FILTRAGEM DE GRANDES DADOS SÍSMICOS TRI-DIMENSIONAIS EM PLACAS GRÁFICAS UTILIZANDO CUDA

Bryan Marinho Hall

Dissertação de Mestrado apresentada ao Programa de Pós-graduação em Engenharia de Sistemas e Computação, COPPE, da Universidade Federal do Rio de Janeiro, como parte dos requisitos necessários à obtenção do título de Mestre em Engenharia de Sistemas e Computação.

Orientador: Ricardo Cordeiro de Farias

Rio de Janeiro
Abril de 2014

FILTRAGEM DE GRANDES DADOS SÍSMICOS TRI-DIMENSIONAIS EM PLACAS GRÁFICAS UTILIZANDO CUDA

Bryan Marinho Hall

DISSERTAÇÃO SUBMETIDA AO CORPO DOCENTE DO INSTITUTO ALBERTO LUIZ COIMBRA DE PÓS-GRADUAÇÃO E PESQUISA DE ENGENHARIA (COPPE) DA UNIVERSIDADE FEDERAL DO RIO DE JANEIRO COMO PARTE DOS REQUISITOS NECESSÁRIOS PARA A OBTENÇÃO DO GRAU DE MESTRE EM CIÊNCIAS EM ENGENHARIA DE SISTEMAS E COMPUTAÇÃO.

Examinada por:

Prof. Ricardo Cordeiro de Farias, Ph.D.

Prof. Alexandre de Assis Bento Lima, D.Sc.

Prof. Esteban Walter Gonzalez Clua, Ph.D.

RIO DE JANEIRO, RJ – BRASIL
ABRIL DE 2014

Hall, Bryan Marinho

Filtragem de Grandes Dados Sísmicos Tri-Dimensionais em Placas Gráficas Utilizando CUDA/Bryan Marinho Hall. – Rio de Janeiro: UFRJ/COPPE, 2014.

x, 37 p. 29, 7cm.

Orientador: Ricardo Cordeiro de Farias

Dissertação (mestrado) – UFRJ/COPPE/Programa de Engenharia de Sistemas e Computação, 2014.

Referências Bibliográficas: p. 36 – 37.

1. Filtragem de Dados Volumétricos. 2. Paginação de Dados. 3. Programação em Placa Gráfica. I. Farias, Ricardo Cordeiro de. II. Universidade Federal do Rio de Janeiro, COPPE, Programa de Engenharia de Sistemas e Computação. III. Título.

A todos que de alguma forma me guiaram até aqui.

Agradecimentos

Gostaria de agradecer aqui aos meus pais, William e Lúcia Helena Hall, que me deram o suporte necessário durante todas as etapas da minha vida.

Aos meus professores ainda na graduação, José Mexas e Dirce Uesu, por todo o incentivo e orientação. Ao meu orientador, Ricardo Farias, que muito além de apenas orientar academicamente, foi um grande amigo e me ajudou a passar por uma fase ruim da minha vida, confiando nas minhas habilidades. À todos os professores deste Programa de Pós-Graduação, especialmente ao Prof. Ricardo Marroquim, que aguentou boa parte dos meus problemas iniciais durante o mestrado.

À Professora Rosa Leonora que me incentivou a perseguir este grau acadêmico e que sempre me inspirou e guiou. Aos meus Chefes da Universidade Federal Fluminense, Professores Alberto Domingues Vianna, Alair Sarmet, Alessandro Severo e Marcelo Nacif por todo seu incentivo e paciência, dados meus atrasos recorrentes ao trabalho durante o último ano. Não esquecendo dos meus colegas, Bené, Janet, Martha e Thiago, por todo incentivo e compreensão.

Aos meus amigos que sempre me apoiaram e em especial ao meu amigo Felipe que nessa última etapa do mestrado, foi essencial para a conclusão do meu curso.

Resumo da Dissertação apresentada à COPPE/UFRJ como parte dos requisitos necessários para a obtenção do grau de Mestre em Ciências (M.Sc.)

FILTRAGEM DE GRANDES DADOS SÍSMICOS TRI-DIMENSIONAIS EM PLACAS GRÁFICAS UTILIZANDO CUDA

Bryan Marinho Hall

Abril/2014

Orientador: Ricardo Cordeiro de Farias

Programa: Engenharia de Sistemas e Computação

Este trabalho tem a atenção voltada ao estudo de filtragem tridimensional de dados grandes, utilizando a técnica demoninada Filtro FKK. Este tipo de filtragem não é muito comum entre geofísicos e engenheiros de petróleo, então alguns fatos importantes serão explorados. Como os dados de aquisição sísmica 3D são geralmente muito grandes se faz necessário um particionamento em diversas etapas de todo o processo, que pode ser lento e computacionalmente custoso. Sendo assim, analisamos também algumas implementações em GPU das ferramentas necessárias para concluir com êxito o objetivo.

Abstract of Dissertation presented to COPPE/UFRJ as a partial fulfillment of the requirements for the degree of Master of Science (M.Sc.)

HUGE THREE DIMENSIONAL SEISMIC DATA FILTERING IN GPU USING CUDA

Bryan Marinho Hall

April/2014

Advisor: Ricardo Cordeiro de Farias

Department: Systems Engineering and Computer Science

This work is the attention devoted to the study of three-dimensional filtering large data, using the technique known as FKK filter. This kind of filtering is not very common among geophysicists and petroleum engineers, so some important facts will be explored. As the data of 3D seismic acquisition are generally very large, partitioning is needed at various stages throughout the process, which can be slow and computationally costly. Thus we also analyze some GPU implementations of the tools necessary to successfully accomplish the goal.

Sumário

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

"In the twenty-first century, the robot will take place which slave labor occupied in ancient civilization."

– Nikola Tesla

Análise e interpretação de dados geofísicos de possíveis campos de exploração são sempre a primeira etapa para a decisão de se instalar ou não um poço de exploração de petróleo, sendo uma etapa que requer tecnologias avançadas e á altamente custosa, tanto em termos econômicos quanto em gasto de pessoal e poder de processamento computacional. Existem diversas empresas nacionais e multinacionais que oferecem esse tipo de serviço especializado, que se baseia dados de geologia e geofísica para determinar quais são as chances de se encontrar ou não petróleo em uma dada região, bem como outras informações relevantes a respeito de fraturas nas rochas, densidade das camadas de sedimentos e estimativa da quantidade de óleo, caso exista um reservatório.

Esta etapa de análise das condições da subsuperfície é composta basicamente de etapas de aquisição de dados, sejam dados amostrados nos métodos sísmicos, levantamentos geológicos e etc, mas também por meio de simulações. Contudo, métodos empíricos como o abordado neste trabalho são costumeiramente muito ruidosos e diversas técnicas diferentes precisam ser utilizadas para preparar o dado para a coleta de informações geológicas pertinentes, a fim de maximizar o conhecimento dos campos, anteriormente ao procedimento de instalação de equipamentos de upstream (nomenclatura utilizada para a série de procedimentos necessários para retirar petróleo dos reservatórios) [1].

Nesse cenário muitas técnicas são utilizadas e em sua maior parte, estudos em 2D por serem mais rápidos e baratos em sua aquisição. Contudo aquisições 3D de dados

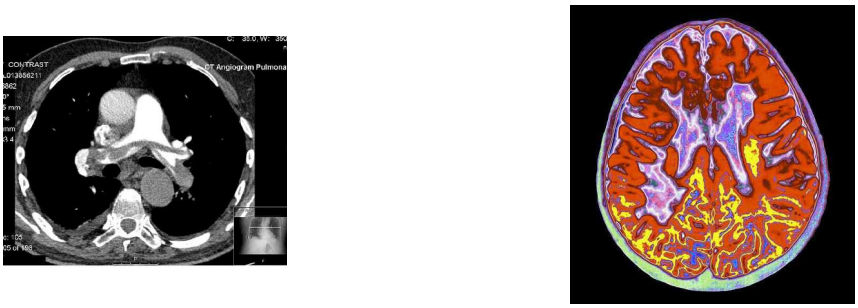


Figura 1.1: Exemplos de Dados Volumétricos - fatias 2D

Figura 1.2: Exemplos de Dados Sísmicos Volumétricos - fatias 2D

sísmicos 1.2 já estão ficando mais acessíveis e mais importantes a cada dia. Não apenas dados sísmicos, mas dados de tomografias computadorizadas, ressonâncias magnéticas 1.1, entre outros, que são dados do mesmo tipo, fazendo com que este estudo, por mais que seja voltado para os dados sísmicos, possa ser utilizado em qualquer dado tri-dimensional, salvo algum ajuste adicional de parâmetros.

Inserir imagem da aquisição 3D sísmica

1.1 Motivação

Com a corrida pela tecnologia para explorar a camada pré-sal em busca de petróleo, todo conhecimento das camadas, tipos de solos, rochas e falhas geológicas é relevante. Seja um estudo geológico ou geofísico, adquirido e tratado de diferentes maneiras, e cada novo pedaço de informação ajudando a mensurar as reservas alvo e minimizar o risco da atividade exploratória, como vazamento (Blow-out) [1] e a consequente interrupção da produção. De uma maneira geral, os dados obtidos através das técnicas de sísmicas 3D [2] é representado como um grande volume de dados, podendo passar dos 100 GB de dados (e possivelmente já bem maiores), e cada vez mais aumentando esse número em termos de precisão e extensão.

