MINISTÉRIO DA CIÊNCIA E TECNOLOGIA
INSTITUTO NACIONAL DE PESQUISAS ESPACIAIS

PROCESSAMENTO DE ALTO DESEMPENHO APLICADO AO PROCESSAMENTO DE IMAGENS DE SATÉLITES COM AS TECNOLOGIAS OPENMP, CUDA E OPENCL

Marcos Lima Rodrigues
Marilyn Menecucci Ibañez
Rodrigo Augusto Rebouças
Viny Cesar Pereira

Trabalho Final da disciplina CAP-378 Tópicos em Observação da Terra

URL do documento original:
<http://urlib.net/>

INPE
São José dos Campos
2016

LISTA DE FIGURAS

Pág.

2.1 Processo de divisão entre as threads. 4
2.2 Comparativo do numero de operações de ponto flutuante entre CPUs e GPUs.
Fonte: Documentação Online CUDA Toolkit versão 7.5 [1] 5
2.3 Diferenças entre as arquiteturas CPU e GPU. 6
2.4 Modelo da organização do OpenCL. 7
2.5 Esquema comparativo entre a abordagem paralela baseada em CUDA e tradicional em CPU para o processamento digital de imagens: (a) CUDA; (b) CPU. 8
2.6 Fluxo do processo de compilação do código fonte e geração dos objetos referentes ao *device* e *host*. 9
2.7 Fluxograma de execução de um processamento de uma imagem com o uso de CUDA. 10
2.8 Sequências de etapas de Processamento Digital de Imagens de Satélite. . 12

3.1 Imagem LandSat 8 utilizada na aplicação das tecnologias de PAD. 15
3.2 Imagem LandSat 8 utilizada na aplicação das tecnologias de PAD. 16
3.3 Imagem LandSat 8 utilizada na aplicação das tecnologias de PAD. 16
3.4 Imagem LandSat 8 utilizada na aplicação das tecnologias de PAD. 17
3.5 Resultado do processamento das imagens do LandSat 8 aplicando o filtro Sobel do Anexo A. (a) Figura 3.1, (b) Figura 3.2, (c) Figura 3.3 e (d) Figura 3.4 19
3.6 *P*rofiling do algoritmo Sobel do Anexo A para processamento da Figura 3.4. 20
3.7 Resultado do Filtro Sobel com OpemMP aplicado na Figura 3.1 21
3.8 Resultado do Filtro Sobel com OpemMP aplicado na Figura 3.2 21
3.9 Resultado do Processamento da LandSat 8 aplicando o filtro Sobel em (a)CPU - OpenCV e (b)GPU - OpenCL da região do aeroporto de SJC; (c)CPU - OpenCV e (c)GPU - OpenCL de SJC e (e)CPU - OpenCV e (f)GPU - OpenCL de SJC colorida 24
3.10 Resultado do processamento das imagens do LandSat 8 aplicando o filtro Sobel em GPU. (a) Figura 3.1, (b) Figura 3.2, (c) Figura 3.3 e (d) Figura 3.4 26

3.11 Resultado do processamento das imagens do LandSat 8 aplicando o filtro Canny em GPU. (a) Figura 3.1, (b) Figura 3.2, (c) Figura 3.3 e (d) Figura 3.4 27

4.1 Estrutura básica de uma FPGA. 30
4.2 Vantagens de usar hardware reconfigurável em processamento de dados de sensoreamento remoto. 32

LISTA DE TABELAS

Pág.

3.1 Tempo de Processamento de CPU do filtro Sobel utilizando código do Anexo A. 19
3.2 Tempo de processamento com uso de OpenMP 20
3.3 Tempo de Processamento do filtro Sobel utilizando o CPU e a GPU para OpenCL. 23
3.4 Tempo de Processamento do filtro Sobel utilizando o CPU e a GPU para CUDA. 25
3.5 Tempo de Processamento do filtro Canny utilizando o CPU e a GPU. . . 26

SUMÁRIO

Pág.

