Pós-Graduação em Ciência da Computação

IGOR RAFAEL DE OLIVEIRA PONA

PROPOSTA DE UMA IMPLEMENTAÇÃO OTIMIZADA DO ALGORITMO RTM.3D EM OPEN.CL PARA PLATAFORMAS BASEADAS EM FPGAS

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

RECIFE
2016

Igor Rafael de Oliveira Pona

PROPOSTA DE UMA IMPLEMENTAÇÃO OTIMIZADA DO ALGORITMO RTM.3D EM OPEN.CL PARA PLATAFORMAS BASEADAS EM FPGAS

Dissertação de Mestrado apresentada ao Programa de Pós-graduação em Ciência da Computação da Universidade Federal de Pernambuco como requisito parcial para a obtenção do título de Mestre em Ciência da Computação.

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

RECIFE
2016

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

P792p Pona, Igor Rafael de Oliveira
Proposta de uma implementação otimizada do algoritmo RTM.3D em OPEN.CL para plataformas baseadas em FPGAs / Igor Rafael de Oliveira Pona. – 2016.
66 f.: il., fig., tab.

Orientador: Manoel Eusébio de Lima.
Dissertação (Mestrado) – Universidade Federal de Pernambuco. CIn, Ciência da Computação, Recife, 2016.
Inclui referências e apêndices.

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

621.39 CDD (23. ed.) UFPE- MEI 2017-116

Igor Rafael de Oliveira Pona

Proposta de uma Implementação Otimizada do Algoritmo RTM.3D em OPEN.CL para Plataformas baseadas em FPGAs

Dissertação de Mestrado apresentada ao Programa de Pós-graduação em Ciência da Computação da Universidade Federal de Pernambuco como requisito parcial para a obtenção do título de Mestre em Ciência da Computação.

Aprovado em: 08/09/2016.

BANCA EXAMINADORA

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

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

Prof. Dr. Manoel Eusébio de Lima
Centro de Informática/UFPE
(**Orientador)**

Agradecimentos

Agradeço a minha mãe Simone, pelo apoio, motivação, amizade e amor. Ao Marcelo que é mais do que um pai, um grande amigo. Aos meus irmãos Ytalo e Yasminy pelo companheirismo e amizade. A todos que fazem parte desta grande família que vai além dos laços sanguíneos.

Agradeço aos meus professores, por seus ensinamentos, dedicação, paciência e amizade. Ao professor Manoel Eusebio, pela oportunidade, conselhos e orientação. Aos professores do grupo HPCin que me deram suporte no decorrer desta pesquisa, e contribuíram com esta dissertação.

Aos amigos que fiz no grupo HPCin, Joelma Souza, Severino José (Biu) e Thyago Maia, pela amizade, ajuda e disposição de compartilhar conhecimentos.

Sou grato a todos que contribuíram na minha caminhada até este momento.

Resumo

A demanda por sistemas de alto desempenho cresce junto ao desenvolvimento científico e econômico e dentro das mais diversas áreas, passando por modelagens científicas, inteligência artificial, criptografia, computação em nuvem, etc. A prospecção de petróleo e gás natural faz parte desses sistemas, exigindo o processamento de dados com um volume acima dos Terabytes e ao custo de semanas ou meses de execução, no intuito de procurar bolsões no subsolo; além de sua importância estratégica devido ao pré-sal. Essa procura faz uso da equação acústica de propagação de onda, e apresenta como uma de suas soluções o método de diferenças finitas (MDF) pelo algoritmo de RTM (*Reverse Time Migration*). Essa solução demanda uma grande quantidade de operações em ponto flutuante, exigindo hardwares com arquiteturas dedicados a essa finalidade como FPGAs e GPGPUs. Neste trabalho fazemos uma análise sobre essas arquiteturas para o algoritmo RTM em OpenCL na sua versão 3D, assim como as possíveis otimizações ao se aproveitar da portabilidade do código em OpenCL de GPGPUs para FPGAs. Avaliamos os recursos utilizados em sínteses feitas pelo SDK OpenCL da Altera para o FPGA Stratix V A7, para em um segundo momento, desenvolver um código que tenta otimizar o uso desses recursos que estão disponíveis no FPGA. E por fim, analisamos os resultados obtidos frente a outras arquiteturas.

Palavras-chave: HPC. FPGA. OpenCL. RTM 3D.

Abstract

The high-performance computing systems increase with scientific and economic development through several fields like scientific modeling, artificial intelligence, cryptography, cloud computing, etc. The oil and natural gas extraction is among of these systems, requiring data processing with sizes greater than Terabytes and with the cost of weeks or months of execution time, in order to look for underground reservoir; as well as its strategic importance due to the pre-salt. The oil extraction makes use of acoustic wave equation, and has the finite difference method (FDM) as one of your solutions through the algorithm of RTM (Reverse Time Migration). This solution requires a lot of floating point operations and a hardware with dedicated architecture as FPGAs and GPGPUs. This work we analyze these architectures to implement the RTM 3D algorithm with OpenCL, as well as the possibly of take advantage of code portability of OpenCL for FPGAs GPGPUs. We evaluate the resources used in syntheses made by the OpenCL SDK Altera Stratix V A7 FPGA, and in a second moment, to develop a code that attempts to optimize the use of these resources that are available in the FPGA. Finally, we analyze the results against other architectures.

Keywords: HPC. FPGA. OpenCL. RTM 3D.

Lista de ilustrações

Lista de tabelas

Lista de abreviaturas e siglas

ALM	Adaptive Logic Modules
API	Application Programming Interface
ARM	Advanced RISC Machine
ASIC	Application Specific Integrated Circuits
AVX	Advanced Vector Extensions
CLB	Configurable Logic Block
CMOS	Complementary Metal-Oxide-semiconductor
CPU	CentralProcessing Unit
CUDA	Compute Unified Device Architecture
EUVL	Extreme Ultraviolet Lithography
FPGA	Field Programmable Gate Array
GDDR	Graphics Double Data Rate
GPGPU	General Purpose GPU
GPU	Graphics Processing Unit
HDL	Hardware Description Language
IOB	Input/Output Block
LAB	Logic Array Blocks
LLVM	Low Level Virtual Machine
LUT	Look-Up Table
MDF	Migração por Diferenças Finitas
MIMD	Multiple Instruction Multiple Data
MISD	Multiple Instruction streams, Single Data stream
OpenCL	Open Computing Language
RTM	Reverse Time Migration
SDK	Software Development Kit
SIMD	Single Instruction Multiple Data

SISD	Single Instruction stream, Single Data stream
SSE	Streaming SIMD Extensions

Sumário

1 Introdução

No decorrer dos anos da fabricação de circuitos integrados em *wafers* (pratos) de materiais semicondutores, como o silício, a fotolitografia é a técnica mais comum em sua produção. Essa técnica permite gravar vias e componentes eletrônicos pela irradiação de luz ultravioleta, passando por mascaras que definem a deposição de metais nos wafers e vão se sobrepor em camadas para a formação do circuito. Assim, é possível produzir circuitos que ocupam uma menor área e um menor consumo de potência elétrica, pois como o menor tamanho é possível obter transistores que operam com uma menor resistência elétrica e irradiam menos calor. No entanto, durante os anos, surgiram questões e paradigmas referentes a sua concepção e utilização, como o limite de frequência de operação em temperatura ambiente, o limite de tensão, o limite no paralelismo de instruções para uma determinada estrutura de dado e o limite na quantidade de vias físicas, ou tamanho de palavra no circuito. Esses paradigmas motivaram o aumento da densidade de transistores em uma mesma área, para obter um maior desempenho e um maior número de funções lógicas, como propõem a lei de Moore 1965 (MOORE'S...). Mas a partir de 2006 esse movimento vem perdendo força (HENNESSY; PATTERSON, 2011, p. 100), principalmente com as dificuldades impostas durante o processo de fabricação.

Essa dificuldade se dá devido ao tamanho do comprimento de onda da luz ultravioleta que é de 193 nanômetros (nm), e é maior do que as fissuras nas mascaras (16-10mn), criando defeitos durante o processo de litografia e por vezes exigindo um maior tempo de exposição e um maior número de mascaras na fabricação. Atualmente, as projeções indicam essa dificuldade de produção em circuitos abaixo de 10 nanômetros, usando as técnicas e materiais vigentes, como o processo de nanolitografia em ultravioleta (EUVL - *Extreme ultraviolet lithography*) Wu e Kumar (2014), janice m golda (2016), Ann Steffora Mutschler (2015).

Em decorrência desta dificuldade de produção dos circuitos e no intuito de manter o crescimento no desempenho, os fabricantes optaram por aumentar o paralelismo pelo número de cores em um mesmo chip, ou pelo uso de instruções vetoriais. No entanto, o conceito de paralelismo é mais antigo e surge com Flynn em 1966, Hennessy e Patterson

(2011, p. 197). Flynn define a taxonomia de arquiteturas paralelas em quatro categorias; são elas: SISD (*Single instruction stream, single data stream*), SIMD *(Single Instruction Multiple Data*), MISD (*Multiple instruction streams, single data stream*) e MIMD (*Multiple instruction streams, multiple data streams*). Esta categorização se dá pelo tipo de dependência lógica das operações ou instruções, e da estrutura do dado a ser processado por essas instruções. Portanto, SISD é a categoria que apresenta a dependência tanto no sequenciamento lógico, quanto na estrutura de dado; SIMD possuímos um conjunto de dados que apresenta o mesmo sequenciamento lógico e é independente na estrutura de dado; MISD é a categoria que permite aplicar múltiplas sequencias lógicas a uma mesma estrutura de dado; e por fim, MIMD é a categoria que não apresenta nenhuma dependência, nem no sequenciamento lógico, e nem na estrutura de dado. Essa abstração lógica pode ser extrapolada tanto a nível de Hardware, quanto a nível de Software, para além dos processos em execução no sistema operacional; como por exemplo, uma aplicação que roda em *Clusters* espalhados em vários *Data Centers*.

Para o cenário no qual o volume de dados processado cresce junto ao número de usuários e sistemas voltados para internet, e junto ao desenvolvimento cientifico que exige modelos de maior complexidade e quantidade de dados - E como exemplo desse cenário podemos citar: o *Big Data*, modelos científicos de física quântica, processamento de imagens, inteligência artificial, criptografia e modelos sísmicos com sensores mais apurados e com uma maior resolução - Em resumo, problemas atuais para os quais o desempenho é obtido pela melhoria no design das arquiteturas para favorecer o paralelismo e que consideram as limitações de desempenho atuais. Para esse cenário, é reforçada a necessidade de se trabalhar com sistemas que permitam tanto o paralelismo no processamento, quanto no paralelismo de dados. Esse último, por vezes, é alcançado ao se fazer interfaces com outros sistemas, que podem ser de arquiteturas distintas, heterogêneas. Essas plataformas heterogênias fornecem uma escalabilidade e estabilidade, sem grandes dependências da arquitetura, e dando suporte as instruções do tipo SIMD e MIMD. Com essa finalidade surge a *Open Computing Language* (OpenCL), uma linguagem (API) para escrever programas que funcionam em diferentes arquiteturas (KHRONOS.ORG).

Contudo, o processo de usar sistemas com arquiteturas distintas pode apresentar uma grande diferença de desempenho e dificuldades ao se portar o código de uma arquitetura para outra. Essa dificuldade se dá principalmente ao tipo de dependência de dados que o problema apresenta, algumas características intrínsecas do algoritmo e da arquitetura empregada. São esses fatores que vão determinar se é possível usar o paralelismo na solução do problema.

Portanto, portar código entre diferentes dispositivos pode apresentar diversos problemas, e dentre eles, o baixo desempenho. E é neste aspecto que esse trabalho se propõem a estudar, demonstrado as otimizações que são necessárias para portar um

código de GPU para FPGA. Realizando uma análise de cada arquitetura envolvida nesse processo com o intuito de diminuir o esforço necessário para implementar esses sistemas heterogêneos.

Essa analise fará uso do algoritmo de *Reverse Time Migration* em sua versão 3D (RTM 3D), o qual é utilizado no processo de modelagem e migração em superfícies de relevo complexos, que possui uma grande importância na prospecção de petróleo e gás natural, principalmente após o pré-sal. Este algoritmo requer o processamento de uma quantidade massiva de dados, portanto, um grande esforço computacional. Problemas reais, como a análise do subsolo em áreas como a do pré-sal, exigem processamento de banco de dados da ordem de terabytes de informação, o que pode representar semanas a meses de computação. (MEDEIROS, 2013, p. 15).

E para viabilizar o processamento dessa quantidade de dados, usamos as seguintes plataformas na implementação deste algoritmo em OpenCL; plataforma da Nallatech 385, com o dispositivo FPGA Stratix V A7 da Altera, plataforma GTX 580 com a microarquitetura *Fermi*, da Nvidia e a plataforma HD 7970 Tahiti da AMD com microarquitetura "*GCN 1st gen*". E com o algoritmo RTM 3D para cada uma dessas plataformas, realizaremos um estudo comparativo do desempenho computacional com ênfase na plataforma que possui FPGA. Demostrando o quão é necessário a otimização do código em OpenCL para estes tipos de dispositivos, frente aos tradicionais sistemas baseados em GPUs, no sentido de se conseguir um melhor desempenho computacional. Assim, o objetivo principal desta dissertação é demonstrar que embora o uso de OpenCL facilite bastante a codificação e compilação de algoritmos para dispositivos com FPGAs, esta linguagem quando escrita para GPU, ainda é ineficiente em aproveitar todos os recursos disponíveis no FPGA. Sendo necessário uma análise e modificação adicional do código em OpenCL.

Esta dissertação está dívida em 7 capítulos; no capítulo 2 é apresentada uma breve revisão bibliográfica sobre a modelagem e migração sísmica, fornecendo alguns fundamentos necessários a esta dissertação, nos conceitos de prospecção e da origem do RTM. Nas sessões seguintes são apresentados os conceitos e fundamentos do OpenCL e as arquiteturas utilizadas na dissertação, FPGA e GPUs. No capítulo 3 apresentamos os trabalhos relacionados. No capítulo 4 discute-se a implementação do algoritmo RTM 3D, o código para GPU e FPGA, assim como a síntese resultante do SDK OpenCL da Altera. No capítulo 5 discute-se em detalhes as otimizações necessárias para melhorar o desempenho do RTM 3D no FPGA. Em seguida, no capítulo 6, os resultados são apresentados. Finalmente no capítulo 7 são apresentadas as conclusões da dissertação e elencados os trabalhos futuros.

2 Fundamentação Teórica

Nesta seção relacionamos os principais conceitos sobre a prospecção de petróleo, incluindo uma visão geral da aquisição de dados, modelagem e migração. Detalhando o processo de migração com o algoritmo RTM (seção 2.1), e apresentamos a API do OpenCL (seção 2.2), que é a linguagem utilizada na implementação, e em seguida mostramos as características das arquiteturas utilizadas neste trabalho (seção 2.3).

2.1 Prospecção de petróleo e gás natural

O processo de prospecção de petróleo e gás natural pode ser divido em três principais etapas: aquisição, processamento e interpretação (YILMAZ, 2001), e ocorre tanto em terra quanto em mares e oceanos. No caso da prospecção em mares e oceanos, um navio de pesquisa sísmica e destacado para a tarefa, e conforme exemplificado na Figura 1, o processo de aquisição depende da produção de ondas sonoras que são geradas por um canhão de ar comprimido, propagando ondas nas camadas do subsolo. Essas ondas são refletidas ou refratadas conforme as diferentes características do solo, até serem capturadas por sensores especiais que neste caso são os hidrofones. Após o processo de aquisição, é gerado um modelo de referência de acordo com equações de onda, que vai se juntar as informações coletadas para, durante a etapa de processamento, ser realizada a migração sísmica, confrontando com as informações capturadas pelos hidrofones para gerar uma saída de dados que corresponde ao um *imageamento* das camadas do subsolo. É esse *imageamento* das camadas que é usado durante a última etapa de interpretação.

Sequem abaixo todas as 16 etapas do processo convencional de prospecção segundo (YILMAZ, 2001):

1. Preprocessing
   Demultiplexing
   Reformatting
   Editing
   Geometric Spreading Correction

Setup of Field Statics
2. Deconvolution and Trace Balancing
3. CMP Sorting
4. Velocity Analysis
5. Residual Statics Corrections
6. Velocity Analysis
7. Normal-Moveout Correction (NMO)
8. Dip-Moveout Correction (DMO)
9. Inverse NMO Correction
10. Velocity Analysis
11. NMO Correction, Muting and Stacking
12. Deconvolution
13. Time-Variant Spectral Whitening
14. Time-Variant Filtering
15. Migration
16. Gain Application

Esse trabalho aborda a 15º etapa do processo de prospecção, que é a migração sísmica - *Migration*. O processo de migração sísmica pode ser obtido através de vários métodos, conforme resumido por KRÜGER (2012) na Figura 2. E cada um desses métodos acabam pertencendo a uma categoria, conforme descrito no artigo de Gray et al. (2001) em um resumo histórico sobre modelagens e migrações sísmicas. O último autor descreve que a migração por tempo, *Time migration*, é executada construindo previamente um campo de velocidade que estima e avalia em cada ponto da imagem, o tempo necessário à onda ir do emissor ao receptor, considerando as reflexões dos obstáculos de ângulos próximos ao perpendicular do emissor. Esses objetos perpendiculares são os vários níveis de camadas do solo, e o tempo medido considerar o efeito NMO/stack (*Normal Moveout*), que é um procedimento para compensar a distância entre emissores e os diferentes receptores, considerando um mesmo ponto no subsolo. Já na migração por profundidade, Gray et al. (2001) descreve que o campo não é de velocidade como no método anterior, mas sim de intervalos construídos por médias de velocidades que podem ser medidas pelo geólogo no local, usando equipamentos e técnicas distintos para apresentar resultados mais próximos das coordenadas reais do subsolo. No entanto, a migração por profundidade é mais custosa do que a migração por tempo, e as análises são interpretadas com ênfase nas frequências e no tempo, com a precisão da distância entre a superfície e as camadas do subsolo, em segundo nível de importância.

