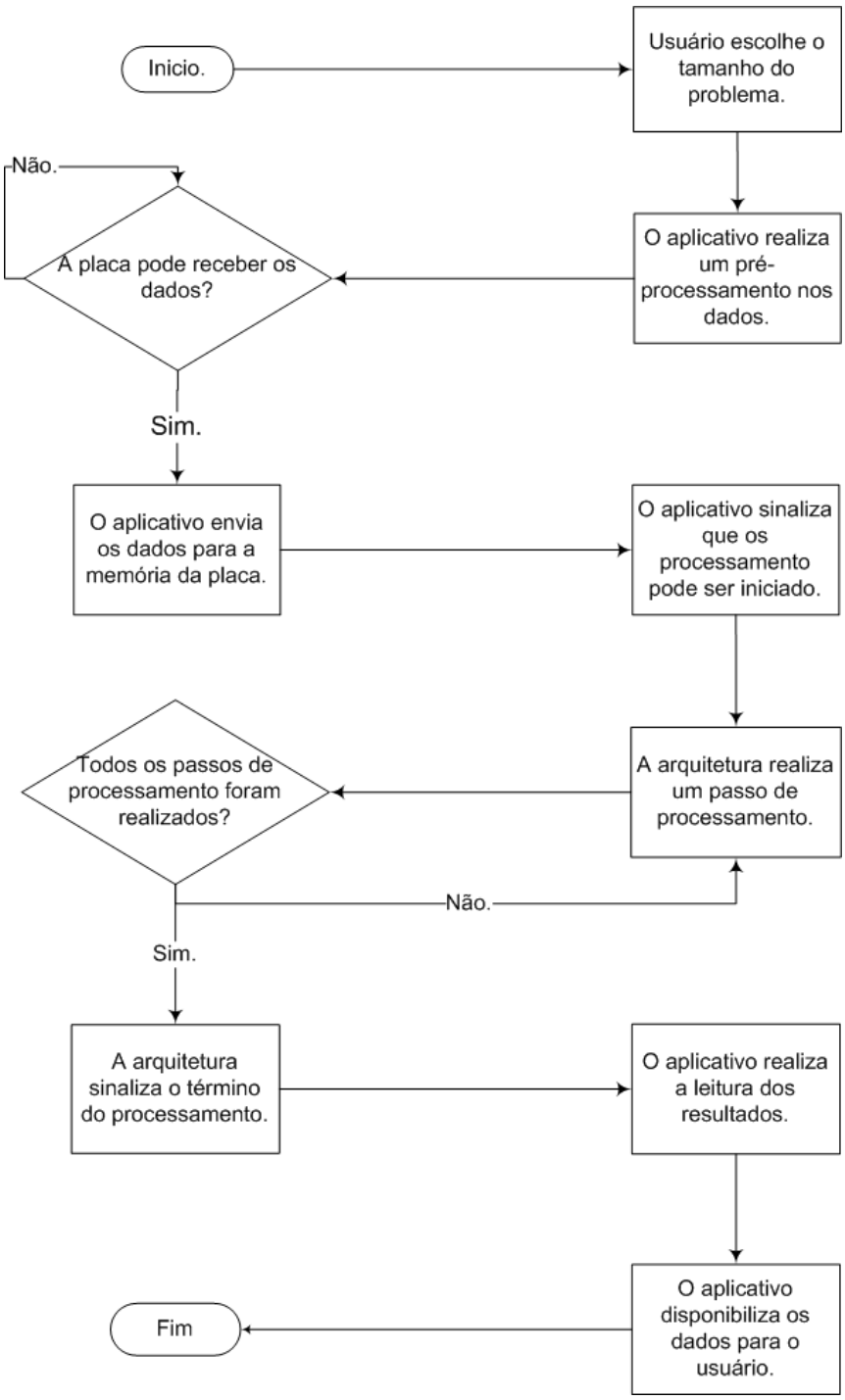


Figura 35 Fluxo de processamento da plataforma.

5.2 Aplicação de Controle e Comunicação

Um aplicativo de Controle e Comunicação foi desenvolvido em *software* com o objetivo controlar toda a arquitetura para o processamento em *stream*. Ou seja, fornecer dados necessários para os núcleos de processamentos realizarem a computação da equação de diferenças finitas, fornecer parâmetros para o controle de fluxo de dados na arquitetura, e ler

os resultados da memória da placa. Para se comunicar com a placa, a aplicação utiliza o *device driver* apresentado no capitulo 3 na seção 4.3.2. A Figura 36 mostra o diagrama de blocos da arquitetura Interna da Aplicação.

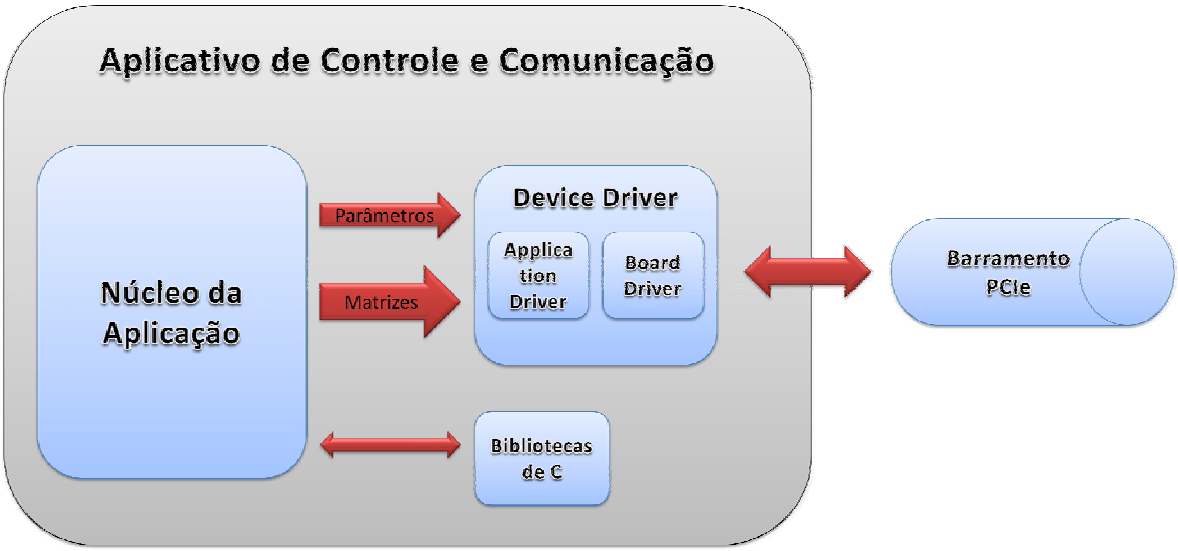


Figura 36 Arquitetura interna do Aplicativo de Controle e Comunicação.

Como citado anteriormente, o Aplicativo de Controle e Comunicação é responsável por enviar todas as matrizes de entrada, o vetor de entrada e os parâmetros de configuração da arquitetura para processamento em *stream*. Após enviar todos esses dados, a aplicação sinaliza para a arquitetura, através de um sinal (*start_process*), que o processamento já pode ser iniciado. Ao final, quando a arquitetura finaliza o processamento, um sinal de *end_process* sinaliza para a aplicação que o processamento chegou ao fim, Após a finalização do processamento, a aplicação faz a leitura da matriz de resposta.

O vetor de entrada é o vetor do pulso sísmico, que representa a fonte de ondas sísmicas inserida no modelo 2D. Ou seja, a onda que é inserida no modelo é representada por um vetor onde os índices desse vetor representam os instante de tempo, e os elementos do vetor representam os valores da amplitude da onda em cada um desses instantes de tempo. Esse vetor é gerado a partir de parâmetros específicos do problema, como a frequência da onda sísmica, o tamanho do *grid* que está representando o terreno, entre outros. Cada elemento desse vetor é representado por um número em ponto flutuante de 32 *bits* que é utilizado em cada passo de processamento da arquitetura. Ou seja, a onda sísmica que é inserida no modelo

é dividida espacialmente em vários valores, e cada um desses valores é inserido no modelo a cada passo de processamento. Após o vetor ser gerado ele é enviado para a memória da placa.

As matrizes de entrada são representadas pela matriz de pressão atual, a matriz de pressão anterior e a matriz de velocidade. As matrizes de pressão atual e anterior são geradas e inicializadas direto na memória da aplicação, com todos os seus elementos com valores iguais a zero. A matriz de velocidade, por sua vez, pode ser lida a partir de um arquivo ou pode ser gerada direto na memória da aplicação. Cada elemento dessas três matrizes é representado por um número, ponto flutuante, com 32 bits. Após a geração dessas matrizes, cada uma delas é transformada em um vetor unidimensional, ou quais, são então, transferidos para a memória da placa.

A conversão matriz x vetor é realizada, a partir da divisão da matriz em conjuntos de quatro colunas inteiras da matriz. A esta estrutura vetorial dá-se o nome de *slice*. Dessa forma, uma matriz se torna um conjunto de *slices* sequenciais, onde o número de *slices* é igual ao número de colunas da matriz dividido por quatro. O vetor resultante possui, portanto, uma largura de dados quatro vezes maior que o tamanho de um elemento da matriz, ou seja, cada elemento desse vetor possui 128 *bits*. O tamanho desse vetor equivale ao número de *slices* vezes o número de linhas da matriz. A Figura 37 mostra esse processo de transformação para uma matriz com 12 linhas e 16 colunas.

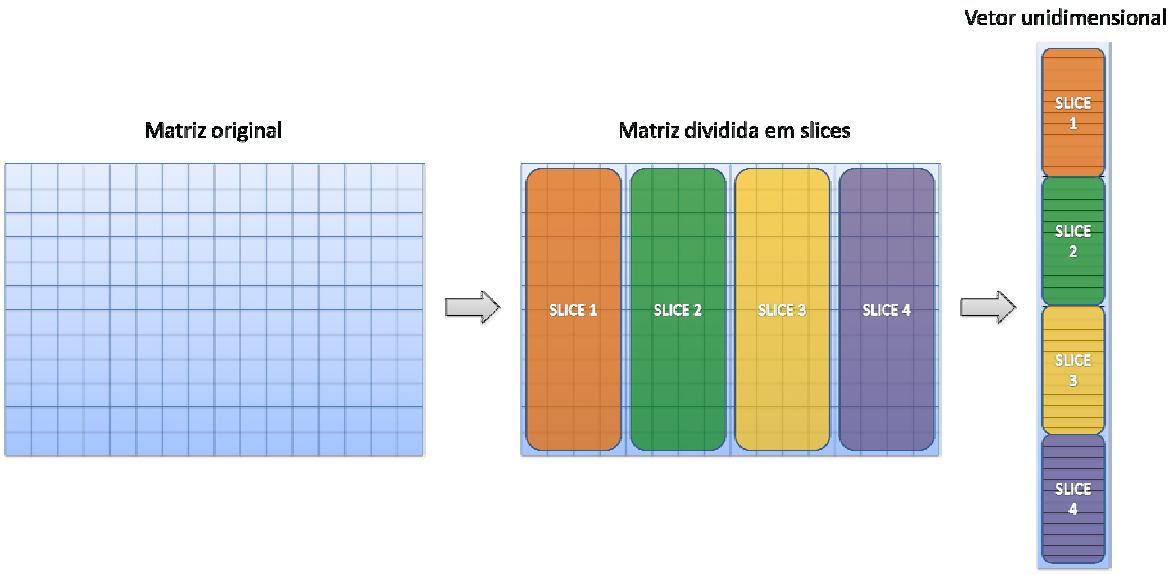


Figura 37 Transformação de uma matriz em um vetor unidimensional.

Além da conversão, alguns parâmetros precisam ser transferidos para a arquitetura, os quais são fornecidos pelo usuário da plataforma, em função do tamanho problema a ser resolvido. Esses parâmetros são: número de linhas da matriz (*PAR_num_lines [15:0]*), número de colunas da matriz (*PAR_num_cols [15:0]*), número de passos desejados de processamento (*PAR_num_times_steps [15:0]*), a linha onde será inserido o pulso sísmico (*PAR_pulse_pos_line [15:0]*) e a coluna onde será insira o pulso sísmico (*PAR_pulse_pos_col [15:0]*). A partir desses cinco parâmetros iniciais são gerados os seguintes parâmetros:

a. número de *slices* da matriz (*PAR_num_slices [15:0]*), como descrito acima;
b. o número de passos de inicialização que são realizados para as matrizes de entrada (*PAR_num_init_steps [15:0]*). Esse valor é igual ao número de linhas vezes dois;
c. o *slice* onde será inserido o pulso sísmico (*PAR_pulse_pos_slice [15:0]*), sendo esse igual ao valor da coluna onde será inserido o pulso dividido por quatro. Caso esse valor não resulte em um número inteiro, deve-se arredonda-lo para o próximo valor inteiro.
d. o número de linhas processáveis da matriz (*PAR_processable_lines [15:0]*), o que representa o número de linhas da matriz menos quatro. Essa subtração de quatro representa as linhas que são bordas na matriz e que não precisam ser processadas;
e. o endereço inicial da matriz de pressão atual (*PAR_apf_start_addr [31:0]*);
f. o endereço inicial da matriz de pressão anterior (*PAR_ppf_start_addr [31:0]*);
g. o endereço inicial da matriz de velocidades (*PAR_vel_start_addr [31:0]*);
h. o endereço inicial do vetor de pulso sísmico (*PAR_seismic_start_addr [31:0]*).

As palavras entre parênteses são os nomes dos atributos da classe do *device driver*, descrito no capitulo anterior, a partir dos quais o aplicativo envia parâmetros para o processamento.

Existe uma limitação para o tamanho de linhas das matrizes de entrada que podem ser enviadas para a arquitetura em *hardware*. Porém, essa limitação não é uma restrição da aplicação, mas sim da arquitetura, ou melhor, do tamanho máximo das FIFOS internas no *FPGA.* Este fato é função direta do modelo de reusabilidade de dados utilizado no modelo proposto. O numero de máximo de linhas que as matrizes de entrada podem ter é 8192.

5.3 Arquitetura para Processamento em Stream

5.3.1 Visão geral da Arquitetura

A Arquitetura para processamento em *stream* tem como objetivo fornecer os dados necessários para que os núcleos de Processamento possam realizar a computação das equações de diferenças finitas. O algoritmo de diferenças finitas possibilita um alto nível de paralelismo, ou seja, várias computações podem ocorrer em paralelo, e a computação de um elemento não necessita de algum tipo de comunicação ou armazenamento de elementos vizinhos que estão sendo computados ao mesmo tempo. Ou seja, a comunicação e armazenamentos globais são mínimos, possibilitando que os dados passem de uma unidade de computação para outra ao longo do Pipeline sem dependência de dados. A arquitetura proposta nessa dissertação explora essa localidade e concorrência, separando a comunicação de dados das estruturas de armazenamentos e de processamento, fazendo com que as unidades de computação possam ser alimentadas de forma eficientemente, o que caracteriza uma arquitetura de processamento em *stream*. Na arquitetura em *stream* o próprio fluxo de dados é quem "dispara" as computações.

A Arquitetura procura manter um fluxo de dados constante para os núcleos de processamento, fazendo com que eles estejam, na maior parte do tempo, processando dados válidos. Esta alimentação de dados é feita através do módulo *PROCMultiPort*, que realiza leituras e escritas na memória da placa e implementa a comunicação com o barramento *PCIe*. De fato, o módulo *PROCMultiPort,* se comunica com o aplicativo para controle e comunicação, através do barramento *PCIe,* para receber dados de entrada para processamento e sinais de controle. A Figura 38 mostra o diagrama de blocos da arquitetura interna para processamento em *stream*.

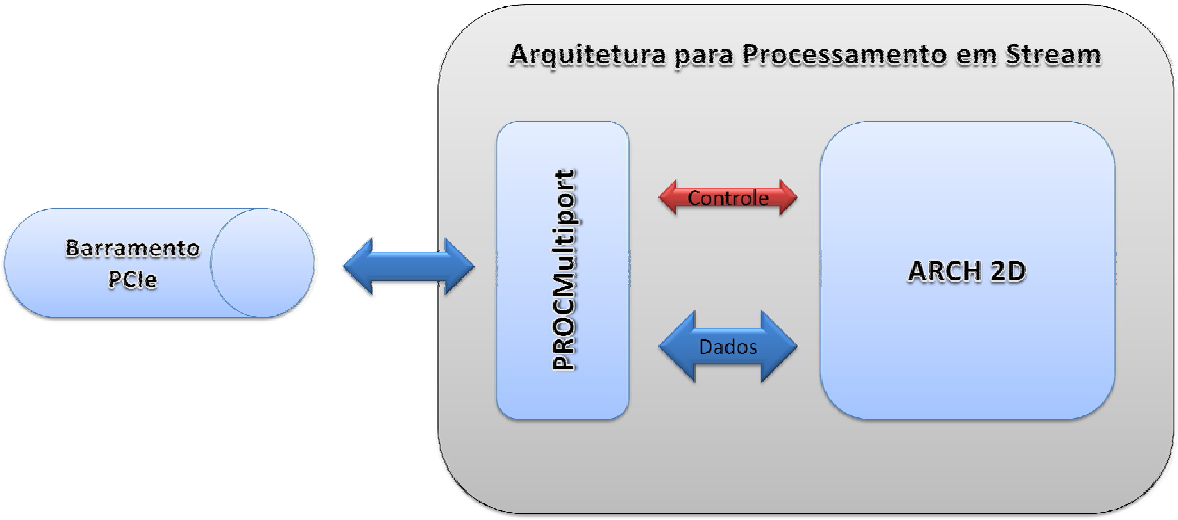


Figura 38 Diagrama de blocos da Arquitetura para Processamento em Stream.

Além do módulo *PROCMultiPort,* o módulo *ARCH 2D,* é o responsável pela interconexão de todos os módulos desenvolvidos para implementar a arquitetura. Como parte de *ARCH 2D,* a unidade *Control Unit* é a que controla toda a arquitetura. Esse módulo é responsável por controlar todo o fluxo de dados, realizando leituras e escritas, além de indicar o final do processamento de todos os dados.

Outros módulos presentes na arquitetura são:

a. os módulos *Ricker* e *Ricker PPF,* responsáveis pela inserção do vetor de pulso sísmico nas matrizes de pressão atual e anterior, ou seja, responsáveis pela implementação da fonte de ondas sísmicas no modelo.

b. o módulo *FIFO_SR,* que possui um conjunto de *FIFOs* e registradores de deslocamento (*Shift Registers - SR*) que implementam o reuso de dados da arquitetura.

c. o Núcleo de Processamento, que é o modulo que implementa a equação discretizada da equação de propagação de ondas. Na Arquitetura proposta nessa dissertação foram utilizados quatro Núcleos de Processamento para realizar a computação dos dados de entrada.

A Figura 39 mostra o diagrama de blocos da arquitetura interna do Módulo *ARCH 2D*, apresentando os módulos citados acima e como eles estão interligado.

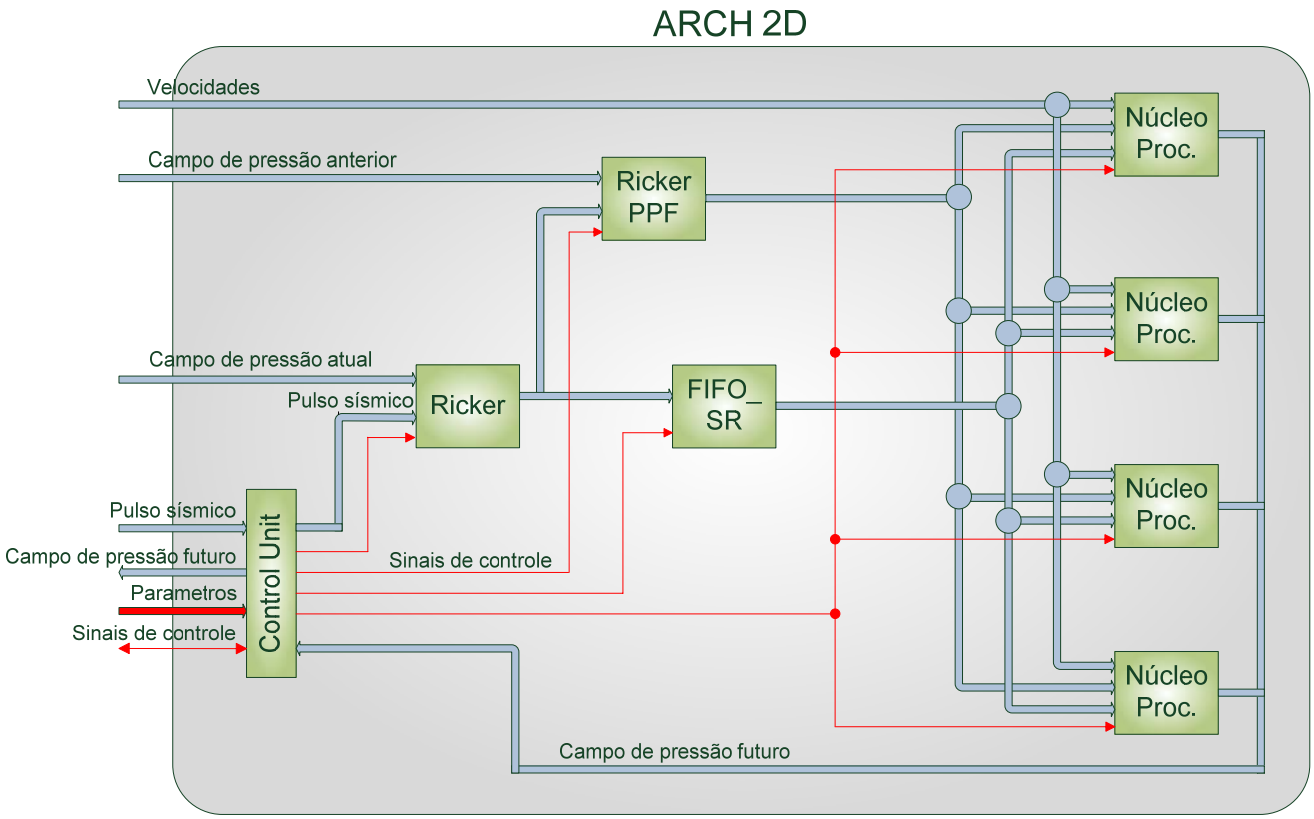


Figura 39 Diagrama de blocos da arquitetura interna do ARCH 2D.