1 Introdução 1

2 Fundamentação Teórica 3

2.1 Processamento de Alto Desempenho 3

2.1.1 OpenMP 3

2.1.2 GPGPU 4

2.1.3 OpenCL 6

2.1.4 CUDA 8

2.2 Pocessamento de Digitais de Imagens 11

2.2.1 Filtro Sobel 11

2.2.2 Imagens de Satélites 13

2.2.3 OpenCV 13

3 Resultados 15

3.1 Otimização do Compilador GCC 18

3.2 OpenMP 20

3.3 OpenCL 22

3.4 CUDA 24

4 Trabalhos Relacionados 29

4.1 *Field-Programmable Gate Array* (FPGA) 29

4.1.1 FPGAs no contexto de Processamento Digital de Imagens 30

5 Conclusões 35

REFERÊNCIAS BIBLIOGRÁFICAS **37**

ANEXO A - CÓDIGO FONTE - PROCESSAMENTO DO FILTRO SOBEL EM LINGUAGEM C **41**

ANEXO B - CÓDIGO FONTE - PROCESSAMENTO DO FILTRO SOBEL COM OPENMP **45**

ANEXO C - CÓDIGO FONTE - PROCESSAMENTO DO FILTRO SOBEL COM CUDA **49**

ANEXO D - CÓDIGO FONTE - PROCESSAMENTO DO FILTRO SOBEL COM OPENCL. **51**

1 Introdução

A evolução tecnológica nos últimos anos está relacionada com a capacidade de processamento dos computadores desenvolvidos. No entanto, essa evolução encontra atualmente barreiras na sua continuidade devido as limitações das tecnologias existentes para o desenvolvimento dos hardwares responsáveis pelo processamento de informações. Na tentativa de contornar essas limitações, novas tecnologias auxiliares aos núcleos de processamento dos computadores estão atualmente sendo melhoradas e consequentemente tendo uma ampliação nas suas utilizações. Dentro dessas novas tecnologias, encontram-se as *General Purpose Computing on Graphic Processing Units* (GPGPU) que são unidades de processamento representadas pelas placas de vídeo e permitem realizar processamento em ponto flutuante aumentando a capacidade do número de execução de tarefas (GONG; HAO, 2014).

Com o aumento da utilização dessas novas tecnologias a área de Processamento de Alto Desempenho (PAD) caminhou também para atender as demandas de utilização das GPGPU. Para isso, surgiu a necessidade do desenvolvimento de novas ferramentas para a programação voltadas para esse tipo de hardware. Até então, a PAD era representada nesse contexto pela *Open Multi-Processing* (OpenMP) que é uma *Aplication Program Interface* (API) para programação paralela. Para atender a nova demanda foram desenvolvidas APIś específcas para programação em GPGPU, dentre elas estão as APIś *Compute Unified Device Architecture* (CUDA) desenvolvida pela empresa NVIDIA e *Open Computing Language* (OpenCL) desenvolvida pela *Advanced Micro Devices* - AMD.

Juntamente com o crescimento da GPGPU, surge o grande aumento do número de informações das mais diversas áreas que necessitam de um processamento rápido e preciso para a geração de novas informações de elevada importância para a sociedade. Um modelo de dados que é muito utilizado nos dias atuais são as imagens de satélites. A análise desse tipo de imagem possibilita a geração de informações relacionadas ao estudo de fenômenos naturais que podem afetar direta ou indiretamente a vida do homem.

Para análise desse tipo de informação aplica-se os conceitos da área de Processamento de Digital de imagens (PDI). A PDI oferece uma grande variedade de metodologias que permitem a extração de informações com grande utilidade para a análise das imagens de satélites. Algumas das metodologias utilizadas em PDI para esse tipo de tarefa são: Filtragem, classificação, segmentação, registro, entre outras (GONZALEZ; WOODS, 2010).

Assim, a utilização do poder de processamento das tecnologias de desenvolvimento para GPGPU de PAD para melhorar o desempenho da aplicação das metodologias da área de PDI na análise de imagens de satélites é de grande interesse para essa comunidade científica. Dentro desse contexto, o objetivo desse projeto é apresentar um exemplo de aplicação de uma metodologia de PDI nas tecnologias de PAD para análise de imagens do satélite LandSat 8. A metodologia para exemplificação escolhida foi a filtragem, sendo representada pelo filtro ou operador de Sobel, que é um operador que permite realizar a identificação de bordas em um imagem (GONZALEZ; WOODS, 2010). No projeto apresenta-se o comportamento da execução desse operador dentro das APIś OpenMP, CUDA e OpenCL e na programação direta na CPU.