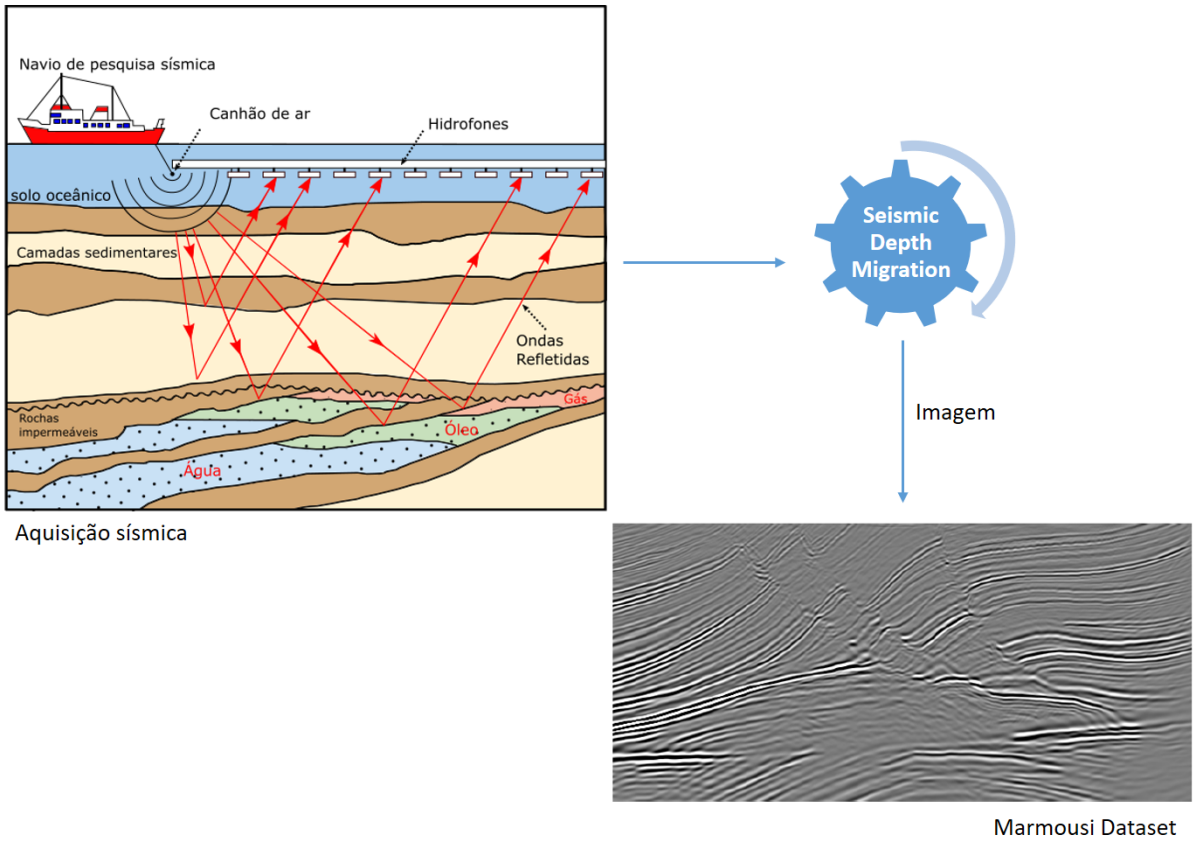
Uma opção mais interessante para a resolução deste tipo de modelagem e migração é o algoritmo RTM. O método, baseado na resolução da equação de onda acústica e seu processamento através de diferenças finitas (MDF), permite uma maior fidelidade

nas imagens geradas em superfícies irregulares. Este método, no entanto, é mais custoso computacionalmente, exigindo uma massa de dados em seu processamento bem maior que o método de Kichhoff e, por conseguinte, mais recursos computacionais.

O método de Kirchhoff, que é menos custoso, gera resultados não tão precisos para regiões subterrâneas onde as camadas de rocha não são regulares e estão situadas a altas profundidades, como no pré-sal. Panetta et al. (2007) faz referência aos problemas de interferência e ruídos no *imageamento*.

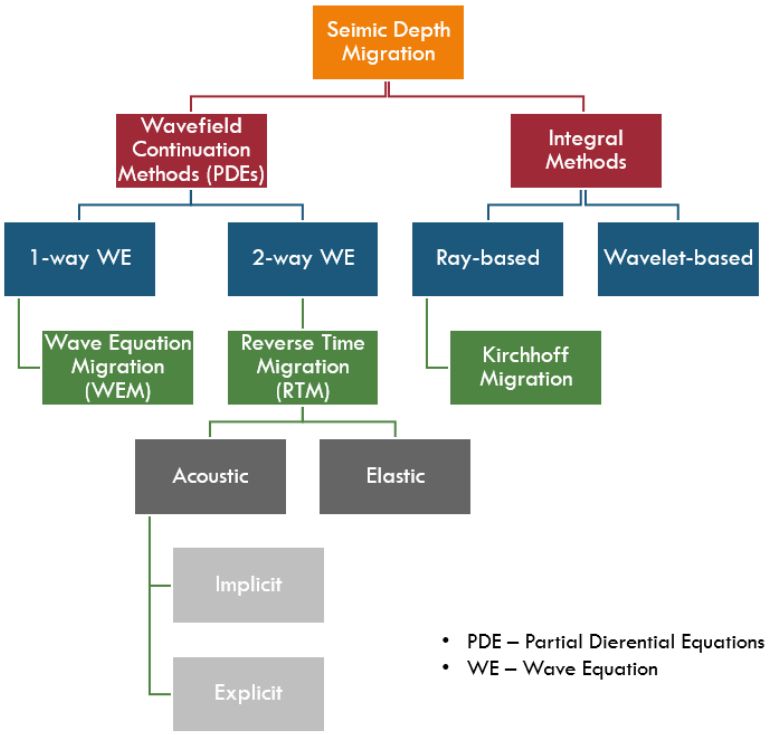
Por fim, Gray et al. (2001) descreve que melhores resultados são obtidos ao reverter o processo de análise, usando diferenças finitas, por onde o problema e abordado da perspectiva inicial aos receptores. Assim o algoritmo de RTM é apresentado como opção valiosa quando o foco é a qualidade do resultado, ao mesmo tempo em que demanda de um relevante esforço computacional Medeiros (2013). O algoritmo RTM em particular é descrito com mais detalhes na seção 4.

Figura 1 – Prospecção de petróleo e gás natural.



Fonte: Imagem de Aquisição sísmica baseada em similar encontrada em: Aquisição... (2016); Marmousei Dataset: KRÜGER (2012).

Figura 2 – Modelagem e migração sísmica - Métodos.



Fonte: KRÜGER (2012).

2.2 OpenCL

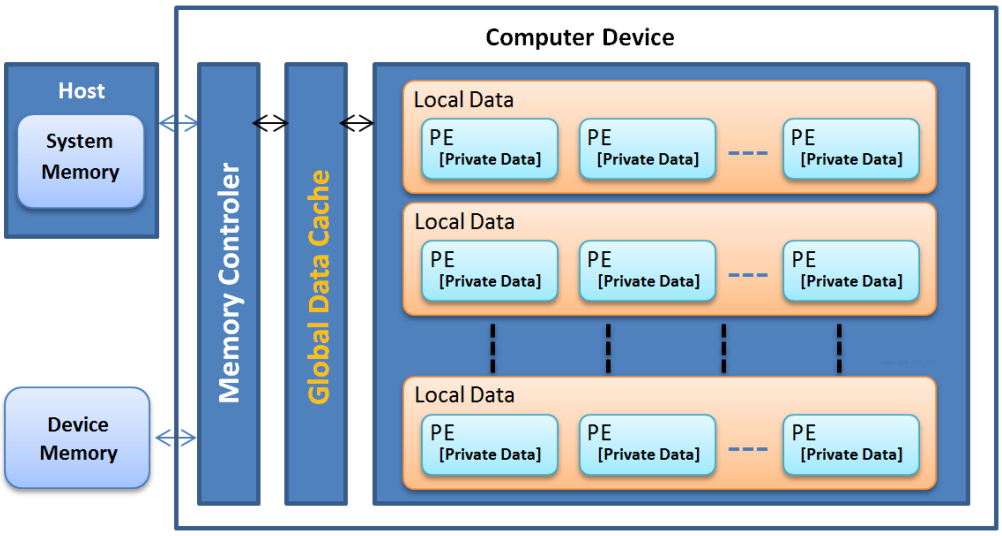
Open Computing Language, ou OpenCL, é um padrão de linguagem para cálculos que pode ser usado em diferentes plataformas, e é mantido por um consórcio chamado Khronos Group, ao qual fazem parte várias empresas como: Altera, AMD, Apple, Intel, Nvidia e Xilinx. Sua linguagem é baseada em C99 (ISO/IEC 9899:1999) e sua API permite controlar dispositivos heterogêneos com ênfase em algumas formas de paralelismo. Desta forma é possível portar o código para diferentes arquiteturas de processadores como CPUs, GPUs, DSPs e FPGAs.

O código necessário para usar esses dispositivos de cálculo é dividido em duas partes, o código do *Host* e o código do *Kernel*. O código do *Host* é responsável por instanciar, configurar e controlar os dispositivos através de uma API, e também instanciar os dados que serão enviados e recebidos para o dispositivo durante a execução. Portanto o código do programa é escrito em C com as extensões "[.c]" e "[.h]", onde podemos obter informações sobre a plataforma e os dispositivos que a API está acessando. Desta forma, com as informações acessadas, podemos alocar os dados em *buffers* de memória que em seguida serão transferidos para o dispositivo. Já o código do *Kernel* recebe os dados dos *buffers* e os envia para cada unidade de processamento, conforme programado no código de extensão "[cl]", que possui os métodos a serem executados no dispositivo. Esse código deve levar em conta o paralelismo intrínseco da arquitetura do dispositivo. Ao final do processamento o *host* pode fazer a leitura dos resultados no dispositivo. Esse controle

também pode ser escrito por chamadas de eventos na API.

Para o OpenCL, cada dispositivo possui um conjunto de unidades de processamento, e dentro de cada unidade, um conjunto de elementos de processamento (PEs, Figura 3). Cada PE é responsável por executar uma *Thread*. Esses dois níveis compartilham quatro níveis hierárquicos de memória: *Global*, *Read-only*, *local* e *private* (KHRONOS.ORG).

Figura 3 – OpenCL - Computer Device.



Fonte: Autor.

- *Global Data* é uma região de memória compartilhada para todos os PE, possuindo uma maior latência.
- O *Read-only* pode ser lido por todos os elementos, assim como o *Global Data*, e é escrito apenas pelo *Host*, possui uma menor latência do que a Global, apesar de estar no mesmo nível.
- A *Local Data* é compartilhada dentro de um mesmo grupo de PE, servindo para compartilhar valores sem uma grande penalidade no tempo de acesso.
- A *Private Data* são os registradores de cada PE e com a menor latência no tempo de acesso, e sem penalidade de concorrência.

Ao se estruturar o código do *Host* e do *Kernel* é importante considerar as estruturas de divisão na memória e nas unidades de processamento de cada dispositivo, adequando o código ao paralelismo do problema. No *Kernel*, além desta hierarquia de memória, temos os comandos que auxiliam na sincronização das *threads* ao acesso de memória em níveis compartilhados, como *barriers*.

Em GPUs e CPUs, o código do *kernel* é compilado em tempo de execução (*run-time*) e enviado ao dispositivo, isso acarreta em uma maior dificuldade em proteger propriedade intelectual. Devido a esse aspecto, recentemente há uma procura em ofuscar esse código através da máquina virtual LLVM (*Low Level Virtual Machine*), como o SPIR (*Standard*

Portable Intermediate Representation) (KHRONOS.ORG). Assim é possível entregar um código pré-compilado para ser executado na LLVM. Esse problema pode ser evitado nos FPGAs devido ao próprio processo de configuração do dispositivo, como será demostrado na seção 4.4.

Para a execução do OpenCL em FPGAs, o código do *kernel* pode sofrer adição de macros e de atributos para serem usados durante a síntese, auxiliando a configuração do paralelismo. A síntese do código é feita em algumas etapas anteriores a execução, onde o código é convertido para uma linguagem de descrição de Hardware, HDL, e também é adicionado um *wrapper* para comunicação com os barramentos e memória. Por fim, é gerado um *bitStream* que configurará o FPGA. Esse processo de otimização será abordado por completo na seção 4.4.

2.3 Arquiteturas FPGA e GPU

2.3.1 FPGA - Field Programmable Gate Array

Dispositivos lógicos reconfiguráveis, como os *Field programmable Gate Arrays* (FPGAs), são circuitos eletrônicos integrados, que podem ter sua lógica modificada, reconfigurada pelo usuário, em função do algoritmo a ser implementado em seu núcleo. Este comportamento é bem diferente da grande maioria dos circuitos tradicionais, como GPUs e CPUs, que embora possam ser programados, não podem ter seus circuitos internos customizados para outras funções diferentes daqueles previamente estabelecidas em seu projeto original.

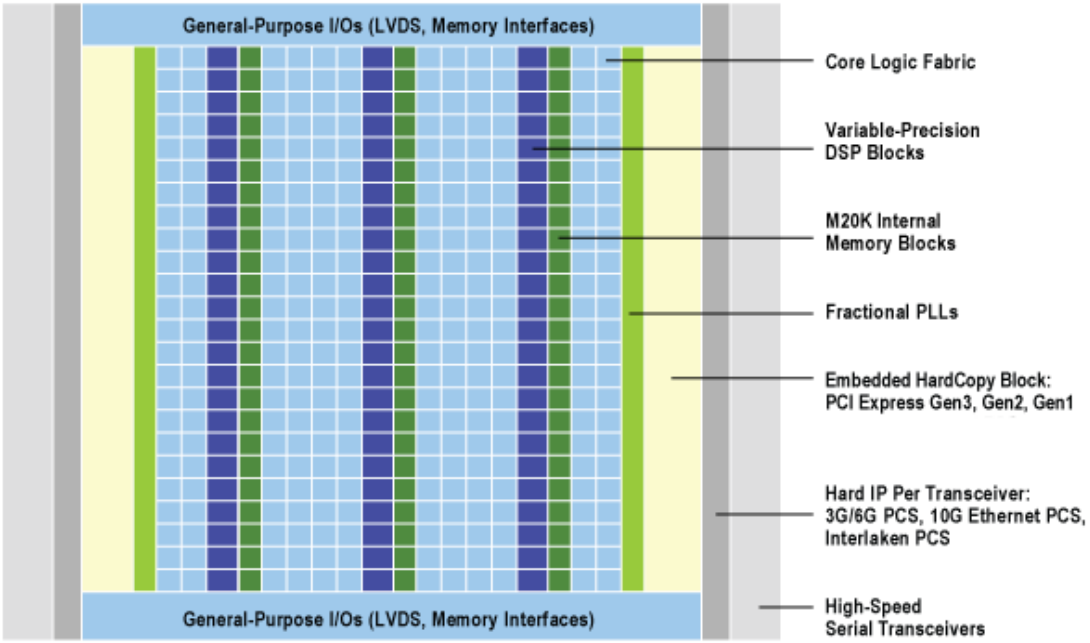
Os FPGAs são compostos por uma série de componentes, tais como: blocos lógicos, bancos de memória, cores de microprocessadores, os quais, através de uma malha de roteamento podem ser utilizados, de acordo com a funcionalidade desejada pelo projetista. Esses blocos, ou células lógicas, recebem o nome de CLBs (*Configurable Logic Block*) que são conectados por várias matrizes de roteamento até os blocos que fazem a entrada/saída (*input/output*), os IOB. Além desses, temos os blocos que são responsáveis por fazer conexões e podem receber vários nomes como *crossbar switch, cross-point switch, matrix switch* e outros, dependendo da tecnologia e do fabricante.

Neste trabalho usamos a placa 385-A7 da Nallatech, a qual possui como elemento principal de processamento o FPGA Stratix V GX A7 da Altera. Sua arquitetura é análoga ao explicado acima, com algumas diferenças discutidas a seguir.

Na Stratix V os CLB são chamados de LAB – *Logic Array Block* – (Mostrados na Figura 4 como o nome de *Core Logic Fabric*). Usando a metade de um LAB pode-se formar uma MLAB ou *Memory LAB*.

Os LAB são formados por blocos chamados ALM – *Adaptive Logic Modules* – que

Figura 4 – Arquitetura Stratix V.