Tendo em vista o grande aumento na quantidade de informações digitais, e aprimoramento dos equipamentos no sentido de aumento de resolução da captura, a questão de otimização de execução de certos algoritmos para, principalmente,

redução de tempo de processamento, passou a ser foco de pesquisas tanto por parte da indústria quanto da academia. Operações que antes tinham alto custo computacional e não eram possíveis de serem executadas *In Loco* já podem ser visualizadas e até mesmo trabalhadas, podendo ser aceitas ou rejeitadas, de acordo com a qualidade ou propósito da aquisição.

1.2 A Proposta do Trabalho

Este trabalho tem como objetivo estruturar algoritmos simples e eficientes para a criação de uma ferramenta de paginação e filtragem de dados tri-dimensionais sísmicos a partir da Transformada de Fourier, extendendo em uma dimensão extra o filtro F-K em forma de leque, tradicionalmente utilizado para duas dimensões.

Além disso vamos utilizar esse conjunto para fazer uma comparação entre duas diferentes versões: uma totalmente em CPU e a outra tirando vantagens da tecnologia de processamento paralelo em GPU - NVIDIA CUDA - para que possamos fazer a comparação entre a eficiência de cada uma delas e discutir sua performance. Existem atualmente muitos trabalhos relacionados na área de paginação e processamento distribuído que vão muito além do escopo desse trabalho, que iremos apenas citá-los, principalmente na discussão geral.

O restante do trabalho fará uma abordagem sobre todos os tópicos que foram necessários para a criação dos algoritmos. São organizados da seguinte maneira:

- Trabalhos Relacionados - apresenta estudos de geologia e computação gráfica utilizados como referência
- Fundamentação Teórica - apresenta um breve resumo das técnicas utilizadas
- Implementação - apresenta os detalhes da solução proposta
- Resultados - discussão dos resultados obtidos
- Conclusão

Capítulo 2

Revisão Bibliográfica

"Everything should be made as
simple as possible, but not simpler."
– Albert Einstein

Para realizarmos toda a série de procedimentos até chegar ao dado filtrado como desejado várias técnicas matemáticas e computacionais foram utilizadas. Este capítulo trata dos trabalhos relacionados aos quais faremos referência no decorrer da dissertação, e está intimamente relacionado com o próximo capítulo, que faz uma fundamentação teórica acerca dos conhecimentos envolvidos.

2.1 Exposição do Problema

Como introduzido pelo Capítulo I, o objetivo deste estudo é o de criar uma interface para a definição de um filtro geofísico específico, o filtro tri-dimensional F-K-K, em formato de cone, bem como avaliar as dificuldades de cada etapa, inclusive a paginação dos dados, desde a definição de parâmetros até o resultado final filtrado, utilizando algorítmos sequenciais e em paralelo, com a tecnologia NVIDIA CUDA.

Desta forma, para cada etapa, existe alguma vantagem e alguma limitação pela utilização de cada uma das estratégias. O fluxograma 2.1 representa toda a cadeia de procedimentos necessários:

A CPU e a GPU foram desenhadas para finalidades distintas: as CPUs têm apenas alguns poucos núcleos, e conseguem rodar apenas uma quantidade relativamente pequena de threads simultâneas, mesmo que em uma velocidade alta. Já a GPU tem uma quantidade bem maior de núcleos especializados, que já ultrapassa algumas centenas, podendo executar milhares de threads ao mesmo tempo [3]. Po-

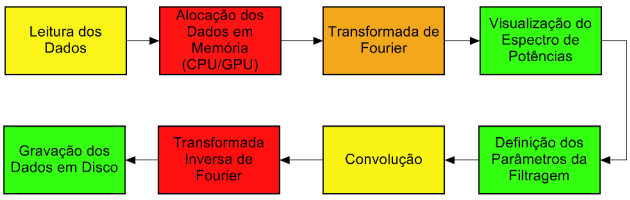


Figura 2.1: Fluxo geral de execução.

demos então esperar que códigos que possuam um grande número de operações, ou dados grandes que executem a mesma operação, podem ser beneficiados pelo uso das GPUs. Levando em conta cada etapa separadamente, todo o conjunto de procedimentos abordados nesse trabalho é altamente paralelizável, como será discutido nos capítulos seguintes.

Porém existem também as desvantagens da GPU:

- Latência na cópia de dados CPU para GPU e vice-versa;
- Necessidade de alto nível de paralelismo na tarefa a ser executada;
- Necessidade de dados organizados (coalescência).

Como as GPUs utilizam o barramento PCI-E a latência na transferência de dados entre CPU e GPU pode desacelerar qualquer procedimento a ser executado. Contudo é necessário frisar que já existem placas gráficas NVidia totalmente integradas à placa mãe, resolvendo este problema. Quanto ao nível de paralelismo devemos destacar que nem todo algorítmo é vantajoso sob o ponto de vista das GPUs. Tarefas que requeiram, por exemplo, várias comunicações com a memória principal do computador ou diversas ordenações nos dados são exemplos de processos custosos e pioram o desempenho das GPUs, muitas vezes sendo solucionados por uma abordagem de computação heterogênea. Ainda no tópico relativo aos dados, como um conjunto de threads será enviado para diferentes gerenciadores na própria GPU, os dados precisam ser bem organizados para que a eficiencia seja efetiva.

Assim o objetivo deste trabalho pode ser sumarizado nesses tópicos:

- Diminuir o trabalho humano no que tange à diminuição de procedimentos necessários para se obter um volume inteiro filtrado, substituindo inúmeros filtros 2D por um filtro global, 3D
- Definir parâmetros fáceis de interpretar (definição da máscara de filtragem)
- Ganhar performance, quando comparadas as versões sequencial e paralela
- Criar uma biblioteca que execute todos os procedimentos necessários para alcançar os objetivos acima para massas de grandes de dados (alguma paginação pode ser necessária)

2.1.1 Arquitetura da GPU

Neste estudo foi utilizada apenas uma configuração de máquina, já que o objetivo não é fazer um benchmark entre placas distintas. A placa de vídeo utilizada foi uma NVidia GeForce m630, com 96 núcleos e 1GB de memória disponível. Esta placa utiliza a arquitetura chamada Kepler [4] e sua estrutura geral se encontra na imagem a seguir.

Figura 2.2: Arquitetura das GPUs NVidia Kepler

Na GPU as tarefas são executadas por *threads*, que por sua vez são agrupadas em blocos que são, por fim, organizados em *grids*. Cada bloco é executado em uma unidade de processamento, ou *streaming multiprocessor*. Também temos que salientar que caso alguma thread esteja ociosa, esperando alguma resposta de outra thread, o gerenciador da placa gráfica pode salvar o estado atual e lançar outras threads para execução. Desta maneira um núcleo que talvez esteja esperando por algum dado fica em *stand-by* e enquanto isso outra *thread* está ativa e trabalhando. As *grids* representam um arranjo dos dados, que pode ser em uma, duas ou três dimensões, e fornecem maneiras de indexarmos os dados.

A Memória Global pode ser acessada por todas as threads e ter os dados coalescentes é de extrema importância. A Memória Compartilhada pode ser acessada apenas pelas *threads* de um mesmo bloco. A quantidade de memória disponível é geralmente pequena e algumas alocações internas reduzem um pouco mais esse espaço. A Memória Constante pode ser uma alternativa para o acesso à memória global da GPU, seu acesso é mais rápido porém é apenas para leitura. Caso algum dado solicitado esteja na memória constante sua leitura é tão rápida quanto a de um dado na memória compartilhada, dentro do mesmo *Multiprocessor*. A Memória

de textura é otimizada para quando precisamos acessar dados localmente próximos, assim como para interpolar valores de dados vizinhos [4].

2.1.2 NVidia CUDA - Computer Unified Device Architecture

CUDA é uma linguagem de programação criada para facilitar a criação de códigos para GPUs. Funciona como um subconjunto da linguagem C, porém com algumas novas funções específicas para o comando de instruções à GPU. As primeiras placas gráficas que suportavam CUDA foram as da série G8x e o pacote SDK foi tornado público em 2006, com a GeForce 8800 GTX.

Diferentemente das placas gráficas anteriores, as placas com suporte à CUDA não funcionam apenas como pipeline gráfico, mas cada unidade de processamento é programável para GPGPU (General Purpose Graphic Processing Units). Em suas primeiras placas, apenas a precisão de ponto fluante simples era suportada, porém com a crescente utilização de GPUs para acelerar projetos científicos de pesquisa, a comunidade acabou criando uma necessidade por maior precisão. Atualmente algumas placas já dispoem de precisão dupla para ponto fluante, bem como um espaço de memória compartilhada para cada uma das unidades de processamento (streaming multiprocessors).

O pacote de desenvolvimento, que pode ser baixado gratuitamente do site da NVidia, fornece toda a interface programável para criar programas na GPU. Durante este estudo utilizamos várias versões enquanto a NVidia desenvolvia seus produtos, porém no estágio final a versão utilizada é a v5.5. Além de todas as funções específicas para o controle da GPU, o pacote provê um compilador exclusivo para os códigos em CUDA, chamado *nvcc* [5].