Para uma melhor apresentação do trabalho, este documento está organizado da seguinte forma: no Capítulo 2 é apresentado a contextualização das tecnologias e metodologias utilizadas no projeto, no Capítulo 3 mostra-se os resultados da aplicação do Filtro Sobel nas imagens escolhida para cada APIś de desenvolvimento estudada, no Capítulo 4 mostra-se alguns trabalhos com aplicações de PAD e PDI em outras tecnologias e por fim apresenta-se a conclusão do trabalho seguida das referências e dos anexos nos quais se encontram os códigos fontes utilizados no projeto.

2 Fundamentação Teórica

2.1 Processamento de Alto Desempenho

A área de alto desempenho têm sido de suma importância em diversas áreas da engenharia, possibilitando o avanço tecnológico das mesmas através de plataformas computacionais cada vez mais rápidas e eficientes (YANG; GUO, 2005). As aplicações dentro de PAD hoje, apesar de tentarem manter uma boa relação entre desempenho e facilidade de desenvolvimento, ajudaram a evoluir linguagens de programação, bibliotecas, compiladores, ferramentas e dispositivos de hardware que diminuíram os esforços humanos necessários para obter bons resultados utilizando PAD. Com o surgimento das técnicas para programação paralela e a adoção delas pela comunidade de desenvolvedores, várias delas hoje são padronizadas e utilizadas em larga escala inclusive para aplicações críticas. Em paralelo com soluções de programação para PAD, surgiram também dispositivos de hardware especializados em aplicações de alto desempenho, que introduziram um novo paradigma de particionar o processamento de uma aplicação complexa em uma grande quantidade de processadores de baixo poder computacional, elevando o nível do paralelismo e mantendo um baixo consumo energético.

A introdução de PAD na área de sensoreamento remoto mudou a forma com que os dados gerados são coletados, analisados, armazenados e transmitidos (PLAZA; CHANG, 2007). As tecnologias de alto desempenho permitiram que a capacidade de processamento desses dados fosse pelo menos próxima de tempo real, o que é fundamental dada a quantidade alta de dados gerados em curtos períodos de tempo. As seções a seguir descrevem algumas das técnicas de PAD utilizadas no processamento de imagens para aplicações espaciais abordadas neste trabalho.

2.1.1 OpenMP

O OpenMP é uma API para programação paralela em C, C++ e Fortran em memória compartilhada (CHAPMAN et al., 2008). Surgiu em 1997, criado pelo grupo *OpenMP Architecture Review Board* (ARB), inicialmente para Fortran, e em 1998 para C/C++, atualmente o OpenMP está na versão 4.5, lançada em novembro de 2015. Uma das grandes vantagens do OpenMP é que o ARB continua trabalhando nele, além de existir um fórum ativo (BOARD, 2008).

OpenMP é formado por um conjunto de diretivas de compilador, rotinas de biblioteca, e variáveis de ambiente, onde faz uso de *multithreading*, na qual existe a *Master*

Thread que executa uma sequência de instruções, onde divide as tarefas entre as *threads*, como ilustrado na Figura 2.1.

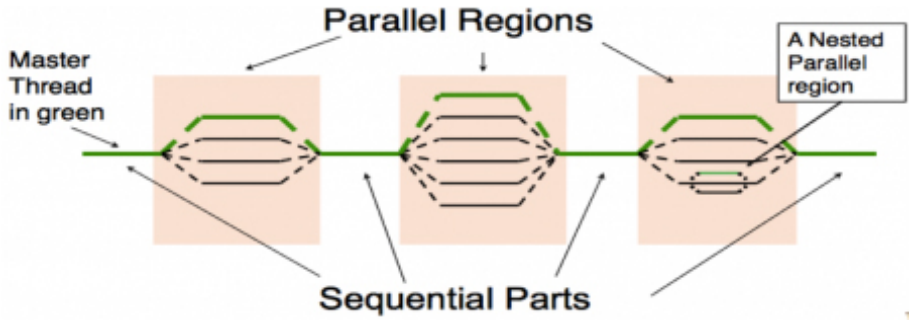


Figura 2.1 - Processo de divisão entre as threads.
Fonte:(NERSC, 2016)

Outra vantagem do OpenMP é a facilidade de se trabalhar, pois não requer grandes mudanças no código para torná-lo paralelo, como por exemplo, apenas com uma simples diretiva o compilador já defini uma região como paralela, para executar com as *threads*, caso o compilador não possua suporte para OpenMP, irá simplesmente tratar como comentário.