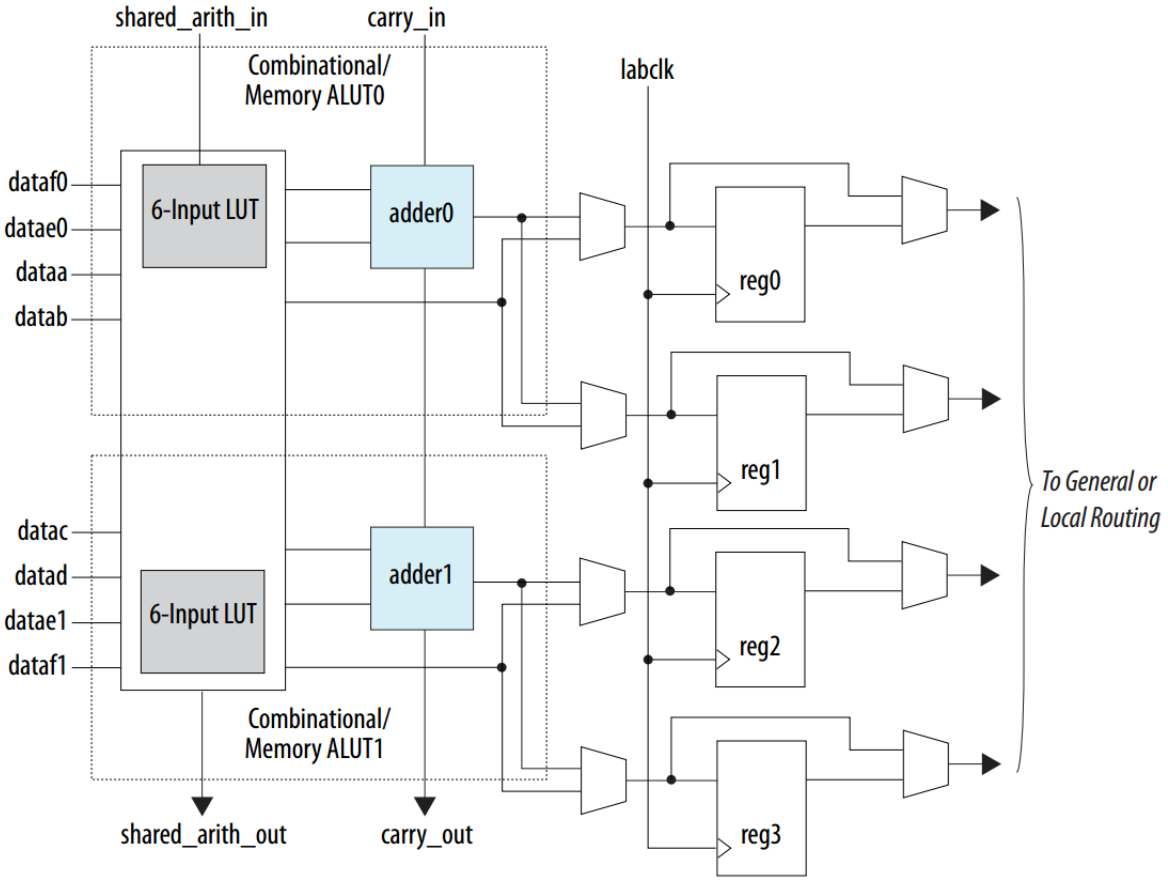
Fonte: Altera (2010a)

são compostos basicamente de duas LUTs, dois somadores e quatro registradores, Figura 5. Um ALM possui quatro modos de operação: *Normal, Extended LUT, Arithmetic e Shared arithmetic*. No modo normal é possível fazer combinações das entradas das duas LUTs, dividindo em duas funções separadas ou usando como uma mesma função. Já no modo *Extended LUT*, quando as sete primeiras entradas não fazem o uso dos registradores, o oitavo pino pode fazer uso direto de um registrador. No modo *Arithmetic* são usados dois conjuntos de quatro entradas para os dois somadores, e é possível fazer *Carry Chain* entre os ALMs. No modo *Shared Arithmetic* é possível usar o somador com três entradas em uma LUT, resultando em uma função de seis entradas, também é possível usar esse modo em *Carry Chain* (Altera, 2010b).

Além das ALMs temos outros blocos que auxiliam ao montar as funções logicas, como os DSP que são responsáveis por cálculos de ponto flutuante de 9, 18, 27 e 36-bit, também é possível somar, subtrair e acumular operações de 64-bit; os *Fractional PLLs* que fazem a sincronia dos pulsos de *clock*, podem dividir ou multiplicar os pulsos; M20k Memory são blocos dedicados ao armazenamento de dados entre as operações. Por fim, temos os blocos de interface *I/O: PciExpress, Transceiver e Serial*.

Abaixo segue a tabela de Especificações do Stratix V.

Figura 5 – ALM – Adaptive Logic Module.



Fonte: Altera (2010b)

Tabela 1 – Stratix V - Especificações

Product Line	5SGXA7
LEs (K)	622
ALMs	234.720
Registers	938.880
M20K memory blocks	2.560
M20K memory (Mb)	50
MLAB memory (Mb)	7.16
Variable-precision DSP blocks	256
18 x 18 multipliers	512

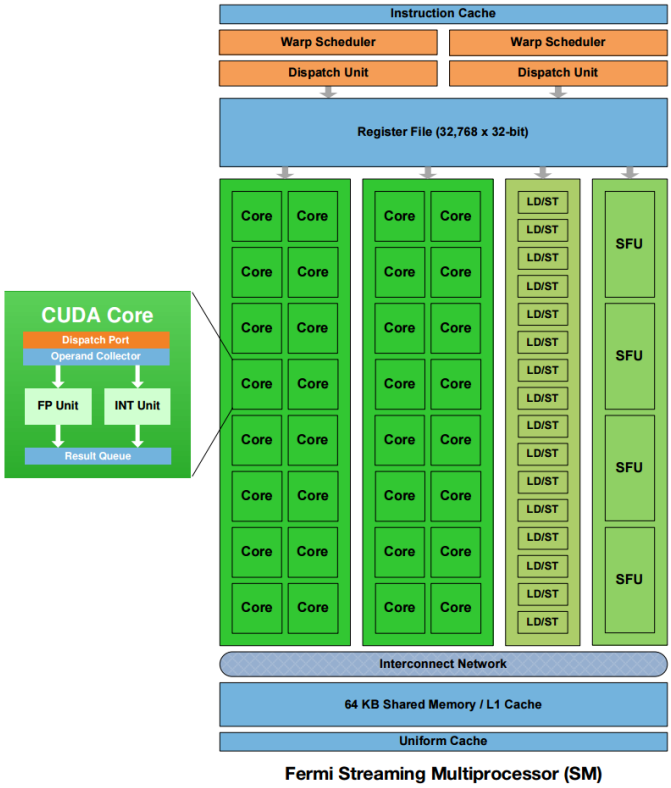
2.3.2 GPU - Graphics Processing Unit

A necessidade de processadores gráficos mais potentes tem aumentado no decorrer dos últimos anos; telas com maiores resoluções e novas tecnologias de captura, exibição, jogos eletrônicos e aplicações 3D levaram a uma grande necessidade de evolução nos processadores gráficos, GPU – *Graphics Processing Unit*. Essa evolução propiciou uma maior capacidade de cálculos paralelos, novas APIs e linguagens de programação que possibilitam sua adoção em outras aplicações científicas e empresariais. Assim surgiram

as GPGPU – *General Purpose GPU*, que é justamente para o uso destes processadores gráficos para além de seu propósito inicial, como simulações de modelos científicos, redes neurais, criptografia e outros. Hoje é possível contratar servidores virtualizados com acesso a essas GPGPUs para serem utilizadas em diversas aplicações. Também podemos citar a grande evolução desses dispositivos em sistemas embarcados e aparelhos móveis.

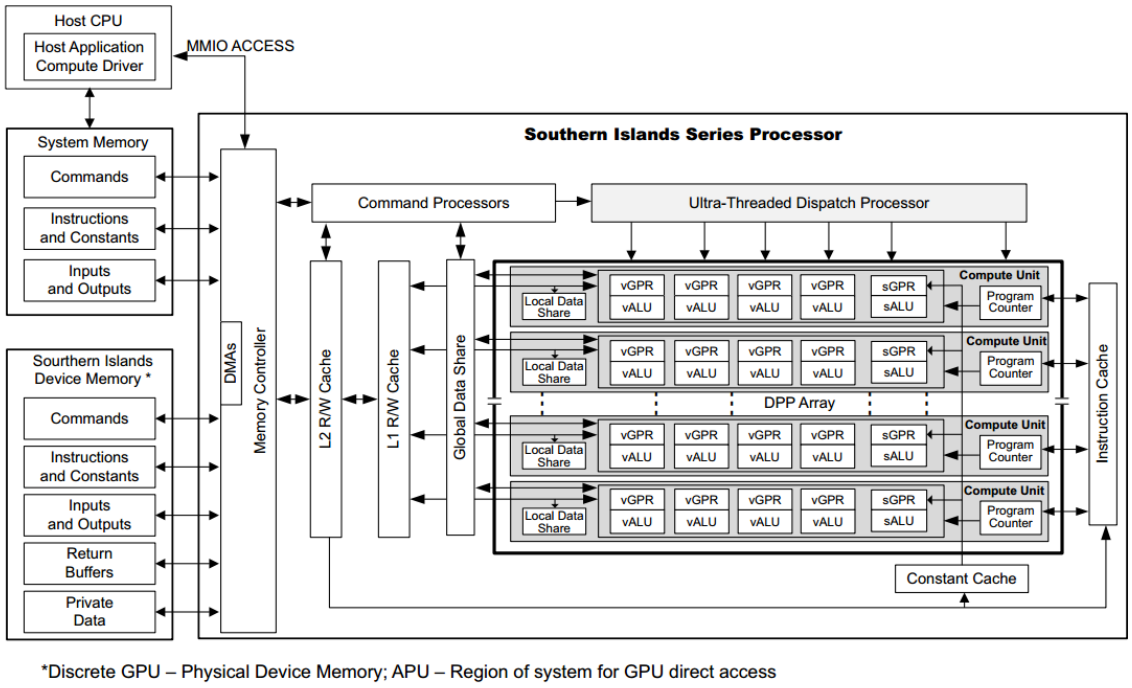
Essa evolução influenciou na arquitetura desses processadores gráficos, privilegiando operações vetoriais através de instruções SIMD – *Single Instruction, Multiple Data* – que são processadas em grupos com uma determinada quantidade de unidades de processamento ou PEs (Figura 3). Na Nvidia essas unidades são chamadas de "*Cuda Cores*" (Figura 6) e na AMD levam o nome de "*Compute Unit*" (Figura 7). São elas que executam *threads* nas GPGPUs. Outro aspecto dessa especialização aconteceu na arquitetura de memória, que foi modificada para trabalhar com uma maior quantidade de dados, as GDDRs. Atualmente, existem memórias GDDRs que trabalham com 256 bits até 512-bits de barramento, ou seja, podem ser acessados o equivalente a 16 valores ponto-flutuante de precisão simples de 32 bits. Hoje as GDDRs estão na sua quinta versão; são nessas memórias que os *buffers* enviados pelo *host* serão armazenados.

Figura 6 – Fermi Streaming Multiprocessor (SM).



Fonte: Nvidia (2010)

Figura 7 – Graphics Core Next "GCN 1.0"AMD Southern Islands Series Block Diagram.



Fonte: AMD - Advanced Micro Devices, Inc. (2012)

3 Trabalhos Relacionados

Os trabalhos relacionados a seguir foram selecionados pela similaridade de algoritmo, hardware ou técnica proposta. O primeiro artigo *Evaluation of successive CPUs/APUs/GPUs based on an OpenCL finite difference stencil* (CALANDRA et al., 2013) faz uso de OpenCL para implementar o algoritmo de diferenças finitas. O segundo artigo *Synthesis of Platform Architectures from OpenCL Programs* (OWAIDA et al., 2011), demonstra outra forma de se sintetizar códigos em OpenCL para FPGA. O terceiro artigo *Analyzing the energy-efficiency of sparse matrix multiplication on heterogeneous systems: A comparative study of GPU, Xeon Phi and FPGA* (GIEFERS et al., 2016), faz uso de algoritmos de multiplicação de matrizes com hardware de GPU e FPGA similares a este trabalho. O artigo *Scaling Reverse Time Migration Performance through Reconfigurable Dataflow Engines* (WEBER et al., 2011), faz uso de uma dataflow engine análogo ao proposto neste trabalho. E por fim, o artigo *Comparing Hardware Accelerators in Scientific Applications: A Case Study* (FU et al., 2014) faz um comparativo com hardwares similares a essa dissertação.

3.1 Evaluation of successive CPUs/APUs/GPUs based on an OpenCL finite difference stencil

No artigo de Calandra et al. (2013), temos uma avaliação do algoritmo de diferenças finitas 3D que foi implementado em OpenCL e faz parte do RTM. O artigo faz uso de três arquiteturas, CPU, GPU discreta e GPU integradas com foco em APUs (*Accelerated Processing Unit*). As APUs são SOCs com CPU e GPU integrados e tem a capacidade de compartilhar memória, evitado assim, a transferência dos *buffers* entre as memórias da CPU e da GPU. Cada um desses três tipos de dispositivos usou a melhor implementação considerando uma máscara de operação de 8º ordem, com uma estratégia de implementação vetorizada em Z similar ao deste trabalho.

Nos resultados são apresentados alguns gráficos, sendo um deles a comparação de performance em um domínio de execução igual a 1024x1024x32. O melhor resultado que CPU obteve foi de 8 GFlop/s, a melhor APU obteve um valor próximo ao de 48 GFlop/s,

já a GPU discreta obteve um valor de 256 GFlop/s. Esses valores foram obtidos em 10 execuções seguidas, mais o tempo de transferência para a memória principal.

Na conclusão o autor comenta que o fato da não existência do gargalo na transferência de dados através da interface PCI Express, favorece o desempenho das APUs. No entanto, a taxa de transferência entre a memória e a GPU da APU e a quantidade de memória, ainda são menores do que as encontradas nas GPUs dedicadas. Portanto, o desempenho ainda é maior nas GPUs dedicadas, por exemplo, a HD 7970 pode fazer mais de 500 Gflops (CALANDRA et al., 2013). Não é feito nenhuma avaliação de consumo elétrico nos testes, porém, o autor cita que pelo TDP em uma proporção de 3 APUs para uma GPU dedicada, pode haver uma melhor razão entre consumo e desempenho.

O artigo não apresenta um dispositivo de FPGA, e a GPU HD 7970 – Tahiti é do mesmo modelo que usamos neste trabalho.

3.2 Synthesis of Platform Architectures from OpenCL Programs

OWAIDA et al., abordam outra forma de se fazer a síntese para um FPGA a partir de um código OpenCL. O *Silicon-OpenCL* (SOpenCL) é uma extensão apresentada pelo autor que configura o FPGA para dar suporte a uma LLVM – *Low Level Virtual Maquine*. Portanto, é gerado um compilador que interpreta o código OpenCL e o transforma em um código intermediário (IR - *Intermediate Representation*) que será executado por essa LLVM.

A avaliação experimental é feita com seis aplicações escritas em OpenCL e C, em três configurações distintas Ca, Cb, Cc. A configurações Ca ocupa o mínimo de recursos (FUs e I/O), a Cc é a que usa o máximo de recursos, e por fim, a Cb é o intermediário entre as duas configurações anteriores. Nos resultados ele apresenta um gráfico para cada uma das seis aplicações, apresentando o tempo de execução e o clock resultante de cada síntese, para cada configuração. No caso das três primeiras aplicações, não há uma grande diferença no tempo de execução e no clock obtido. Isso se deve ao fato de serem simples benchmarks de adição, convolução e multiplicação vetorial. Assim o autor demostra que não há grandes perdas ao se adotar o SOpenCL nestes casos, e em seguida, nas três últimas aplicações, fica demostrado que na maioria dos casos há uma melhora no tempo de execução, quando o SOpenCL e a estrutura de cache proposta são utilizados.

A diferença deste método para o empregado nessa dissertação se deve ao fato de como o FPGA é configurado. No modelo do SDK OpenCL da Altera, o *Kernel* é convertido em módulos de HDL que serão integrados ao invólucro que dá suporte aos outros componentes da placa, sendo sintetizado em um binário que configurará o FPGA. Ou seja, cada *Kernel* requer uma nova síntese. Já o modelo de SOpenCL, o FPGA é configurado apenas uma única vez com a LLVM.

3.3 Analyzing the energy-efficiency of sparse matrix multiplication on heterogeneous systems: A comparative study of GPU, Xeon Phi and FPGA

Giefers et al. (2016) faz análise de performance e eficiência elétrica em quatro sistemas: CPU, Acelerador com múltiplos CPUs, GPUs e FPGAs usando respectivamente Intel Xeon E5-2630v2, Intel Xeon Phi 5110P, Nvidia Tesla K20 GK110 e Nallatech 385N_A7. Neste último caso foi feita a síntese usando o SDK OpenCL da Altera. E para executar essa análise foram usados dois tipos de algoritmos de multiplicação de matrizes o SpMV (*Sparse matrix-vector*) e SpMM (*Sparse matrix-matrix*), com dados obtidos de um subgrupo de 181 matrizes de uma coleção chamada *UF sparse matrix collection*.