5.3.2 Organização de dados na memória

Nessa seção será mostrado como foram organizados os dados das matrizes do campo de pressão atual, anterior e futuro, da matriz de velocidade e do vetor de pulso sísmico na memória. O objetivo dessa organização foi posicionar os dados na memória de forma que a leitura e escrita desses dados seja feita com a menor latência possível. Para atingir essa baixa latência, os dados foram organizados de uma forma que possibilita a utilização de transferências em modo *burst* entre a memória e seu controlador, ou seja, os dados são lidos da forma mais sequencial possível.

O controlador de memória utilizado na arquitetura desenvolvida nessa dissertação, como citado anteriormente, é o *PROCMultiPort*. Este controlador, descrito em detalhes na seção 4.2, permite a abstração de vários detalhes da implementação da memória, tais como tamanho de dados, frequência, etc., facilitando sua utilização.

A organização dos dados para melhor reuso e, por conseguinte, melhor desempenho no processamento, pode ser visto na Figura 40. Uma das características interessantes do módulo *PROCMultiPort* é que ele permite que um bloco de dados na memória tenha um tamanho diferente de um outro bloco de dados. No caso da organização mostrada na Figura 40, os dados do vetor de pulso sísmico possuem 32 *bits* e o restante dos dados possuem 128 *bits*, sendo o acesso transparente para o *Control Unit*. Cada posição da memória que possui 128 *bits* contem quatro dados de 32 bits, onde cada um desses dados representa um valor em ponto flutuante com precisão simples.

Os primeiros dados nos endereços mais baixos da memória são aqueles referentes ao vetor de pulso sísmico. Cada posição da memória possui um elemento do vetor. Em seguida, um bloco de dados iguais a zero separam o vetor de pulso sísmico do próximo conjunto de dados. O próximo conjunto de dados na memória são os referentes à matriz de pressão atual. Nessa região da memória temos os *slices* da matriz de pressão atual organizados de forma sequencial. Esses *slices* formam o vetor unidimensional, mostrado na Figura 37, que é enviado pelo aplicativo. Após o último *slice* dessa matriz, um novo bloco de zeros é inserido, delimitando os dados dessa matriz atual da próxima matriz.

Os próximos dados na memória são os referentes à matriz de pressão anterior. Esses dados seguem a mesma organização dos dados da matriz de pressão atual. Após os dados da matriz de pressão anterior temos o bloco de zeros e depois os dados da matriz de velocidades que também segue a mesma organização dos dados dessas últimas duas matrizes.

Todas as matrizes acima possuem o mesmo tamanho e o vetor de pulso sísmico possui a quantidade de elementos igual ao número de passos de processamento. A matriz de pressão futura é escrita no mesmo espaço de endereçamento da matriz de pressão anterior, já que essa última não é utilizada nos próximos passos de processamento. Assim, no final do passo de processamento, no local da matriz de pressão anterior, temos a matriz de pressão futura, que se torna a nova matriz de pressão atual, e a matriz de pressão atual se torna a nova matriz de pressão anterior. Dessa forma, essas duas matrizes ficam trocando de posições durante os passos de processamento. Isso possibilitou a utilização de apenas três espaços de endereçamento, ao invés de quatro, para armazenar as quatro matrizes de campo de pressão. A Figura 40 mostra como é feita a organização dos dados na memória.

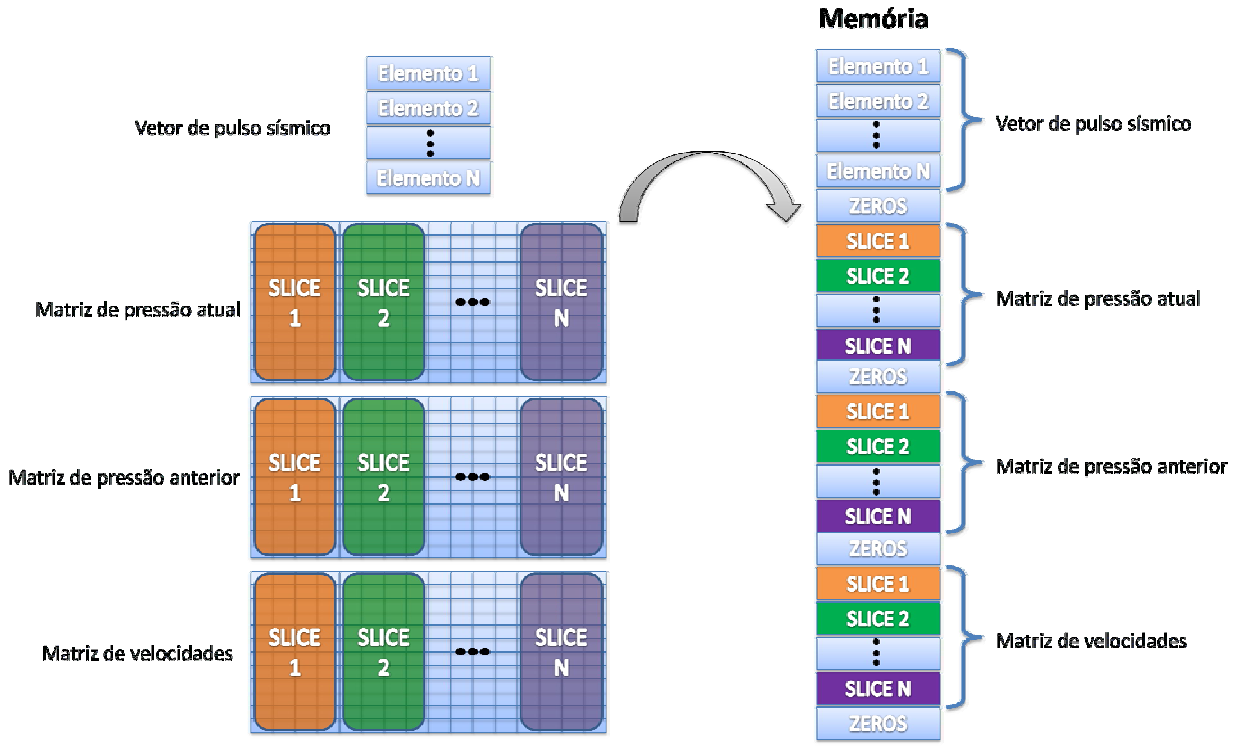


Figura 40 Organização de dados na memória.

5.3.3 A Utilização do PROCMultiPort

Nessa seção será mostrado como o módulo *PROCMultiPort* foi configurado e utilizado na arquitetura para processamento em *stream*. Como citado anteriormente, este *IP-core* da *GiDEL* foi utilizado nessa arquitetura para realizar todas as transferências de dados com a memória e com o barramento *PCIe*.

Para implementar a função de envio de parâmetros do aplicativo de Controle e Comunicação para a arquitetura, foram gerados registradores especiais no *PROCMultiPort*, para cada um dos parâmetros enviados pela aplicação. Os valores desses parâmetros são disponibilizados para a unidade de controle, através de portas de saídas do *PROCMultiPort*. A Figura 41 mostra as portas de saída do *PROCMultiport* que são utilizadas para enviar os parâmetros para a arquitetura, e as portas de entrada da arquitetura (*ARCH 2D*) que recebe esses parâmetros.

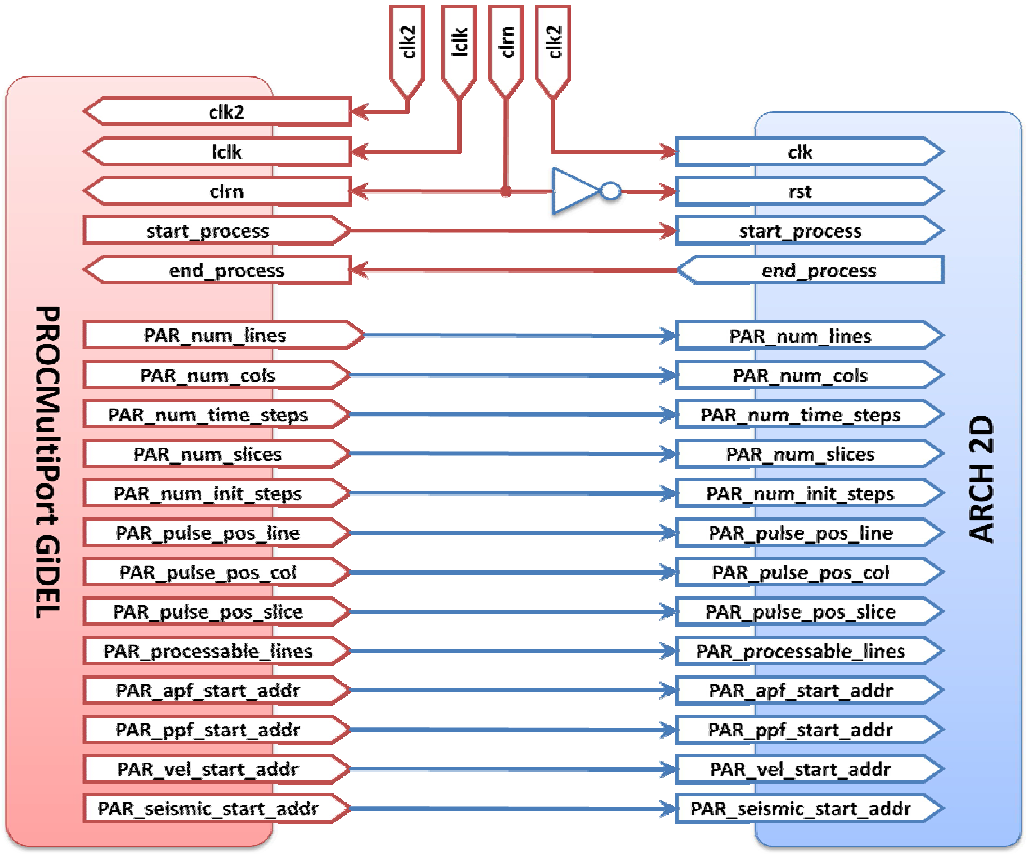


Figura 41 Portas de parâmetros, clock e reset.

Na Figura 41 também pode ser visto o sistema de *clock* utilizado na arquitetura. A placa possui um *clock* principal, o *clk0*, a partir do qual todos os outros *clocks* podem ser derivados. A frequência do *clk0* pode ser configurada na ferramenta *PROCWizard.* O valor configurado para esse *clock* foi de 100 MHz. O sinal *lclk* representa o *clock* do barramento local da placa, onde estão ligados o *FPGA* e o barramento *PCI*. A frequência do *lclk* é de 50 MHz. O sinal *clk2* é um *clock* auxiliar que pode ser utilizado como o *clock* em emulações mais lentas. Este sinal pode possuir uma frequência com o valor máximo igual à metade da frequência do *clk0*. A frequência desse *clock* é configurada na ferramenta *PROCWizard.* Seu valor configurado foi de 50MHz. O sinal de *reset* utilizado pela plataforma é definido como *clrn*, gerado a partir da chamada de uma função do *device driver*. Este sinal é ativo em nível lógico baixo.

Ainda na Figura 41 podem ser vistos os sinais utilizados na indicação do início e do término do processamento, que são os sinais *start_process* e *end_process*. O *start_process* é o sinal que indica para a arquitetura que o processamento pode ser iniciado, e o *end_process* é o sinal que indica para o aplicativo que a arquitetura finalizou o processamento.

As matrizes do campo de pressão atual e anterior, da matriz de velocidade e do vetor de pulso sísmico são lidas através de portas de leitura sequenciais no *PROCMultiPort,* uma porta para cada um conjuntos de dados. Da mesma forma, para se realizar a escrita da matriz do campo de pressão futuro, é utilizada uma porta de escrita sequencial no mesmo *PROCMultiPort*.

As portas de leitura sequencial utilizadas para realizar as leituras dos dados da matriz de pressão anterior, e atual, e da matriz de velocidade possuem uma largura de dados igual a 128 *bits*, equivalentes a quatro números ponto flutuante de precisão simples. Cada porta possui um sinal de status que informa ao módulo de controle quando sua *FIFO* interna está ficando vazia. Já a porta de leitura do vetor de pulso sísmico possui uma largura de dados igual a 32 *bits*, possuindo o mesmo sinal de status.

A Figura 42 mostra as portas dos módulos *PROCMultiPort* e *ARCH 2D* que implementam as funcionalidades das portas de leitura da matriz de pressão atual, anterior e da matriz de velocidades. Na parte superior da Figura 43 pode ser visto as portas dos módulos *PROCMultiPort* e *ARCH 2D* que implementam as funcionalidades da porta de leitura do vetor do pulso sísmico.

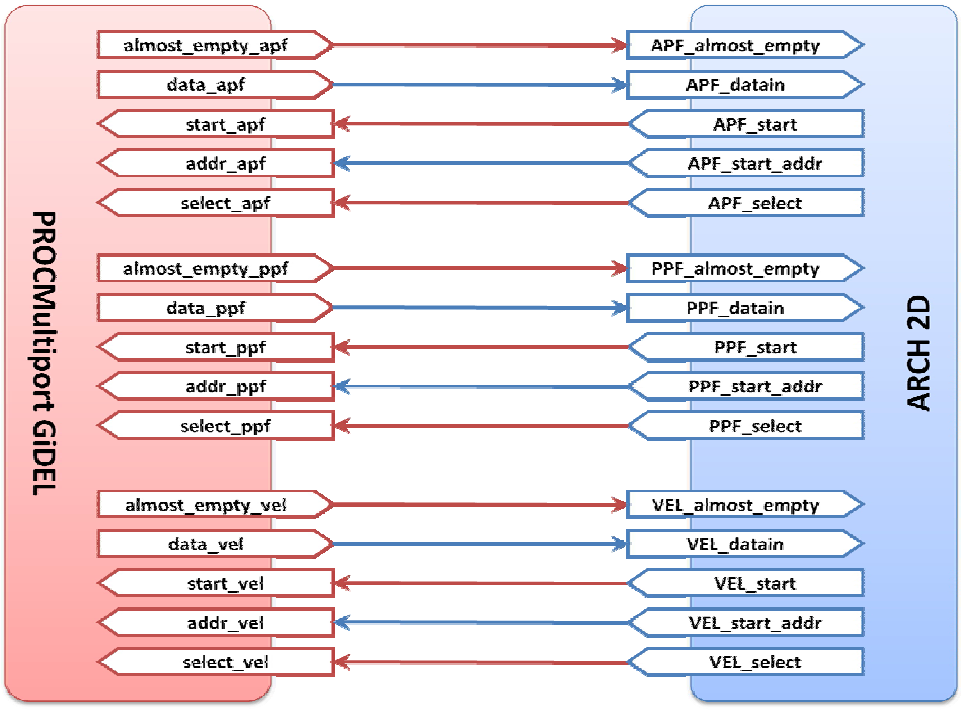


Figura 42 Conjunto de portas que forma as Portas de Leitura Sequencial das matrizes de pressão atual, anterior e de velocidade.

A escrita da matriz do campo de pressão futuro é realizada através de uma porta de escrita sequencial, de largura igual 128 bits (quatro palavras de 32 bits), correspondente a quatro números ponto flutuante com precisão simples. Dois sinais de *status* desta porta indicam as condições de sua *FIFO* interna, mostrando se a mesma está quase cheia ou vazia. Na parte inferior da Figura 43 pode ser visto as portas dos módulos *PROCMultiPort* e *ARCH 2D* que implementam as funcionalidades da porta de escrita da matriz do campo de pressão futuro.

Todas as portas acima possuem a mesma configuração de *clock*, com um valor igual a 50 MHz.

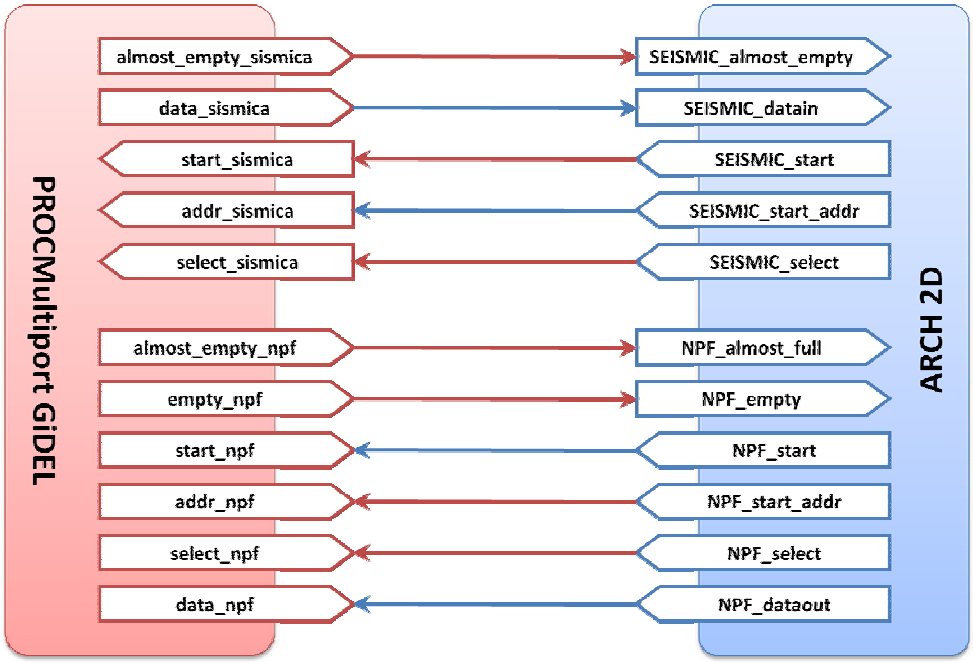


Figura 43 Conjunto de portas que formam a Porta de Leitura Sequencial do vetor de pulso sísmico e a Porta de Escrita Sequencial da matriz do campo de pressão futuro.

O módulo *Control Unit* é o modulo que faz o controle de todas as portas descritas acima. Esta unidade é a responsável por manter o fluxo de dados o mais constante possível na arquitetura, de forma que, a cada ciclo de *clock*, os núcleos de processamento tenham dados válidos em suas portas para serem processados.

5.3.4 FIFOs, Shift Registers e o reuso de dados

O módulo *FIFO_SR* é o módulo responsável por implementar todo o reuso de dados que a arquitetura utiliza e também por disponibilizar os dados da matriz do campo de pressão atual, organizados de maneira correta, para os quatro núcleos de processamento.

O módulo *FIFO_SR* é composto por uma janela de registradores e por um conjunto de *FIFOs* [39]. O modulo possui seis *FIFOs*, onde cada uma dessas *FIFOs* tem como objetivo armazenar uma coluna inteira da matriz. Cada uma delas, por sua vez, está ligada a um conjunto de registradores que forma uma coluna de registradores de deslocamento. A Figura 44 mostra a arquitetura interna do *FIFO_SR*.