Exemplo de Diretiva de compilador, para definir uma região paralela:

- C/C++:

  #pragma omp parallel

- Fortran:

  !$OMP PARALLEL

Além das diretivas, faz uso de funções, como *omp_set_num_threads()* para definir o número de *threads*, ou definir variável de ambiente *OMP_NUM_THREADS*, antes da execução do código para fazer o mesmo.

2.1.2 GPGPU

O termo *General Purpose computing on Graphic Processing Units* (GPGPU)[1] reflete a utilização cada vez maior desse tipo de hardware específico para o desenvolvimento

[1]GPGPU - Utilização de GPUs para computação de propósito geral.

de aplicações nas mais diversas áreas do conhecimento como: medicina, ciências espaciais, processamento digital de imagens (PDI), computação gráfica, entre outras[2]. O desenvolvimento desse tipo de aplicação surgiu como alternativa a utilização de processamento por multiprocessadores, por apresentar a capacidade de realizar maior numero de operações de ponto flutuante como demonstra a Figura 2.2, com menor custo em geral com ganhos de até 10 vezes em relação à *Central Process Unit* (CPU) (GONG; HAO, 2014).

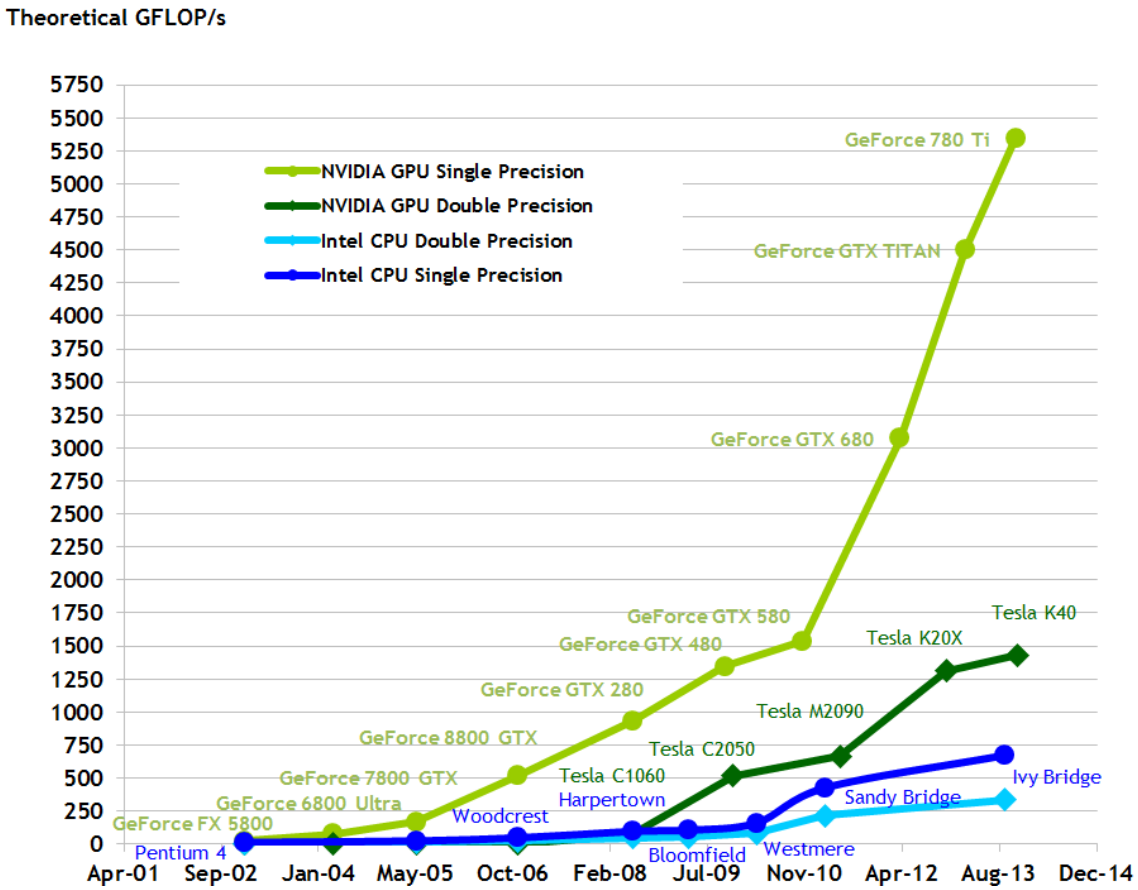


Figura 2.2 - Comparativo do numero de operações de ponto flutuante entre CPUs e GPUs. Fonte: Documentação Online CUDA Toolkit versão 7.5 [4]