O autor faz menção a limitação ao se transferir dados pela Pci Express, o que favorece a CPU por não depender desse fator para sua execução. Mas quando eliminado o custo da transferência de dados, a GPU e o Xeon Phi demostram um melhor desempenho do que CPU e o FPGA. No entanto, o FPGA foi notoriamente superior ao se analisar a eficiência energética (Gflops/W).

3.4 Scaling Reverse Time Migration Performance through Reconfigurable Dataflow Engines

Na referência (FU et al., 2014), o autor explica que a exploração de petróleo e gás exigem algoritmos de grande demanda computacional, e devido aos limites na taxa de transferência das memorias, é difícil escalar a desempenho em geral, mesmo aumentando o número de cores, pois o acesso a memória aumenta de forma geométrica. Assim ele demostra qual é a eficiência obtida ao se adotar os FPGAs, projetados com ênfase no fluxo de dados (*data-flow engine*).

Para a avaliação de desempenho ele usa dois modelos de FPGA: MAX2, MAX3, comparando com um sistema de dois processados Intel – Sandy Bridge, cada um com 8 núcleos, totalizando 16. Na CPU foi feita uma versão do algoritmo em OpenMP, assim foi obtido um speedup de 4.5x em relação a CPU e uma eficiência no consumo de energia de 10.2x. No entanto, esse artigo não faz nenhuma comparação com GPUs.

3.5 Comparing Hardware Accelerators in Scientific Applications: A Case Study

Neste estudo de caso foi usado o algoritmo de *Quantum Monte Carlo* para avaliar as seguintes arquiteturas de GPU, Randeon 4870, Firestream 9170, GTX 285, Geforce

9400m e o FPGA Virtex-4 LX160 Weber et al. (2011) utilizando as linguagens OpenCL, CUDA, Brook+ e C++, conforme a disponibilidade no dispositivo alvo. A implementação do FPGA foi realizada em VHDL.

Os autores comentam que o desempenho da Randeon 4879 não foi satisfatório quando comparado com as outras GPUs, tanto na linguagem destinada a plataforma, quanto usando o OpenCL. Eles acreditam que esse fato esteja ligado ao não suporte, na época, do nível de memória local, restando apenas o nível global. Assim quando o número de partículas usadas na simulação passa de 10^4, a Nvidia GTX 285 se mostrou superior no desempenho. Também demostra que o fato da linguagem Book+ usar o cache de textura, nível local de memória, da Firestream 9170 contribuiu para uma diferença significante no resultado final.

Eles ainda comentam que a Radeon 4870 possui uma configuração de hardware superior. Eles ainda afirmam que, com a exceção do VHDL, as linguagens forneceram uma facilidade no desenvolvimento e na depuração, mas esse fato citado ja é superado pelas ferramentas atuais, tanto da Altera quanto da Xilinx. Os autores também comentam que devido as essas dificuldades, o tempo de desenvolvimento em VHDL levou aproximadamente um ano. E por fim, eles comentam sobre a facilidade de sintaxe devido as abstrações para gerenciar *threads*, tanto no OpenCL, quanto em CUDA, junto à similaridade de suas sintaxes. E ainda afirmam que existe uma diferença de desempenho que favorece o uso de CUDA sobre OpenCL, em dispositivos da Nvidia.

3.6 Conclusões

Não foi encontrada, nas principais publicações da área nenhuma abordagem similar, e infelizmente, não foi possível a discussão diretamente a trabalhos relacionados com essa implementação, em OpenCL, do algoritmo para modelagem e migração computacional RTM 3D. Sendo assim, abordou-se trabalhos que tratam de exemplos que usam OpenCL em arquiteturas de interesse da dissertação, e seus respectivos desempenhos, em classes de problemas que requerem processamento de alto desempenho.

A Tabela 2 é utilizada apenas para referenciar os trabalhos nas tabelas a seguir. As Tabela 3 e 4 apresentadas mostram as arquiteturas utilizadas em cada trabalho descrito acima, e as análises que foram realizadas em cada um destes respectivamente, em termos de arquitetura. Nesta dissertação a ênfase será dada a três itens importantes quando tratamos de FPGAs: o speedup alcançado, por se tratar da solução de problemas que requerem um alto desempenho, a eficiência energética, medida aqui em termos de dissipação de potência e os limites encontrados nestes dispositivos em função dos recursos de lógica associada a cada componente.

A proposta desta dissertação, a ser apresentada no próximo capítulo, é portanto,

mostrar o quão ainda é deficiente a geração de sistemas descritos em OpenCL para FPGAs, em termos de desempenho, quando comparado a GPUs. Por outro lado, pode-se ver que é possível se obter, com pequenos ajustes no código do programa em OpenCL, um aumento expressivo de ganho de desempenho em FPGAs. Também é importante ressaltar, nesta discussão entre arquiteturas, o ganho relativo em dissipação de potência nos FPGAs em relação a CPUs e GPUs. Neste sentido, como uma segunda contribuição faz-se também uma análise do desempenho relativo das arquiteturas por Watt dissipado, desta feita tendo um problema RTM 3D como exemplo. Esta métrica é importante devido a necessidade cada vez maior de se economizar energia em *Data Centers*.

Tabela 2 – Trabalhos - Índice

	Trabalhos
1	Evaluation of successive CPUS/APUS/GPUS based on an OpenCL finite difference stencil
2	Synthesis of platform architectures from OpenCL programs
3	Analyzing the energy-efficiency of sparse matrix multiplication on heterogeneous systems: A comparative study of GPU, Xeon Phi and FPGA
4	Scaling reverse time migration performance through reconfigurable dataflow engines
5	Comparing hardware accelerators in scientific applications: a case study
6	Esta Dissertação

Tabela 3 – Trabalhos - Hardwares

	CPU	**GPU**	**FPGA**	**APU**	**Xeon Phi**
1	x	x	-	x	-
2	-	-	x	-	-
3	x	x	x	-	x
4	-	-	x	-	-
5	x	x	x	-	-
6	x	x	x	-	-

Tabela 4 – Trabalhos - Características

	Speedup	**Power Efficiency**	**Área - FPGA**	**Técnica**
1	x	x	-	Memória compartilhada - Pcie x Intregrated GPU
2	x	-	-	SOpenCL (Silicon-OpenCL)
3	x	x	x	SpMV, SpMM
4	x	x	x	Dataflow-Oriented (DFE)
5	x	-	-	QUANTUM MONTE CARLO
6	x	x	x	RTM 3D OpenCL FPGA

4 Implementação

Neste capítulo será demostrado como foi desenvolvida a etapa de modelagem sísmica do algoritmo RTM 3D, explicado em detalhes na seção 4.1, e sua implementação em GPU e FPGA são discutidas na seção 4.2 e seção 4.3. E por fim, é demonstrado como o código é sintetizado no arquivo que será aplicado ao FPGA seção 4.4.

4.1 O algoritmo RTM 3D

A prospecção de petróleo ou gás natural representa a procura por bolsões de menor densidade nas camadas do subsolo e independe se a pesquisa é feita em solo ou em meio líquido (mar, oceano, rio, etc.). Para detectar esses bolsões no subsolo um pulso sísmico é gerado, fazendo ondas se propagarem no meio até serem refletidas ou refratadas de volta aos sensores. Esses sensores podem ser de dois tipos: geofones ou hidrofones; para a terra ou mar respectivamente. As informações desses sensores serão usadas para gerar um modelo de referência, com a participação de geólogos em etapas do processo para gerar as entradas do algoritmo de RTM que resulta em um mapeamento dos possíveis locais das jazidas.

O modelo de referência é produzido a partir da equação de onda acústica/elástica para, de forma reversa ao tempo, ir comparando com os sinais detectados pelos sensores. E então formar o *imageamento* das camadas de diferentes densidades no subsolo. O algoritmo é baseado na equação de onda abaixo:

The laplace-transformed homogeneous 3D acoustic wave equation (SHIN; SHIN; CALANDRA, 2016) (LIU et al., 2013):

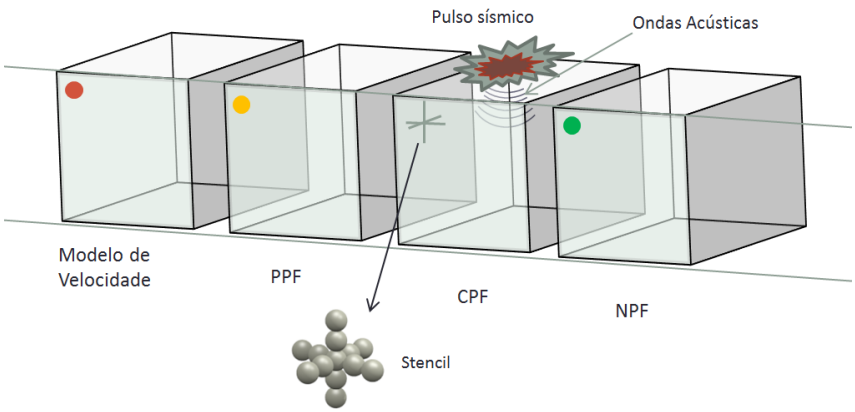
$$\frac{s^2}{c^2}p = v^2\left(\frac{\partial^2 p}{\partial x^2} + \frac{\partial^2 p}{\partial y^2} + \frac{\partial^2 p}{\partial z^2}\right) \tag{4.1}$$

Onde: *s - Positive Laplace damping constant; c - Acoustic wave velocity; p - Pressure wave field.*

O algoritmo RTM 3D usa um operador de aproximação de quarta ordem no espaço, pois são usados quatro pontos em cada dimensão além do ponto central. E de segunda

ordem no tempo, pois os valores atuais e o valor da matriz na última rodada também entram na equação. Assim, é necessária uma matriz para os valores de pontos no instante atual ou CPF (*Current Pressure Field*) e, outra de mesmo tamanho para o instante anterior ou PPF (*Previous Pressure Field*). Além delas, é necessária a matriz VEL que contém o modelo de velocidades. Os novos valores de campo de pressão calculados através dessas três matrizes são armazenados na matriz de campo de pressão seguinte, denominada NPF (*Next Pressure Field*). E para operar a propagação, é feita uma varredura de máscara (*Stencil*) na matriz de CPF como é demostrado na Figura 8, junto com a varredura de cada matriz, que são inicializadas com valores de ponto flutuante com 32 bit. Na Figura 9 o pseudocódigo de referência demonstra os valores do cálculo e como o valor do *Stencil* é obtido. Por fim, temos um diagrama de execução do *Host/Device* na Figura 10.

Figura 8 – *Stencil* e os pontos de VEL, PPF e CPF que resultam no NPF



Fonte: Autor.

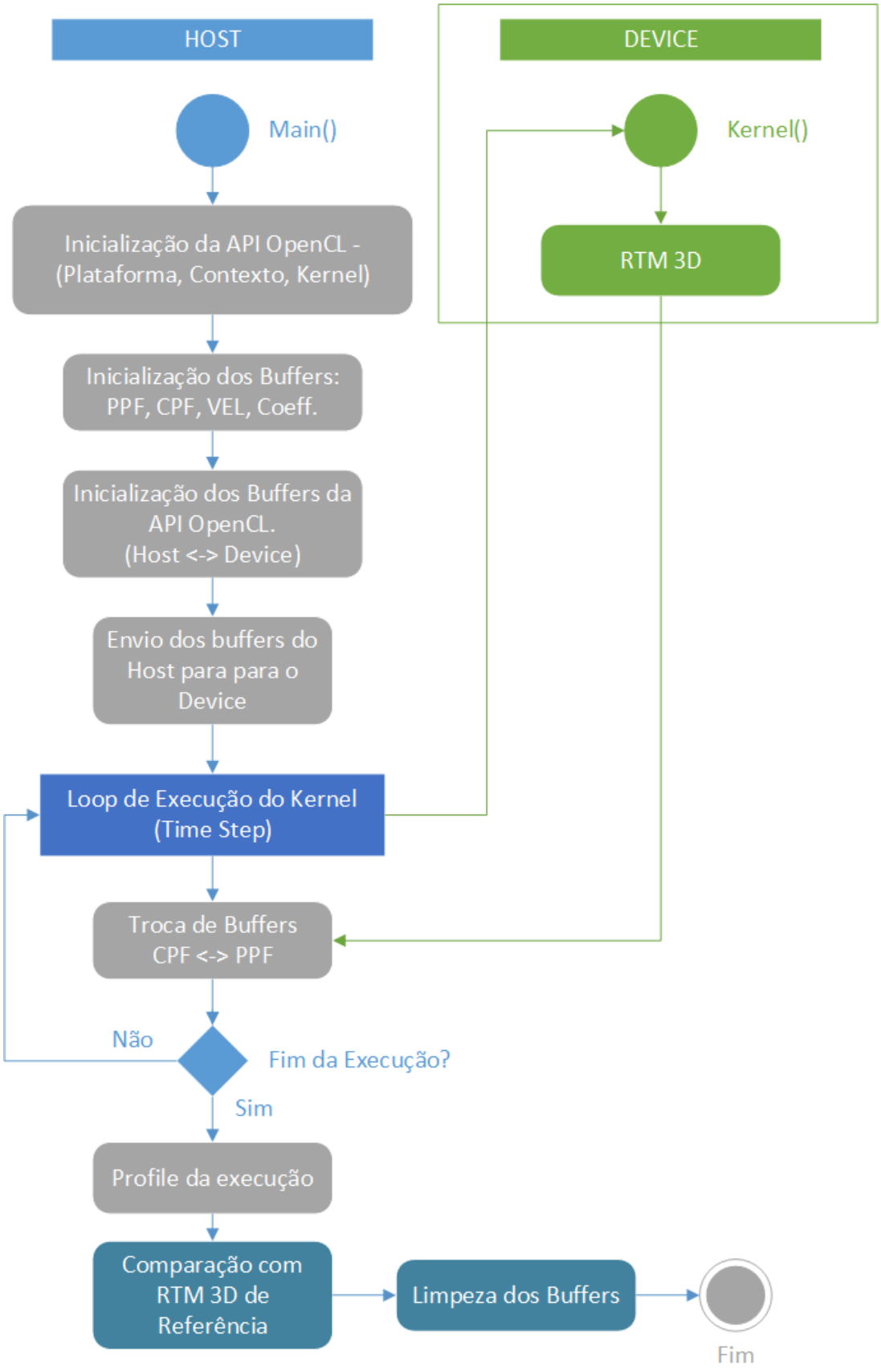
Figura 9 – Pseudocódigo RTM 3D

```
//Loop no tempo
//radius=2
for (i=2;i<dimZ-2;i++) {
    for(j=2;j<dimY-2;j++) {
        for (k=2;k<dimX-2;k++) {
            npf[i][j][k] = (2.0f * cpf[i][j][k] +
                (vel[i][j][k] *
                ((coeff[0] * (cpf[i][j-2][k]+cpf[i][j+2][k]+cpf[i-2][j][k]+
                    cpf[i+2][j][k]+cpf[i][j][k-2]+cpf[i][j][k+2])) +
                (coeff[1] * (cpf[i][j-1][k]+cpf[i][j+1][k]+cpf[i-1][j][k]+
                    cpf[i+1][j][k]+cpf[i][j][k-1]+cpf[i][j][k+1])) +
                (coeff[2] * cpf[i][j][k])) - ppf[i][j][k]));

        }
    }
}
```

Fonte: Baseado no código apresentado por Medeiros (2013).

Figura 10 – Diagrama de Execução do Host/Device (OpenCL)



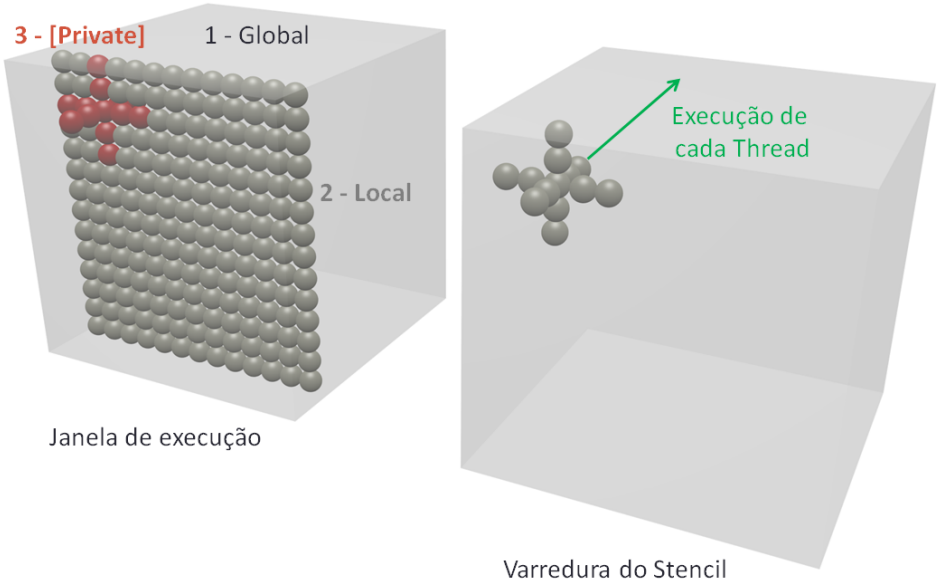
Fonte: Autor.