Na Figura 44 pode ser vista a interconexão das *FIFOs* com os registradores e como os próprios registradores estão interligados, formando os registradores de deslocamento, onde o valor de um registrador e passado para o seguinte a cada pulso do relógio. Esses registradores formam a janela de convolução que é utilizada no método de diferenças finitas de quarta ordem quando quatro núcleos de processamento são colocados lado a lado. Como citado anteriormente, o número máximo de linhas das matrizes que podem ser armazenados nessas *FIFOs* é de 8192. Estes registradores estão organizados de modo a fornecer os dados para os quatro núcleos de processamento da arquitetura. Na Figura 44, com os nomes NP1, NP2, NP3 e NP4, é mostrada a localização dos núcleos de processamento na janela de registradores. Os dados necessários para que esses núcleos de processamento possam realizar suas computações são armazenados nos registradores mostrados na Figura 44, esses registradores estão nomeados com a palavra *out* e um índex que indica sua linha e coluna na janela de registradores.

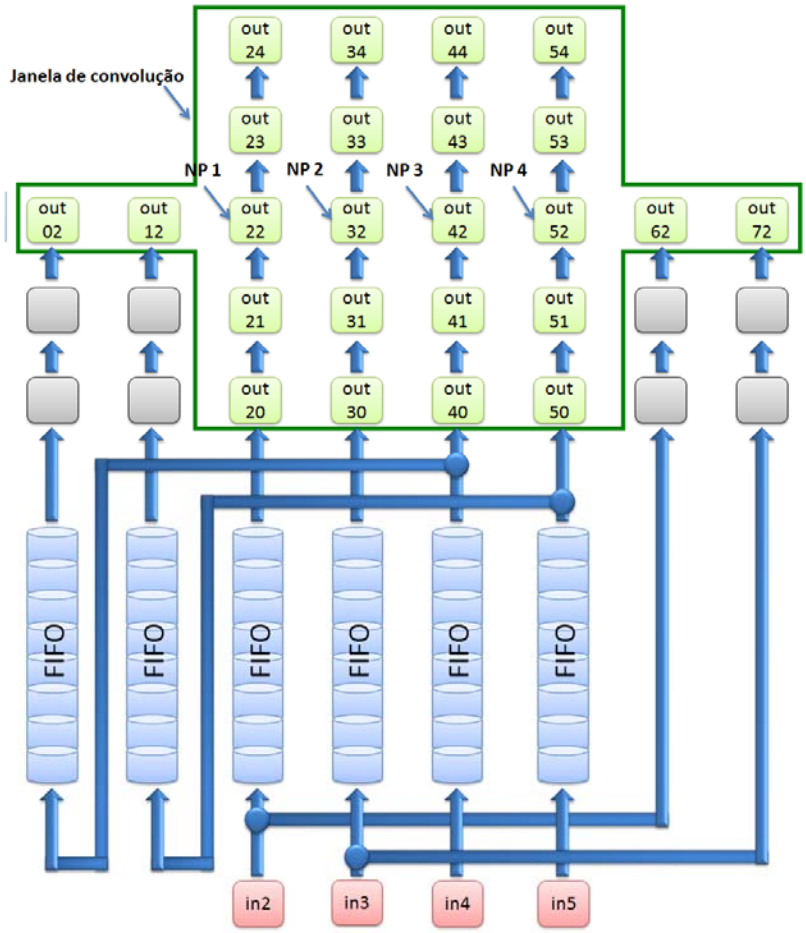


Figura 44 Arquitetura interna do FIFO_SR.

A ideia por trás do fluxo de dados desse módulo é fazer com que seja necessária apenas uma leitura da matriz de pressão atual da memória para que a computação de quatro valores da matriz do campo de pressão futuro possa ser realizada. A Figura 45 mostra essa idéia. Após o armazenamento dos dois primeiros *slices* da matriz de pressão atual nas *FIFOs* do *FIFO_SR*, a cada leitura de uma linha do terceiro *slice*, uma computação pode ser realizada. Na verdade, as quatro primeiras leituras ainda não possibilitaram a computação, pois nelas a janela de registradores ainda não está completa com dados válidos. Após a quinta leitura, a janela de registradores está completa, fazendo com que cada leitura possibilite uma computação de uma linha do segundo *slice* da matriz do campo de pressão futuro. Pode ser visto na Figura 45 como é realizada a movimentação da janela de registradores (janela de convolução) sobre a matriz de pressão atual.

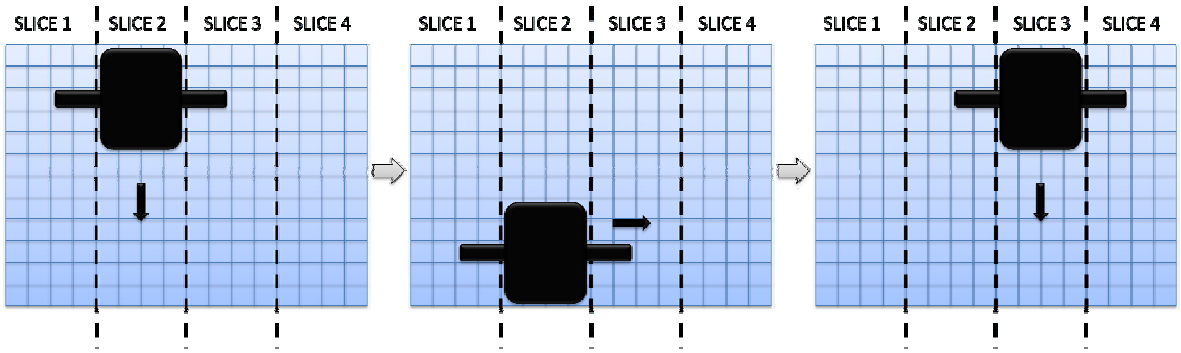


Figura 45 Movimentação da janela de registradores.

O controle desse fluxo de dados dentro da arquitetura interna do *FIFO_SR* é feito através de uma máquina de estados. O objetivo dessa máquina de estado é reutilizar os dados da matriz de pressão atual, de forma que um dado dessa matriz seja lido apenas uma única vez da memória para cada passo de processamento. O outro objetivo dessa máquina é fornecer os dados de maneira correta para os núcleos de processamento. Essa maquina de estados segue o fluxograma apresentado na Figura 46.

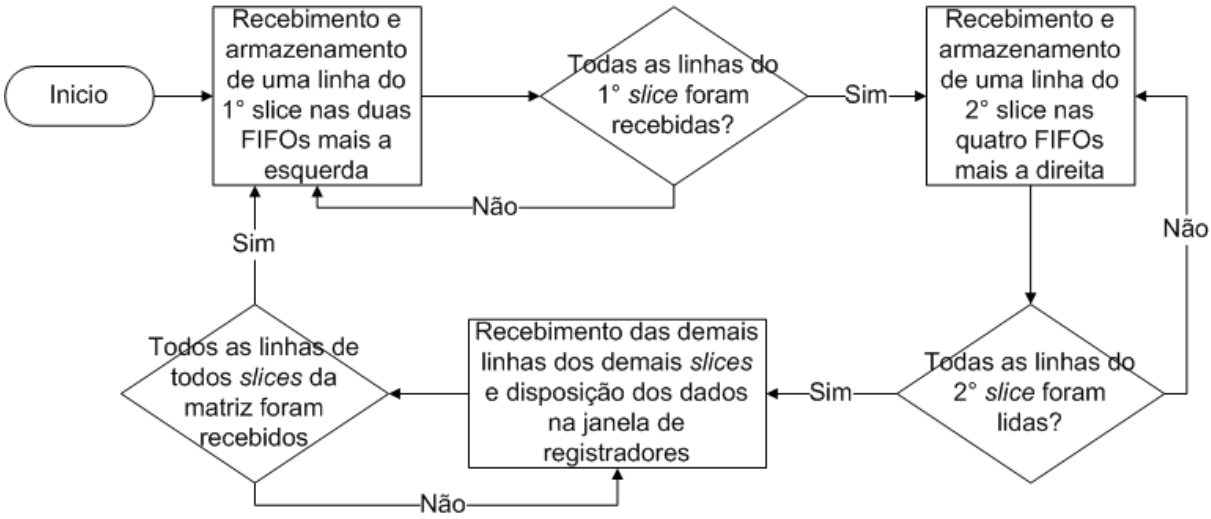


Figura 46 Fluxograma da máquina de estados do módulo FIFO Shift Registers.

Assim, pode-se notar que é através da interconexão das *FIFOs* que o reuso dos dados é implementado, fazendo com que os dados da matriz de pressão atual sejam lidos apenas uma vez da memória.

5.3.5 Inserção do pulso Sísmico

A fonte de ondas sísmicas é responsável por fornecer a energia necessária para que a sondar o terreno cuja estrutura geofísica pretende-se levantar. O vetor que representa essa fonte de onda é gerado na aplicação, e a cada passo de processamento um elemento desse vetor deve ser inserido na matriz de pressão atual. Essa inserção é feita através da adição do elemento do vetor com um elemento da matriz onde o pulso será inserido. O resultado dessa adição é repassado para os núcleos de processamento.

A estratégia desenvolvida nessa dissertação foi armazenar todo o vetor do pulso sísmico na memória, e a cada passo de tempo (passo de processamento) um elemento desse vetor é lido da memória e armazenado em um registrador para ser adicionado à matriz de pressão atual. Dessa forma, à medida que os dados da matriz de pressão atual são lidos da memória, é verificado se uma das quatro palavras que foram lidas, já que uma leitura de um dado da matriz de pressão atual fornece quatro palavras, representa a posição na matriz que o usuário escolheu para ser colocada a fonte de ondas sísmicas. Caso uma dessas palavras seja essa posição, o elemento do vetor do pulso sísmico armazenado no registrador é adicionado à palavra da matriz de pressão atual.

O controle da verificação de qual palavra deve ser adicionado o vetor de pulso sísmico na matriz de pressão atual é realizado pelo módulo *Control Unit*, e a adição é realizada pelo módulo *Ricker*. Assim, como o controle da verificação de qual palavra deve ser substituída na matriz de pressão anterior é realizado pelo módulo *Control Unit*, e a substituição é realizada pelo módulo *Ricker PPF*. Os módulos *Ricker* e *Ricker PPF* são apresentados a seguir.

5.3.5.1 Módulo *Ricker*

O módulo de *Ricker* tem como objetivo inserir o pulso sísmico na matriz de pressão atual a cada passo de processamento. Para realizar essa tarefa o módulo recebe como entrada o valor atual do pulso, as quatro palavras que representam um dado que foi lido da matriz de pressão atual, um sinal que indica qual das quatro palavras será adicionada ao pulso, e um sinal

indicando quando inserir (adicionar) esse pulso. O resultado da computação são quatro palavras, onde uma dessas palavras foi adicionada ao pulso.

A arquitetura interna do módulo *Ricker* é formada basicamente de multiplexadores, registradores de deslocamento e um adicionador em ponto flutuante com precisão simples [40]. Esse adicionador foi gerado no *MegaWizard Plug-Ins* da *Altera* [41]. A Figura 47 apresenta o diagrama de blocos da arquitetura interna do módulo de *Ricker*.

Para alcançar o objetivo de inserir o pulso sísmico na matriz de pressão atual, o módulo precisa realizar algumas tarefas, quais sejam:

- Armazenar um valor do pulso sísmico no registrador interno: o primeiro passo da inserção do pulso sísmico é armazenar o valor do pulso sísmico que será usado no passo de processamento atual em um registrador interno do módulo *Ricker*. A escrita nesse registrador só é habilitada quando o sinal *wr_seismic_pulse* é ativado, fazendo com que o valor da porta *seismic_pulse_value [31:0]* seja armazenado no registrador.
- Selecionar o valor que será adicionado a uma das quatro entradas: o módulo sempre está adicionando um valor a uma das quatro palavras de 32 bits que entram no mesmo. O valor que é adicionado a uma dessas palavras é definido pela saída do multiplexador ao qual o registrador interno está ligado. Como o pulso sísmico é adicionado a apenas uma posição da matriz, a cada passo de processamento, o valor do registrador só é utilizado nesse momento, acionado através do sinal *pulse_position_flag.* Nos demais momentos, o valor adicionado a uma das quatro palavras é zero.
- Selecionar uma das quatro entradas para ser somada ao pulso sísmico: essa seleção é feita de acordo com a entrada *pulse_position_sel [1:0]*, que é o sinal de seleção do multiplexador no qual as entradas estão ligadas. Este sinal de seleção representa os dois bits menos significativos do parâmetro (*PAR_pulse_pos_slice [15:0]*), que a arquitetura de controle recebe como entrada, e que indica a coluna onde o pulso sísmico deve ser inserido, com a seguir:
    - 00: a entrada *data_in_0 [31:0]* é adicionada ao pulso sísmico.
    - 01: a entrada *data_in_1 [31:0]* é adicionada ao pulso sísmico.
    - 10: a entrada *data_in_2 [31:0]* é adicionada ao pulso sísmico.
    - 11: a entrada *data_in_3 [31:0]* é adicionada ao pulso sísmico.

- Sincronização: um atraso de processamento é realizado para sincronização da saída do módulo adicionador ponto flutuante com precisão simples, com as demais entradas que não foram adicionadas. Um atraso de sete *clocks* é aplicado através de registradores de deslocamento, a todas as entradas, já que o pulso sísmico pode ser aplicado em qualquer uma delas.
- Selecionar as saídas: após todas as tarefas anteriores terem sido realizadas, deve ser realizada a seleção das saídas. A seleção é feita através de quatro multiplexadores e um módulo que faz o controle desses multiplexadores. Cada um desses multiplexadores seleciona entre uma das saídas dos registradores de deslocamento e a saída do adicionador. Por exemplo, caso o pulso sísmico tenha sido adicionado à entrada *data_in_0 [31:0]*, então o módulo de controle faz a seleção dos multiplexadores de forma que a saída do somador mais as saídas dos registradores de deslocamento das entradas *data_in_1 [31:0]*, *data_in_2 [31:0]* e *data_in_3 [31:0]* sejam repassados para as saídas *data_out_0 [31:0]*, *data_out_1 [31:0]*, *data_out_2 [31:0]* e *data_out_3 [31:0]*, respectivamente.

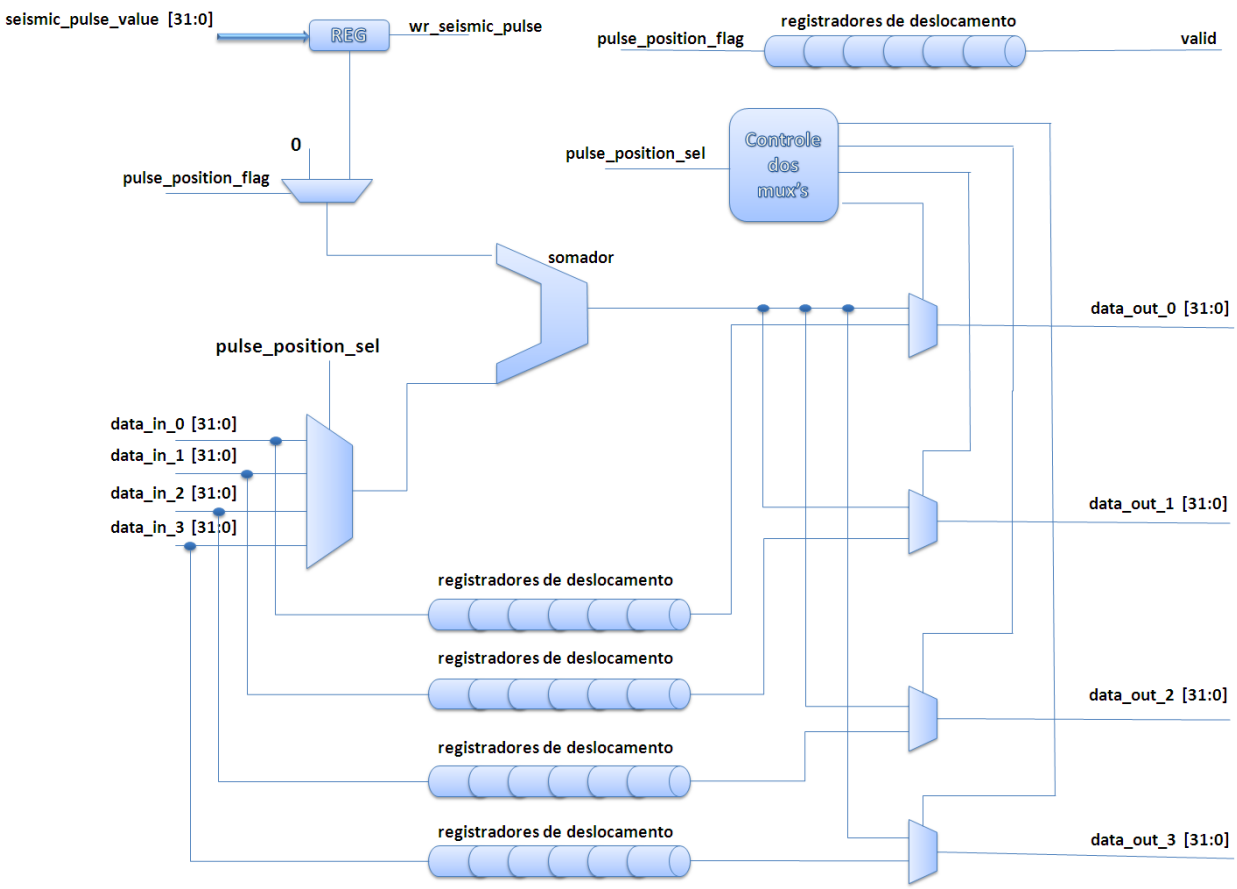


Figura 47 Arquitetura interna do módulo Ricker.

5.3.5.2 Módulo *Ricker PPF*

O módulo *Ricker PPF* tem como objetivo inserir o pulso sísmico na matriz de pressão anterior a cada passo de processamento, com exceção do primeiro. O *Ricker*, que é responsável pela inserção do pulso sísmico na matriz de pressão atual, não escreve na memória o resultado do seu processamento, o que faz com que, a matriz de pressão atual fique inconsistente com o valor que foi passado para os Núcleos de Processamento. Assim, para cada passo de processamento o módulo *Ricker PPF* guarda o valor de saída do módulo *Ricker* para utilizá-lo no próximo passo de processamento.

A arquitetura interna desse módulo é formada por um multiplexador, registradores e uma máquina de estados que controla a escrita de alguns desses registradores. A Figura 48 mostra o diagrama de blocos dessa arquitetura.

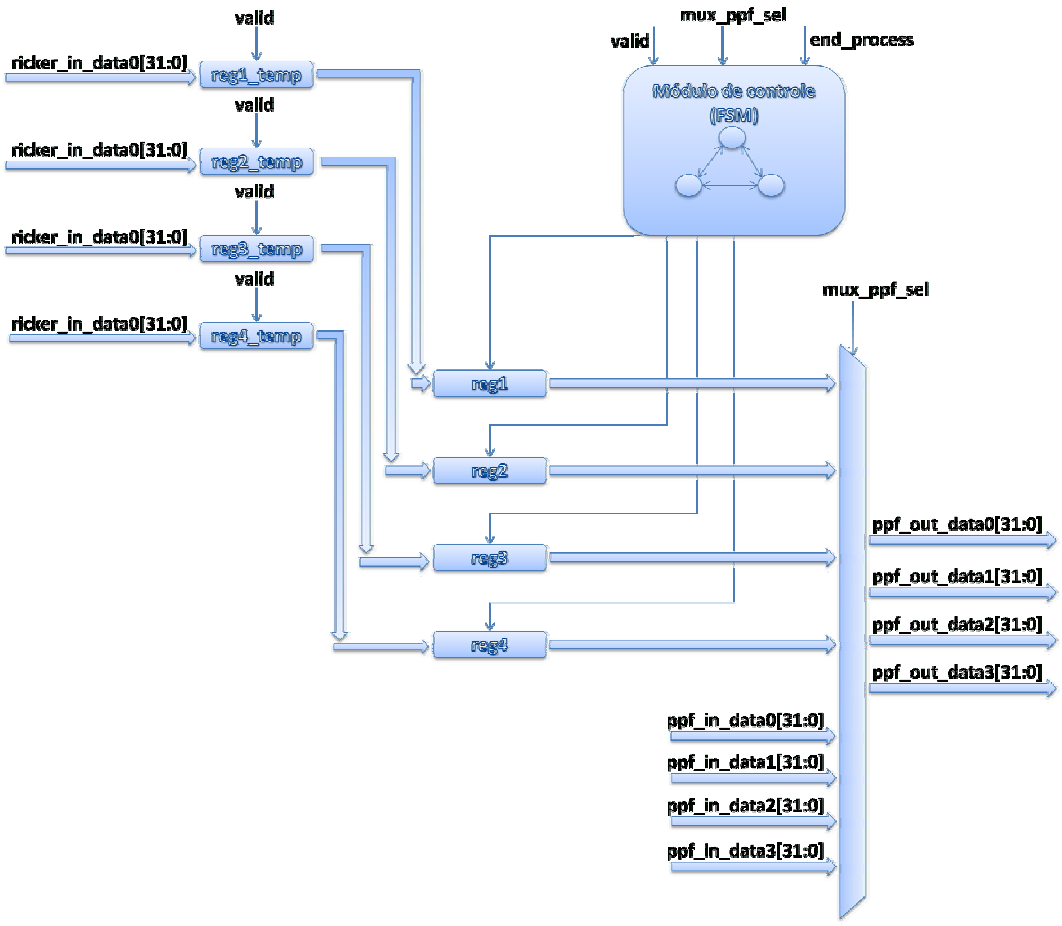


Figura 48 Arquitetura interna do módulo Ricker PPF.