2.1.3 Paginação e Processamento Distribuído

O algoritmo da Transformada de Fourier é muito utilizado para pesquisas sobre paginação e processamento distribuído por conta da necessidade da integralidade dos dados para calcular cada uma das saídas. Por conta desse tipo *All-to-All* de comunicação, a maior parte das implementações de Transformadas de Fourier para dimensões acima de 1-D executam a divisão total em subproblemas de 1-D. Essa abordagem faz todo sentido, uma vez que problemas em dimensões maiores podem ser totalmente escritos como em 1-D, porém para sistemas de processamento distribuído pode não ser a melhor opção, por talvez exigirem mais uma cópia *All-to-All* e mais uma transposição de dados [6]. De acordo com os resultados de [7][8] devemos esperar algum ganho no uso da GPU para esta finalidade. A paginação dos dados em

si sempre vão requerer comunicação com a memória da CPU e/ou disco. O objetivo maior da abordagem utilizando placas gráficas é justamente realizar processamento ao mesmo tempo que outras tarefas, como leitura/escrita do/em disco. Este tipo de abordagem é o alvo da nossa implementação.

Existem 2 abordagens em se tratando de Transformadas de Fourier de dados 3D: executa-se transformações 2D, de todos os planos em um determinado sentido, executa-se uma transposição e em seguida transforma-se a última dimensão; ou executa-se transformação em 1D seguida da transposição, até que as 3 dimensões sejam calculadas. A estratégia utilizada neste trabalho aborda principalmente a primeira vertente, pois em se tratando do processamento em placas gráficas a quantidade de cópias de dados entre CPU e GPU é reduzido, contudo o processamento necessário apenas para a transformação é o mesmo [9]. Contudo, apesar de nossos experimentos todos realizarem a transformação em 2D seguida de outra em 1D, também criamos funções para sequenciar 3 etapas em 1D, caso os dados sejam realmente muito grandes.

Capítulo 3

Fundamentação Teórica

"I seldom end up where I wanted to go, but almost always end up where I need to be."

– Douglas Adams

Este capítulo tem o objetivo de expor alguns detalhes e propriedades importantes de cada conhecimento necessário para a realização da filtragem F-K-K, bem como provê um panorama abrangente sobre a Transformada de Fourier e FFT.

3.1 Transformada de Fourier e FFT

3.1.1 Introdução à análise de sinais digitais

A qualidade dos dados se fazem tão importantes quanto as técnicas utilizadas para chegar a algum resultado. Assim se torna imprescindível definir um sinal e suas características para conseguirmos extrair informações relevantes. Dependendo de sua fonte o sinal pode ser classificado como analógico ou digital, onde a diferença básica é que num sinal analógico as amostras são tomadas em todo instante de tempo, enquanto que o sinal digital é proveniente de uma amostragem desse dado analógico.

Um sinal digital é uma sequência indexada de valores, que podem ser reais ou complexos. Podemos então definir uma função, com domínio no conjunto dos inteiros, de forma que:

$$f(n) = x_n \tag{3.1}$$

Onde $f(n)$ é a sequência de valores e x_n representa cada um dos valores, amostrados ou criados. No caso real essa sequência pode ser convenientemente escrita em forma de vetor e para o caso complexo são necessários 2 vetores, um para gaurdar a informação real e outro para a imaginária. Como será útilizado mais adiante, os números complexos podem ser expressos da seguinte maneira:

$$z(n) = a(n) + ib(n) \tag{3.2}$$

$$M(z(n)) = \sqrt{a(n)^2 + b(n)^2} \tag{3.3}$$

$$\theta(z(n)) = \arctan b(n)/a(n) \tag{3.4}$$

onde $z(n)$ é o número complexo, $a(n)$ é sua parte real e $b(n)$ sua parte imaginária. Também vale frisar que $M(z(n))$ é o módulo do número complexo e $theta(z(n))$ é sua fase. Por último se $z(n)$ é um número complexo da forma $z(n) = a(n) + ib(n)$, então dizemos que seu conjugado é $z(n) = a(n) - ib(n)$.

Ainda sobre os números complexos destaca-se a fórmula de Euler, que é extremamente importante para sua representação, uma vez que expressa números complexos como sendo uma soma seno e cosseno:

$$e^{ix} = cos(x) + isin(x) \tag{3.5}$$

Esta representação é muito útil na forma polar de um número complexo:

$$z = a + ib = |z|(cos(\theta) + isin(\theta)) = M(z)e^{i\theta} \tag{3.6}$$

3.1.2 Transformada de Fourier e FFT

A Transformada de Fourier, originalmente publicada em 1822, é muito útil para a análise de sinais pois transfere o dado original a um domínio diferente, onde características não facilmente perceptíveis podem ser analisadas e modificadas. Este fato se torna realmente importante pois essa transformação é totalmente reversível, através da Transformada Inversa de Fourier. Essa característica de se poder reconstruir o dado original integralmente é a parte fundamental deste tipo de filtragem, porém a grande utilidade está em modificar o resultado da Transformada de Fourier antes de executar a inversão.

- Filtro Passa Baixa (efeito de blur)
- Filtro Passa Banda

- Filtro Passa Alta (realça contornos)
- Filtro Unsharp

Neste trabalho estamos interessados apenas em um tipo especial de filtragem, utilizado em dados de aquisições sísmicas, chamado de filtragem f-k, que será explorado em uma seção própria.

A formulação da Transformada Contínua de Fourier é a seguinte:

$$F(\xi) = \int_{-\infty}^{\infty} f(x) e^{-2\phi i x \xi} \, \mathrm{d}x \tag{3.7}$$

E a transformada inversa, que possui uma formulação muito similar:

$$f(x) = \int_{-\infty}^{\infty} F(\xi) e^{2\phi i x \xi} \, \mathrm{d}\xi \tag{3.8}$$

Para o caso de uma dimensão, a transformada leva o conjunto original de dados para um espaço onde a frequência das ocorrências pode ser visualizada. Como a saída da Transformada de Fourier é um conjunto de dados complexos, podemos desta maneira desenhar um gráfico desses dados, como no exemplo da figura 3.1, que foi obtido através da transformação de uma função retangular.

Figura 3.1: Uma função retangular e sua transformada de Fourier

Contudo é necessária uma versão que seja utilizável em um conjunto discreto de dados, de forma a tornar este conhecimento aplicável às tecnologias de captação (de luz, de ondas, etc) e ao aparato computacional. Estima-se que este algoritmo do caso discreto em questão é executado bilhões de vezes todos os dias [10].

A representação da Transformada de Fourier para o caso discreto é:

$$X_k = \sum_{n=0}^{N-1} x_n e^{-i 2\phi k n / N} \tag{3.9}$$

Onde X_k é a transformada de Fourier do conjunto na posição k e N é o número de amostras do dado original. Dois fatos importantes podem ser elevados diante dessa formulação:

- Apenas o dado em si e o seu tamanho são suficientes para realizar a transformação

- Cada ponto do conjunto transformado depende de todos os dados originais

Existem diversas bibliotecas computacionais para calcular a Transformada Discreta de Fourier e como a formulação é relativamente simples, criar um programa específico para este fim não é difícil. Porém este algorítmo tem complexidade N^2 e para dados grandes se torna inviável seu uso. Contudo em 1965, Cooley e Tukey criaram um algorítmo com complexidade $Nlog(N)$, nomeado *Fast Fourier Transform* [11]. A figura **??** mostra a relação do número de multiplicações necessárias para a Tranformada Discreta de Fourier (DFT) e Transformada Rápida de Fourier (FFT).

Este algorítmo se aproveita de uma outra propriedade da Transformada de Fourier Discreta. Supondo que o tamanho do dado é uma potência de 2, podemos reescrever a equação da seguinte maneira:

$$N = 2^p \tag{3.10}$$

$$N = 2M \tag{3.11}$$

$$X_k = \sum_{n=0}^{N-1} x_n e^{-i2\phi kn/N} \tag{3.12}$$

Mudando o parâmetro do somatório, temos:

$$X_k = \sum_{n=0}^{2M-1} x_n e^{-i2\phi kn/N} \tag{3.13}$$

Que pode então ser convenientemente reescrito da seguinte maneira:

$$X_k = \sum_{n=0}^{M-1} x_{2n} e^{-i2\phi k2n/N} + \sum_{n=0}^{M-1} x_{2n+1} e^{-i2\phi k(2n+1)/N} \tag{3.14}$$

$$X_k = \sum_{n=0}^{M-1} x_{2n} e^{-i2\phi k2n/N} + \sum_{n=0}^{M-1} x_{2n} e^{-i2\phi k2n/N} + \sum_{n=0}^{M-1} x_{2n} e^{-i2\phi k/N} \tag{3.15}$$

Podemos então definir:

$$X_{even}k = \sum_{n=0}^{M-1} x_{2n} e^{-i2\phi k2n/N} \tag{3.16}$$

$$X_{odd}k = \sum_{n=0}^{M-1} x_{2n+1} e^{-i2\phi k(2n+1)/N} \tag{3.17}$$

E a equação fica escrita da seguinte maneira:

$$X_k = X_{even}k + X_{odd}k \sum_{n=0}^{M-1} x_{2n} e^{-i2\phi k/N} \tag{3.18}$$

Vale a pena notar que, dada a natureza cíclica das funções seno e cosseno, existe uma outra propriedade interessante acerca da Transformada de Fourier:

$$X_{k+K} = X_{even}k - X_{odd}k \sum_{n=0}^{M-1} x_{2n} e^{-i2\phi k/N} \tag{3.19}$$

Podemos notar então que basta calcular apenas a metade das trasnformadas e para a outra metade basta que reutilizemos os resultados da primeira etapa [12]. Também devemos frisar que para o cálculo do espectro de potência da Transformada de Fourier o dado na posição X_k tem o mesmo valor de X_{k+K}, reduzindo também a necessidade de memória.