4.2 Código - GPU

O código de OpenCL de GPU foi produzido com base no exemplo de *Diferenças Finitas* disponibilizado pela Nvidia e o código do Host é implementado conforme descrito na sessão anterior. No algoritmo para GPU a configuração do paralelismo é definida pela quantidade de *workItens* em cada *workgroup*, conforme suportado por cada dispositivo. Esses valores são configurados na etapa de Inicialização da API mostrado na Figura 10.

Após a chamada do *Kernel* que foi distribuído entre os *workgroups* e *workitens*, é possível realizar a chamada de métodos que identificam qual o índice da *thread*, em relação ao contexto Global, Local e do Item no paralelismo. E com esses índices é possível criar uma janela de execução (Figura 11) localizada na memória local e responsável por compartilhar os valores de um mesmo *workgroup*. Além da janela, cada thread cria valores privados correspondentes aos valores laterais do *Stencil*, que estão fora da janela.

Figura 11 – GPU - Exemplo de Varredura pelo *Stencil* lendo os pontos de CPF



Fonte: Autor.

É importante notar que os valores laterais vão ser aproveitados durante o deslocamento em profundidade da *thread*, eliminado o menor valor em z. E que cada *thread* e responsável por alimentar o valor atual da janela, obrigando a sincronização após os deslocamentos e após alimentar o valor no grupo local.

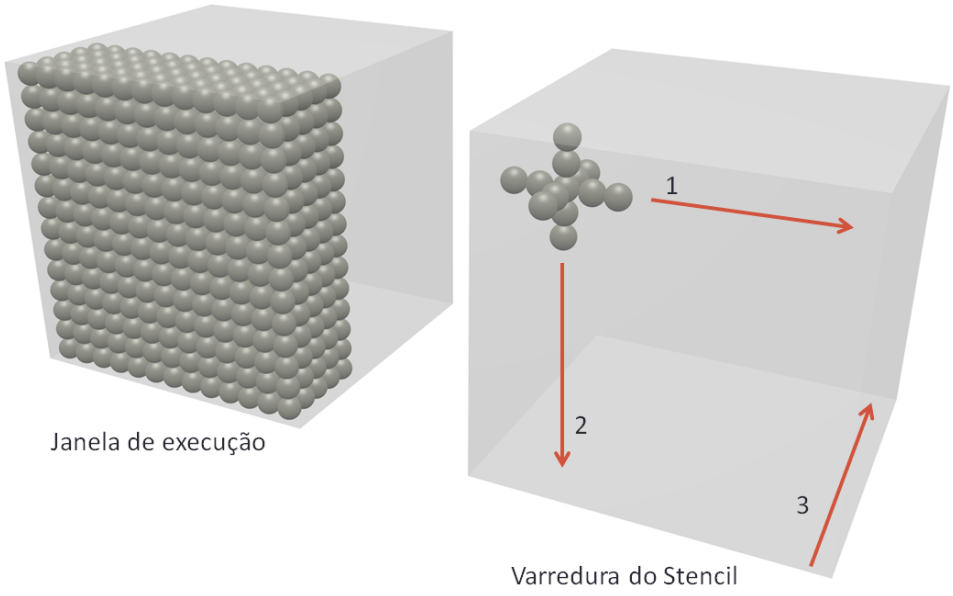
4.3 Código - FPGA

O exemplo de *Diferenças Finitas* foi disponibilizado pela Altera (ALTERA...) e serviu como base para o algoritmo RTM portado para OpenCL.

Ao contrário de como ocorre em outras arquiteturas, tais como GPU e CPU, o OpenCL para FPGA não é compilado em tempo de execução. Ele é sintetizado em uma linguagem de descrição de Hardware (HDL), e dele é gerado um *bitStream* que é gravado no FPGA para configurá-lo. Outro ponto de divergência é o código do dispositivo escrito em OpenCL. Ele não segue a estrutura habitual de se trabalhar com múltiplas *threads* divididas em *workGroup* e *workItem*, pois se pressupõe uma fraca dependência e concorrência ao acesso a memória. E no caso do algoritmo de RTM, existe a dependência dos valores no *stencil* como explicado na Figura 8, onde os valores dos vizinhos são usados para calcular o valor central no *stencil*. Assim o adequado foi utilizar-se do modelo de uma única *thread*, onde o paralelismo do cálculo é obtido através da técnica de desenrolar os loops. Essa técnica de codificação ao OpenCL (*#pragma unroll [n]*) permite que o compilador desenrole os loops instanciando um *pipeline* de hardware executando um determinado número de interações simultaneamente.

Para o cálculo, o algoritmo lineariza o volume de entrada e desliza através de uma janela de varredura na qual a matriz é dividida em volumes menores ou maiores. Essa divisão sistemática para varredura permite utilizar os valores do *stencil* alocados na memória local do FPGA, economizando acessos a memória global do dispositivo. Esses sub-cubos são de tamanho: *2 * RADIUS * DIMX * DIMY + PAR_POINTS*, aonde *DIMX* e *DIMY* correspondem as dimensões da janela de leitura que devem ser múltiplos dos *PAR_POINTS*. Esses valores são multiplicados por duas vezes o valor do *RADIUS* para acumular quatro camadas na ordenada Z.

Figura 12 – FPGA - Exemplo de Varredura pelo *Stencil* lendo os pontos de CPF



Fonte: Autor.

Este sub-cubo, apresentado na Figura 12 mantem os valores do *stencil* na memória local, durante o deslizamento da janela. A cada interação essa janela é alimentada com a

quantidade é igual ao *PAR_POINTS*, que é o número correspondente a quantidade de cálculo simultâneo ao se fazer o desenrolar do loop. Quando essa janela chega ao final da coordenada Z, uma nova janela é processada até percorrer toda a extensão da matriz.

4.4 Síntese do código OpenCL em FPGA

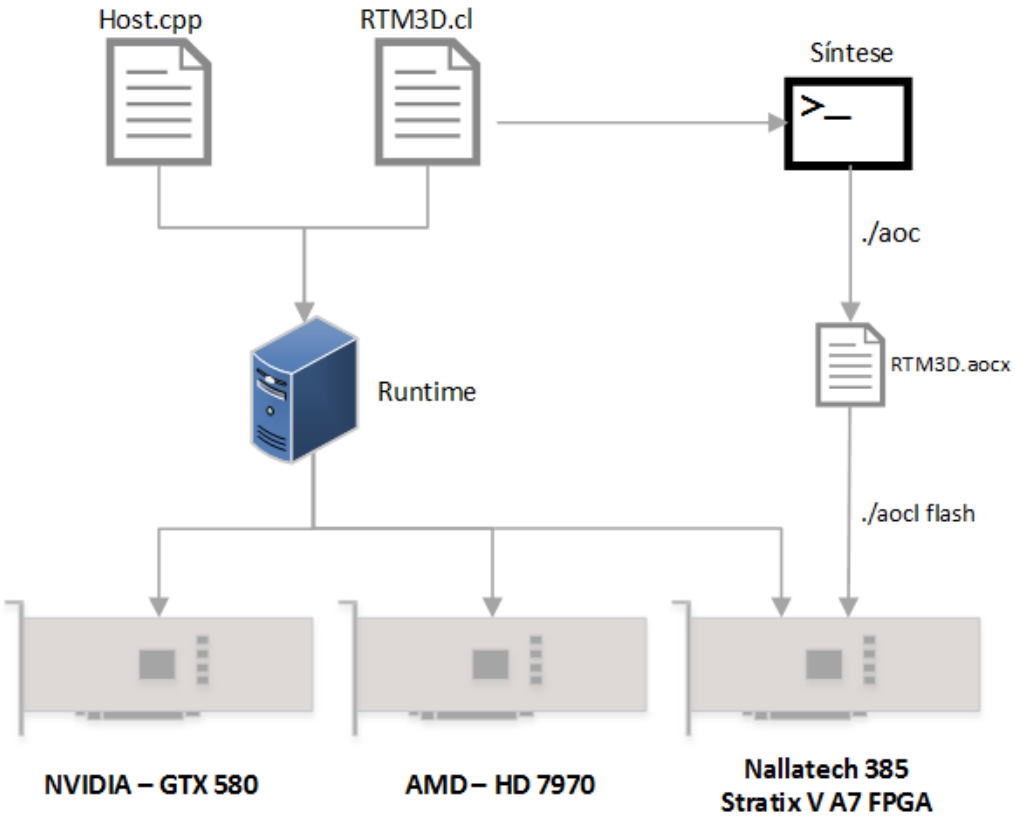
A síntese do SDK OpenCL da Altera é feita a partir do arquivo .cl, que contêm o método *Kernel*. Ele é transformado em seu correspondente Verilog e a ele é adicionado um *wrapper* que fará a comunicação com a camada acima, que é um segundo arquivo que faz as ligações aos arquivos de acesso aos recursos da placa como pinos I/O e controlador de memória. Assim, o projeto fica constituído de 4 principais arquivos TOP.v que é o arquivo principal e mais três que são, **board.qsys** com os competentes da placa como PciExpress, DMA e os bancos de memória DDR3; **kernel_system.qsys** que é a versão HDL correspondente ao arquivo OpenCL mais o invólucro; e **system.qsys** que faz a ligação entre os dois arquivos anteriores. Esses arquivos são projetos do Qsys demonstrados no Apêndice A.

Todo esse processo de composição é feito durante a execução do **aoc**, que é responsável por compilar o arquivo .cl que é enviado com um dos parâmetros. Também é necessário informar o nome da placa do FPGA para o qual foi instalado o drive do SDK OpenCL da Altera. Além desses parâmetros podemos informar se vai ser necessário gerar recursos de profile, para acompanhar o desempenho do programa. Ainda é possível separar os bancos de memória, o que força a síntese a separar o endereçamento de cada um, e alguns outros parâmetros que controlam as operações de ponto-flutuante. Durante a execução o programa vai informando as etapas da síntese e quanto de recurso foi utilizado. A Figura 13 mostra o fluxo para executar a síntese e a Figura 14 mostra os IPs e os elementos de hardware resultantes.

Ao término da execução do **aoc**, um arquivo com extensão .aocx é gerado, Este arquivo é responsável por configurar o FPGA através de outro comando, chamado aocl flash. O programa **aocl** é responsável por instalar o *drive*, configurar e diagnosticar o FPGA.

Para o arquivo RTM3D.cl, devido ao tamanho de memória do dispositivo, foram programadas matrizes de 352 pontos em cada dimensão, além de 4 pontos adicionais de borda para acomodar o *Stencil*, resultando em matrizes de 356 pontos. Os recursos utilizados foram de acordo com a Tabela 5. O tamanho das janelas que fazem a pré-copia dos valores das dimensões x e y foi de 180. O número máximo de pontos a serem executados em paralelo é 6.

Figura 13 – Fluxo de Execução SDK OpenCL - Altera.



Fonte: Autor.

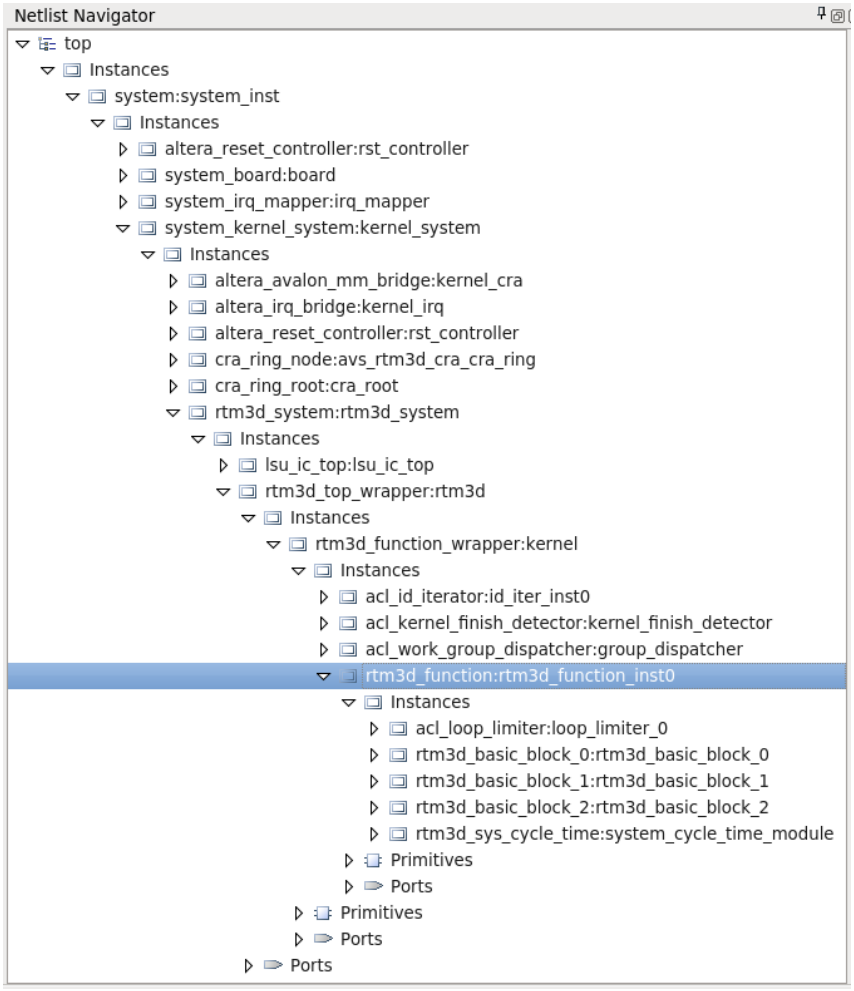
Tabela 5 – Stratix V - Recursos utilizados

LEs:	26.9334%
FFs:	18.9825%
RAMs:	44.5703%
DSPs:	34.3750%
Total Util.:	43.5587%

4.5 Conclusões

Os códigos de OpenCL são abordados e implementados de formas diferentes em cada arquitetura. O código apresentado na seção 4.2 é utilizado tanto na execução em GPU quanto para na síntese e execução no FPGA, na versão portada diretamente entre as plataformas. Já a versão de código apresentada na seção 4.3 e seção 4.4 é a versão otimizada, que será objeto de discussão no próximo capítulo.

Figura 14 – Síntese OpenCL RTM 3D - Árvore hierárquica



Fonte: Autor.

5 Otimizações

A abordagem direta do uso do código de GPUs para FPGA apresenta um baixo desempenho devido às diferenças de arquitetura. Portanto, iremos explicar as otimizações necessárias para se melhor aproveitar os recursos disponíveis do OpenCL para FPGAs.

O primeiro aspecto importante no processo de portabilidade do código, discutido na seção 4.2, é a configuração do paralelismo que depende do tamanho de *Workitens* em cada *Workgroup*, influenciando na quantidade de recursos utilizados no FPGA. Essa configuração é feita ao se adicionar duas linhas de código antes do método do *kernel*, a primeira é o *attribute* ***reqd_work_group_size***, informando o tamanho do *workgroup* local de 32x32 itens. O segundo indica o número de operações de instruções simultâneas, *attribute* ***num_simd_work_items***, que é igual a 8. É importante alinhar o número de instruções com a janela local, neste caso $8 * 4 = 32$. Foram feitas tentativas de síntese com tamanhos diferentes, como 64x64 que resulta em falha por falta de área, assim como usar tamanhos menores de 16x16 que resultaram em um menor desempenho. Variações no número de instruções SIMD apresentaram o mesmo comportamento.

Outro aspecto está na característica de varredura e de *thread* do algoritmo da GPU, que exige um maior número de acesso à memória principal do FPGA, pois o cálculo utilizado neste trabalho lê 15 valores de ponto flutuante de 32 bits de dado, ignorando as constantes que vão vir de caches separadas. São 13 valores do *Stencil*, 1 de velocidade e 1 do PPF, em um total de 480 bits para cada valor calculado no NPF. Esses valores devem passar pelo barramento de 72 bits que o FPGA tem para com os dois bancos de DDR3, determinando um máximo de dois valores 32 bits em cada operação de acesso na memória. Isso constitui em um gargalo, mesmo se consideramos o aproveitamento de valores na memória local e privada de cada *thread* na execução e o acesso de rajada na memória do dispositivo.

Para superar esses dois aspectos foi necessário trazer uma maior quantidade de valores para as regiões internas do FPGA, aumentando o tamanho da janela de varredura até cobrir o *Stencil* por completo. Assim quando um segundo ponto na mesma região do *Stencil* for calculado, não será necessário trazer uma grande quantidade de valores para a

memória local. Porém existem outras características a serem observadas.

Ainda assim, devemos observar o problema de concorrência ao se calcular múltiplos pontos, pois é necessário sincronizar o acesso de cada *thread* aos valores globais para pré-alimentar os valores locais que serão compartilhados no *Workgrop* usando *barrier*. Essa sincronia é um consumo extra de recursos que pode ser otimizado na execução do código em FPGA, já que há uma dependência entre os pontos do mesmo plano, Figura 11.