O processo de atualização da matriz de pressão anterior é descrito em seguida, lembrando-se que, o controle do módulo *Ricker PPF* é feito pelo *Control Unit*:

- Armazenar a saída do módulo *Ricker* nos registradores temporários: inicialmente as saídas do módulo de *Ricker* são armazenadas em registradores temporários. A escrita nesses registradores só é habilitada quando o sinal *valid* é ativado, fazendo com que o valor das portas "ricker_in_data0 [31:0]", "ricker_in_data1 [31:0]", "ricker_in_data2 [31:0]" e "ricker_in_data3 [31:0]" sejam armazenado nos registradores. A porta de entrada *valid* do módulo *Ricker PPF* está ligada a porta de saída *valid* do módulo *Ricker*. Ou seja, o módulo *Ricker PPF* só armazena em seus registradores temporários o valor da matriz de pressão atual que foi adicionado ao pulso sísmico.
- Armazenar nos registradores ligados ao multiplexador os valores que estão nos registradores temporários: a escrita dos valores dos registradores temporários nesses registradores é controlada pela máquina de estados mostrada na Figura 49. O funcionamento dessa máquina é mostrado a seguir.

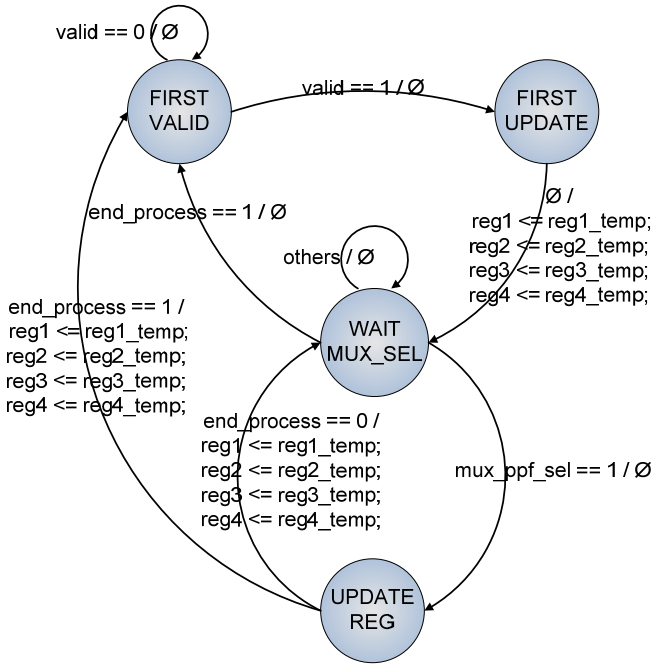


Figura 49 Máquina de estados que atualiza os registradores ligados ao multiplexador.

- FIRST_VALID: este estado é o estado inicial da máquina de estados. Ele espera pelo sinal de *valid* do módulo *Ricker* que é ativo no primeiro passo de processamento. Quando esse sinal é ativado, indicando que a saída do módulo de *Ricker* foi

adicionada ao pulso sísmico, a máquina de estados faz a transição para o estado FIRST UPDATE.

- FIRST_UPDATE: nesse estado os registradores são atualizados com os valores dos registradores temporários. A máquina faz então a transição para o estado WAIT MUX_SEL.

O objetivo desses dois primeiros estados é armazenar o valor da matriz de pressão atual adicionado ao valor do pulso sísmico do primeiro passo de processamento. Esse valor será usado no próximo passo de processamento.

- WAIT_MUX_SEL: o objetivo desse estado é esperar pela ativação do sinal de seleção *mux_sel_ppf*, que é ativado pelo módulo *Control Unit*, para que no próximo estado, os valores dos registradores sejam atualizados com os valores temporários.

- UPDATE_REG: nesse estado os registradores são atualizados com os valores dos registradores temporários. Caso o sinal *end_process* esteja ativado, a máquina faz a transição para o estado FIRST_VALID, caso ele esteja desativado a máquina faz a transição para o estado WAIT_MUX_SEL. O objetivo desses dois últimos estados é ficar atualizando os valores dos registradores internos com os valores dos registradores temporários a cada passo de processamento, para que eles possam ser utilizados na saída do módulo no próximo passo de processamento.

- Selecionar quais dos dois conjuntos de entradas serão passados para as saídas: a seleção do multiplexador, que seleciona as saídas ppf_data_out0 [31:0], ppf_data_out1 [31:0], ppf_data_out2 [31:0] e ppf_data_out0 [31:0] do módulo, é feita de acordo com o sinal *mux_sel_ppf*. Caso ele esteja ativado as saídas receberão o valor dos registradores reg1, reg2, reg3 e reg4 respectivamente, caso ele não esteja ativado as saídas recebem os valores das entradas ppf_in_data0 [31:0], ppf_in_data1 [31:0], ppf_in_data2 [31:0] e ppf_in_data3 [31:0] respectivamente. Ou seja, quando o sinal *mux_sel_ppf* é ativado o significa que o pulso sísmico deve ser inserido na matriz de pressão anterior.

5.3.6 Núcleo de Processamento

O Núcleo de processamento é o módulo da arquitetura para processamento em *stream* responsável por implementar a equação de diferenças finitas (2.2), mostrada na seção 2.1.2.

Este módulo é ativado assim que um pulso de relógio é aplicado em sua porta. O núcleo possui um *reset* assíncrono, e possui uma arquitetura organizada em *pipeline*, com uma profundidade de 30 estágios. O núcleo de processamento gera valores em sua saída a cada ciclo de relógio. Contudo, um sinal de controle é responsável por sinalizar se esta saída é válida ou não. O fluxo funciona da seguinte forma: o núcleo de processamento recebe os dados de entrada e uma sinalização de que estes dados são válidos; após os 30 estágios do *pipeline* um dado de saída é computado e o núcleo de processamento sinaliza a validade deste dado. Cada saída válida corresponde a um ponto da matriz do campo de pressão futuro.

5.3.7 Unidade de Controle

O *Control Unit* é a Unidade de Controle da arquitetura. Ele é responsável por controlar todo o fluxo de dados entre os elementos do sistema, a saber: fornecer os dados das matrizes de entrada para os Núcleos de Processamento, lendo esses dados da memória, através do *PROMultiPort*, e é responsável por escrever os dados gerados pelo Núcleo de Processamento de volta para a memória, também através do *PROCMultiPort*.

O *Control Unit* foi desenvolvido visando uma arquitetura para processamento em *stream* parametrizável, permitindo ao usuário escolher o número de linha e colunas das matrizes de entradas, o número de passo de processamentos, a linha e a coluna onde o pulso sísmico é inserido. Para implementar essa parametrização, o *Control Unit* recebe vários parâmetros que são enviados pelo aplicativo de Controle e Comunicação. Como citado anteriormente esses parâmetros são enviados à *Control Unit*, residente no *FPGA*, através do barramento *PCIe* e do *PROCMultiPort*.

A seguir será mostrado todo o fluxo de dados da Arquitetura. Esse fluxo pode ser dividido em dois subfluxos: o primeiro e mais importante é o subfluxo de dados relativo à

leitura das matrizes de campo de pressão, de velocidade e do vetor de pulso sísmico, e a distribuição desses dados para os Núcleos de Processamento. O segundo subfluxo de dados é o relativo à escrita dos dados gerados pelos Núcleos de Processamento na memória. Cada um desses fluxos possui uma máquina de estados que os controla. Além dessas duas máquinas de estados, existem mais duas máquinas, onde uma controla a inserção do pulso sísmico na matriz de pressão atual, e a outra controla a inserção do pulso sísmico na matriz de pressão anterior.

5.3.7.1 Fluxo dos dados lidos da memória.

Nessa seção será explicado o fluxo dos dados que saem da memória e chegam até os Núcleos de Processamento. A Figura 50 mostra o fluxograma que é implementado pela máquina de estado que controla esse fluxo de dados. A seguir está a descrição de cada uma dos passos desse fluxograma.

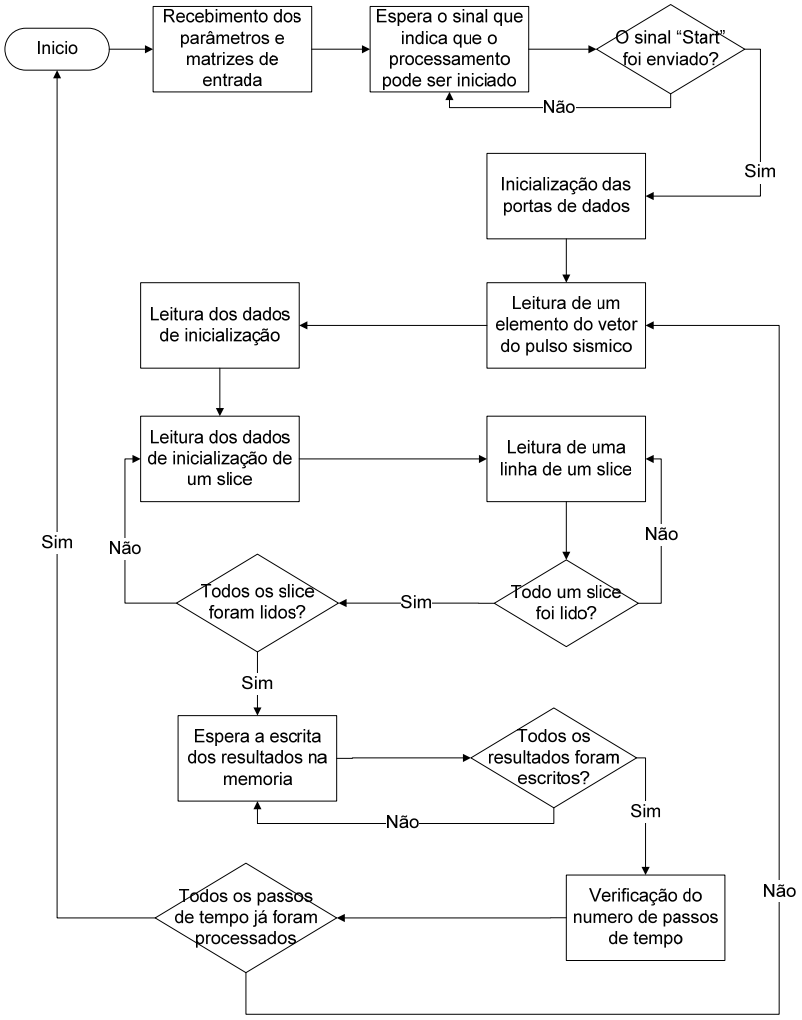


Figura 50 Fluxograma implementado pela FSM que realiza a leitura dos dados.

O primeiro passo do fluxo é o recebimento dos parâmetros e dos dados de entrada. Os parâmetros iniciais são passados pelo usuário da plataforma através do Aplicativo de Controle e Comunicação. Esses parâmetros representam o número de linhas e colunas da matriz de entrada, o número de passo de processamento, a linha e a coluna onde o pulso sísmico será inserido. A partir desses cinco parâmetros iniciais, vários outros parâmetros são gerados na aplicação, assim como, as matrizes de entrada, e o vetor do pulso sísmico.

Todos os parâmetros são repassados para a arquitetura através do módulo *PROCMultiPort*. As matrizes e o vetor do pulso sísmico são armazenados na memória, também, através do *PROCMultiport*. Após o envio de todos esses dados o Aplicativo sinaliza para o *Control Unit*, através do *PROCMultiport* (ativa o sinal *start_process*), que o processamento pode ser iniciado.

Quando o *Control Unit* recebe o sinal indicando que o processamento pode ser inicializado, sua máquina de estados de leitura é ativada, permitindo o processamento de todo o fluxo de dados.

O primeiro passo do *Control Unit* é inicializar as portas de leitura das matrizes de pressão atual e anterior, da matriz de velocidade e do vetor de pulso sísmico, e inicializar a porta de escrita do campo de pressão futuro.

O próximo passo é ler o valor do pulso sísmico que será utilizado nesse passo de processamento.

O próximo passo do *Control Unit* é fazer a leitura dos dados de inicialização. Nesse passo o *Control Unit* faz à leitura dos primeiros dois *slices* da matriz de pressão atual. Como citado na seção que explica o FIFO_SR, esses dois *slices* são armazenados nas *FIFOs* internas do *FIFO_SR*. Nesse passo de inicialização o *Control Unit* também faz a leitura de um *slice* menos duas linhas da matriz de pressão anterior e de velocidades. Antes dos dados da matriz de pressão atual serem enviados para o módulo FIFO_SR, eles passam primeiro pelo módulo *Ricker*, pois o pulso sísmico pode estar em um desses dois primeiros *slices*. Na verdade todos os dados da matriz de pressão atual passam pelo módulo *Ricker*. Os dados da matriz de pressão anterior e de

velocidade são descartados, já que eles não são utilizados no processamento. A visualização das leituras realizadas nesse passo nas matrizes pode ser vista na Figura 51, essas leituras estão destacadas na figura com o nome de INIT.

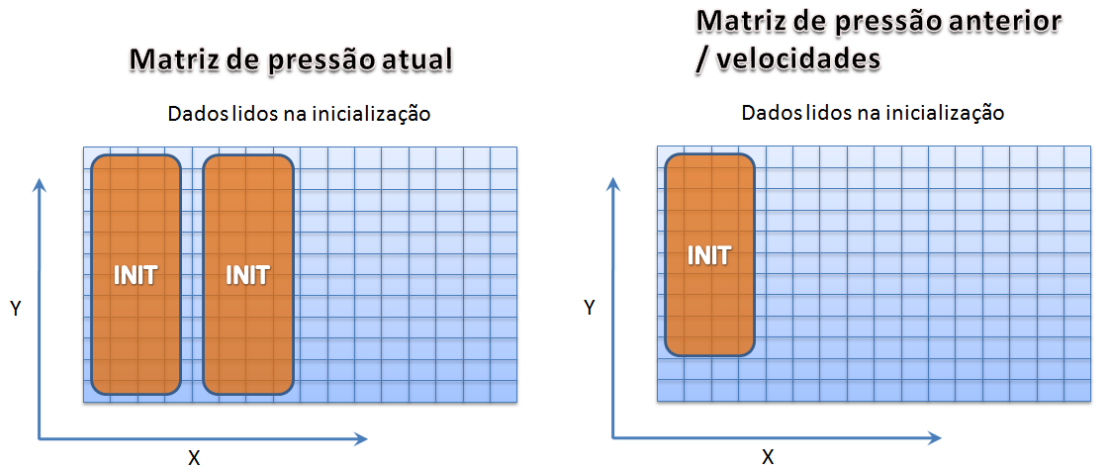


Figura 51 Dados lidos das matrizes de entradas na inicialização.

O próximo passo é a inicialização do *slice*, nesse passo o *Control Unit* faz a leitura de quatro linhas de um *slice* das matrizes de pressão atual e anterior, e da matriz de velocidade. Na matriz de pressão atual essas linhas estão no mesmo *slice*, já nas matrizes de pressão anterior e na matriz de velocidade duas dessas quatro linhas estão no final de um *slice*, e as outras duas no inicio do *slice* adjacente. As leituras da primeira inicialização de um slice podem ser vista na Figura 52, destacadas como com o nome INIT SLICE. É nesse passo que o *Control Unit* prepara a janela de registradores do *FIFO_SR* de forma que a partir da próxima leitura de dados a computação do núcleo de processamento pode ser iniciada. Os valores da matriz de pressão atual vão para o módulo *Ricker* e depois passam para o módulo *FIFO_SR*. Os valores da matriz de pressão anterior e da matriz de velocidades são descartados.

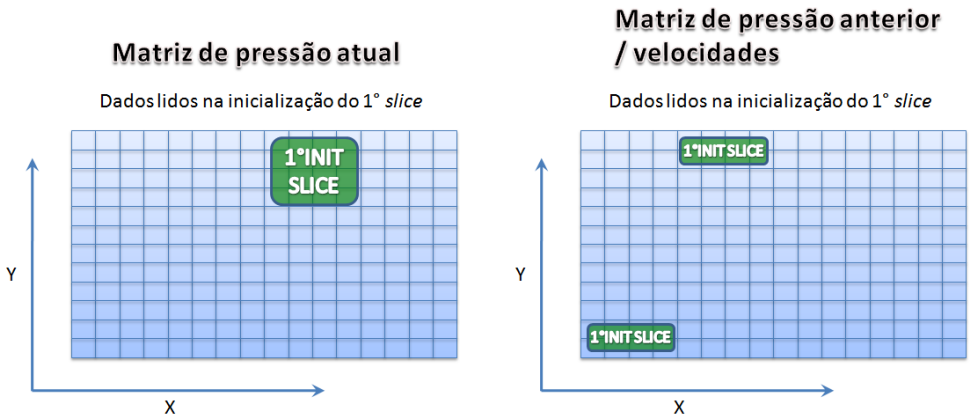


Figura 52 Dados lidos das matrizes de entrada no 1° *init slice*.

O passo seguinte é o chamado de passo de processamento. Nesse passo o *Control Unit* faz a leitura de um dado da memória de cada uma das matrizes e indica para os Núcleos de Processamentos que os dados que estão na janela de registradores do *FIFO_SR* são dados válidos. A janela de registradores desse módulo está ligada as entradas dos Núcleos de Processamento. Os valores da matriz de pressão atual são passados para o módulo de *Ricke*r, depois passam para o *FIFO_SR*, e depois para os Núcleos de Processamento. Os valores da matriz de pressão anterior são passados para o módulo *Ricker PPF*, e depois são passados para os Núcleos de Processamento. Os valores da matriz de velocidade são passados para os Núcleos de Processamento. O *Control Unit* permanece lendo dados e indicando para o núcleo de processamento que há dados validos em suas portas até ele ler a ultima linha do *slice* da matriz de pressão atual. A quantidade de dados lidos nesse passo é a mesma para as três matrizes, porem na matriz de pressão anterior e na matriz de velocidade as leituras não chegam ao final da matriz, já que a leitura dessas duas matrizes é iniciada e terminada em linhas diferentes nesses três passos quando compara a matriz de pressão atual. A Figura 53 mostra os dados que são lidos no primeiro passo de processamento, eles estão destacados com o nome PROCESS. Quando o *Control Unit* finaliza a leitura de todas as linhas de um *slice* da matriz de pressão atual, ele volta para o passo de inicialização de *slice*, caso ainda existam *slices* para serem lidos, e fica nesse loop (inicialização de *slice* e processamento) até o final da matriz.

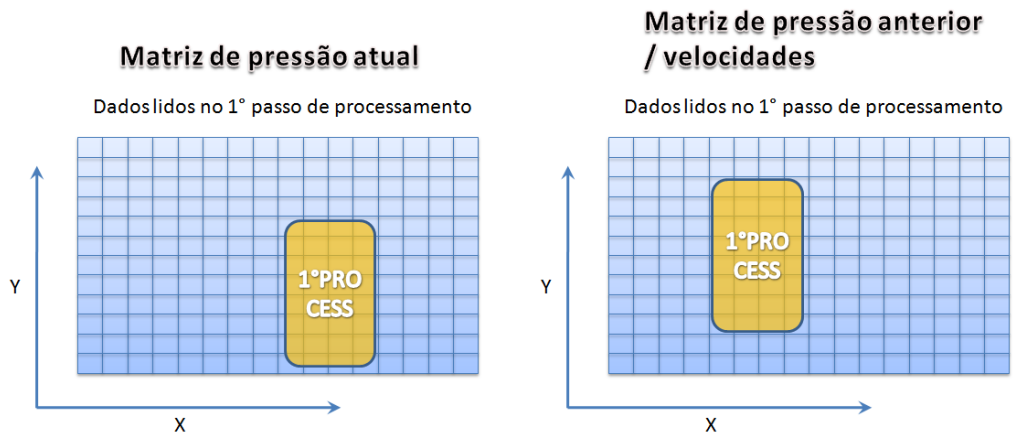


Figura 53 Dados lidos das matrizes de entrada no 1° passo de processamento.