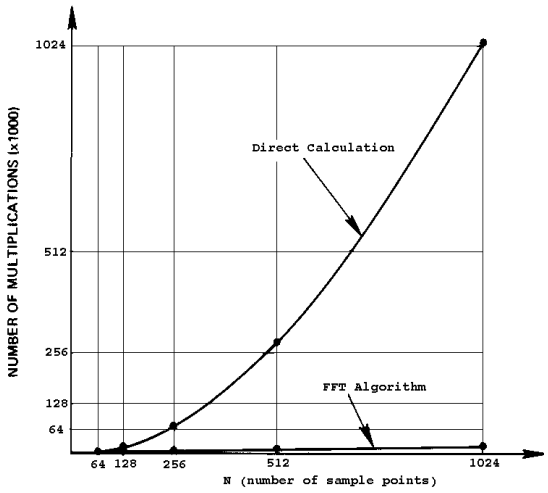


Figura 3.2: Vantagem na utilização da FFT sobre DFT

Para chegar ao resultado final, executa-se uma etapa chamada de *butterfly* que consiste exatamente na reutilização dos dados já calculados nas etapas posteriores [13].

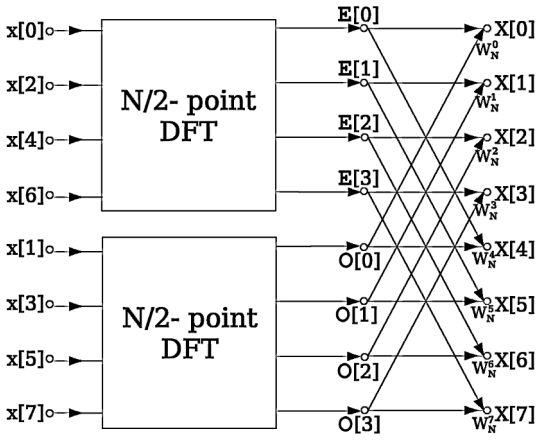


Figura 3.3: Multiplicação em butterfly para FFT

Porém é importante notar que não são apenas os dados múltiplos de 2 que tem essa proproedade. Segundo [14], é possível reescrever a equação de maneira análoga para números primos.

Para finalizar essa seção, vamos enumerar algumas propriedades importantes da Transformada de Fourier, apenas para o caso discreto. Para o caso contínuo pode-se consultar [15].

Separabilidade: Se uma função $f(k, p)$ puder ser escrita como produto de duas outras funções $f_x(k)$ e $f_y(p)$, e dois números reais a e b então:

$$F(w, q) = F_x(w)F_y(q) \tag{3.20}$$

Linearidade:

$$O_f(af(k) + bg(p)) = aF(w) + bG(q) \tag{3.21}$$

Escalamento:

$$O_f(f(ak)) =$$

$1/\mathrm{a}F(w/a)(3.22)$

Convolução: O operador linear convolução é muito útil para realização de filtragem de dados e tem a seguinte formulação, para o caso discreto: dadas duas sequências f e g,

$$(f \star g)(k) = h(k) = \sum_{j=0}^{k} f(j)g(k - j) \tag{3.23}$$

Onde f e g tem tamanho n, e $h(k)$ representa o resultado da Convolução. Contudo, para o interesse presente é necessário explorar as propriedades do operador

convolução em dados transformados, através do algorítmo de Fourier. Aplicando a Transformada à convolução, chegamos ao seguinte resultado:

$$(f \star g)(k) = F(x)G(x) \tag{3.24}$$

Onde $F(x)$ e $G(x)$ são as transformadas de fourier de $f(k)$ e $g(k)$, respectivamente. Analisando essa fórmula (descrita com mais detalhes em [12]) percebemos que podemos trocar o custo computacional do um somatório por apenas uma multiplicação ponto a ponto. Este teorema é a base central de todos os filtros de frequência e utilizaremos esse conhecimento para desenhar o filtro F-K-K.

3.1.3 Desafios para Tratamentos de Dados Grandes

Um dos problemas mencionados na introdução consiste na falta de memória para alocar a grande massa de dados do tipo aquisição sísmica 3-D. Já é notável o crescimento recente no volume de dados de aquisições sísmicas e o mesmo se apoia no fato de que as sísmicas tri-dimensionais estão cada vez mais acessíveis. Ainda hoje o tipo mais utilizado são as aquisições de seções 2-D, por serem mais baratas, mais fáceis de serem interpretadas e processadas. O dado volumétrico tem algumas vantagens nítidas sobre bidimensional como continuidade lateral [16] e melhor definição da área total explorável [2].

Deste modo algumas massas de dados conseguem chegar até terabytes de informação, mesmo que não sejam tão facilmente obtíveis. Nos dados analisados neste trabalho as massas de dados não são tão grandes, porém, para fins de benchmarking, criamos massas de dados nulas, já que o trabalho para o cálculo da transformada de Fourier seria o mesmo.

Capítulo 4

Solução Proposta

"A dream doesn't become reality
through magic; it takes sweat,
determination and hard work."
– Colin Powell

A solução proposta deste trabalho será descrita em detalhes nesse capítulo. Tratam-se de duas implementações, uma em CPU e outra em GPU, a fim de fazer comparações antes de expor os resultados. Em ambos os casos utilizamos a biblioteca OpenCV para a leitura e visualização dos dados, ainda em 2D, a fim de compararmos os resultados com o observado nas referências do filtro F-K. A cadeia de procedimentos pode ser resumida para ambos os casos, como exemplificado no fluxograma 4.1a seguir:

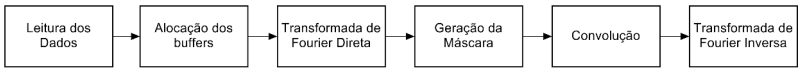


Figura 4.1: Esquema geral das duas implementações

4.1 Solução baseada em CPU

Esta implementação tem como características gerais a utilização da biblioteca FFTW [17][18] e o código é totalmente sequencial. Na etapa de leitura o dado completo é dividido de maneira a maximizar a utilização da memória disponível. Essa divisão é necessária em dois momentos diferentes da execução: uma anteriormente à execução da tranformada em 2D e a outra em 1D.

A fim de ocupar o maior espaço disponível calculamos previamente a quantidade de dados ótima, levando em consideração apenas a quantidade de memória

disponível no sistema naquele instante. Na transformação em 2D temos o dado organizado da forma inicial, padronizada para dados sísmicos, nas direções Z-Y-X. Desta maneira queremos conhecer a maior quantidade de fatias bi-dimensionais de tamanho dimZ*dimY para os espaços de memória de entrada e de saída de dados. Para alocar o espaço referente à saída de dados, a matriz é organizada em fatias bi-dimensionais de dimY*(dimZ/2+1) de números complexos, utilizando a simplificação obtida através de uma entrada puramente real, conforme descrito no manual do FFTW [19]. E por fim para alocar o espaço refente à entrada de dados, a matriz é organizada em fatias de tamanho dimY*dimZ. Desta maneira, a quantidade de fatias alocáveis é:

$$NumSlices_{2D} = (RAMDisponível/2(DimZ + 1)DimYsizeof(float)) \quad (4.1)$$

De maneira análoga, para a alocação em 1D necessitamos do dado de entrada, proveniente da etapa anterior e um buffer de saída, desta vez do mesmo tamanho do buffer de entrada. Cada um dos buffers possui o máximo de fatias de dimX*(dimZ/2+1) de números complexos. Assim, a quantidade nesta etapa é:

$$NumSlices_{1D} = (RAMDisponível/2Dim(X)(dimZ/2 + 1)sizeof(complex)) \quad (4.2)$$

As figuras 4.2 e 4.3 demonstram a utilização da memória para a entrada e saída dos dados necessários para as distintas etapas:

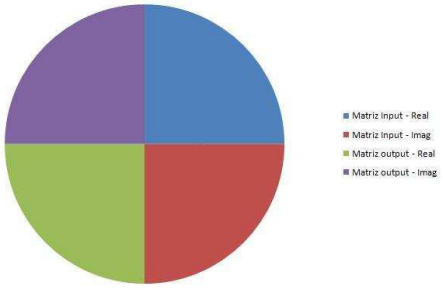


Figura 4.2: Memória utilizada para a transformação em 1D

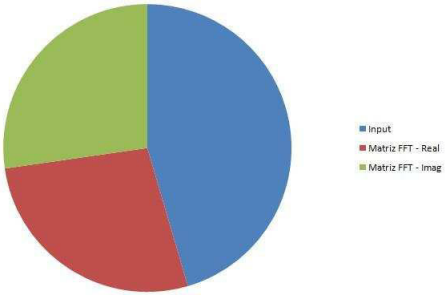


Figura 4.3: Memória utilizada para a transformação em 2D

Para o caso do algorítmo para CPU devemos guardar algumas ressalvas acerca da paginação do próprio sistema operacional - apesar de utilizarmos informações presentes da quantidade de memória disponível nesse momento, o sistema pode necessitar de memória e resolver paginar alguma área utilizada pelo algorítmo. Sabendo disso, a fim de minimizar problemas com o sistema operacional, devemos deixar uma parcela da memória RAM livre.