Essa vantagem se deve em razão do hardware especializado das GPUs para renderização de imagens, que realizam muitas operações sobre um pequeno numero de dados. As GPUs possuem para esse fim uma grande quantidade de *Aritmetic Lo-*

[2]GPU catalog aplications disponível em http://www.nvidia.com.br/content/gpu-applications/PDF/GPU-apps-catalog-mar2015.pdf, consultado em set. 2016

[4]Disponível em: http://docs.nvidia.com/cuda/cuda-c-programming-guide/#axzz4JaDIOQR4. Acesso em set. de 2016.

gic Unit (ALU) em relação as CPUs, ver Figura 2.3, que podem ser aplicadas para aplicações massivamente paralelas.

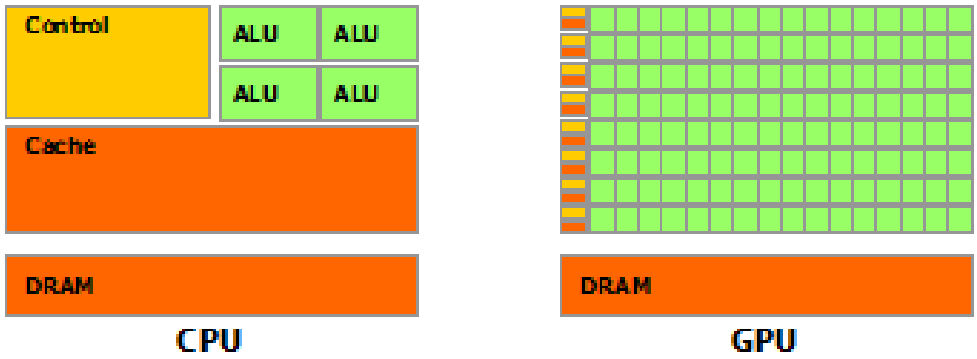


Figura 2.3 - Diferenças entre as arquiteturas CPU e GPU.
Fonte: Documentação Online CUDA Toolkit versão 7.5.

2.1.3 OpenCL

O OpenCL é um *framework* aberto, multiplataforma para programação paralela em diversos modelos de processadores. Esse *framework* pode ser utilizado em variados hardware como: computadores pessoais, servidores, dispositivo móveis e plataformas embarcadas. Uma das suas características principais é melhorar muito a velocidade e capacidade de resposta das aplicações das mais diversas áreas: jogos, entretenimento, médica e outras áreas científicas. As aplicações desenvolvidas com OpenCL conseguem acessar o processamento combinado da GPU e CPU de um computador, servidor ou de núcleos de APU em uma única plataforma unificada. Essas características contribuem muito na área de processamento de alto desempenho (PAD) (KRONOS, 2016).

A API OpenCL possui as seguintes características:

- Suporta programação paralela de dados e programação baseada em tarefas
- Faz uso do subconjunto ISO C99 com extensões para o paralelismo
- Utiliza requisitos numéricos baseados na IEEE754
- Possibilita a configuração para dispositivos portáteis e embarcados
- Permite a integração com OpenGL, OpenGL ES e outras APIs gráficas

- Define uma linguagem de programação baseada na linguagem C

O OpenCL visualiza um sistema computacional como um conjunto de vários dispositivos de computação. Assim, um único dispositivo computacional consiste de várias unidades de processamento, que englobam múltiplos elementos de processamento. Os programas executados em um dispositivo OpenCL são denominados de *kernels*. A execução de um único *kernel* permite executar vários elementos de processamentos em paralelo.

Considerando a linguagem de programa baseada em C, o OpenCL define uma *Application Programming Interface* (API) que permite que os programas em execução na máquina carreguem o *kernels* no computador e gerenciei os dispositivos de memória. O OpenCL possui a API padrão para as linguagens C e C++; no entanto, exitem API's de terceiros para as linguagens java, Python e .NET. (AMD, 2016). Um modelo da organização do OpenCL pode ser visualizado na Figura 2.1.3