Quando todos os *slices* da matriz de pressão atual são lidos, ou seja, todos os dados que são necessários para a geração da matriz do campo de pressão futuro já foram entregues aos Núcleos de Processamento, o *Control Unit* vai para um estado que espera que todos os dados que foram gerados pelos Núcleos de Processamento tenham sido escritos na memória. Isso é necessário, já que a parte da memória que possui dados da matriz de pressão atual se torna a

parte da memória que terá os dados da matriz de pressão anterior no próximo passo de tempo (passo de processamento), e a parte da memória que tem os dados da matriz do campo de pressão futuro se torna a parte da memória que terá os dados da matriz de pressão atual no próximo passo de processamento.

Quando todas as escritas são realizadas o *Control Unit* vai para um estado que verifica se ainda existem passos de tempo (passos de processamento) a serem realizados. Caso ainda existam passos de processamentos para serem realizados, ele volta para o passo de inicialização das portas, caso contrário, o *Control Unit* volta para o estado inicial indicando para o Aplicativo que o processamento foi terminado (*end_process*), e fica esperando que o Aplicativo indique que uma nova modelagem deve ser iniciada.

Na Figura 54 pode ser visto como cada um dos passos do fluxo de leitura realiza as leituras dos dados das matrizes de entrada, para uma matriz com 12 linhas e 16 colunas, em cada um dos passos de tempo.

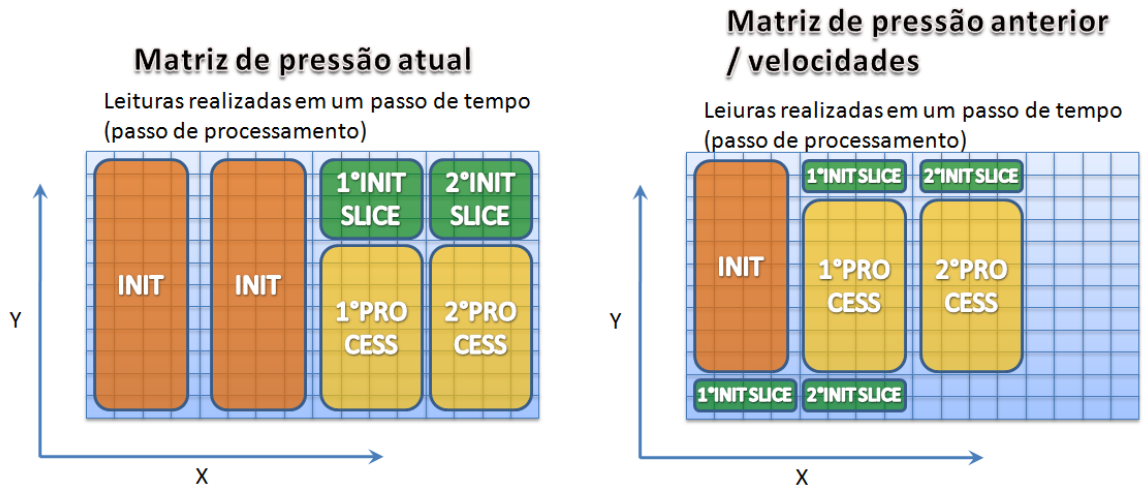


Figura 54 Leitura dos dados das matrizes de entrada em um passo de tempo.

5.3.7.2 Fluxo dos dados gerados pelos Núcleos de Processamento

Nessa seção será explicado como os dados resultantes dos Núcleos de Processamento chegam à memória da placa de prototipação através do *PROCMultiPort*. O fluxo dos dados que devem ser escritos na memória é controlado pelo *Control Unit*, através de uma *FIFO* que armazena temporariamente os dados que são gerados pelos Núcleos de Processamento. Dessa

forma, o objetivo da FSM, que controla esse fluxo de dados dentro do *Control Unit,* é ler os dados dessa *FIFO* e enviá-los para a porta de escrita do *PROCMultiPort*.

A utilização dessa *FIFO* foi necessária porque a porta de escrita do *PROCMultiport* pode ficar cheia, não podendo receber novos dados. E uma característica dos núcleos de processamento é que enquanto dados validos são aplicados em suas entradas, eles não param de gerar resultados. Assim, caso a porta de escrita fique cheia, os resultados que são gerados pelos núcleos de processamento neste momento devem ser armazenados temporariamente, justificando, assim, a utilização dessa *FIFO*. Caso essa *FIFO* não fosse utilizada, e os resultados dos núcleos de processamento fossem escritos diretamente na porta de escrita do *PROCMultiport*, e caso a porta de escrita ficasse cheia, os resultados que fossem gerados neste momento seriam perdidos. Por isso, as saídas dos quatro núcleos de processamento são concatenadas e armazenadas nessa *FIFO*.

A Figura 55 mostra o fluxograma que é implementado pela FSM que faz o controle dos dados que devem ser escritos na memória.

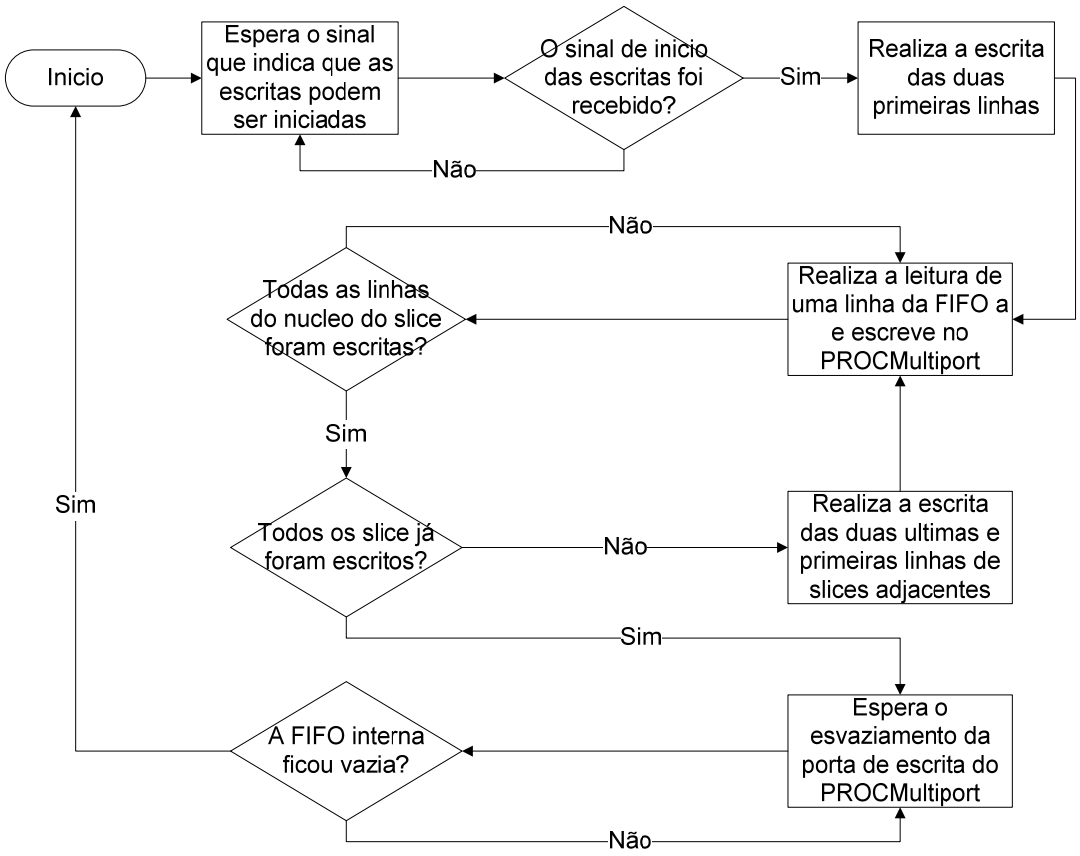


Figura 55 Fluxograma da FSM que implementa a escrita dos resultados na memória.

O fluxograma pode ser mais bem entendido com a descrição de passo, como a seguir:

O primeiro passo desse fluxo é esperar pelo sinal de autorização de escritas na memória. Quando esse sinal é recebido o fluxograma faz a transição para o próximo passo.

O segundo passo do fluxo tem como objetivo efetuar a escrita na memória das duas primeiras linhas do segundo *slice* da matriz do campo de pressão futuro, as quais não são geradas pelos Núcleos de Processamento, por se tratarem de dados de borda da matriz. Isto é necessário desde que o endereço inicial de escrita está posicionado no inicio do segundo *slice* da matriz do campo de pressão futuro, já que o primeiro slice representa dados de borda (duas ultimas colunas do slice) e dados não utilizados na computação (dados adicionados utilizados para homogeneização do fluxo de dados), dados estes que não são gerados pelos núcleos de processamento, fazendo com que o endereço de escrita deva ser inicializado no segundo slice. A Figura 56 mostra essas duas escritas iniciais.

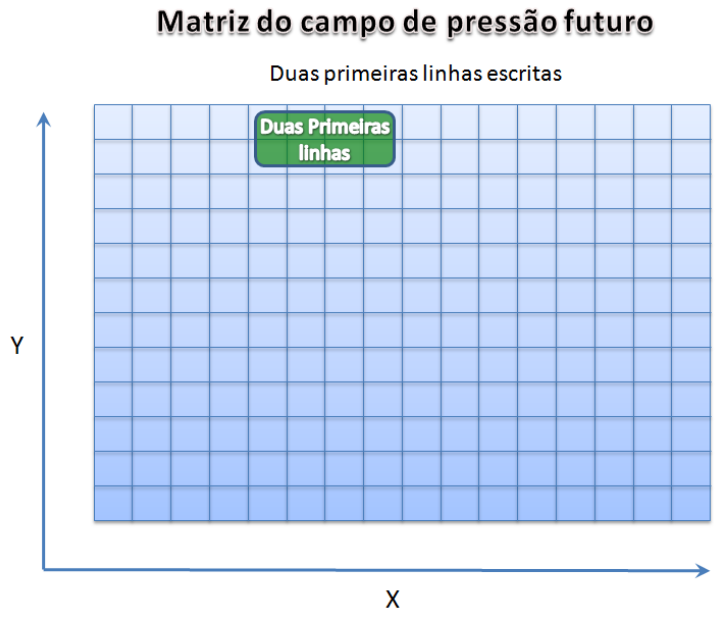


Figura 56 Escritas das duas primeiras linhas do segundo *slice*.

O terceiro passo do fluxo é o responsável pela escrita de dados do *slice* que não são bordas. A máquina verifica se existem dados na *FIFO* para serem escritos na memória. Caso existam, ela faz a escrita desses dados na porta de escrita do PROCMultiPort, caso não, ela fica esperando que os mesmos sejam escritos na FIFO. A máquina permanece nesse estado até que todo um slice, tirando as duas primeiras e duas ultimas linhas, que são borda da matriz, seja

escrito. Ou seja, esse passo é responsável por escrever na memória os dados dos *slices* que são gerados pelos núcleos de processamento. Na Figura 57 pode ser visto o primeiro *slice* de dados gerados pelos núcleos de processamento, que são escritos pelo terceiro passo do fluxo. Após escrever esse bloco de dados, o fluxograma faz a transição para o quarto passo caso ainda existam *slices* para serem escritos, caso contrario, o fluxograma faz a transição para o quinto passo.

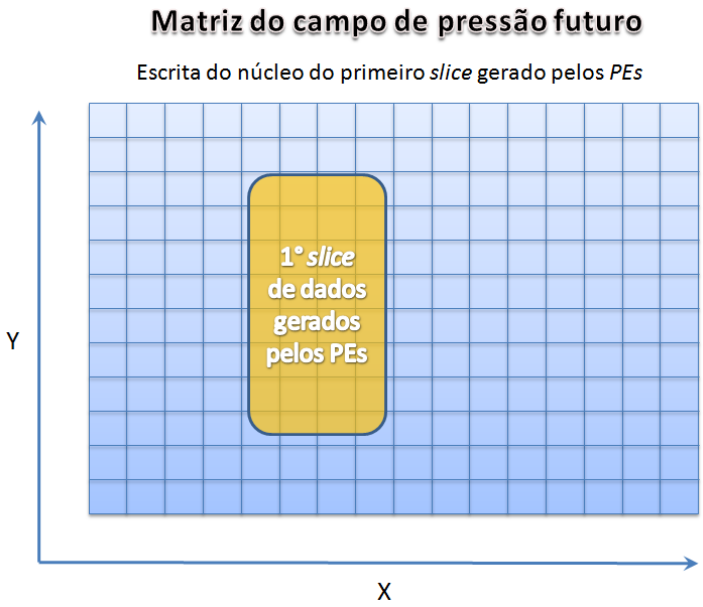


Figura 57 Dados do primeiro *slice* gerado pelos núcleos de processamento que são escritos na memória.

O quarto passo do fluxo é responsável por escrever as duas linhas de borda do final do *slice* que acabou de ser escrito na memória, e por escrever as duas linhas de borda do inicio do próximo *slice* que será escrito na memória. Ou seja, esse estado é responsável por escrever as bordas que existem entre dois *slices* adjacentes na memória. A Figura 58 mostra as escritas que são realizadas para o caso no qual os dois *slices* adjacentes serem o primeiro e o segundo *slices* gerados pelos núcleos de processamentos. Essas escritas estão destacadas com o nome "Bordas entre *Slices*".

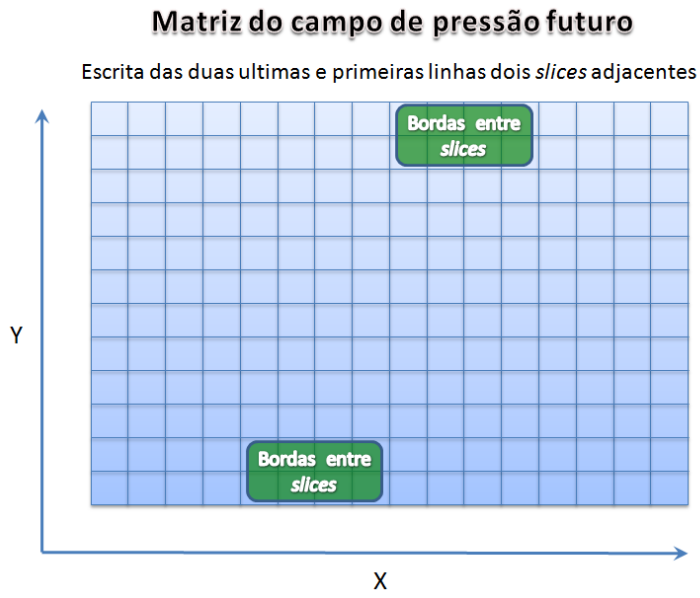


Figura 58 Escritas das bordas entre *slices* adjacentes.

O quinto passo do fluxo é responsável por esperar o esvaziamento da FIFO interna da porta de escrita do PROCMultiPort. O objetivo desse estado é garantir que todos os dados que foram enviados para a porta de escrita já foram escritos na memória. Assim que é constatado o esvaziamento da FIFO, essa máquina sinaliza para a máquina de leitura de dados que as escritas de todos os *slices* foram realizadas, e realiza a transição para o estado inicial.

Essa espera é necessária, pois segundo o algoritmo de diferenças finitas que utiliza a equação (2.2), ao final de um passo de processamento a matriz de pressão atual se torna a matriz de pressão anterior, e a matriz do campo de pressão futuro se torna a matriz de pressão atual. E essa atualização de matrizes é realizada na arquitetura através de uma troca dos endereços iniciais das matrizes de pressão atual e anterior, já que, como citado na seção 5.3.2, os dados gerados pelos núcleos de processamento são armazenados na região da memória onde está localizada a matriz de pressão anterior. Dessa forma, a computação de um novo passo de processamento só pode ser iniciada quando todos os dados do passo de processamento atual já estiverem escritos na memória. E como se pode ter percebido o primeiro *slice*, as duas últimas linhas do penúltimo *slice* e o ultimo *slice* não são escritos na memória. Isso não é um problema já que os dados da matriz do campo de pressão futuro estão sendo escritos na mesma região de memória da matriz do campo de pressão anterior, assim os dados referentes a esses *slices* na matriz de pressão anterior serão os mesmo na matriz de pressão futura. Como esses dados são bordas dessas matrizes, eles nunca são alterados, possuindo sempre o mesmo valor.

A Figura 59 mostra todas as escritas que são realizada por cada posso do fluxograma apresentado acima para uma matriz com 12 linhas e 16 colunas, em cada passo de tempo.

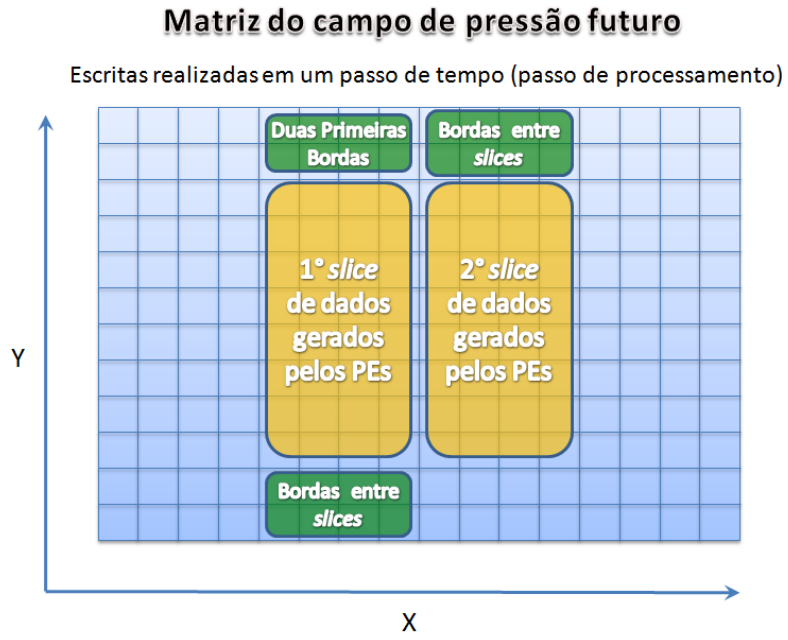


Figura 59 Todas as escritas realizas em um passo de tempo.

5.3.7.3 Controle da inserção do pulso sísmico

Alem dos dois fluxos de dados mostrados acima e suas máquinas de estados, existem mais duas outras máquinas de estado que controlam a inserção do pulso sísmico nas matrizes de pressão atual e anterior. Essas máquinas não controlam exatamente um fluxo especifico de dados, elas apenas modificam os dados que estão sendo passados para os Núcleos de Processamento, ou seja, elas fazem modificações em alguns dados que estão no fluxo de dados de leitura.

A Figura 60 mostra o tipo de máquina de estados que implementa esse controle de inserção do pulso sísmico em ambas as matrizes. As máquinas são similares, mas apresentam diferenças, tais como:

- cada uma controla a inserção do pulso sísmico em uma matriz diferente. Uma controla a inserção na matriz de pressão atual, verificando as leituras que são realizadas na porta do *PROCMultiport* que forneces os dados desta matriz, e a outra

controla a inserção do pulso na matriz de pressão anterior, verificando as leituras na porta do *PROCMultiPort* que fornece os dados da matriz desta última matriz.

- outra diferença é que na máquina que controla a inserção na matriz de pressão anterior existe um estado adicional, o estado de inicialização. Esse estado garante que não há processamento no primeiro passo, já que a inserção do pulso na matriz de pressão anterior só ocorre no segundo passo de processamento.

A seguir será descrito cada um dos estados, e através dessa explicação será mostrado como o *Control Unit* indica quando a inserção do pulso sísmico deve ser realizada.

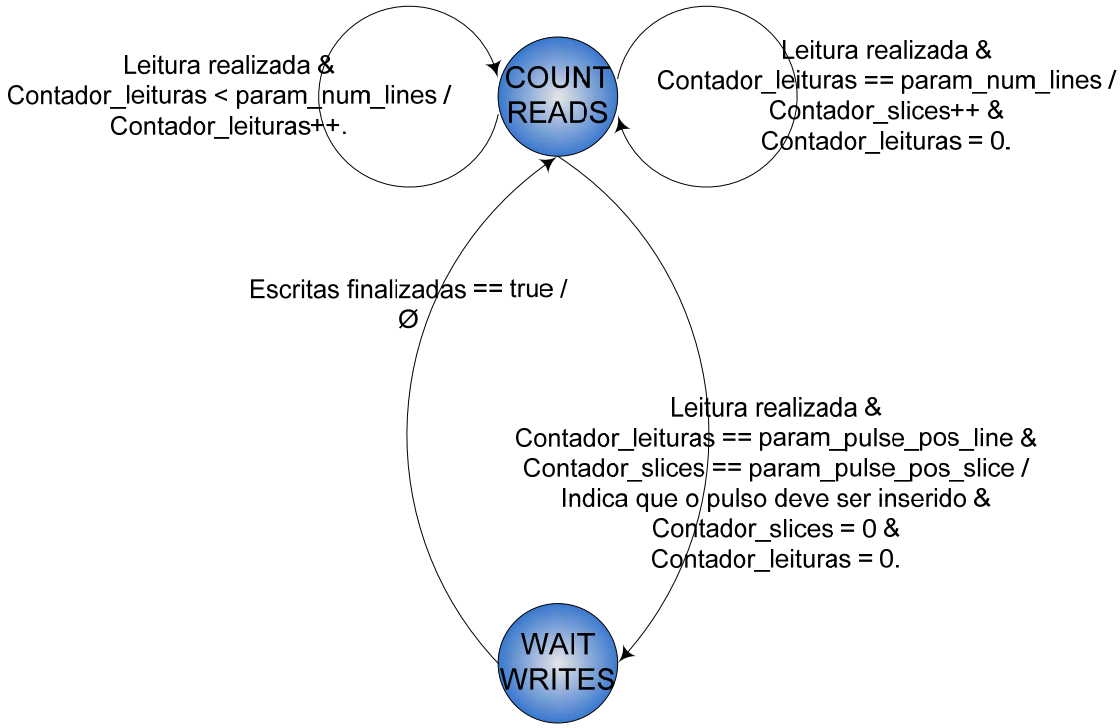


Figura 60 Máquina de estados que controla a inserção do pulso sísmico.