Por isso, o sistema de *multi-thread* é substituído por um paralelismo obtido por um desenrolar de loop para processar uma determinada sequência de pontos, em um modelo de *dataflow* que não faz uso de um indexador matricial para armazenar a janela de varredura. Deste modo, para fazer andar com a janela de execução, é necessário reindexar todos os pontos da janela, para alimentar os novos pontos de acordo com a quantidade no PAR_POINTs. Contudo, esse processo é feito a um custo menor de execução, do que o acesso à memória externa. No caso de se manter o indexador matricial, implica como consequência programar uma lógica de endereços por ponteiros que aumentaria a necessidade de espaço em memória, ou em reaproveitar o índice no deslocamento, acrescentando lógica ao problema e podendo diminuir o desempenho. A quantidade de vezes que esse loop é desenrolado é configurado pelo número de PAR_POINTs.

Então para conseguir maximizar o tamanho de área é importante descobrir o maior janelamento possível, respeitando o alinhamento SIMD de itens simultâneos no PAR_POINTs. A configuração que apresentou o melhor resultado no desempenho foi a de 6 PAR_POINTs, com uma janelamento de 180x180x4, sendo 180 um múltiplo direto de 6 por uma questão de alinhamento, conforme exemplificado na Figura 12. E o tamanho externo deve seguir a mesma lógica, ou seja, ser múltiplo de 180, e considerar os valores de borda para acomodar o *Stencil*.

A quantidade de memória e o tamanho do problema de entrada é mais um aspecto a ser avaliado na otimização de acordo com cada dispositivo. Desta forma a maior janela possível no FPGA deve ser de dimensão menor que 360, que neste caso foi de 352 em cada dimensão, mais a borda do *Stencil*, 356. Se considerando as matrizes tridimensionais VEL, PPF, CPF e NPF, o tamanho total aproximado de memória utilizada no dispositivo é de $356^3 * 32\ bits * 4\ Matrizes = 5.775.106.048\ bits$, que convertidos para bytes resultam em 688 Mb. O próximo múltiplo de 180 é 540, resultando em 2.403 Mb, valor superior ao do menor dispositivo, a GTX 580 com 1.5Gb.

Pequenas mudanças de configuração resultaram em grandes alterações no desempenho, como remover em cada dimensão do problema 4 valores, 352 itens, o desempenho cai de 0.5100 Gpoints/s para 0.1029 Gpoints/s. Configurações com 4 itens a mais em cada dimensão, portanto 360, resultam em erro de síntese na etapa de *placement*. Todas as configurações em que usamos mais de 6 PAR_POINTs, também resultaram em falha de síntese. Em um dos testes com 8 PAR_POINTs e um tamanho de janelamento de

192x192X4 o erro de *placement* foi devido à adição de 32 fifos a mais que não couberam no FPGA por falta de blocos de memória interna, e devido a localização dessas FIFOs, o seu provável uso seria na sincronização dos dados ao acessar a controladora de memória do FPGA para alimentar os valores no janelamento.

No decorrer dos testes, o SDK mostrou uma determinada instabilidade, pois cada síntese desta etapa necessitava de 8hs de execução e muitas vezes o processo quebrou por falta de recurso, demostrando possível vazamento de memória ou falha no algoritmo, que resultava em um uso de mais de 50GB na memória RAM.

5.1 Conclusões

Tabela 6 – Otimizações

	Código da GPU no FPGA	Código Otimizado para FPGA	% Otimizada
ALUTs	165.980	108.783	-34,46%
Registers	239.274	140.516	-41,27%
Logic utilization	140.994 / 234.720 (60 %)	91.410 / 234.720 (39 %)	-35,17%
I/O pins	385 / 664 (58 %)	385 / 664 (58 %)	0,00%
DSP blocks	26 / 256 (10 %)	88 / 256 (34 %)	238,46%
Memory bits	6.277.412 / 52.428.800 (12 %)	12.562.768 / 52.428.800 (24 %)	100,13%
RAM blocks	1.128 / 2.560 (44 %)	1.006 / 2.560 (39 %)	-10,82%
Actual clock freq	196,42	212,40	8,14%
Kernel fmax	196,42	212,40	8,14%
Block interconnects	484.754 / 1.596.384 (30 %)	313.177 / 1.596.384 (20 %)	-35,4%

A otimização obtida fica caracterizada pelo tamanho máximo de pontos processados em paralelo, que consideram os aspectos explicados acima e mantiveram o alinhamento síncrono com a controladora de memória do FPGA. Assim temos o valor de 6 PAR_POINTs simultâneos, bem como o maior janelamento possível de síntese, que é igual a 180 nas duas dimensões vezes o tamanho do *stencil*. Esses parâmetros estão definidos no código que está no apêndice seção B.2 nas linhas 3, 4 e 6, bem como o tamanho de janelamento que é definido na linha 17.

O código completo do *kernel* do FPGA, com essas otimizações desenvolvido em OpenCL para o FPGA está no Apêndice B e o resultado comparativo é apresentado na seção seguinte. A Tabela 6 demostra os recursos de hardware resultantes de cada síntese.

6 Resultados

Nesta seção são mostrados os dados referentes à etapa de modelagem sísmica do algoritmo RTM nas plataformas GPU, CPU e FPGA. Os experimentos estão focados principalmente em um comparativo de desempenho para cada arquitetura, no teste de implementação de portabilidade do OpenCL e no desempenho da FPGA usando OpenCL ao invés de uma HDL. As implementações do algoritmo sísmico, restrito a etapa de modelagem sísmica se deu na versão 1.2 do OpenCL. A Tabela 7 mostra as configurações usadas.

Tabela 7 – Configurações

Placa	Fabricante	Nallatech	NVIDIA	AMD
	Nome	p385_A7	GTX 580	HD 7970
	Processors Units	6 PAR_POINT	16 SM / 512 cuda cores	2048 Stream Processors
	Clock (MHz)	212,9	772	950
	TDP (W)	16	244	300
	Tec. Fabr. (nm)	28	40	28
	Transistors (Million)	14.3M ASIC gates or up to 1.19M logic elements	3000	4313
	Die Size (mm2)	-	520	352
	S.O.	Centos 6.8	Centos 6.8	Win 8.1
Memória	Tec.	DDR3	GDDR5	GDDR5
	Bandwidth (GB/s)	25,6	192	264
	Clock (MHz)	933	2004	1375
	Bus width (bit)	72	384	384
	Size (GB)	8	1,5	3

Os principais parâmetros de execução do algoritmo são as dimensões das matrizes. Por uma questão de alinhamento de memória, que é diferente em cada uma das arquiteturas exploradas, foram testados diferentes tamanhos de matrizes, de acordo com a lógica programada para as bordas do *stencil*. O critério final de escolha para o tamanho das matrizes considerou os valores mais próximos, que obtivessem o melhor desempenho em cada uma das plataformas. O referencial de desempenho dos resultados está centrado no *Throughput* com a saída em *GigaPoints* por segundo, calculados a partir da média de 100 passos do algoritmo.

A primeira etapa de testes consistiu na verificação da portabilidade de códigos desenvolvidos em OpenCL. Para isto o código escrito para GPU foi usado sem alterações no FPGA, e em seguida foi escrito uma versão que considera as modificações para aproveitar melhor os recursos do FPGA. Os resultados obtidos neste primeiro teste são apresentados na Tabela 8.

Tabela 8 – Resultados – Portabilidade OpenCL

Versão	Plataforma	Cód.	Hardware	Throughput Gpoints/Sec	Time (s)
OpenCL 1.2	FPGA - Altera	GPU	Nallatech p385_A7	0,2224	0,1957
OpenCL 1.2	FPGA - Altera	FPGA	Nallatech p385_A7	0,5100	0,0879

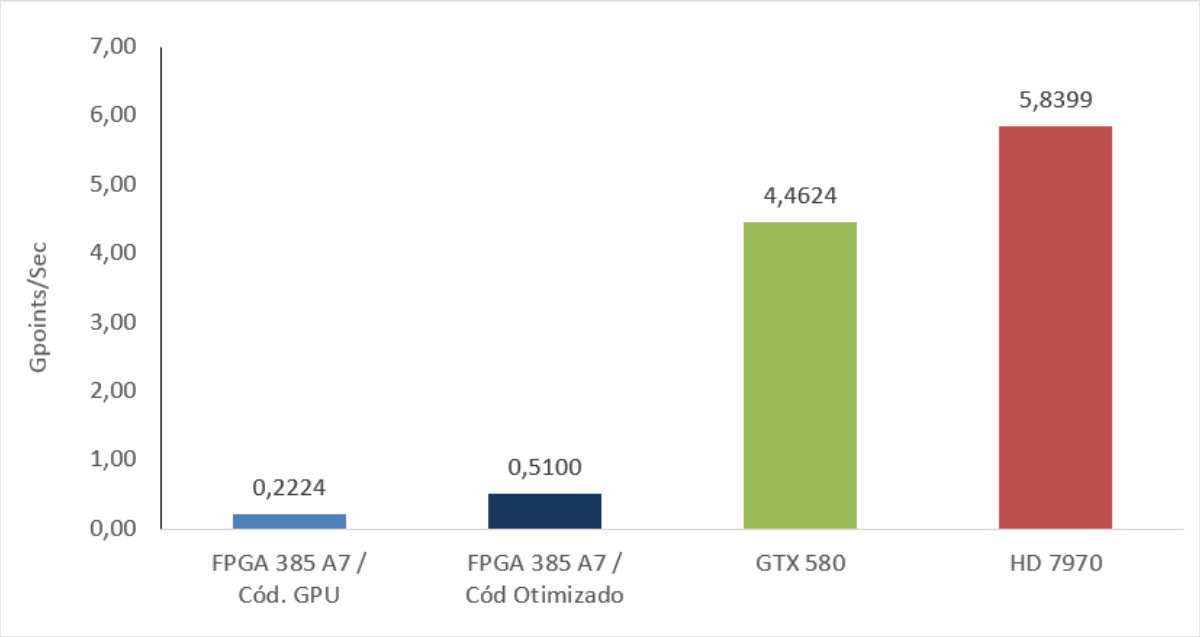
Percebe-se que o desempenho do código otimizado para a plataforma FPGA é cerca de 2,29 vezes melhor do que o código sem qualquer otimização. Isto mostra que o OpenCL realmente atende a sua proposta de ser uma linguagem portável para ambientes heterogêneos; no entanto, se a demanda por alto desempenho for crítica na aplicação em questão, é fundamental a adaptação e otimização do código para a plataforma alvo.

Tabela 9 – Resultados – CPU X GPU X FPGA

Versão	Arq.	Cód.	Fab.	Hardware	Throughput Gpoints/Sec	Time(s)	TDP(W)	(TDP)/(Gp/s)
Opencl 1.2	FPGA	GPU	Altera	p385_A7	0,2224	0,19570	25	112,410
Opencl 1.2	FPGA	FPGA	Altera	p385_A7	0,5100	0,08790	16	31,372
Opencl 1.2	GPU	GPU	AMD	HD 7970	5,8399	0,00747	300	51,371
Opencl 1.2	GPU	GPU	Nvidia	GTX 580	0,0579	0,77858	244	54,679
Opencl 2.0	CPU	GPU	Intel	i7-2600k	0,0130	3,34436	95	7307,692

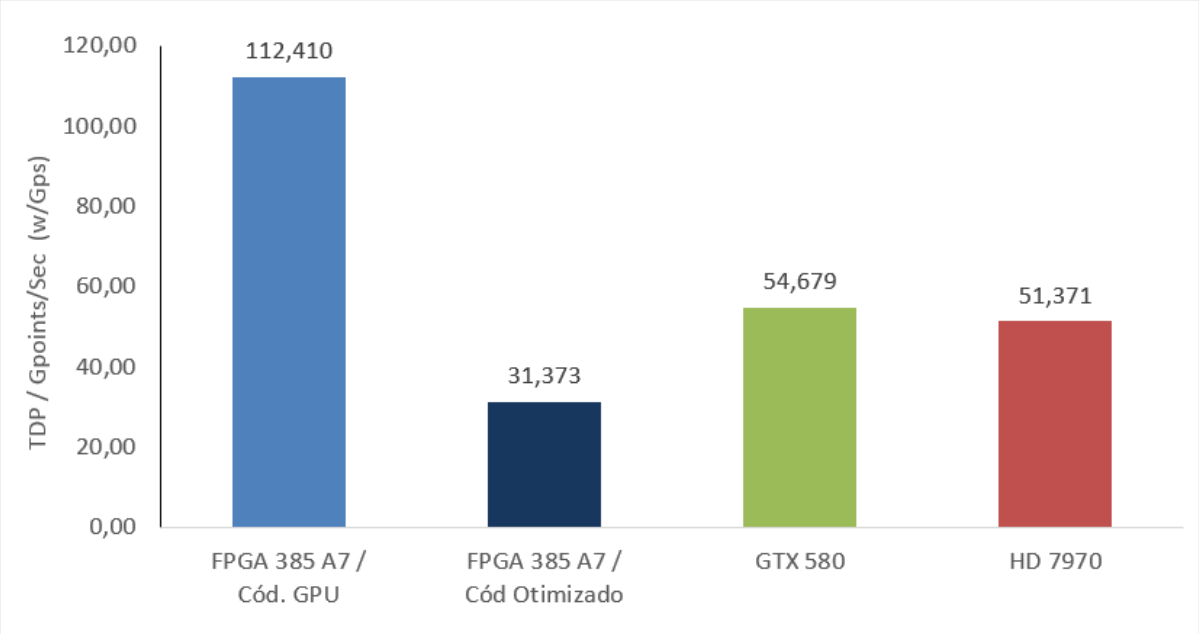
Outra etapa de testes focou no comparativo entre as diferentes arquiteturas: CPU, GPU e FPGA. A Tabela 9 está ordenada de acordo com o valor obtido entre a relação de TDP - *Thermal Design Power* – e a coluna de *Throughput*, demostrando o consumo médio para cada *Gpoints* por segundo. As imagens resultantes da modelagem estão no Apêndice C. Observa-se que para as GPUs, com suas arquiteturas SIMD e suas memórias GDDR5, obteve-se um melhor desempenho; porém, o FPGA possui vantagem ao gerar uma melhor relação de energia consumida por pontos calculados.

Figura 15 – Throughput Gpoints/Sec



Fonte: Autor.

Figura 16 – TDP Watts / Gpoints/Sec



Fonte: Autor.

6.1 Conclusões

O uso de OpenCL para desenvolvimento de projetos em FPGAs parece promissor. Suas vantagens são o tempo de desenvolvimento de projetos e possibilidades de se ter modelos de referência, com códigos similares aos de produção. Sua curva de aprendizagem é rápida já que se assemelha a linguagens de alto nível compatíveis com outras arquiteturas de mercado. Particularmente para FPGA, embora seja promissora, em suas versões atuais, ajustes de código ainda precisam ser feitos para se ter desempenho aceitável em aplicações de alta complexidade e processamento massivo de dados. Um outro aspecto relevante demonstrado aqui é que, mesmo apresentando um baixo desempenho computacional em relação GPUS, os FPGAs continuam demonstrando vantagens sobre CPUs, e apresentando uma relação *Gflops/s/w* menor que todas as arquiteturas aqui discutidas.

7 Conclusão

Durante o desenvolvimento dessa pesquisa e processos de teste, houveram três importantes atualizações no SDK OpenCL da Altera. Elas trouxeram melhorias no compilador OpenCL e novos recursos, por exemplo, o suporte a emulação de execução em CPU. O OpenCL vem sendo constantemente trabalhado e melhorado pela comunidade responsável, sua implementação é cada vez mais eficiente e facilitada. A tendência de mercado é que a plataforma OpenCL seja cada vez mais competente perante seus concorrentes e se fortaleça no mercado de HPC.

No decorrer dos testes foi evidenciado que o uso consciente dos parâmetros de síntese dentro do código OpenCL, devem levar em conta as características das arquiteturas dos dispositivos usados, pois isso interfere diretamente no desempenho. Entre alguns desses parâmetros de síntese testados, foram o número de passos no tempo e o tamanho das dimensões de cada matriz, o que impacta diretamente no desempenho. O tamanho final é de acordo com o alinhamento mais próximo ao número de pontos a serem processados, pois há diferenças entre o tratamento de borda entre os dispositivos.

Outro aspecto demostrado, é o tamanho da janela mantida em memória local, e principalmente, como ela é atualizada, o que implica em uma imensa diferença no desempenho. Principalmente devido ao gargalo de acesso a memória no próprio dispositivo.

A otimização na síntese é demostrada, pela a quantidade economizada de recurso do FPGA, que tem a lógica necessária diminuída em 35,17% (Tabela 6). E o desempenho adquirido é de 2,29 vezes melhor do que o código não otimizado (Tabela 8) para o FPGA da Nallatech que utilizamos.

Por fim, apresentamos um comparativo de desempenho em GigaPoints/sec relativos ao TDP estimado pelos respectivos fabricantes (Tabela 9 e Figura 16), como um índice indicativo da eficiência de pontos processados pelo consumo de energia.

Em trabalhos futuros, poderia ser feita a comparação com o algoritmo de RTM 3D implementado em HDL para a mesma placa, pois o refinamento na utilização dos recursos da placa pode obter um desempenho ainda maior, (MEDEIROS, 2013) e (LIMA, 2014).

Por fim, o uso de uma linguagem com um alto nível de abstração como o OpenCL, e o conhecimento específicos de algumas características do hardware utilizado, permitem reduzir drasticamente o tempo e a dificuldade na implementação de sistemas, para obter um melhor desempenho usando FPGAs.