O esquema de paginação de memória, para o caso do algorítmo para CPU baseia-se na divisão do dado total em partições iguais, constituidas de um determinado número de fatias 2D do dado sísmico. Porém, a fim de utilizar melhor a eficiência de leitura e gravação, e posteriormente adaptar este esquema para a GPU, alocamos sempre duas partições distintas de dado de entrada simultaneamente. O algorítmo foi pensado desta maneira a fim de possibilitar o uso de multicores também para CPU, como disparar nova leitura ou nova gravação em disco de threads executadas em núcleos distintos, porém este caso já estaria fora do escopo de análise deste trabalho.

A etapa de processamento consiste basicamente na utilização da biblioteca FFTW e por isso ela merece alguns comentários. Atualmente ela é tida como a mais rápida rotina implementada para realizar o Fast Fourier Transform e baseia-se na pre-definição de parâmetros e cálculos heurísticos a fim de determinar o algorítmo ótimo para cada caso particular. Estão implementados os algoritmos de Coley-Tukey, em sua versão para potências de números primos, algoritmo de Rader para tamanhos primos, e uma versão split-radix [19].

Apesar de todos estes algoritmos, o que realmente apresenta a grande utilidade e performance da FFTW é a idéia de planejar o futuro. Existe uma classe chamada de *fftw plan* onde são definidos todos os parâmetros do dado a ser transformado, que são o tamanho de cada uma das dimensões do dado, ponteiros para o dado de entrada e espaço de memória de saída, uma flag definindo se a transformada é direta ou inversa, e a última flag do próprio sistema. Esta última flag determina o como os testes heurísticos serão realizados a fim de determinar o algorítmo que melhor se adapta aos parâmetros passados.

As flags de performance são: FFTW ESTIMATE, que usam parâmetros razoáveis e resultam, em sua maioria, em velocidades sub-ótimas de execução. Porém, caso dada transformada seja executada uma única vez, talvez seja a melhor opção em termos de tempo; FFTW MEASURE que efetivamente realiza cálculos a fim de determinar o melhor algorítmo a ser utilizado. Como no caso de dados sísmicos serem de um tamanho razoavelmente grande, e a transformada a ser realizada em etapas de iguais parâmetros, este método se torna ideal. Ao início determina-se o melhor método e replica-se o mesmo plano para fatias diferentes do dado; FFTW MEASURE, a FFTW PATIENT e FFTW EXHAUSTIVE realizam testes mais demorados, com a gama de algorítmos disponíveis para escolha sempre aumentando. A última opção foi desenhada para linhas de execução e provavelmente com alguma fonte de alimentação contínua de dados; FFTW WISDOM ONLY utiliza dados garimpados pelos algorítmos e complementados pelos próprios usuários, resultando num plano adequado para um problema bem específico.

Abaixo encontra-se um trecho do código com a implementação do cálculo da memória necessária disponível e a inicialização dos planos:

```
void FKKFilter::allocMemory(char machine, int dimX, int dimY, int dimZ)
{
  ...
  cpuMemory = getSystemMemory(); //memória RAM disponível na CPU
  ...

  memoryAvailable = cpuMemory*(1 - kernelReservedMemory); //hardcode a
      fim de não utilizar toda a RAM do sistema
  partitions2D = floor( memoryAvailable / 2*(dimZ+1)*dimY*sizeof(float)
      );
  partitions1D = floor( memoryAvailable / 2*(dimZ/2+1)*dimX*sizeof(
      float));

  ...
  //definição dos parâmetros da transformada, usando FFTW
  planForward2D = fftwf_plan_dft_r2c_2d(dimY, dimZ, inputReal,
      outputComplex, FFTW_MEASURE);
  planBackward2D = fftwf_plan_dft_c2r_2d(dimY, dimZ, inputComplex,
      outputReal, FFTW_MEASURE);

  planForward1D = fftwf_plan_dft_c2c_1d(dimX, inputComplex,
      outputComplex, FFTW_MEASURE);
  planBackward1D = fftwf_plan_dft_c2c_1d(dimX, inputComplex,
```

```
        outputComplex, FFTW_MEASURE);

}
```

A classe *fftw plan* mencionada anteriormente, inicializada no trecho do código acima, fica totalmente caracterizado a partir das seguintes especificações:

- Escolha do tipo de entrada de dados (real ou complexo - r2c significa entrada real, c2c entrada complexa);
- Definição dos tamanhos do dado a ser transformado;
- Definição dos ponteiros de entrada e saída;
- Escolha da *flag* de performance.

Após a execução da transformação em 2D, a fim de otimizar a próxima etapa, é necessário realizar uma reordenação nos dados antes da transformada de Fourier 1D. O dado é reorganizado levando em consideração a ordenação dos dados, que anteriormente eram Z x Y x X e serão transformados para X x Z x Y. Esta reordenação é custosa mas ordenar o dado de maneira a simplesmente enviá-lo para a linha de processamento, bloco a bloco, além de agilizar o processo permite uma melhor organização das tarefas.

Os dados sísmicos são então transformados da dimensão tempo-espaço-espaço para frequência-wavenumber-wavenumber [2] pela aplicação da FFT a fim de se apresentar o espectro de potência simplificado para a determinação dos parâmetros.

O cálculo do espectro de potência é calculado da seguinte maneira [20]:

$$Spec = Log_{10}(1 + \sqrt{real^2 + imag^2}) \tag{4.3}$$

Figura 4.4: Fatia 2D com seu respectivo spectro de potência

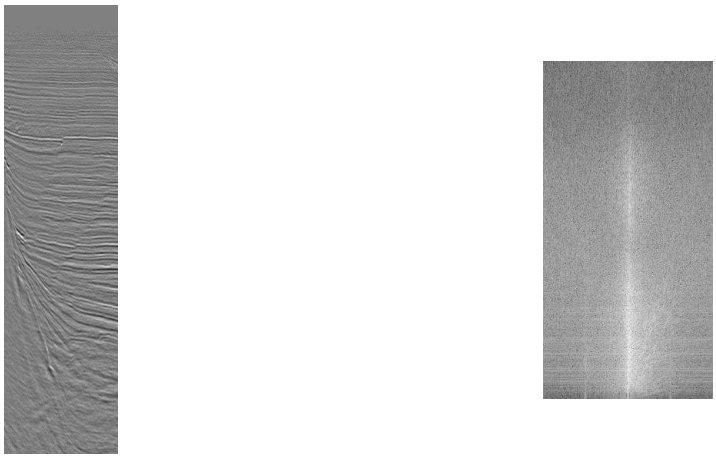


Figura 4.5: Fatia 2D com seu respectivo spectro de potência

Figura 4.6: Fatia 2D com seu respectivo spectro de potência

Como anteriormente descrito nos capítulos 2 e 3 uma máscara para filtragem utilizando FFT é apenas um conjunto de dados com o mesmo tamanho do dado de entrada (ou simplificado, para o caso de entradas exclusivamente reais) com um valor por posição, no intervalo de 0 a 1. Várias máscaras diferentes podem ser criadas e cada uma proporciona um resultado distinto, porém neste trabalho procuramos estender a filtragem F-K-K em 3D.

Figura 4.7: Fatia 2D e o resultado da aplicação do filtro F-K-K

De posse do espectro de potências do dado selecionado, pode-se visualmente definir os parâmetros do filtro a ser aplicado. Utilizamos apenas um formato de filtro, que é totalmente definido por duas equações: a parte superior de dois cones concêntricos, um com os braços maiores que o outro. O cone interno serve para delimitar a região do espaço da máscara onde os valores dos pontos serão 1.0, ou seja, não modificarão a Transformada de Fourier e o cone externo delimita a região onde os valores serão 0.0, significando que aqueles eventos serão removidos do resultado

da filtragem. A região que fica entre os dois cones tem valores que variam de 1.0, a partir da borda do cone interno, decaindo suavemente até 0.0, na borda do cone externo.

Os dados de entrada para a definição dos cones são baseados nas dimensões do dado original e são eles:

- Definição do braço do cone interno (região de aceitação total)
- Definição do braço do cone externo (região de *tapering*)
- Centro do cone (deslocamento)

Onde:

- Na região de aceitação total a máscara recebe o valor 1.0.
- Na região de *tapering* ocorre o decaimento do valor da máscara de 1.0 até 0.0, para evitar o efeito Gibbs.
- Deslocamento, mesmo que muitas vezes não seja necessário do ponto de vista prático.

Os tempos de execução de cada uma das tarefas estão demonstrados nos gráficos do próximo capítulo para tamanhos que cabem na memória (totalmente alocados) e um outro gráfico para dados que necessitam ser particionados (com uma observação: computadores e sistemas operacionais com funcionamento em 64 bits podem alocar uma quantidade muito maior que sistemas 32 bits, sendo que todos os dados utilizados podem ser alocados de uma única maneira. Porém existem dados maiores e nem todos os computadores teriam tanta memória quanto alguns volumes de dados. Desta forma forçamos o algorítmo a alocar partições do dado para fins de teste e benchmarking, sem alteração à logica ou o funcionamento do algorítmo.)

Como foi descrito no capítulo 2, os procedimentos para aplicação da transformada inversa são análogos aos da transformada direta. De fato, basta mudar um sinal da exponencial e dividir por um valor que depende das dimensões do dado. Criamos apenas uma função que executa tanto a transformada direta quanto a inversa, visto que ambas tem apenas uma diferença sutil em seu cálculo 3, que são ajustadas passando um parâmetro diferente em cada chamada da função. Desta maneira, após a aplicação do filtro, visualização e aceitação dos resultados, executa-se as mesmas rotinas, com essas pequenas modificações na etapa da transformada, a fim de retornar o dado do domínio F-K-K para o T-X-Y.