- COUNT_READS: estado utilizado para contar o número de leituras que estão sendo realizadas. A contagem é realizada através de dois contadores, um que conta quantas linhas de um *slice* já foram lidas, e outro que conta quantos *slices* já foram lidos. Onde uma linha do *slice* representa as quatro palavras que são fornecidas quando uma leitura é realizada no *PROCMultiPort*. Assim, quando esse dois contadores ficam iguais aos parâmetros que representam a linha e o *slice* onde o pulso sísmico deve ser inserido, a máquina indica para o módulo de *Ricker* ou *Ricker*

PPF que a inserção do pulso deve ser realizada, limpa os dois contadores e faz a transição para o próximo estado.

- WAIT_WRITES: estado que aguarda a escrita de todos os dados gerados na memória, nesse passo de processamento. O objetivo desse estado é não realizar mais contagens até que o próximo passo de processamento seja iniciado, já que a inserção do pulso sísmico do passo de processamento atual já foi realizada. A máquina se mantém nesse estado até que a máquina de escrita dos dados indique que todos os dados do passo de processamento atual já foram escritos na memória. Quando a escrita desses dados termina, a máquina faz a transição para o passo inicial.

5.4 Conclusões

Este capítulo apresentou a estrutura da plataforma desenvolvida neste trabalho. Esta plataforma utiliza uma arquitetura híbrida, tendo um dispositivo *FPGA*, como um coprocessador, conectado via barramento *PCIe*, a uma *CPU* de propósito geral

A plataforma foi customizada para a resolução de um modelo para modelagem-2D, em sísmica . O desenvolvimento da arquitetura em hardware foi realizado de forma que o tamanho dos modelos dos terrenos possa variar, ou seja, a arquitetura em hardware possui características de fácil parametrização dos modelos em análise. Entretanto, existe uma limitação para o tamanho de linhas das matrizes de entrada que podem ser enviadas para a arquitetura em *hardware*. Isso ocorre devido o tamanho máximo das FIFOS internas no *FPGA.* O numero de máximo de linhas que as matrizes de entrada podem ter é 8192.

Embora a arquitetura apresente vantagens, observa-se que alguns aspectos de ordem geral, da arquitetura, poderiam ser otimizados, porém em outra versão de plataforma, mais robusta em recursos, tais como: maior largura de banda da memória, maior quantidade de elementos lógicos do *FPGA* e elementos especializados na implementação de funções aritméticas, como operadores de ponto flutuante com precisão simples.

Capítulo

6

6 Exemplos e Resultados

Neste capítulo serão apresentados os resultados experimentais da plataforma desenvolvida, utilizando dois modelos de velocidades, sendo um com velocidades constantes, e o outro o modelo de Marmousi [48].

6.1.1 Resultados

O desenvolvimento de cada um dos módulos da arquitetura para processamento em *stream* foi feito no *ModelSim* [42]. Testes funcionais foram realizados em cada um deles. Os módulos da arquitetura foram desenvolvidos em *Verilog*, visando o mapeamento na *FPGA Stratix III EP3SE80* da *Altera*. Toda a arquitetura foi sintetizada e validada na ferramenta *Quartus* [38].

Para a validação da arquitetura proposta nesta dissertação, assim como para verificar o seu ganho de desempenho em relação a sua implementação em software, uma implementação em software do algoritmo foi também implementada em um PC com uma arquitetura Intel *Core 2 Quad 6600* 2.4 GHz, e 2GB de memória RAM.

Na implementação em software da modelagem 2D, que foi escrita em C, duas abordagens foram realizadas:

- Abordagem baseada no compilador C++ 11.1 da Intel para Linux [43]: nessa abordagem foi utilizado o Ubuntu 10.04 como sistema operacional e foi utilizado o compilador da Intel para gerar o arquivo executável. Foi utilizada a otimização de compilação *–fast* [66], visando obter um arquivo executável com o melhor desempenho possível.
- A segunda abordagem foi baseada no Visual Studio 2008 [44]: aqui foi utilizado o Windows XP como sistema operacional e o compilador utilizado foi o do Visual Studio, utilizando as configurações de otimizações padrões.

Para demonstração e análise das abordagens hardware e software, foram desenvolvidos dois exemplos: Modelo Constante e de Marmousi.

A Tabela 6 abaixo mostra o número total de linhas de toda a plataforma desenvolvida nessa dissertação. E a Tabela 7 mostra o resultado da utilização dos recursos disponíveis no FPGA pela arquitetura desenvolvida.

Tabela 6 Número de linhas da plataforma.

Módulo da plataforma	Número de linhas
Aplicativo de controle e comunicação	3481 (C)
Arquitetura em hardware	6113 (verilog)

Tabela 7 Resumo da utilização dos recursos do FPGA.

Fitter Status	Successful - Fri Jul 09 18:14:43 2010
Quartus II Version	8.1 Build 163 10/28/2008 SJ Full Version
Revision Name	dma
Top-level Entity Name	dma
Family	Stratix III
Device	EP3SE80F1152C4
Timing Models	Final
Logic utilization	94 %
Combinational ALUTs	36,366 / 64,000 (57 %)
Memory ALUTs	320 / 32,000 (1 %)
Dedicated logic registers	40,500 / 64,000 (63 %)
Total registers	44499
Total pins	669 / 744 (90 %)
Total virtual pins	0
Total block memory bits	1,122,992 / 6,331,392 (18 %)
DSP block 18-bit elements	16 / 672 (2 %)
Total PLLs	2 / 8 (25 %)
Total DLLs	1 / 4 (25 %)

6.1.1.1 Modelo Constante

Nesse exemplo, o modelo do terreno, que é representado pela matriz de velocidades, possui um valor constante. Esse exemplo é geralmente utilizado para se verificar a corretude da arquitetura, pois ele é bastante simples para constatar o deslocamento da onda sísmica pelo modelo. Neste modelo não existem interfaces entre rochas no terreno, o que faz com que a onda sísmica se propague de maneira constante no meio.

Nesse exemplo foi utilizado o pulso Ricker [67] como a fonte de onda sísmica. O pulso foi posicionado na oitava linha das matrizes de entrada e na coluna que se encontra no meio das mesmas, ou seja, o pulso foi inserido na parte superior e central das matrizes.

Nesse exemplo o número de passos de processamento garante que a onda sísmica se desloque por todo e modelo e retorna para a superfície do mesmo. A Tabela 8 mostra o tamanho das matrizes do campo de pressão atual e anterior, da matriz de velocidades, e o número de passos de processamento utilizados para cada uma delas.

Tabela 8 Número de passos de processamentos para cada tamanho de matriz.

Tamanho da matriz	N° de passos de processamento
600x600	5999
800x800	7999
1000x1000	10000
1200x1200	11999
1400x1400	13999
1600x1600	15999
1800x1800	18000
2000x2000	20000
2200x2200	21999
2400x2400	23999

Na Figura 61 pode-se ver o gráfico que mostra o tempo de processamento que a plataforma levou para processar essas matrizes e o tempo do processamento que a versão em software, compilado com o compilador da Intel, levou para processar as mesmas matrizes.

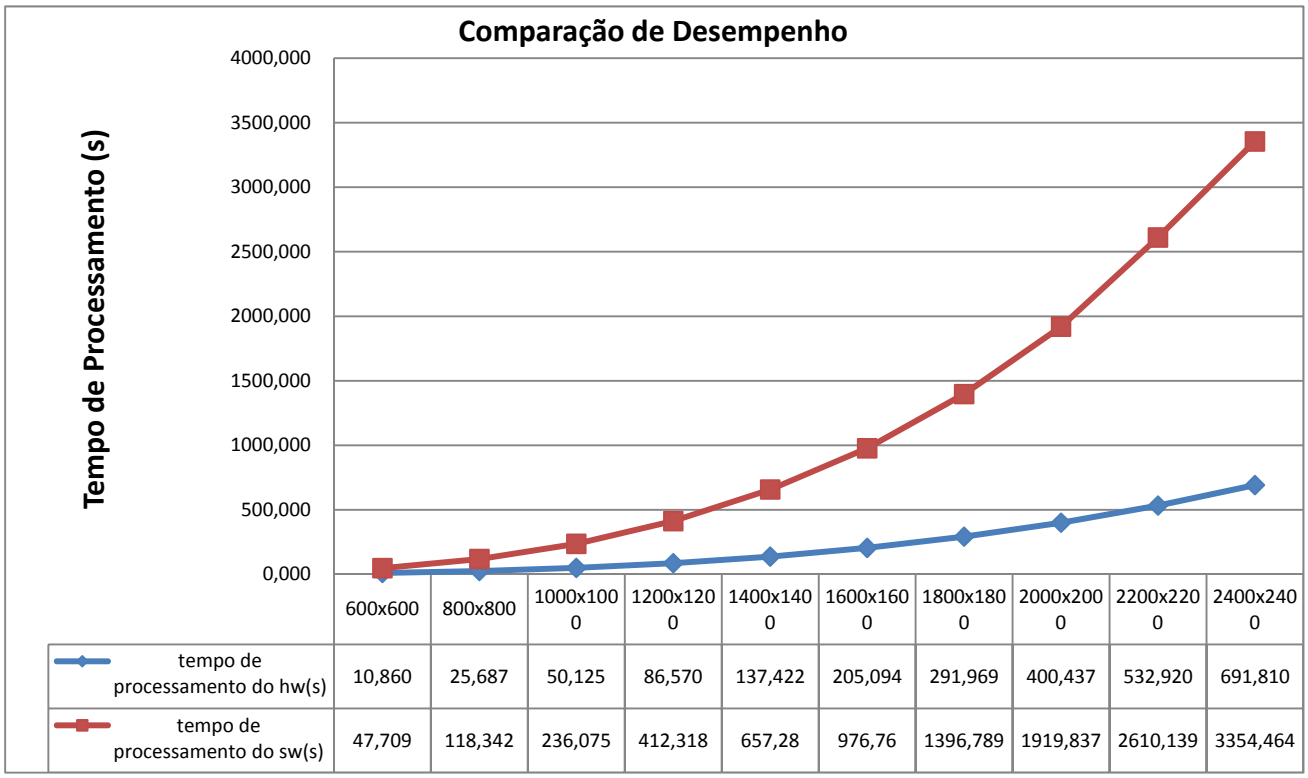
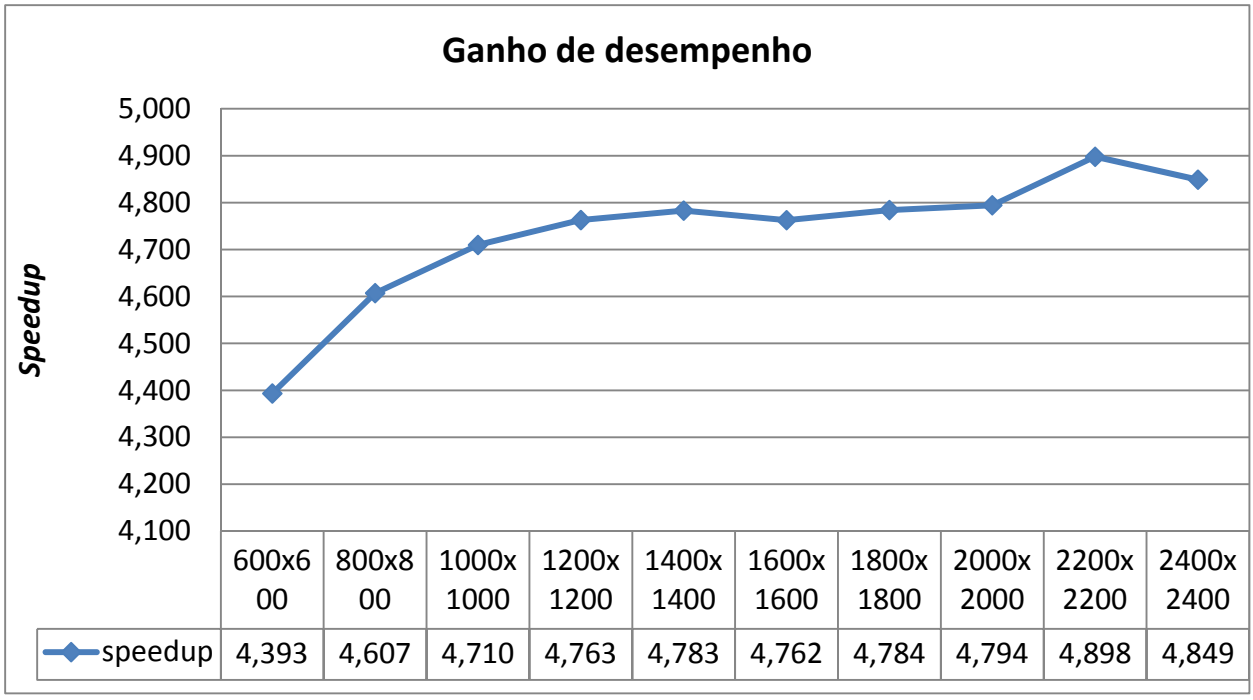


Figura 61 Tempo de Processamento da Plataforma Proposta e do Software compilado pelo compilador da Intel.

Na Figura 62 pode ser visto o gráfico que mostra o *speedup* que a plataforma obteve quando comparada a versão em software, para cada uma das matrizes. Pode-se notar que o *speedup* obtido pela plataforma proposta foi em torno de 4,6x quando comparado com o software gerado pelo compilador da Intel. Esse resultado é bastante interessante já que esse compilador que possui um desempenho melhor que o compilador *GNU gcc* [46] [47]. O compilador Intel também é bastante utilizado pelos desenvolvedores de métodos sísmicos, como o método sísmico de diferenças finitas [45], e na comparação de novas plataformas que implementam métodos sísmicos[7].



	600x600	800x800	1000x1000	1200x1200	1400x1400	1600x1600	1800x1800	2000x2000	2200x2200	2400x2400
speedup	4,393	4,607	4,710	4,763	4,783	4,762	4,784	4,794	4,898	4,849

Figura 62 Speedup para o modelo constante utilizando o compilador da Intel.

As quatro figuras abaixo mostram o deslocamento da onda com o crescimento do numero de passos de processamento através de um modelo com 1000 linhas e 1000 colunas, e com o pulso Ricker sendo posicionado na linha 8 e coluna 500. Estes resultados foram gerados na plataforma desenvolvida neste trabalho.

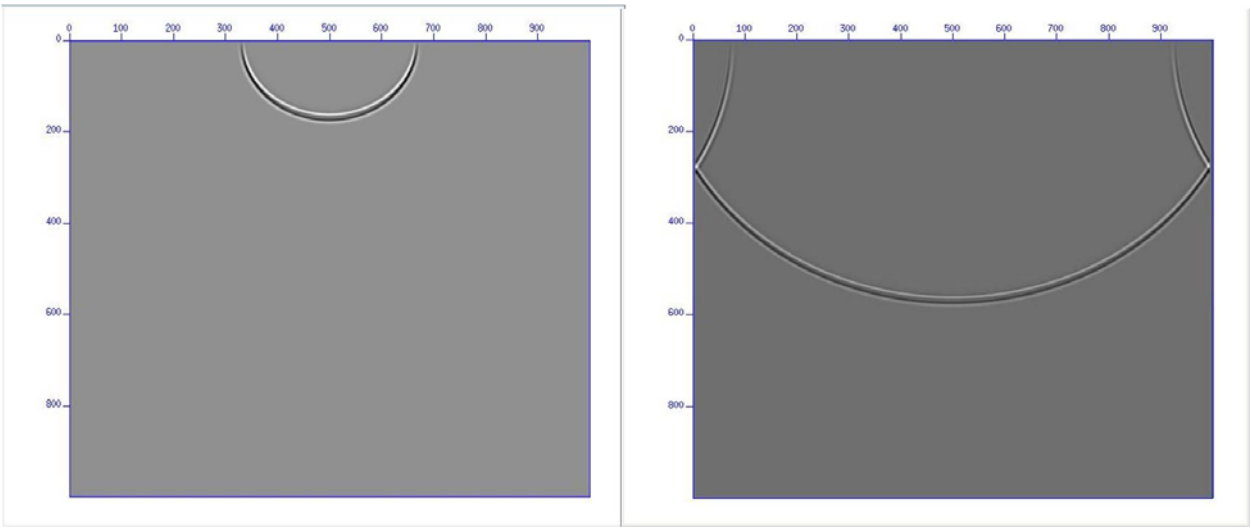


Figura 63 Deslocamento do pulso através de um modelo constante nos passos de processamento 1000 (esquerda) e 3000 (direita).

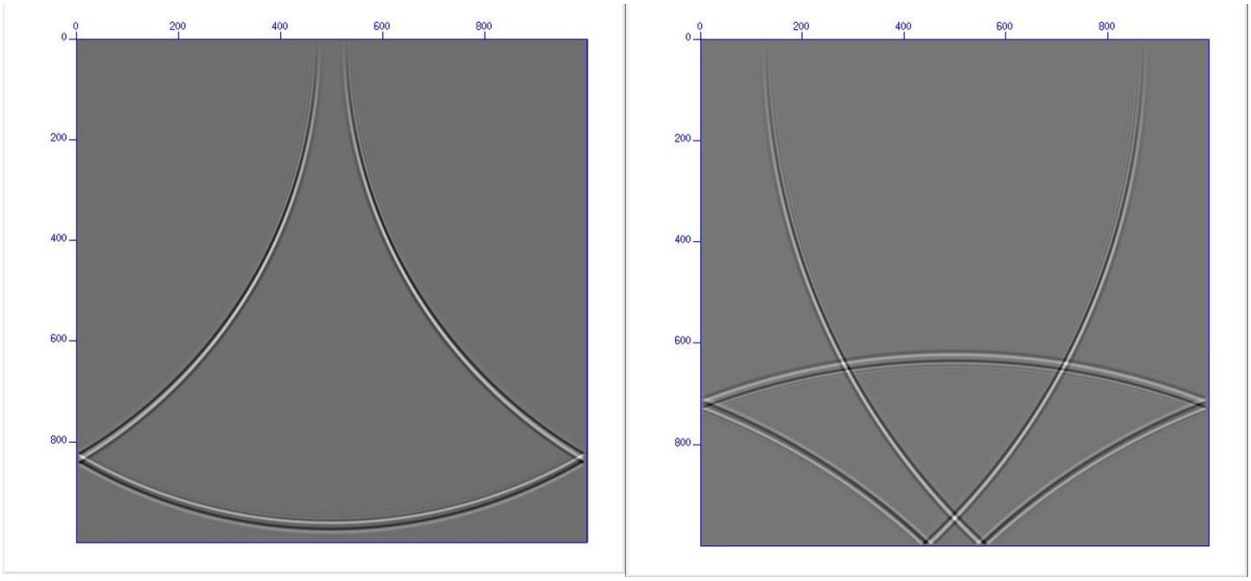


Figura 64 Deslocamento do pulso através de um modelo constante nos passos de processamento 5000 (esquerda) e 7000 (direita).

6.1.1.2 Modelo de *Marmousi*

Nesse exemplo, foi utilizado o modelo de Marmousi [48] como o modelo de velocidade. Esse modelo foi criado pelo *Institut Français du Pétrole* (*IFP*) em 1988. O modelo de *Marmousi* possui uma distribuição das camadas de rochas bastante complexa, sendo por isso, bastante utilizado na indústria de processamento de dados sísmicos, na realização de benchmarks que

verificam se a qualidade da imagem do terreno gerada pelos algoritmos é satisfatória. A Figura 65 mostra um exemplo desse modelo.