Referências

Altera. *Altera - FPGA Stratix V Architecture*. 2010. Disponível em: <https://www.altera.com/products/fpga/stratix-series/stratix-v/features.html#Stratix-V-GX-FPGA-Overview>. Acesso em: 22 jan. 2016.

Altera. *Stratix V Device Handbook Volume 1: Device Interfaces and Integration*. 2010. Disponível em: <https://www.altera.com/en_US/pdfs/literature/hb/stratix-v/stx5_core.pdf>. Acesso em: 19 jan. 2016.

ALTERA SDK for OpenCL. Disponível em: <http://www.altera.com/products/software/opencl/opencl-index.html>. Acesso em: 06 mar. 2015.

AMD - Advanced Micro Devices, Inc. *Reference Guide - Southern Islands Series Instruction Set Architecture*. 2012. Disponível em: <http://developer.amd.com/wordpress/media/2012/12/AMD_Southern_Islands_Instruction_Set_Architecture.pdf>. Acesso em: 19 jan. 2016.

Ann Steffora Mutschler. *Semiconductor Engineering .:. Is EUV Making Progress?* 2015. Disponível em: <http://semiengineering.com/is-euv-making-progress/>. Acesso em: 09 jul. 2016.

AQUISIçãO Sísmica. 2016. Disponível em: <http://www.uff.br/geofisica/index.php/aquisicao-sismica>. Acesso em: 09 jul. 2016.

CALANDRA, H. et al. Evaluation of successive CPUs/APUs/GPUs based on an OpenCL finite difference stencil. In: *2013 21st Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*. IEEE, 2013. p. 405–409. ISBN 978-1-4673-5321-2 978-0-7695-4939-2. Disponível em: <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6498582>.

FU, H. et al. Scaling reverse time migration performance through reconfigurable dataflow engines. *Micro, IEEE*, v. 34, n. 1, p. 30–40, 2014. ISSN 0272-1732.

GIEFERS, H. et al. Analyzing the energy-efficiency of sparse matrix multiplication on heterogeneous systems: A comparative study of GPU, xeon phi and FPGA. In: *2016 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 2016. p. 46–56. Disponível em: <http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=7482073>.

GRAY, S. H. et al. Seismic migration problems and solutions. *Geophysics*, v. 66, n. 5, p. 1622–1640, 2001.

HENNESSY, J. L.; PATTERSON, D. A. *Computer architecture: a quantitative approach*. 5. ed. [S.l.]: Elsevier, 2011.

janice m golda. *EUV Lithography – Progress on the Journey to Manufacturing Magic - Technology@Intel*. 2016. Disponível em: <http://blogs.intel.com/technology/2016/02/euv-progress/>. Acesso em: 09 jul. 2016.

KHRONOS.ORG. *OpenCL - The open standard for parallel programming of heterogeneous systems*. 2015. Disponível em: <https://www.khronos.org/opencl/>. Acesso em: 12 fev. 2015.

KRÜGER, J.-T. Green wave: A semi custom hardware architecture for reverse time migration. 2012. Disponível em: <http://archiv.ub.uni-heidelberg.de/volltextserver/id/eprint/13552>.

LIMA, I. P. d. *Implementação do algoritmo (RTM) para processamento sísmico em arquiteturas não convencionais*. Dissertação (Mestrado) — Universidade Federal do Rio Grande do Norte, 2014. Disponível em: <http://repositorio.ufrn.br/handle/123456789/13004>.

LIU, G. et al. 3d seismic reverse time migration on GPGPU. v. 59, p. 17–23, 2013. ISSN 00983004. Disponível em: <http://linkinghub.elsevier.com/retrieve/pii/S0098300413001519>.

MEDEIROS, V. *fastRTM: Um Ambiente Integrado para Desenvolvimento Rápido da Migração Reversa no Tempo (RTM) em Plataformas FPGA de Alto Desempenho*. Tese (Doutorado) — UFPE, 2013.

MOORE'S Law 40th Anniversary. 2016. Disponível em: <http://www.intel.com/pressroom/kits/events/moores_law_40th/index.htm>. Acesso em: 09 jul. 2016.

Nvidia. *Whitepaper NVIDIA's Next Generation CUDA Compute Architecture: Fermi*. 2010. Disponível em: <http://www.nvidia.com/content/pdf/fermi_white_papers/nvidia_fermi_compute_architecture_whitepaper.pdf>. Acesso em: 19 jan. 2016.

OWAIDA, M. et al. Synthesis of platform architectures from OpenCL programs. In: *Field-Programmable Custom Computing Machines (FCCM), 2011 IEEE 19th Annual International Symposium on*. IEEE, 2011. p. 186–193. ISBN 978-1-61284-277-6. Disponível em: <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5771271>.

PANETTA, J. et al. Computational characteristics of production seismic migration and its performance on novel processor architectures. In: *Computer Architecture and High Performance Computing, 2007. SBAC-PAD 2007. 19th International Symposium on*. [S.l.]: IEEE, 2007. p. 11–18.

SHIN, J.; SHIN, C.; CALANDRA, H. Laplace-domain waveform modeling and inversion for the 3d acoustic–elastic coupled media. v. 129, p. 41–52, 2016. ISSN 09269851. Disponível em: <http://linkinghub.elsevier.com/retrieve/pii/S0926985116300805>.

WEBER, R. et al. Comparing hardware accelerators in scientific applications: A case study. v. 22, n. 1, p. 58–68, 2011. ISSN 1045-9219. Disponível em: <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5482576>.

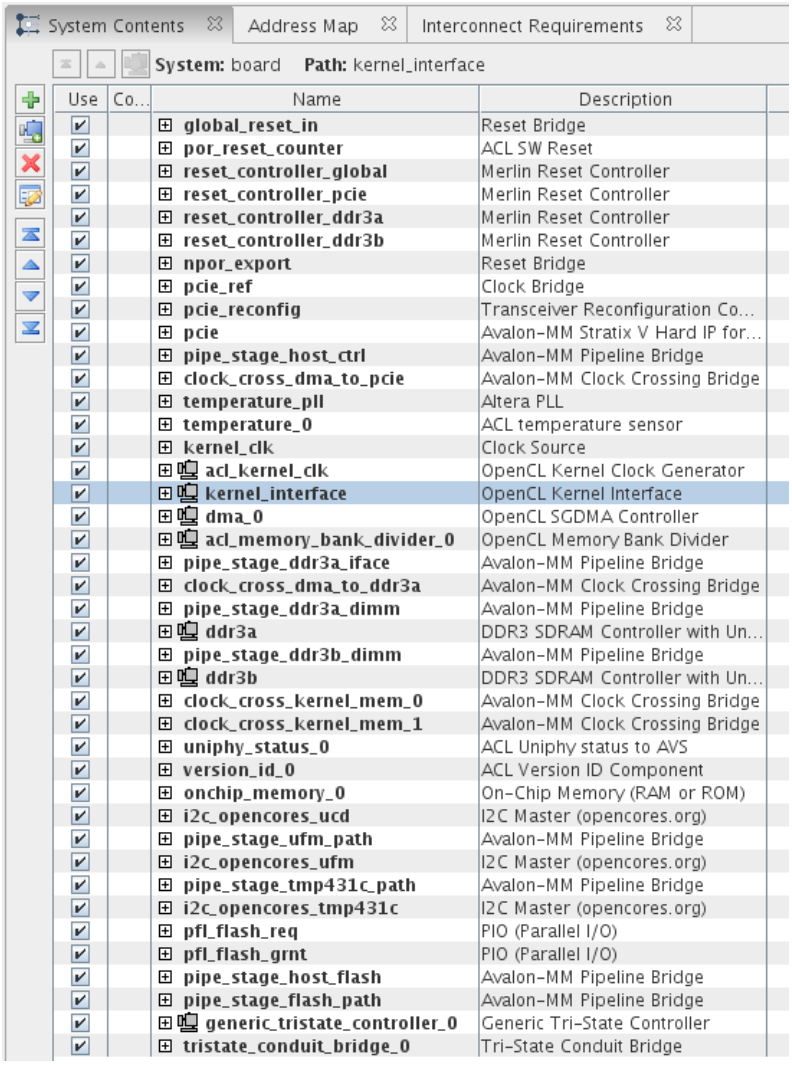
WU, B.; KUMAR, A. Extreme ultraviolet lithography and three dimensional integrated circuit—a review. v. 1, n. 1, p. 011104, 2014. ISSN 1931-9401. Disponível em: <http://scitation.aip.org/content/aip/journal/apr2/1/1/10.1063/1.4863412>. Acesso em: 09 jul. 2016.

YILMAZ, z. *Seismic data analysis*. Society of exploration geophysicists Tulsa, 2001. v. 1. Disponível em: <http://library.seg.org/doi/pdf/10.1190/1.9781560801580.fm>.

Apêndices

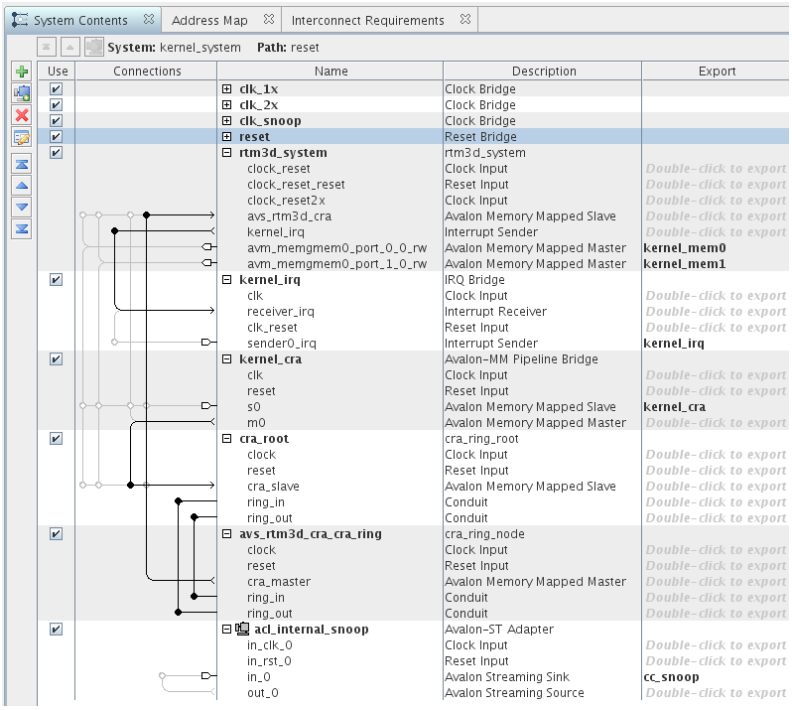
A Síntese OpenCL RTM 3D

Figura 17 – Síntese OpenCL RTM 3D - Qsys Board



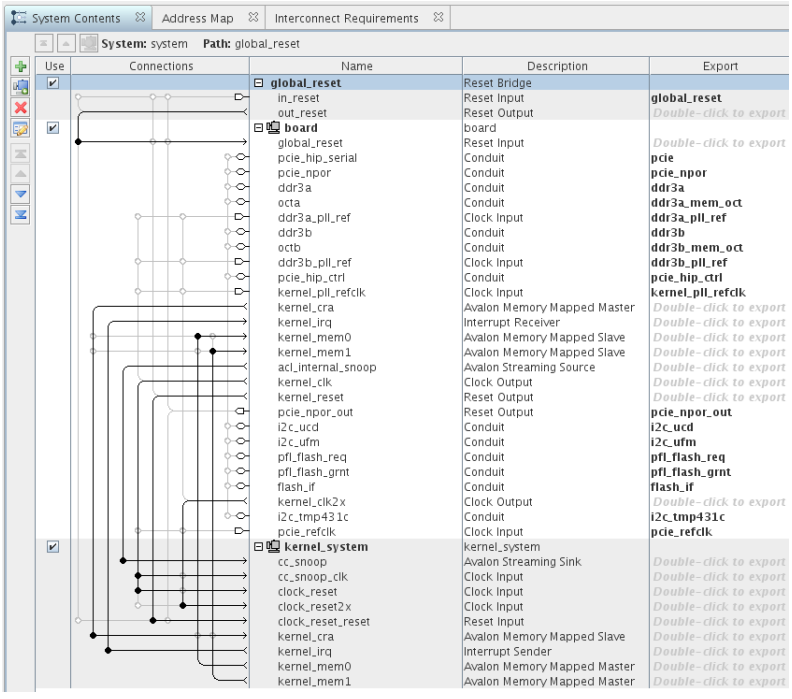
Fonte: Autor.

Figura 18 – Síntese OpenCL RTM 3D - Qsys Kernel System



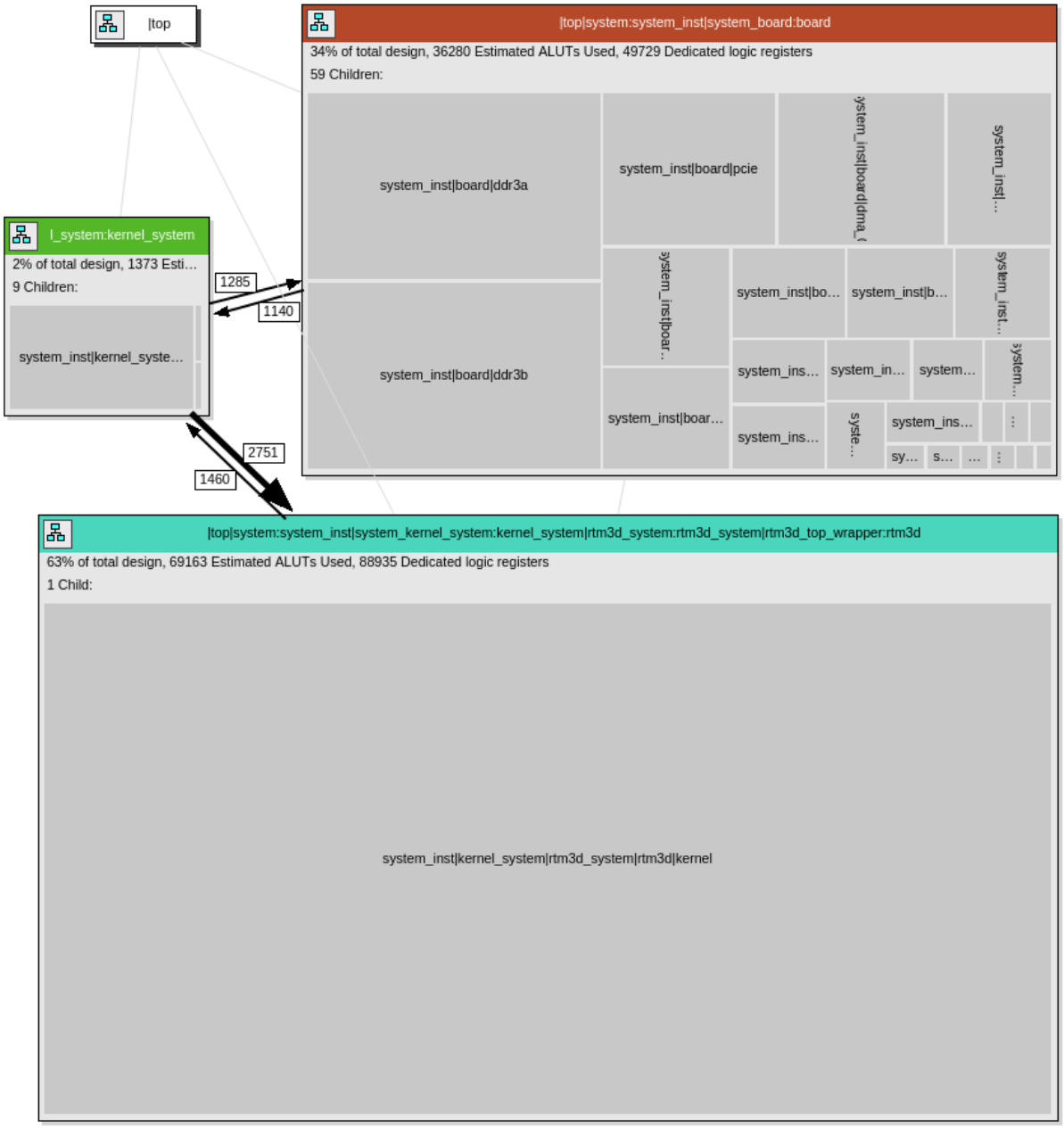
Fonte: Autor.

Figura 19 – Síntese OpenCL RTM 3D - Qsys System



Fonte: Autor.

Figura 20 – Síntese OpenCL RTM 3D



Fonte: Autor.