4.2 Solução Baseada em GPU

Antes de detalharmos esta implementação, se faz necessário descrever as características da placa gráfica utilizada para este estudo. A placa gráfica utilizada foi a NVidia GeForce 630m, com 1 GB de memória disponível, 96 Cuda Cores e com arquitetura Kepler (marcas registradas da NVidia). Esta arquitetura possibilita a execução simultânea de duas categorias distintas de tarefas, que serão exploradas adiante: execução das threads enviadas para a GPU e cópia de memória entre Host (CPU) e Device (GPU), bem como uma funcionalidade muito útil e extremamente impactante sobre tarefas que demandem alto tráfego de dados entre CPU e GPU, chamada de streams.

Apesar de ser possível executar códigos diferentes nas streams, o objetivo aqui é de apenas garantir que todos os kernels da GPU estão trabalhando em alguma tarefa o tempo todo. Ao executar essa linha de cópia, processamento e cópia estamos utilizando todo o poder da placa gráfica, e aproveitando o fato de que o controlador consegue gerenciar as threads e realizar uma cópia (CPU-GPU ou GPU-CPU) ao mesmo tempo.

Esta implementação tem como características gerais: utilização da biblioteca CUFFT [21] e o código executado em paralelo em GPU (NVIDIA CUDA). Inicialmente limitamos o escopo do algoritmo para apenas dados que cabiam na memória da GPU, mas posteriomente estendemos os algoritmos para uma versão com paginação de dados.

Assim como no filtro sequencial, precisamos definir os parâmetros do filtro a partir do resultado do volume já transformado, visualizando apenas o módulo de cada ponto (que é um número complexo). O procedimento de transformação é similar ao exposto no item anterior, portanto nos limitaremos a identificar as diferenças e acrescentar algumas informações que são válidas para este caso.

Na versão para GPU a etapa de alocação ganha um espaço extra para cada espaço de memória descrito na seção 4.1, um para entrada de dados e outro para saída, porém esses novos buffers são para a GPU. Antes de lançarmos qualquer kernel, necessitamos efetuar uma cópia de dados para a GPU e, como no caso específico da configuração utilizada e provavelmente de qualquer configuração de memória atualmente, a quantidade de memória disponível na GPU limita a alocação na CPU. Porém os cálculos para a alocação de memória são totalmente análogos.

Utilizamos nesta implementação dois buffers distintos para entrada e apenas um para saída. O algorítmo foi desenhado desta maneira para podermos utilizar as streams conjuntamente com a biblioteca cuFFT. Abaixo apresentamos um trecho do código utilizado para este fim:

```
void FKKFilter::allocMemory(char machine, int dimX, int dimY, int dimZ)
```

```
{
  ...
  cudaHostAlloc( (void**)&hostFFTMatrix, partitions2D*dimY*(dimZ/2+1)*
      sizeof(cufftComplex), cudaHostAllocDefault);

  //dois buffers diferentes a fim de utilizar streams
  cudaMalloc( (void**)&gpuInput2D_0, (paritions2D/2)*dimY*(dimZ/2+1)*
      sizeof(float));
  cudaMalloc( (void**)&gpuInput2D_1, (paritions2D/2)*dimY*(dimZ/2+1)*
      sizeof(float));

  //os resultados das transformadas são copiados em seus respectivos
      endereços, dentro deste buffer
  cudaMalloc( (void**)&gpuOutput2D, (paritions2D)*dimY*(dimZ/2+1)*
      sizeof(cufftComplex));

  //para a placa gráfica e versão CUDA utilizada, a melhor configuração
       foi resultado da utilização de 2 streams simultaneamente
  cufftPlanFwd2D = new cufftHandle[simultaneousStreams];
  cufftPlanFwd1D = new cufftHandle[simultaneousStreams];
  cufftPlanInv2D = new cufftHandle[simultaneousStreams];
  cufftPlanInv1D = new cufftHandle[simultaneousStreams];

  streams = new cudaStream_t[simultaneousStreams];

  for (int c=0; c<simultaneousStreams; c++)
  {
    //definição dos planos da FFT
    cufftPlan2d(&cufftPlanFwd2D[c], dimY, dimZ, CUFFT_R2C);
    cufftPlan2d(&cufftPlanFwd1D[c], dimX, CUFFT_C2C, 1);
    cufftPlan2d(&cufftPlanInv2D[c], dimY, dimZ, CUFFT_C2R);
    cufftPlan2d(&cufftPlanInv1D[c], dimX, CUFFT_C2C, 1);

    //para que as streams sejam efetivamente utilizadas, precisamos
        associar cada plano a uma das streams
    cufftSetStream(cufftPlanFwd2D[c], streams[c]);
    cufftSetStream(cufftPlanFwd1D[c], streams[c]);
    cufftSetStream(cufftPlanInv2D[c], streams[c]);
    cufftSetStream(cufftPlanInv1D[c], streams[c]);
```

```
}
```

Durante a fase de testes verificamos que as configurações para mais de 2 streams simultâneas não traziam benefícios de tempo, contudo existe um ganho pela utilização de 3 ou mais streams nas novas gerações das placas NVIDIA. A solução proposta pode ser iniciada com quantas streams forem solicitadas pelo usuário, através de um parâmetro na inicialização.

A classe *cufftPlan* é muito parecida com a classe *fftw plan*, da biblioteca FFTW. A definição de parâmetros é análoga: necessita-se dos dados relativos à dimensão do dado de entrada e que tipo de transformação está sendo feita. No caso dos planos bi-dimensionais utilizamos a propriedade de simetria da transformada de Fourier para dados reais.

Ao utilizar streams para o processamento, retornamos o controle de execução para a thread principal antes, liberando a CPU. Porém, sempre antes de iniciar outro bloco de processamento, devemos sincronizar as threads para evitar que a GPU execute instruções em ordem errada, ou principalmente com o dado errado. Após a transformação em 2D o algorítmo realiza a transposição dos dados anteriormente à execução da última dimensão. Esta etapa é de suma importância para a GPU, uma vez que sua arquitetura foi desenhada para trabalhar com dados concorrentes.

```
bool FKKFilter::organizeStreams2d(int maxPart, int actualPart, int
    finalPart)
{
  int i;
  int outputSize = dimY*(dimZ/2+1);
  int inputSize = dimY*dimZ;
  cufftResult result;
  cudaError_t error;

  for (i = actualPart; i < finalPart; i++)
  {
    //cópias dos pedaços de memória que serão utilizados em seguida
    error = cudaMemcpyAsync(gpuInput2D_0+i*inputSize, hostInputMatrix+i
        *inputSize, inputSize*sizeof(float), cudaMemcpyHostToDevice);

    error = cudaMemcpyAsync(gpuInput2D_1+(i+1)*inputSize,
        hostInputMatrix+(i+1)*inputSize, inputSize*sizeof(float),
        cudaMemcpyHostToDevice);

    //os planos de execução já foram linkados às streams anteriormente
```

```
    result = cufftExecR2C(cufftPlanFwd2D[0], gpuInput2D_0+i*inputSize,
        gpuOutput2D+i*outputSize);
    result = cufftExecR2C(cufftPlanFwd2D[1], gpuInput2D_1+(i+1)*
        inputSize, gpuOutput2D+(i+1)*outputSize);

    error = cudaMemcpyAsync(hostFFTMatrix+i*outputSize, gpuOutput2D+i*
        outputSize, outputSize*sizeof(cufftComplex),
        cudaMemcpyDeviceToHost);

    error = cudaMemcpyAsync(hostFFTMatrix+(i+1)*outputSize, gpuOutput2D
        +(i+1)*outputSize, outputSize*sizeof(cufftComplex),
        cudaMemcpyDeviceToHost);

  }
}
```

A sequencia de cópia CPU-GPU e GPU-CPU é onde temos maior gasto de tempo. Cada kernel necessita de 2 sequencias de cópia, então idealmente queremos fazer que o tempo necessário para que essas 2 cópias sejam iguais ao tempo de execução do kernel. A utilização das streams ajuda a mitigar esse problema, porém vale lembrar que a GPU está executando kernel ao mesmo tempo em que copia dados de/para a CPU.

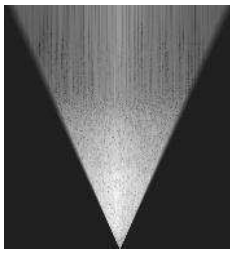
A sequência de Paginação para a aplicação da transformada de fourier é similar ao apresentado no item anterior, e será descrito em detalhes mais adiante.