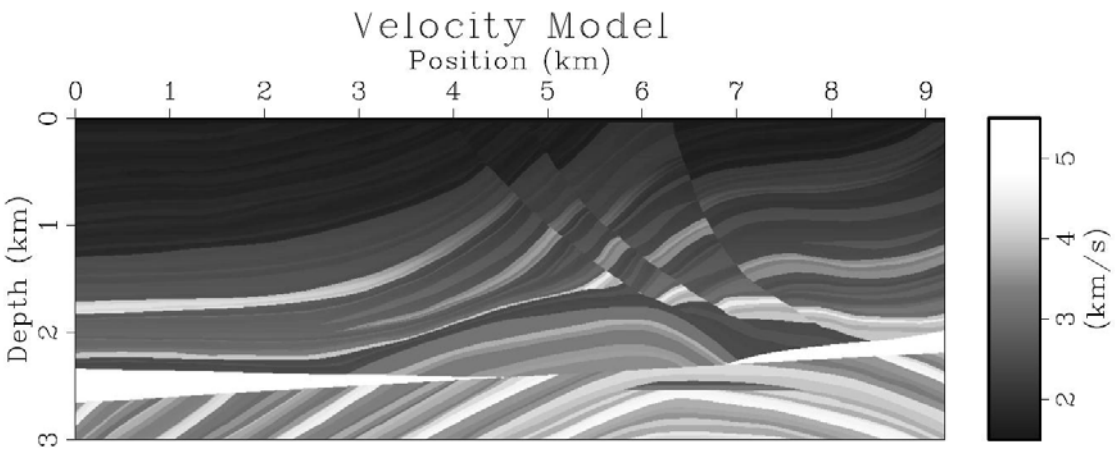


Figura 65 Modelo de Marmousi.

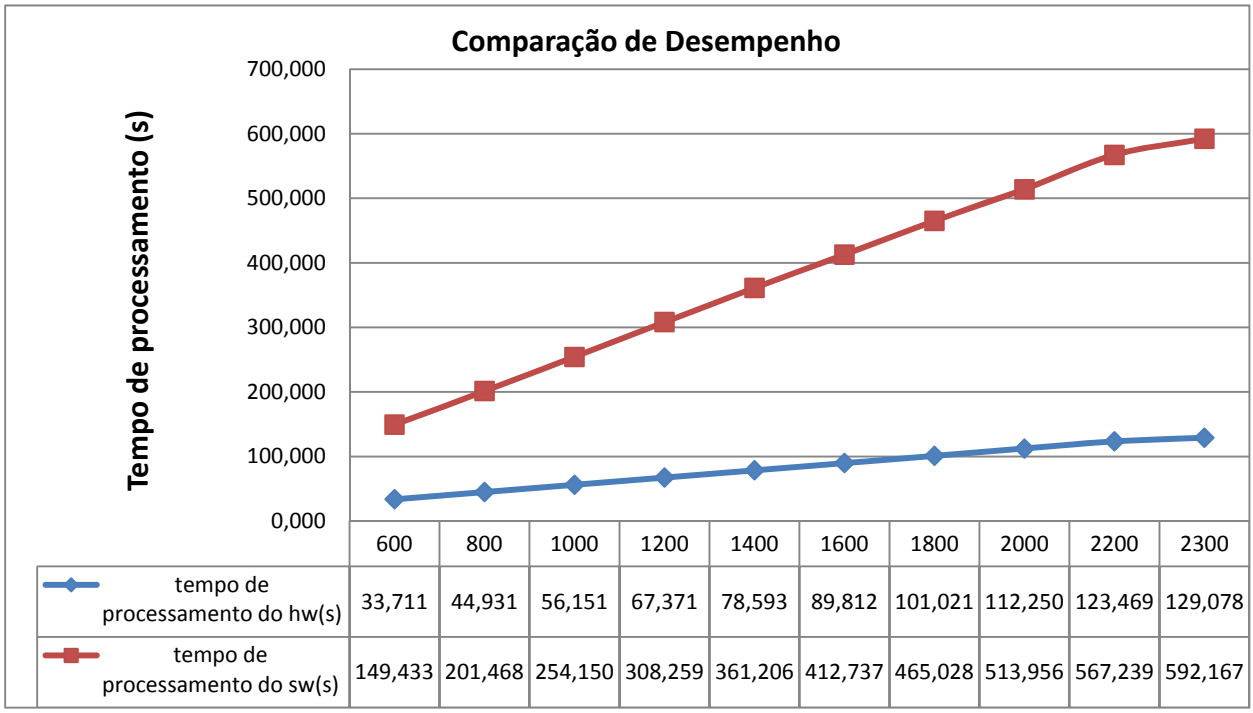
Neste trabalho foram executados vários exemplos utilizando o modelo de *Marmousi* com tamanhos diferentes. O modelo de *Marmousi* originalmente possui 751 linhas e 2301 colunas. Uma função foi criada para geração de modelos de *Marmousi* de menor tamanho. Esta função divide trechos contínuos da matriz original, customizando-os para as dimensões que o usuário escolher. Nesses exemplos também foi utilizado o pulso *Ricker* [67] como a onda sísmica a ser inserida no modelo, inserida na mesma posição daquela utilizada no modelo com velocidade constante.

Os exemplos executados, utilizando o modelo de *Marmousi,* mantiveram o número de linhas constante, igual a 748, variando-se o número de colunas, como a saber: 600; 800; 1000; 1200; 1400; 1600; 1800; 2000; 2200; e 2300.

O número de passos de processamento foi o mesmo para todos os modelos. Este número foi obtido utilizando-se a mesma fórmula citada anteriormente. Como o número de passos gerados por essa fórmula depende apenas do número de linhas do modelo, o mesmo foi constante e igual a 14960.

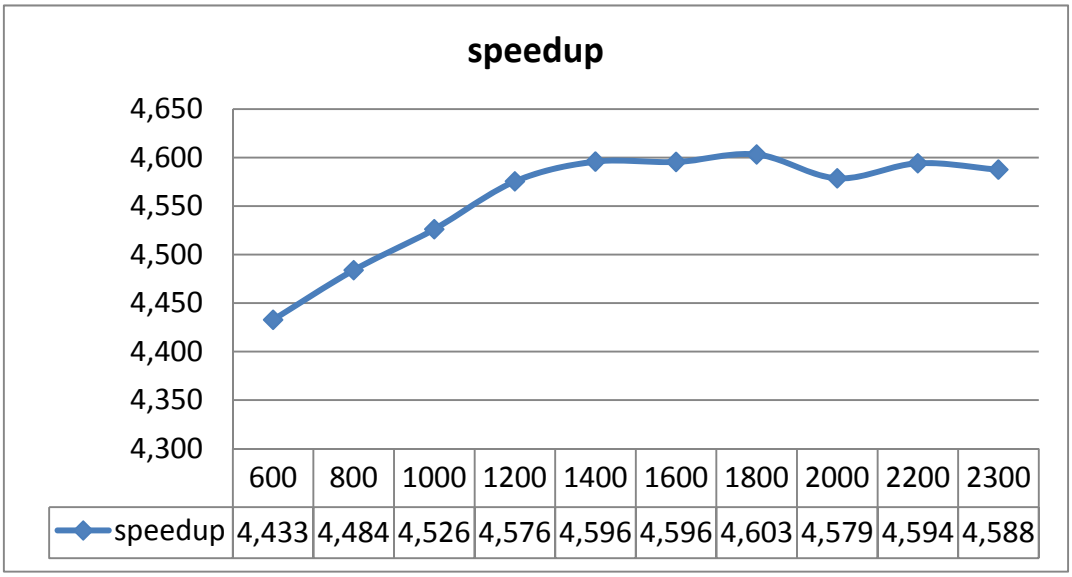
A Figura 66 é apresentado o gráfico que possui o tempo de processamento da plataforma proposta e do software que foi compilado e executado no *Visual Studio 2008* para os vários tamanhos de matrizes utilizando o modelo de *Marmousi*. Como o número de linha é

fixo, apenas o número de colunas é mostrado no eixo X do gráfico. Na Figura 67 pode ser visto o gráfico que mostra o *speedup* que a plataforma alcançou quando comparada com o software.



	600	800	1000	1200	1400	1600	1800	2000	2200	2300
tempo de processamento do hw(s)	33,711	44,931	56,151	67,371	78,593	89,812	101,021	112,250	123,469	129,078
tempo de processamento do sw(s)	149,433	201,468	254,150	308,259	361,206	412,737	465,028	513,956	567,239	592,167

Figura 66 Tempo de processamento da plataforma e do software compilado no Visual Studio 2008 para o modelo de Marmousi.



	600	800	1000	1200	1400	1600	1800	2000	2200	2300
speedup	4,433	4,484	4,526	4,576	4,596	4,596	4,603	4,579	4,594	4,588

Figura 67 Speedup alcançado pela plataforma quando comparada com o software compilado e executado no Visual Studio.

Pode-se notar que o *speedup* diminuiu devido ao grande desempenho que o *Visual Studio 2008* possui em seu compilador. Porém, um *speedup* de 4,5x ainda é um valor bastante motivador para a plataforma proposta.

Outros fatores relevantes a serem considerados no desenvolvimento desta plataforma são os recursos da placa de prototipação, como o número de pentes de memória que estão sendo utilizados e o modelo da *FPGA* utilizada, uma unidade pequena, o que restringe a instanciação de um pequeno número de Núcleos de Processamento, além da baixa frequência de operação destes núcleos.

Caso fosse utilizada uma placa de prototipação de maior escala, como por exemplo, a PROCStar III [68], mais núcleos de processamentos poderiam ser utilizados na arquitetura. Na placa utilizada nessa dissertação, a utilização de lógica de um único núcleo foi de 19%, como foram utilizados quatro núcleos, a quantidade de lógica utilizada apenas pelos núcleos de processamento na arquitetura foi 76%, aproximadamente. Na PROCStar III, um núcleo de processamento utiliza 5% do total de lógica disponível nessa placa. Assim, se fosse utilizada a PROCStar III, e fossem utilizados 16 núcleos, a lógica utilizadas pelos mesmos ficaria em 80%, um numero bem próximo ao que ocorreu na plataforma desenvolvida.

O baixo valor na frequência de operação foi devido à utilização de apenas um pente de memória na plataforma, o que gerou um gargalo no fornecimento de dados necessários para a computação dos pontos da matriz de resposta, em cada um dos quatro Núcleos de Processamento. Assim a frequência da plataforma teve que ser bem inferior a frequência real alcançada pelo controlador de memória.

Teoricamente, caso o *PROCMultiPort* trabalhasse a frequência de 200MHz, a frequência máxima que a arquitetura poderia trabalhar seria 50MHz, já que nela são utilizadas 4 portas de comunicação com a memória, ao mesmo tempo. Essas portas são as três portas que realizam as leituras da matriz do campo de pressão atual, anterior e da matriz de velocidade, e a porta que realiza as escritas da matriz do campo de pressão futuro. Assim, foi exatamente essa frequência de 50MHz que foi utilizada em todos os módulos da aplicação desenvolvida nesta dissertação.

As figuras 68 a 71 a seguir mostram o deslocamento da onda com o crescimento do número de passos de processamento através do modelo de *Marmousi*. O tamanho do modelo utilizado nesse exemplo abaixo foi 748 linhas e 2300 colunas, com o numero de processamento igual a 3000, 5000, 7000 e 9500. O pulso *Ricker* foi inserido na linha 8 e coluna 1150.

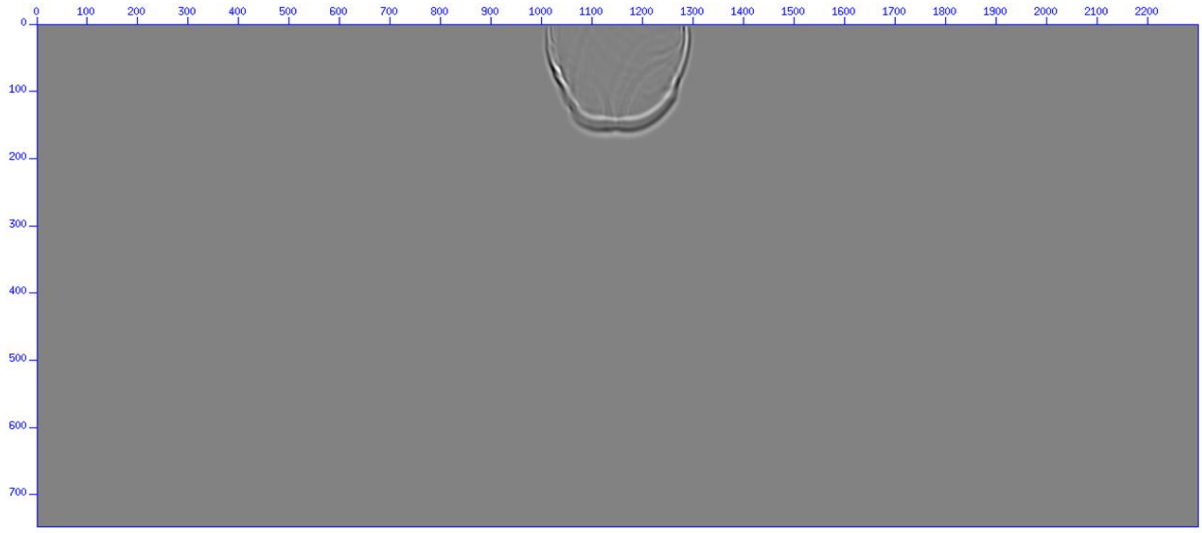


Figura 68 Deslocamento do pulso através do modelo de Marmousi no passo de processamento 3000.

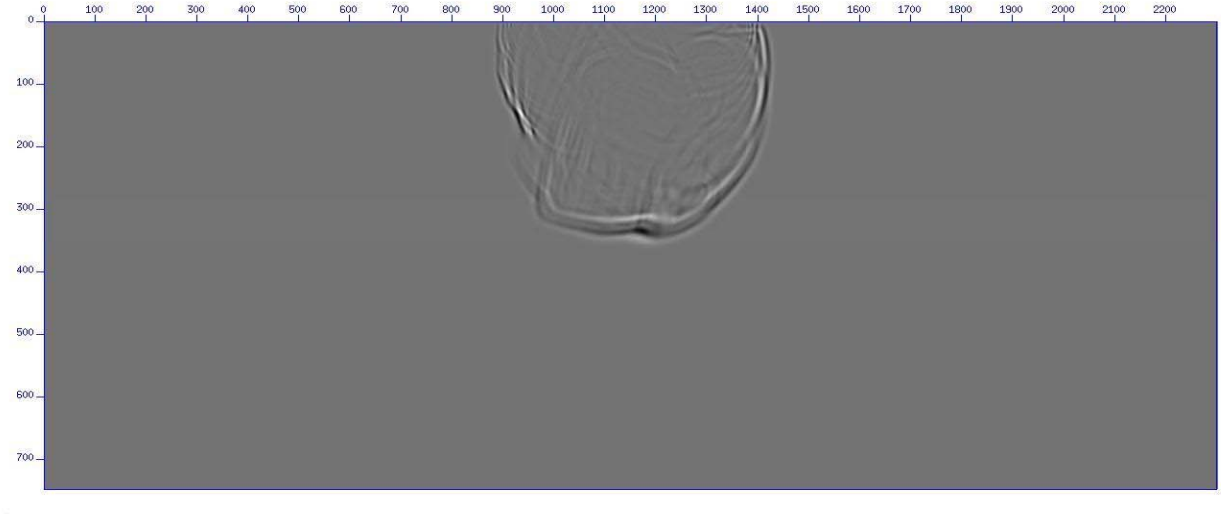


Figura 69 do pulso através do modelo de Marmousi no passo de processamento 5000.

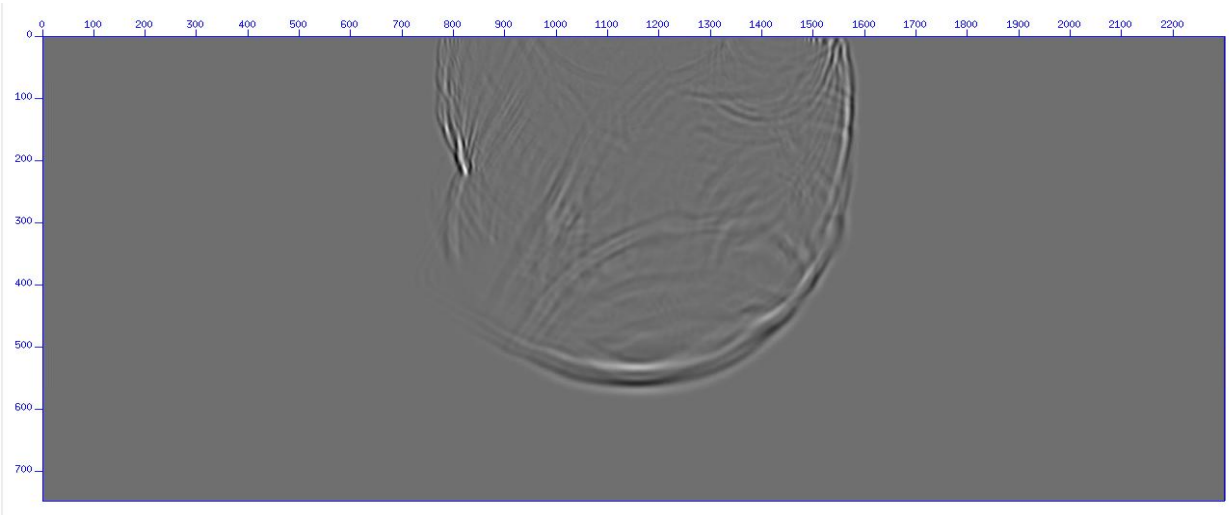


Figura 70 do pulso através do modelo de Marmousi no passo de processamento 7000.

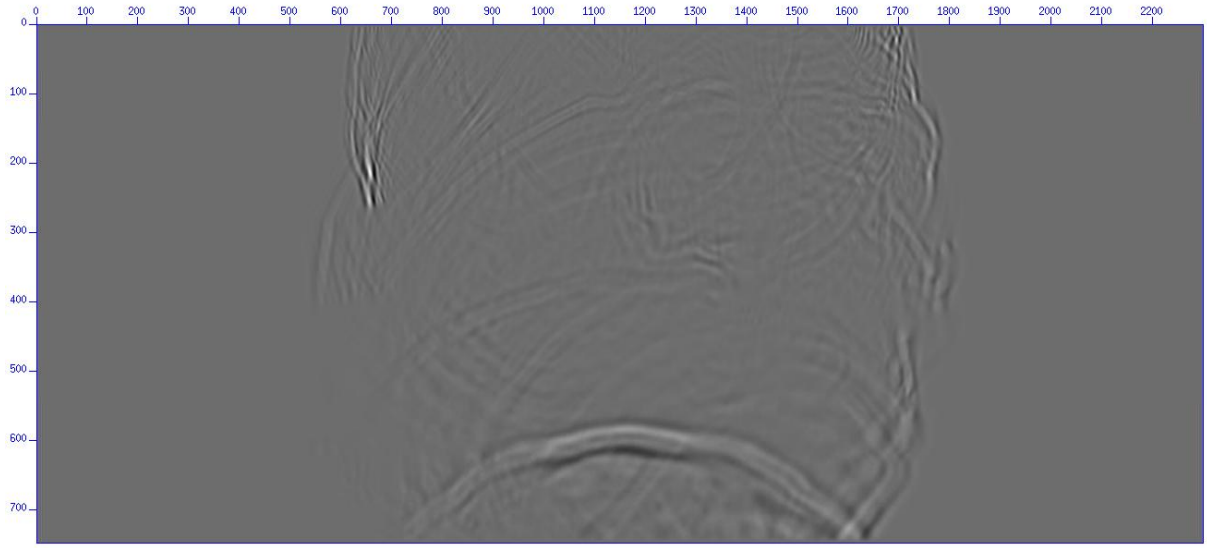


Figura 71 do pulso através do modelo de Marmousi no passo de processamento 9500.

Pode ser visto nas figuras acima que elas apresentam reflexões que não ocorreram no caso do modelo constante. Isso acontece devido o modelo de Marmousi apresentar várias camadas de rochas com composições diferentes, fazendo com que o pulso sísmico sofra reflexões nas interfaces dessas camadas. E são exatamente essas reflexões que são captadas na superfície formando o sismograma.

Capítulo

7

7 Conclusões e Trabalhos Futuros

Este capítulo tem como objetivo apresentar algumas considerações finais sobre os principais tópicos abordados nesta dissertação, incluindo as contribuições alcançadas e indicações para trabalhos futuros.