B Código RTM 3D OpenCL

B.1 Código OpenCL - FPGA - Arquivo rtm3d_config.h

```
1
2 // Specifies the RADIUS of the stencil
3 // For an order-k stencil, RADIUS = k / 2
4 #define RADIUS 2
```

B.2 Código OpenCL - FPGA - Arquivo rtm3d.cl

```
1  #include "../host/inc/rtm3d_config.h"
2
3  #define DIMX 180
4  #define DIMY 180
5
6  #define PAR_POINTS 6
7
8
9  __attribute__((task))
10 kernel void rtm3d(global float * restrict out,
11         global float * restrict ppf,
12         global float * restrict cpf_in,
13         global const float * restrict vel,
14         constant float *coeff,
15         const int dimx, const int dimy, const int dimz) {
16
17         float taps[2 * RADIUS * DIMX * DIMY + PAR_POINTS];
18
19         #pragma unroll
20         for (int i = 0; i < 2 * RADIUS * DIMX * DIMY + PAR_POINTS
               ; i++) {
21                 taps[i] = 0;
```

```
22          }
23
24          int x = 0, y = 0, xtile = 0, ytile = 0, ztile = 0;
25          do {
26                  #pragma unroll
27                  for (int i = 0; i < 2 * RADIUS * DIMX * DIMY; i
                       ++) {
28                          taps[i] = taps[i + PAR_POINTS];
29                  }int inOffset = ztile * dimx * dimy + (y + ytile)
                        * dimx + (x + xtile);
30
31                  #pragma unroll
32                  for (int i = 0; i < PAR_POINTS; i++) {
33                          if (inOffset + i >= 0 && inOffset + i <
                               dimx * dimy * dimz) {
34                                  taps[2 * RADIUS * DIMX * DIMY + i
                                       ] = cpf_in[inOffset + i];
35                          }
36                  }
37
38                  int xoutput = x + xtile, youtput = y + ytile,
                       zoutput = ztile - RADIUS;
39                  int outOffset = zoutput * dimx * dimy + youtput *
                        dimx + xoutput;
40
41                  float value[PAR_POINTS];
42                  #pragma unroll
43                  for (int j = 0; j < PAR_POINTS; j++) {
44                          value[j] = coeff[0] * taps[RADIUS * DIMX
                               * DIMY + j];
45
46                          #pragma unroll 2
47                          for (int i = 1; i <= RADIUS; i++) {
48                                  value[j] += coeff[i] * taps[(
                                       RADIUS - i) * DIMX * DIMY + j];
49                                  value[j] += coeff[i] * taps[(
                                       RADIUS + i) * DIMX * DIMY + j];
50                                  value[j] += coeff[i] * taps[DIMX
                                       * (RADIUS * DIMY - i) + j];
51                                  value[j] += coeff[i] * taps[DIMX
                                       * (RADIUS * DIMY + i) + j];
52                                  value[j] += coeff[i] * taps[
```

```
                                    RADIUS * DIMX * DIMY - i + j];
53                                value[j] += coeff[i] * taps[
                                    RADIUS * DIMX * DIMY + i + j];
54                        }
55
56                        //taps==cpf
57                        value[j] = 2.0f * taps[RADIUS * DIMX *
                             DIMY + j] + (vel[outOffset+j] * value[j
                             ] - ppf[outOffset+j]);
58                }
59
60
61                #pragma unroll
62                for (int i = 0; i < PAR_POINTS; i++) {
63                        bool haloX = ((xoutput + i < RADIUS) || (
                             xoutput + i >= dimx - RADIUS));
64                        bool haloY = ((youtput < RADIUS) || (
                             youtput >= dimy - RADIUS));
65                        bool haloZ = ((zoutput >= 0 && zoutput <
                             RADIUS) || (zoutput >= dimz - RADIUS));
66
67
68                        if ((xtile + i)>= RADIUS && (xtile + i) <
                              (DIMX - RADIUS) && xoutput + i < dimx
                             - RADIUS &&
69                                ytile >= RADIUS && ytile < (DIMY
                                    - RADIUS) && youtput < dimy -
                                    RADIUS &&
70                                ztile >= 2 * RADIUS && ztile <
                                    dimz) {
71                                out[outOffset + i] = value[i];
72                        } else if ((haloX || haloY || haloZ) &&
                             ztile >= RADIUS && xoutput + i < dimx
                             && youtput < dimy ) {
73                                out[outOffset + i] = taps[RADIUS
                                    * DIMX * DIMY + i];
74                        }
75                }
76
77                xtile = xtile < DIMX - PAR_POINTS ? xtile +
                     PAR_POINTS : 0;
78                ytile = xtile == 0 ? ytile < DIMY - 1 ? ytile + 1
```

```
                    : 0 : ytile;
79                ztile = xtile == 0 && ytile == 0 ? ztile < dimz +
                      RADIUS - 1 ? ztile + 1 : 0 : ztile;
80
81                bool intile = (xtile != 0 || ytile != 0 || ztile
                     != 0);
82                x =  intile ? x : x < dimx - DIMX ? x + DIMX - 2
                     * RADIUS : 0;
83                y = intile ? y : x == 0 ? y + DIMY - 2 * RADIUS :
                      y;
84        } while (y < dimy - 2 * RADIUS);
85 }
```

B.3 Código OpenCL - GPU - Arquivo rtm3d.cl

```
1  __constant const float const coeff[3] = {-90, 16, -1};
2
3  __kernel void rtm3d(__global float * const output,
4          __global const float * const input,
5          __global const float * const vel,
6          const int dimx,
7          const int dimy,
8          const int dimz,
9          const int padding)
10 {
11         bool valid = true;
12         const int gtidx = get_global_id(0);
13         const int gtidy = get_global_id(1);
14         const int ltidx = get_local_id(0);
15         const int ltidy = get_local_id(1);
16         const int workx = get_local_size(0);
17         const int worky = get_local_size(1);
18         __local float tile[MAXWORKY + 2 * RADIUS][MAXWORKX + 2 *
              RADIUS];
19
20         const int stride_y = dimx + 2 * RADIUS;
21         const int stride_z = stride_y * (dimy + 2 * RADIUS);
22
23         int inputIndex  = 0;
24         int outputIndex = 0;
25
26         inputIndex += RADIUS * stride_y + RADIUS + padding;
```

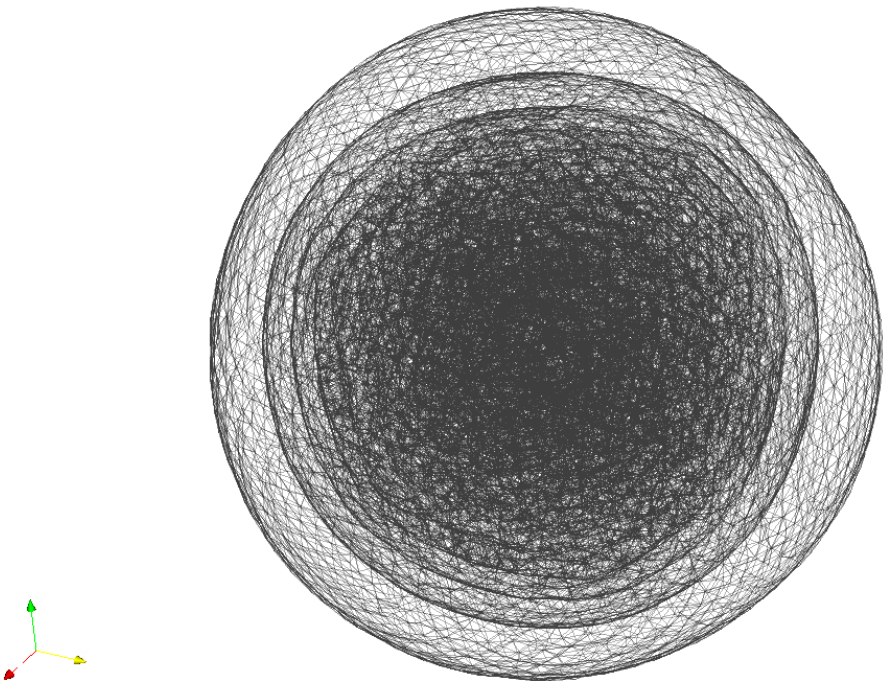
```
27
28          inputIndex += gtidy * stride_y + gtidx;
29
30          float infront[RADIUS];
31          float behind[RADIUS];
32          float current;
33          float currentVel;
34          float outputVal;
35
36          const int tx = ltidx + RADIUS;
37          const int ty = ltidy + RADIUS;
38
39          if (gtidx >= dimx)
40          valid = false;
41          if (gtidy >= dimy)
42          valid = false;
43
44
45          for (int i = RADIUS - 2 ; i >= 0 ; i--)
46          {
47                  behind[i] = input[inputIndex];
48                  inputIndex += stride_z;
49          }
50
51          current = input[inputIndex];
52          outputIndex = inputIndex;
53          inputIndex += stride_z;
54
55          for (int i = 0 ; i < RADIUS ; i++)
56          {
57                  infront[i] = input[inputIndex];
58                  inputIndex += stride_z;
59          }
60
61          // Step through the xy-planes
62          for (int iz = 0 ; iz < dimz ; iz++)
63          {
64                  // Advance the slice (move the thread-front)
65                  for (int i = RADIUS - 1 ; i > 0 ; i--)
66                  behind[i] = behind[i - 1];
67                  behind[0] = current;
68                  current = infront[0];
```

```
69                  for (int i = 0 ; i < RADIUS - 1 ; i++)
70                  infront[i] = infront[i + 1];
71                  infront[RADIUS - 1] = input[inputIndex];
72
73                  inputIndex  += stride_z;
74                  outputIndex += stride_z;
75
76                  currentVel = vel[outputIndex];
77                  outputVal = output[outputIndex];
78
79                  barrier(CLK_LOCAL_MEM_FENCE);
80
81                  // Update the data slice in the local tile
82                  // Halo above & below
83                  if (ltidy < RADIUS)
84                  {
85                          tile[ltidy][tx]                   = input[
                               outputIndex - RADIUS * stride_y];
86                          tile[ltidy + worky + RADIUS][tx] = input[
                               outputIndex + worky * stride_y];
87                  }
88                  // Halo left & right
89                  if (ltidx < RADIUS)
90                  {
91                          tile[ty][ltidx]                   = input[
                               outputIndex - RADIUS];
92                          tile[ty][ltidx + workx + RADIUS] = input[
                               outputIndex + workx];
93                  }
94                  tile[ty][tx] = current;
95
96                  barrier(CLK_LOCAL_MEM_FENCE);
97
98                  // Compute the output value
99                  float value = coeff[0] * current;
100                 #pragma unroll RADIUS
101                 for (int i = 1 ; i <= RADIUS ; i++)
102                 {
103                         value += coeff[i] * (infront[i-1] +
                               behind[i-1] + tile[ty - i][tx] + tile[
                               ty + i][tx] + tile[ty][tx - i] + tile[
                               ty][tx + i]);
```

```
104                     }
105
106                     value = 2.0f * current + (currentVel * value -
                            outputVal);
107
108                     // Store the output value
109                     if (valid)
110                     output[outputIndex] = value;
111             }
112     }
```

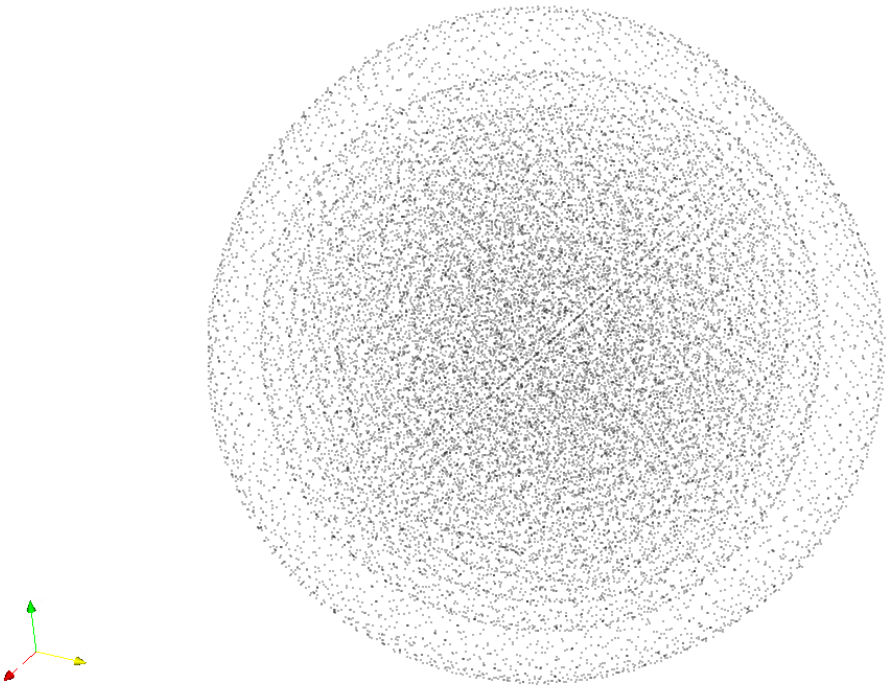
C Imagens Resultantes

Figura 21 – Modelagem RTM 3D - Dim. 356 - 100 Steps



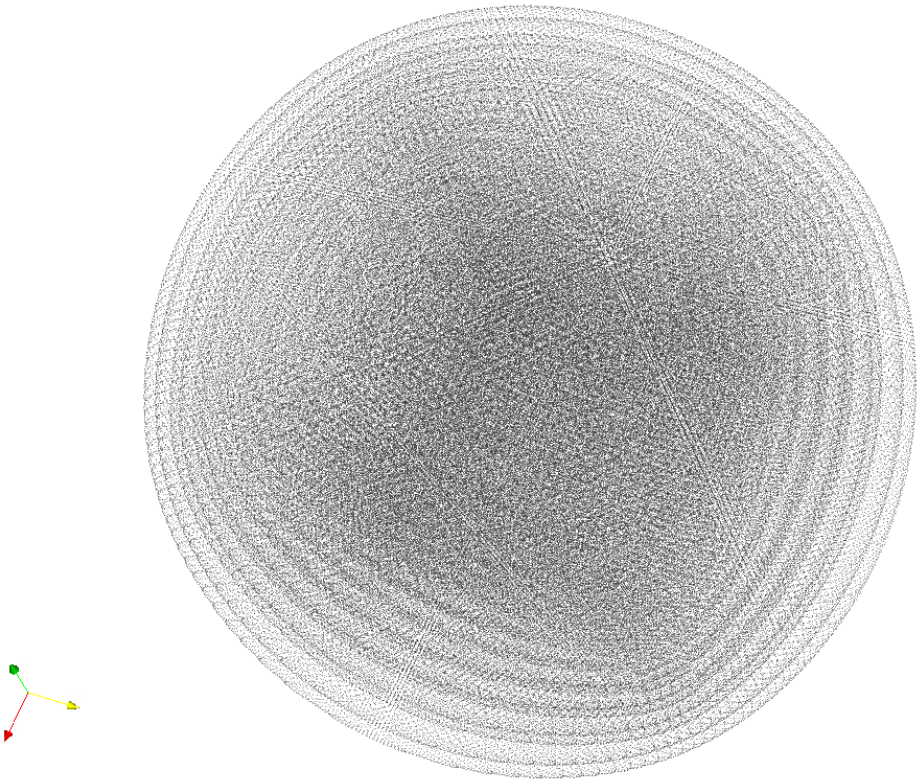
Fonte: Autor.

Figura 22 – Modelagem RTM 3D - Dim. 356 - 100 Steps / Pontos



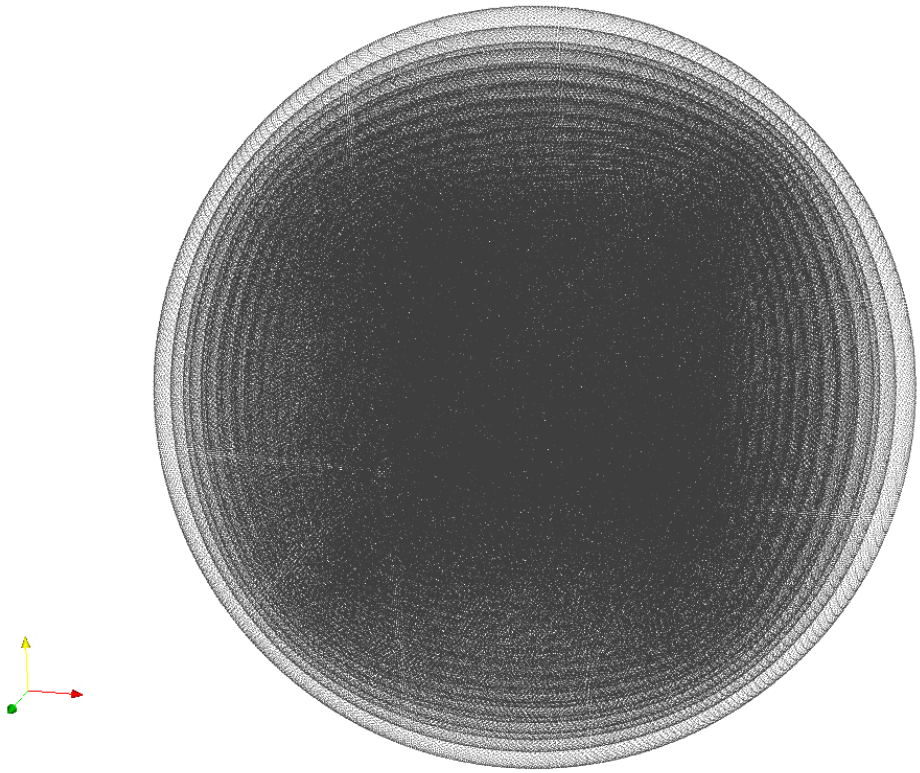
Fonte: Autor.

Figura 23 – Modelagem RTM 3D - Dim. 356 - 300 Steps / Pontos



Fonte: Autor.

Figura 24 – Modelagem RTM 3D - Dim. 356 - 500 Steps / Pontos



Fonte: Autor.