4.3 Criação e Aplicação da Máscara

A máscara para convolução é definida a partir de dois de cones, com mesmo centro. Os parâmetros necessários para o desenho desses cones são o centro, um valor para o braço desenhado na direção X e outro na direção Y. Não podemos esquecer que, a fim de evitar o efeito de Biggs, devemos fazer um *tapering*, decaindo do cone mais interno ao cone mais externo. Assim:

```
bool FKKFilter::maskCreation(int centerX, int centerY, int centerZ,
    float innerX, float outerX, float innerY, float outerY)
{
  ...
  float tgCutYStart = (innerY/timeSample)/dimY/4;
  float tgTapperYStart = (innerY+outerY/timeSample)/dimY/4;
```

```
    float tgCutXStart = (innerX/timeSample)/dimX/4;
    float tgTapperXStart = (innerX+outerX/timeSample)/dimX/4;

    //calcula proporções entre os cones interno e externo
    float xRatio = tgTapperXStart/tgCutXStart;
    float yRatio = tgTapperYStart/tgCutYStart;

    // as variáveis i,j,k repesentam a posição atual a ser calculada

    hip = sqrt((float) (j-centerY)*(j-centerY)+(i-centerX)*(i-centerX));
    sinO = (i-centerX)/hip;
    cos0 = (j-centerY)/hip;

    //calcula o tappering, radialmente, a partir da borda do cone interno
    outerLimit = (-k+centerZ) *(xRatio * cosO + yRatio * sinO
        -1);
    //valor da função elíptica
    innerCone = (i-centerX) /tgCutXStart + (j-centerY) /tgCutYStart
        - (-k+centerZ)
    //valor da máscara, em casa posição
    finalValue = 1 - innerCone/outerLimit;

    if (innerCone<=0)finalValue=1.0;
    elseif(innerCone>outerLimit)finalValue=0.0;
    if (k>centerZ)finalValue=0.0;
}
```



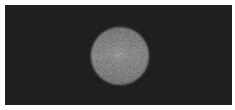
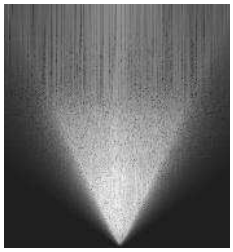


Figura 4.8: Vistas do Filtro F-K-K sem tappering



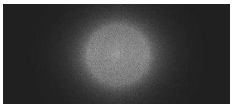


Figura 4.9: Vistas do Filtro F-K-K com tappering

Criamos duas funções diferentes para a criação da máscara: uma que calcula e mantém o resultado guardado na memória, e outra que aplica imediatamente o valor calculado à matriz da transformada. Fizemos isso a fim de futuramente experimentarmos outras máscaras que não são definidas de modo simples, como a parte superior de um cone.

Pode-se usar esferas, curvas de bezier, nurbs, ou até mesmo uma coleção desses itens, porém a quantidade requerida de cálculos poderia ser processada em uma etapa paralela, e necessitariam de espaço em memória para guardar os resultados desses cálculos, nos casos onde a descrição da máscara não seja trivial.

4.4 Paginação da Memória

A segunda parte da implementação foi a extensão dos códigos acima descritos para versões onde haveria paginação do dado, que estamos supondo que não é possível alocar totalmente os dados necessários em memória da GPU. Ao final da implementação, tínhamos duas abordagens distintas para a questão da paginação. A primeira era a suposição de que havia memória suficiente na CPU para armazenar todo o dado sísmico original e a segunda que não havia memória para tal.

O esquema de paginação em si representa um grande atraso em ambas as implementações. Isso porque para a versão em CPU ela representa apenas uma etapa intermediária indispensável para concluir a tarefa. Para a GPU, sempre na transição dos dados de entrada, as streams precisam ser sincronizadas a fim de obtermos o resultado final e podermos finalmente limpar os buffers de entrada, para que o novo bloco possa ser alocado.

Desta maneira esperamos ter o mesmo gasto computacional para ambas as versões. A paginação foi implementada seguindo este procedimento: uma massa de dados cujo processamento tenha sido realizado (seja transformações 2D ou 1D) é escrito em disco em arquivos sequenciais com um header mínimo, descrevendo apenas a quantidade de dados e qual o número da partição foi atribuído, seguido de todo o dado, real ou complexo.

Então realiza-se o processamento do próximo bloco até terminar cada um dos

estágios bi-dimensionais. Ao iniciar a transformação unidimensional, o dado lido já está dentro de cada um dos novos arquivos criado com os dados temporários, porém precisamos realizar uma reordenação dos dados, a fim de sequenciar a última dimensão que ainda não foi transformada.

Na primeira etapa são realizadas transformações bi-dimensionais e a totalidade dos dados necessários para realizar a última transformação está separada em cada etapa intermediária. Neste momento realiza-se um processo bem lento, que consiste na abertura de todos os arquivos temporários da etapa anterior e seguindo-se da seleção de apenas o trecho importante para cada novo subvolume de dados.

Como os algoritmos são exatamente os mesmos, tanto pra CPU quanto pra GPU, esperamos que não haja mudanças no que diz respeito à superioridade da GPU sobre a CPU, que foi verificada quando fizemos os testes de tempo para dados totalmente alocados. No próximo capítulo mostramos alguns gráficos e tabelas com os tempos alcançados a fim de comprovar este fato.

Capítulo 5

Resultados e Discussões

"The important thing is that men should have a purpose in life. It should be something useful, something good."

– Dalai Lama

Nesta seção faremos as comparações entre os tempos de execução para cada etapa do processo e também para toda sequência de procedimentos até o resultado final. Além disso, faremos algumas considerações acerca da estratégia utilizada para a paginação dos dados. Os testes de desempenho para ambas as implementações foram realizados em um computador Intel i5, 2.5Ghz, com 6 GB de memória RAM, placa gráfica NVidia GeForce 630m, com 1GB de memória RAM e 96 Cuda Cores, no sistema operacional Windows 7, 64 bits. Por questões de direitos sobre os volumes de dados utilizados, não podemos divulgar informações acerca dos mesmos, então apenas vamos nos referir pelas siglas Vol1, Vol2 e Vol3, com as seguintes propriedades:

Dataset	DimX	DimY	DimZ	Total Bytes
Vol1	246	503	751	371708952
Vol2	176	401	377	106428608
Vol3	271	221	876	209858064

Tabela 5.1: Dimensões dos dados sísmicos e tamanho total em bytes.

5.1 Comparação de tempos

Os gráficos 5.1 5.2 5.3 a seguir sumarizam os testes iniciais, levando em conta apenas o tempo de processamento dos algorítmos sem a necessidade de paginação dos dados. Apesar da peculiaridade do tipo de transformação necessária para o procedimento, os tempos de execução em paralelo na GPU apresentam um ganho significativo sobre a execução sequencial.

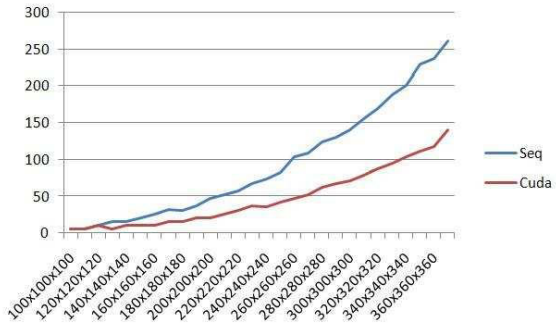


Figura 5.1: Tempos registrados para geração da máscara

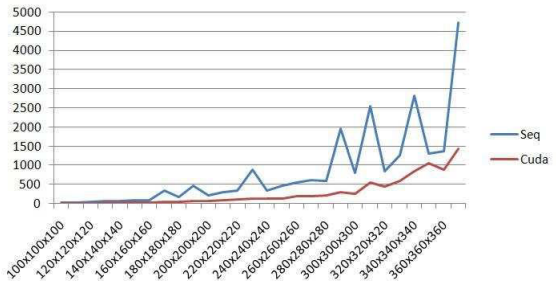


Figura 5.2: Tempos registrados para a execução da Convolução

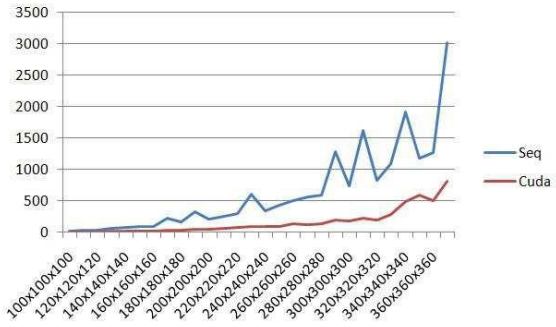


Figura 5.3: Tempos Registrado para a Execução da Transformada de Fourier

Como se pode notar, o tempo gasto pela GPU é sempre inferior ao da CPU e pode ser verificado através dos seguintes dados que foram coletados:

- Para o cálculo da máscara, 2.0 vezes em média, 3.0 vezes máximo;

- Para a execução da convolução, 3.1 vezes em média, 6.6 vezes máximo;
- Para o cálculo da Transformada de Fourier, 4.5 vezes em média, 7.3 vezes máximo.

A partir dos dados coletados vemos claramente que conforme a quantidade de dados aumenta, a diferença nos tempos de execução entre as implementações fica cada vez mais distante. Apesar de a média das operações não revelar grandes melhorias, quanto maior o dado, maior a diferença percentual, que é altamente desejável. Seria necessário uma placa gráfica com mais memória a fim de determinarmos exatamente a extensão desse ganho, e até onde vale a pena seguir a implementação em GPU, mas os dados apontam para uma melhoria significativa se levarmos em conta todo o procedimento.

Pode-se reparar também uma peculiaridade no gráfico: existem alguns picos de tempo maior para algumas amostras de menor tamanho. Isso se deve ao fato de a biblioteca FFTW, tanto quanto a cuFFT, não dispor de um algorítmo muito eficiente para aquele tamanho específico. Segundo o manual, as duas implementações tem algorítmos ótimos para a execução de dados de dimensões múltiplas dos números primos 2, 3, 5, 7, 9 e 11 [19]. De acordo com a documentação, sempre que os planos encontram uma configuração para a qual não possuem um algorítmo específico, eles recorrem a um algorítmo genérico.