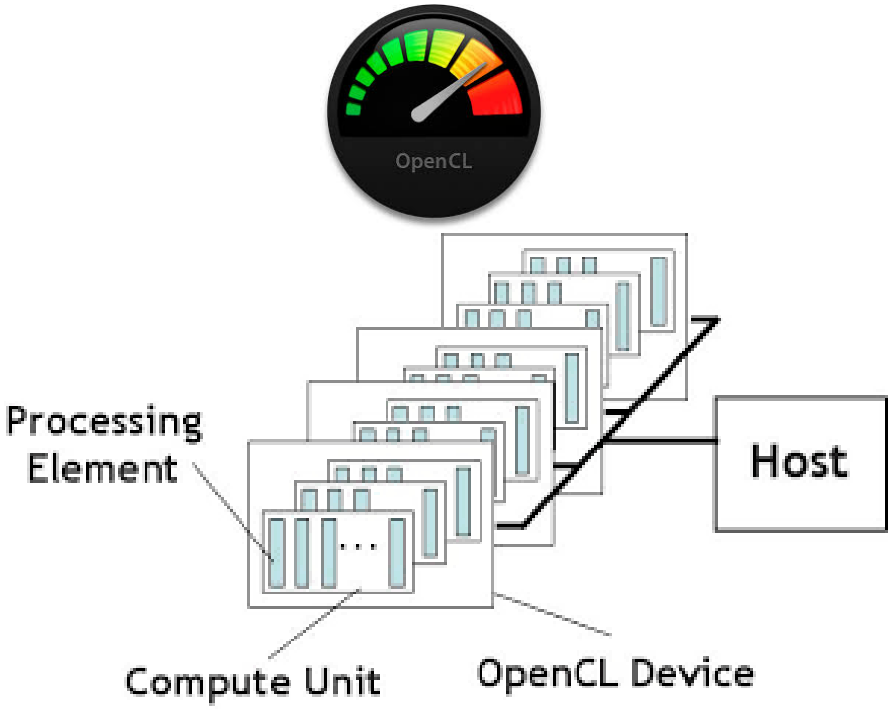


Figura 2.4 - Modelo da organização do OpenCL.
Fonte:(KRONOS, 2016)

A utilização da OpenCL pode ser encontrada em aplicações destinadas aos usuários da Apple na aplicação *Photos for macOS*, Sony na aplicação *Movie Studio HD* -

Sony, da CyberLink na aplicação *PowerDirector*, entre outros.

2.1.4 CUDA

A utilização de GPUs têm sido empregada no desenvolvimento de pesquisas relacionadas ao processamento digital de imagem, como forma de aproveitar a vantagem do processamento massivamente paralelo nativo nesses dispositivos. Além da sua capacidade elevada de processamento, essa tecnologia apresenta menor consumo de energia e menor curva de aprendizado. Por exemplo para as placas gráficas desenvolvidas para NVIDIA, existe um modelo de programação paralela baseado em linguagem C, denominado *Compute Unified Device Architecture* (CUDA). Nesse modelo as funções são chamadas de *kernels*, que mapeiam múltiplos *threads* para processar partes dos dados de entrada simultaneamente, como demonstra a Figura 2.5, num comparativo com a abordagem via CPU (GAO et al., 2009).