7.1 Contribuições e Considerações Finais

A proposta desta dissertação foi verificar a possibilidade da utilização de dispositivos lógicos reconfiguráveis, no aumento de desempenho computacional de métodos de modelagem 2D em sísmica. Para tal foi desenvolvida uma plataforma reconfigurável baseada em *FPGA* que implementa a modelagem 2D, utilizando o método de Migração Reversa no Tempo. A fim de validarmos a plataforma e verificar sua eficiência, exemplos de modelagem 2D foram executados em software e em hardware, considerando diferentes aspectos de implementação. Os resultados das comparações de desempenho entre as duas abordagens mostraram que as implementações em *FPGA* apresentaram um melhor desempenho computacional que aqueles implementados em software, atingindo, em média, um *speedup* de 4,5x.

Concluímos que os ganhos, no entanto, não foram tão expressivos quanto desejado devido a aspectos operacionais, tais como:

- baixa frequência de operação dos núcleos de processamento (50 MHz) provocado pelas restrições de acesso a memória da plataforma *FPGA*. O acesso aos dados na memória da placa precisou ser multiplexado para permitir uma distribuição adequada aos quatro Núcleos de processamento. Estima-se que esta velocidade poderia atingir até 3 vezes o seu valor implementado, o que elevaria o desempenho em até 3 vezes, caso fossem utilizadas mais bancos de memória da placa, eliminando a necessidade de multiplexação.

- A complexidade dos dados, representados por números ponto flutuante, 32 bits. Isto complica substancialmente toda a aritmética envolvida no processamento do algoritmo de modelagem em sísmica. Além disso, como *FPGAs* não possuem unidades aritméticas de ponto flutuante, suas implementações se tornam onerosas em área e ciclos de processamento. Para amenizar o problema da grande quantidade de computação, a arquitetura dos núcleos de processamento foi implementada utilizando massivamente técnicas de *pipeline*.

- O acesso aos dados oriundos da CPU via barramento PCI também possui uma latência importante no desenvolvimento do sistema. Para pequenas matrizes de entrada, que requerem poucos passos de tempo, essa latência se torna considerável, pois o tempo de processamento não é tão grande. Entretanto, para grandes matrizes de entrada, que requerem um grande numero de passos de tempo, o tempo de processamento se torna muito grande, deixando a latência de transmissão entre a CPU e FPGA muito pequena. De qualquer forma, barramentos mais rápidos ou tratamento mais eficiente na relação processamento x transferência de dados poderia ajudara a melhorar o desempenho da plataforma como um todo.

Observamos por outro lado, que estratégias com uso de técnicas *pipeline* e reuso de dados puderam, sobremaneira, melhorar o desempenho de vários estágios da arquitetura, tais como:

- uso explicito das características de paralelismo intrínseco da arquitetura dos FPGA, possibilitando, por exemplo, a execução de várias funções aritméticas de forma concorrente. Esta arquitetura permitiu também a geração de uma plataforma em *stream*, onde vários dados podem ser processados por várias operações diferentes em estágios de *pipelines* diferentes.

- O uso de estruturas tipo *FIFOs* e registradores de deslocamento dentro da *FPGA* permitiu que técnicas com um grande reuso de dados pudessem ser implementadas com facilidade. Desta forma, uma grande quantidade de dados necessita ser lida apenas uma única vez da memória, para cada passo de processamento, mantendo-os armazenados nas *FIFOs* enquanto necessário.

- Como a vantagem do uso de *FPGAs* é seu baixo consumo de energia quando comparado à tecnologia como as *CPUs* de propósito geral e *GPUs*.

Podemos dizer, em síntese, que a tecnologia *FPGA* é uma tecnologia ainda emergente na área de processamento de alto desempenho, se mostrando uma boa solução para

problemas de processamento massivo em dados com pouca complexidade aritmética (aritmética ponto fixo ou inteiro), mas ainda com restrições de desempenho em aritmética ponto flutuante, por falta de suporte da arquitetura.

A arquitetura da plataforma proposta neste trabalho pode ser ainda utilizada não apenas na modelagem 2D, mas também em outras aplicações que utilizam o método de diferenças finitas, onde um operador em forma de estêncil é aplicado a matrizes de entrada.

7.2 Trabalhos Futuros

Apesar do resultado obtido pela plataforma proposta nesta dissertação ser bastante promissor, varias otimizações podem ser integradas a mesma, tais como:

- utilização dos três bancos de memória disponíveis na plataforma *PROCe III* da *GiDEL*. Dessa forma, cada uma das matrizes de entrada poderia ficar armazenada em uma memória. Com isso, a frequência de trabalho da arquitetura poderia ser aumentada, melhorando o ganho de desempenho obtido pela plataforma. Também, o número de núcleos de processamento poderia ser aumentado, aumentando da mesma forma o ganho de desempenho.

- Um estudo para redução da precisão de dados de forma que essa diminuição não degradasse o resultado final da imagem. Essa redução de precisão, que acarreta um menor número de bits para representação de dados, permite um aumento na largura de banda da memória.

- Mais uma melhoria que poderia ser implantada na plataforma seria a utilização de vários passos de processamentos simultâneos, como mostrado na seção 3.3. Essa estratégia se baseia na ideia dos resultados de um núcleo de processamento ir direto para a entrada de outro núcleo de processamento que já está computando o próximo passo de processamento. Assim, o acesso à memória seria realizado apenas no primeiro passo de processamento, onde as leituras são realizadas, e no último, onde as escritas são realizadas. Dessa forma, o número de passos de

processamentos computados é aumentado sem aumentar o acesso à memória, consequentemente aumentando o ganho de desempenho obtido pela plataforma.

- Uma melhoria substancial seria a utilização da plataforma PROCStar III da Gidel [68], com quatro Stratix III [35]. Neste caso o modelo poderia ser dividido em quatro partes e cada uma dessas partes poderia ser computada por um dos FPGAs, aumentando o desempenho obtido pela plataforma. Com a divisão do modelo, uma comunicação deve ser realizada para troca de dados entre os FPGAs, porém caso a plataforma consiga sobrepor o tempo de computação com o tempo de troca de dados, o ganho de desempenho seria de pelos menos quatro vezes mais.

Capítulo

8

8 Referências Bibliográficas

Este capítulo tem como objetivo apresentar as referências bibliográficas detalhadas utilizadas no processo de elaboração desta Dissertação de Mestrado.

[1] Petroleum Geo-Service. Reverse Time Migration. In: *Tech Link, A Publication of Petroleum Geo-Service* Vol. 7, No. 1, September de 2007.

[2] Robert G. Clapp, Haohuan Fu, and Olav Lindtjorn. Selecting the right hardware for reverse time migration. In: The Leading Edge 29: 48-58, January 2010.

[3] C. He, M. Lu, and C. W. Sun. Accelerating seismic migration using FPGA-based coprocessor platform. In: Proceedings of the 12th Annual IEEE Symposium on Field-Programmable Custom Computing Machines, 2004, pp.207-216.

[4] D.S. William, R.S. Austars. Towards an RCC-based Accelerator for Computational Fluid Dynamics Applications. In: Journal of Supercomputing, Vol. 30, 239–261, December 2004.

[5] N. Azizi, I. Kuon, A. Egier, A. Darabiha, P. Chow. Reconfigurable molecular dynamics simulator. In: Proceedings of the 12th Annual IEEE Symposium on Field-Programmable Custom Computing Machines, 2004, pp. 197–206.

[6] Yilmaz, Ö. Seismic data analysis: Processing, inversion, and interpretation of seismic data, vol. 1. 2ª edition. Tulsa: Society of Exploration Geophysicists, 2001.

[7] C. He. Numerical Solutions of Differential Equations on FPGA-Enhanced Computers. 2007. Dissertation (DOCTOR OF PHILOSOPHY). Texas A&M University, Texas.

[8] Robert E. Sheriff, Lloyd P. Geldart. Exploration Seismology. 2ª edition. Cambridge: Cambridge University Press, 1995.

[9] Milton B. Dobrin, Carl H. Savit. Introduction to Geophysical Prospecting. 4ª edition. Singapore: McGraw-Hill Book Companies, 1988.

[10] S. H. Gray. Seismic migration problems and solutions. In: Y2K Review Article, Geophysics, 66, 1622-1640, October 2001.

[11] Silva, Josias José. Migração reversa no tempo na determinação das amplitudes de reflexão em função do ângulo de incidência. 2009. Tese (Doutor em Engenharia Civil). Universidade Federal do Rio de Janeiro, Rio de Janeiro.

[12] Altera. Accelerating High-Performance Computing With FPGAs, Agosto 2007. URL: http://www.altera.com/literature/wp/wp-01029.pdf. Consultado em: 14 de julho de 2010.

[13] top500. http://www.top500.org/overtime/list/35/archtype. Consultado em: 14 de julho de 2010.

[14] L. House, S. Larsen, J. B. Bednar. 3-D elastic numerical modeling of a complex salt structure. In: Expanded Abstracts of SEG 70th Annual Meeting, 2000, pp. 2201-2204.

[15] P. Moczo, M. Lucka, and M. Kristekova. 3D displacement finite differences and a combined memory optimization. In: Bulletin Seismological Society of America, v. 89, p. 69-79, February 1999.

[16] S. Larsen, J. Grieger. Elastic modeling initiative, part III: 3-D computational modeling. In: Expanded Abstracts of SEG 68th annual meeting, 1998, pp. 1803-1806.

[17] K. D. Underwood, and K. S. Hemmert. Closing the gap: trends in sustainable floating-point BLAS performance. In: Proceedings of the 11th Annual IEEE Symposium on Field-Programmable Custom Computing Machines, 2004, pp. 219-228

[18] J. Makino and M. Taiji. Special-purpose Computers for Scientific Simulations: The GRAPE Systems, John Wiley & Sons, Hoboken, NJ, 1998.

[19] The GRAPE Project, GRAPE: A programmable multi-purpose computer for many-body simulations, < http://grape-dr.adm.s.u-tokyo.ac.jp/project-en.htm>. Consultado em: 14 de julho de 2010.

[20] C. Cheng, J. Wawrzynek, and R. W. Brodersen. A high-end reconfigurable computing system. In: IEEE Design and Test of Computers, 22 (2005), 114-125.

[21] Starbridge. HC-62 board specifications, http://www.starbridgesystems.com/. Consultado em: 14 de julho de 2010.

[22] C. Petrie, C. Cump, M. Devlin, and K. Regester, High performance embedded computing using field programmable gate arrays, in: Proceedings of the 8th Annual Workshop on High-performance Embedded Computing, 2004, pp. 124-150.

[23] L. Gray, R. Woodson, A. Chau, and S. Retzlaff, Graphics for the long term: An FPGA-based GPU, <http://www.vmecritical.com/articles/id/?823>, 2005. Consultado em: 14 de julho de 2010.

[24] Cray. CrayXD1datasheet, http://www.hpc.unm.edu/~tlthomas/buildout/Cray_XD1_Datasheet.pdf. Consultado em: 14 de julho de 2010.

[25] SGI. SGI RASC RC100 blade datasheet, http://www.silicongraphics.ru/pdf/rasc_data.pdf. Consultado em: 14 de julho de 2010.

[26] Chuan He, Guan Qin, Mi Lu, Wei Zhao. Optimized high-order finite difference wave equations modeling on reconfigurable computing platform. In: Microprocessors & Microsystems, v. 31 , pp.103-115, March 2007.

[27] IEEE. Padrão para números em ponto flutuante. <http://grouper.ieee.org/groups/754/>. Consultado em: 17 de Julho de 2010.

[28] Xilinx. Virtex-4 ML401 Evaluation Platform. <http://www.xilinx.com/products/devkits/HW-V4-ML401-UNI-G.htm>. Consultado em 17 de Julho de 2010.

[29] Xilinx. Virtex-4 User Guide. <http://www.xilinx.com/support/documentation/user_guides/ug070.pdf>. Consultado em: 17 de Julho de 2010.

[30] P. Durbano, F. E. Ortiz, J. R. Humphrey, P. F. Curt, and D. W. Prather. FPGA-based acceleration of the 3D Finite-Difference Time-Domain (FDTD) method. In: Proceedings of the 12th Annual IEEE Symposium on Field- Programmable Custom Computing Machines (FCCM), pp. 156-163, 2004.

[31] PLX Technology. PCI 9656. <http://www.plxtech.com/products/io/pci9656>. Consultado em: 17 de Julho de 2010.

[32] Xilinx. Virtex II 8000 User Guide. <http://www.xilinx.com/support/documentation/user_guides/ug002.pdf>. Consultado em: 17 de Julho de 2010.

[33] Haohuan Fu, William Osborne, Robert G. Clapp, Oskar Mencer, and Wayne Luk. Accelerating seismic computations using customized number representations on FPGAs. In: EURASIP Journal on Embedded Systems, v. 2009, January 2009.

[34] GiDEL. Plataforma PROCe III. <http://www.gidel.com/PROCe%20III.htm>. Consultado em: 16 de Julho de 2010.

[35] Altera. Overview da Stratix III. <http://www.altera.com/products/devices/stratix-fpgas/stratix-iii/overview/st3-overview.html>. Consultado em: 16 de Julho de 2010.

[36] GiDEL. PROCMultiPort. <http://www.gidel.com/PROCMultiport.htm>. Consultado em: 16 de Julho de 2010.

[37] GiDEL.PROCWizard. <http://www.gidel.com/ProcWizard.htm>. Consultado em: 16 de Julho de 2010.

[38] Altera. Introdução ao Quartus. <http://www.altera.com/literature/manual/intro_to_quartus2.pdf>. Consultado em: 16 de Julho de 2010.

[39] Altera. User Guide da FIFO da Altera. <http://www.altera.com/literature/ug/ug_fifo.pdf>. Consultado em: 16 de Julho de 2010.

[40] Altera. User Guide do Somador. <http://www.altera.com/literature/ug/ug_altfp_mfug.pdf>. Consultado em: 16 de Julho de 2010.

[41] Altera. MegaWizard Plug-Ins <http://www.altera.com/products/ip/altera/megawizd.html>. Consultado em 16 de Julho de 2010.

[42] Mentor Graphics. ModelSim SE. < http://model.com/content/modelsim-se-high-performance-simulation-and-debug>. Consultado em: 17 de Julho de 2010.

[43] Intel. Product Brief do compilador C++ 11.1 da Intel. <http://software.intel.com/sites/products/collateral/hpc/compilers/clin_brief.pdf>. Consultado em: 17 de Julho de 2010.

[44] Microsoft. Visual Studio 2008. < http://www.microsoft.com/visualstudio/pt-br/products/2008-editions>. Consultado em: 17 de Julho de 2010.

[45] Igor S. Terentyev. Framework Design and Implementation of Finite Difference Based Seismic Simulations. 2008. Thesis (Master of Arts). RICE UNIVERSITY, Houston, Texas.

[46] Principled Technologies. Performance comparison of Intel C++ Compiler 9.1 for Linux and GNU gcc 4.1.1 on AMD- and Intel-processor-based system. In: Test Report, June 2006.

[47] Dale Schouten, Xinmin Tian, Aart Bik, Milind Girkar. Inside the Intel Compiler. In: *Linux Journal*. February 2003. < http://www.linuxjournal.com/article/4885>. Consultado em: 17 de Julho de 2010.

[48] Trevor Irons. Marmousi Model. <http://www.ahay.org/RSF/book/data/marmousi/paper.pdf>. Consultado em: 18 de Julho de 2010.

[49] John L. Hennessy, David A. Patterson. Computer Architecture: A Quantitative Approach. 4ª edition. Elsevier, 2007.

[50] Michael John Sebastian Smith. Application-Specific Integrated Circuits. Addison-Wesley. 1997.

[51] NVIDIA. Graphic Processing Unit. <http://www.nvidia.com.br/object/gpu.html>. Consultado em: 20 de Julho de 2010.

[52] Minh Tri Do Dinh. GPUs - Graphics Processing Units. July 2008. <http://informatik.uibk.ac.at/teaching/ss2008/seminar/gpu.pdf>. Consultado em: 20 de julho de 2010.

[53] C. Cheng, J. Wawrzynek, and R. W. Brodersen. A high-end reconfigurable computing system. In: *IEEE Design and Test of Computers*, 22 (2005), 114-125.

[54] S. Brown. Performance comparison of finite-difference modeling on Cell, FPGA and multi-core computers. In: SEG/San Antonio 2007 Annual Meeting, 2007, pp. 2110–2114.

[55] D.B. Thomas, L. Howes, W. Luk. A comparison of CPU's, GPU's, FPGA's, and massively parallel processor arrays for random number generation. In: *ACM/SIGDA International Symp. on Field-Programmable Gate Arrays*, 2009, pp. 63-72.

[56] S. Che, J. Li, J.W. Sheaffer, K. Skadron, and J. Lach. Accelerating Compute Intensive Applications with GPUs and FPGAs. In: *Proc. Symp. Application Specific Processors*, pp. 101-107, 2008.

[57] D. Bevc. Imaging Complex structure with semi-recursive Kirchhoff migration, Geophysics, 62 (1997) 577-588.

[58] Maya B. Gokhale, Paul S. Graham. Reconfigurable computing: Accelerating computation with field-programmable gate arrays. Springer-Verlag, New York, 2005.

[59] Jerry L. Potter. The Massively Parallel Processor. MIT Press. 1985.

[60] Austin Hung, William Bishop, Andrew Kennings. Symmetric Multiprocessing on Programmable Chips Made Easy. In: *Proceedings of the conference on Design, Automation and Test in Europe*, p.240-245, March 07-11, 2005.

[61] Pablo Huerta, Javier Castillo, Carlos Sanchez, Jose Ignacio Martinez. Operating System for Symmetric Multiprocessors on FPGA. In: International Conference on Reconfigurable Computing and FPGAs, pp. 157-162, 2008.

[62] The MathWorks. Least Squares. <http://www.mathworks.com/moler/leastsquares.pdf>. Consultado em: 27 de Julho de 2010.

[63] H. Fu, R. G. Clapp, O. Mencer and O. Pell. Accelerating 3D convolution using streaming architectures on FPGAs. In: 79th Annual Meeting, Society of Expoloration Geophysicists (SEG), Houston, October 2009.

[64] Xilinx. Virtex-5 FPGA Family. <http://www.xilinx.com/products/virtex5/index.htm>. Consultado em: 30 de Julho de 2010.

[65] Xilinx. Virtex-6 FPGA Family. <http://www.xilinx.com/products/virtex6/index.htm>. Consultado em: 31 de Julho de 2010.

[66] Intel. Intel® C++ Compiler 11.1 User and Reference Guides. <http://software.intel.com/sites/products/documentation/hpc/compilerpro/en-us/cpp/lin/compiler_c/copts/common_options/option_fast.htm#option_fast>. Consultado em: 2 de Agosto de 2010.

[67] J. K. Costain, C. Coruh, Basic theory in reflection seismology, Elsevier Science, Amsterdam, Netherlands, 2004

[68] GiDEL.Plataforma PROCStart III. <http://www.gidel.com/PROCStar%20III.htm>. Consultado em: 7 de Agosto de 2010.

[69] Medeiros, Shelly Cristiane D'Avila. Inversão de Parâmetros em Dados Sísmicos por Algoritmos Genéticos. 2005. Dissertação de Mestrado. PUC-Rio, Rio de Janeiro.

[70] F. Berthelot, F. Nouvel, D. Houzet. Partial and dynamic reconfiguration of FPGAs: a top down design methodology for an automatic implementation. In: Parallel and Distributed Processing Symposium, International, p. 209, Proceedings 20th IEEE International Parallel & Distributed Processing Symposium, 2006.

[71] The Group of Twenty (G-20). <http://www.g20.org/index.aspx>. Consultado em: 26 de agosto de 2010.

[72] Goldman Sachs Global Research Centres. DreamingWith BRICs: The Path to 2050. Global Economics Paper No: 99. Outubro de 2003.

[73] Projeto pré-sal. <http://www.petrobras.com.br/pt/energia-e-tecnologia/tecnologia-e-pesquisa/atuacao-no-presal/>. Consultado em: 26 de agosto de 2010.

[74] Jianjiang Li, Dan Hei, Lin Yan. Partitioning Algorithm of 3-D Prestack Parallel Kirchhoff Depth Migration for Imaging Spaces. In: Grid and Cooperative Computing, 2009. GCC '09. Eighth International Conference, 2009.

[75] CUDA. < http://www.nvidia.com.br/object/what_is_cuda_new_br.html>. Consultado em: 26 de agosto de 2010.

[76] Ian Kuon, Russell Tessier, Jonathan Rose, FPGA Architecture: Survey and Challenges. In: *Foundations and Trends in Electronic Design Automation*, Vol. 2, No. 2, pp 135-253, 2008.

[77] James Myers. The Importance of Recurring + Nonrecurring Costs. March 2007.

[78] Silva, Max Wanlemberg Xavier. Migração Reversa no Tempo com diferentes condições de imagem. 2009. Dissertação de Mestrado. Universidade Federal do Rio de Janeiro, Rio de Janeiro.