No que diz respeito à paginação dos dados as etapas intermediárias de leitura e gravação em disco acontecem independentemente da GPU, tornando-se assim um fardo com o mesmo tempo de execução para as duas implementações. Porém ao fazer a organização da linha de processamento na GPU liberamos a CPU para outras atividades. Nesta implementação a CPU faz a escrita dos dados já processados pela GPU enquanto a mesma está processando o próximo subvolume de dados.

Tamanho	Bytes	GPU (ms)	CPU (ms)
400x400x400	256000000	3011	14056
450x450x450	364500000	27877	52089
500x500x500	500000000	31434	81721
550x550x550	665500000	308896	420780
600x600x600	864000000	380657	612753

Tabela 5.2: Tempos e execução da Paginação dos dados.

Os tamanhos de dados relacionados na tabela 5.2 foram escolhidos de forma que todas as etapas tivessem alguma modificação. Aumentar por uma quantidade constante apenas uma das dimensões representaria apenas uma nova repetição do trabalho que já fora realizado e por isso decidimos incrementar em todas as direções ao mesmo tempo. Desta maneira modificamos a quantidade de fatias a serem alocadas

tanto nas transformações bi-dimensionais quanto nas uni-dimensionais, modificando o tamanho total de cada subvolume do processamento. Vale salientar que apenas o primeiro item da tabela 5.2 é totalmente alocado na GPU e por isso existe uma discrepância grande em relação aos outros tempos medidos.

Nota-se que a GPU continua tendo benefícios na utilização sobre uma implementação puramente em CPU. Apesar de existir um gargalo na transição de dados para a GPU, este não mais representa um problema tão grande, ao menos no que tange o cálculo de transformadas de Fourier. Porém devemos notar que a velocidade da comunicação melhorou muito nas últimas inovações, e atualmente está limitada pela velocidade do barramento PCI-E (máximo de 8GB/sec em cada direção), para o computador utilizado nos testes.

Capítulo 6

Conclusões

"Divide each difficulty into as many parts as is feasible and necessary to resolve it."
– René Descartes

Diante do exposto acima percebemos que, apesar dos algorítmos serem simples e facilmente paralelizáveis, a paginação de memória que se faz necessária para dados muito grandes constitui um problema fundamental para a Transformada de Fourier, tanto para CPUs quanto para GPUs, porém foi possível obter um ganho de velocidade razoável em relação à CPU. Existem outros meios de aceleração para CPU, como o processamento utilizando vários cores do processador central, que talvez chegassem perto das velocidades obtidas pela GPU.Tendo em vista que a paginação de memória representa o mesmo custo computacional para ambas as implementações, podemos inferir que a GPU sempre terá vantagem em tempo de execução em comparação com a CPU.

As etapas de processamento em questão foram bastante simplificadas, o que já garantiu mais velocidade nas duas implementações, mas os códigos na GPU ainda têm a desvantagem de comunicação mais lenta, através do barramento PCI-E. Segundo Gordon Moore, o poder de processamento dos computadores dobra a cada 18 meses. Formulada em 1965 e conhecida como Lei de Moore, foi compravada estatisticamente que esta lei se aproxima muito da nossa realidade, e de fato basta observar os aparatos eletrônicos que são lançados no mercado todo ano. Desta maneira podemos nos perguntar: as GPUs ainda poderão superar o poder de processamentos das CPUs? Até a presente data, e de acordo com nossos e muitos outros estudos, parece que GPGPU ainda tem muito para evoluir e se extrapolarmos os gráficos, podemos inferir que, para muitas tarefas, as GPUs podem nos fornecer ganhos reais

significativos.

Ao final deste trabalho criamos uma biblioteca de paginação de dados para filtragem utilizando a transformada de Fourier, implementada na FFTW ou na cuFFT. Apesar de codificado apenas um tipo de filtro, várias adaptações são fáceis de realizar, tendo em vista que apenas se necessita alterar uma equação. Desta maneira este trabalho pode ser eficientemente usado para tratamento de qualquer tipo de dado volumétrico regular, captado através de qualquer equipamento que faça a discretização de fenômenos físicos.

Como este trabalho é apenas uma abordagem inicial de um tópico ainda alvo de muitas pesquisas e desenvolvimento, existem diversas melhorias posteriores. Inicialmente, por mais que já esteja facilmente codificável, o benchmarking na utilização de mais de uma placa gráfica simultaneamente; teste com multithreading para CPU, usando MPI; implementação própria de um algorítmo distribuído para a Transformada de Fourier; criação de um framework com diversos filtros para diversas finalidades; enfim, uma gama de futuros estudos a serem conduzidos.

Outros trabalhos multidisciplinares podem seguir naturalmente desta biblioteca. Como criamos funções capazes de realizar a paginação e o envio de determinadas partes de uma massa de dados para processamento, o mesmo algoritmo pode ser adaptado para realizar o processamento de arquivos de vídeo ou entradas de dados de dispositivos móveis, como dados do acelerômetro de um celular ou qualquer outros dispositivos.

Referências Bibliográficas

[1] THOMAS, J. E. *Fundamentos de Engenharia de Petróleo*. 2 ed. Rio de Janeiro, RJ, Editora Interciência, 2011.

[2] YILMAZ, O., DOHERTY, S. M. *Seismic Data Analysis: Processing, Inversion, and Interpretation of Seismic Data*. Investigations in Geophysics, No. 10. 2 ed. , Society of Exploration, 2000.

[3] RICHARDSON, A., GRAY, A. *Utilisation of the GPU architecture for HPC*. Relatório técnico, EPCC, The University of Edinburgh, 2008.

[4] CORPORATION, N. "Kepler GK110 Whitepaper". 2012. Disponível em: <http://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture

[5] SANDERS, J., KANDROT, E. *CUDA by Example: An Introduction to General-Purpose GPU Programming*. 1st ed. , Addison-Wesley Professional, 2010. ISBN: 0131387685, 9780131387683.

[6] JOACHIM HEIN, HEIKE JAGODE, U. S. A. S., TREW, A. "Parallel 3D-FFTs for Multi-Core Nodes on a Mesh Communication Network". In: *CUG 2008 Conference*, 2008.

[7] GLENN-ANDERSON, J. *Ultra Large-Scale FFT Processing on Graphics Processor Arrays*. Relatório técnico, enParallel Inc, 2009.

[8] GOVINDARAJU, N., LLOYD, B., DOTSENKO, Y., etal. "High performance discrete Fourier transforms on graphics processors". In: *High Performance Computing, Networking, Storage and Analysis, 2008. SC 2008. International Conference for*, pp. 1–12, Nov 2008. doi: 10.1109/SC.2008.5213922.

[9] CHEN, S., LI, X. "A hybrid GPU/CPU FFT library for large FFT problems". In: *Performance Computing and Communications Conference (IPCCC), 2013 IEEE 32nd International*, pp. 1–10, Dec 2013. doi: 10.1109/PCCC.2013.6742796.

[10] GUPTA, A. "Fourier Transform and Its Application in Cell Phones". In: *International Journal of Scientific and Research Publications, Volume 3, Issue 1*, 2013.

[11] COOLEY, J. W., TUKEY, J. W. "An algorithm for the machine calculation of complex Fourier series", *Math. Comput.*, v. 19, pp. 297–301, 1965. doi: 10.2307/2003354.

[12] GONZALEZ, R. C., WOODS, R. E. *Digital Image Processing (3rd Edition)*. Upper Saddle River, NJ, USA, Prentice-Hall, Inc., 2006. ISBN: 013168728X.

[13] HAYES, M. H. *Schaum's Outline of Theory and Problems of Digital Signal Processing*. McGraw Hill Companies, Inc., 1999. ISBN: 0070273898.

[14] BRIGHAM, E. O. *The Fast Fourier Transform and Its Applications*. Upper Saddle River, NJ, USA, Prentice-Hall, Inc., 1988. ISBN: 0-13-307505-2.

[15] KHAN, S. A. *Digital Design os Signal Processing Systems A Practical Approach*. John Wiley and Sons Ltd., 2011. ISBN: 9789479741832.

[16] PRESS, F., SIEVER, R., JORDAN, T., etal. *Para entender a Terra*. Bookman, 2006. ISBN: 9788536306117. Disponível em: <http://books.google.com.br/books?id=K8yWAAAACAAJ>.

[17] FRIGO, M., JOHNSON, S. G. "The Design and Implementation of FFTW3", *Proceedings of the IEEE*, v. 93, n. 2, pp. 216–231, 2005. Special issue on "Program Generation, Optimization, and Platform Adaptation".

[18] FRIGO, M. "A fast Fourier transform compiler". In: *Proc. 1999 ACM SIGPLAN Conf. on Programming Language Design and Implementation*, v. 34, pp. 169–180. ACM, May 1999.

[19] FRIGO, M., JOHNSON, S. G. "FFTW Manual". 2012. Disponível em: <http://www.fftw.org/fftw3.pdf>.

[20] MITRA, S. K. *Digital Signal Processing A Computer Based Approach*. University of California, Santa Barbara, McGraw-Hill, 1999.

[21] CORPORATION, N. "CUDA Cufft Library". 2007. Disponível em: <http://moss.csc.ncsu.edu/~mueller/cluster/nvidia/0.8/NVIDIA_CUFFT_Library