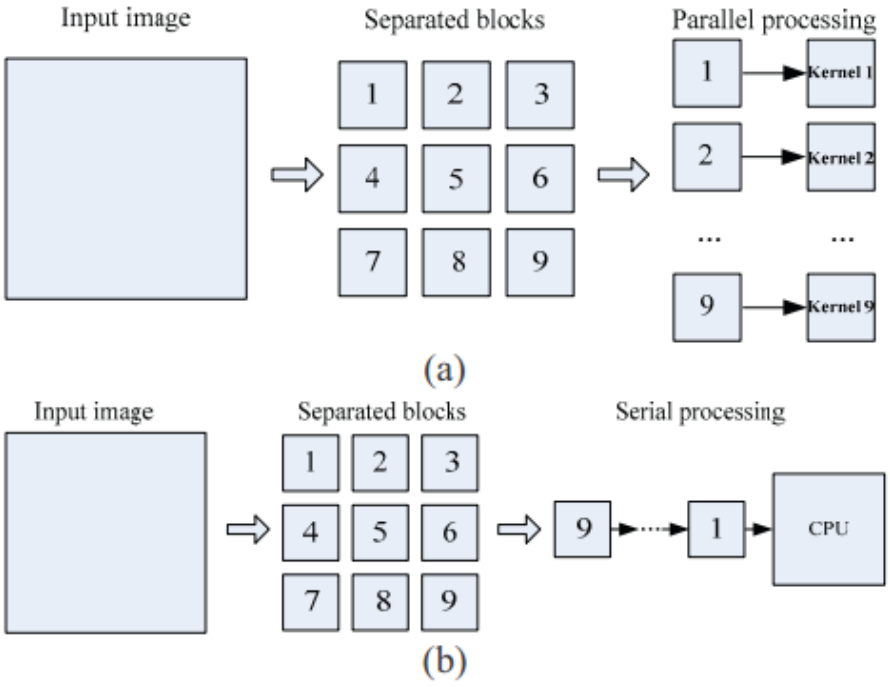


Figura 2.5 - Esquema comparativo entre a abordagem paralela baseada em CUDA e tradicional em CPU para o processamento digital de imagens: (a) CUDA; (b) CPU.

Fonte: Adaptada de (SI; ZHENG, 2010)

A programação em CUDA funciona em cooperação com o processador e a memória principal do computador, essa cooperação é realizada pelo *NVIDIA CUDA Compiler*

Driver (NVCC), que identifica as partes que devem ser processadas de forma paralela pelo "*device*" ou seja arquitetura (GPU) e a parte que será executada no "*host*" ou seja na CPU, conforme ilustra a Figura 2.6.

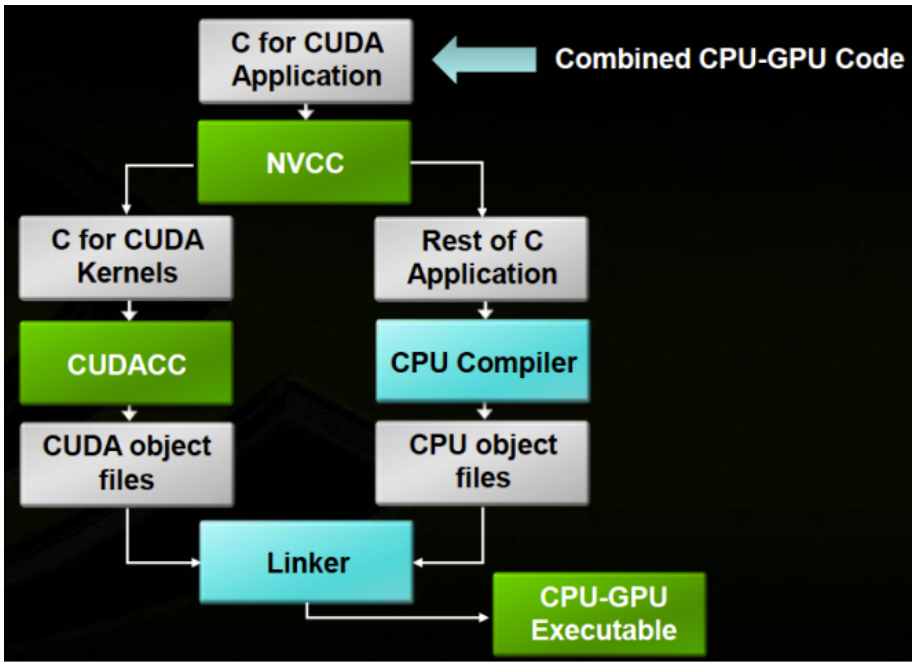


Figura 2.6 - Fluxo do processo de compilação do código fonte e geração dos objetos referentes ao *device* e *host*.
Fonte: Adaptada de (TSE, 2012)

A comunicação entre o *device* e o *host* é realizada através da porta PCI-Express, que pode alcançar velocidades de até 1 Gb/s por faixa de transmissão[5], o mapeamento realizado para os kernels é baseado na arquitetura *single-instruction, multiple-thread* (SIMT)(GAO et al., 2009).

A Figura 2.7 ilustra o fluxo de informações entre os elementos responsáveis pela processamento serial e paralelo, esse fatiamento do código é obtido através da compilação realizada pelo nvcc, que trata de forma separada instruções da GPU, enquanto o resto do código é tratado pelo compilador padrão C. Importante mencionar que a GPU não tem acesso direto a memória da CPU, desta forma a informação precisa ser transferida para a memória global compartilhada (SI; ZHENG, 2010).

O importante para se obter o máximo de performance da GPU é a etapa de prepara-

[5]PCI Express Performance disponível em http://www.pcisig.com/news_room/faqs/pcie3.0_faq/#EQ2, consultado em set. 2016.

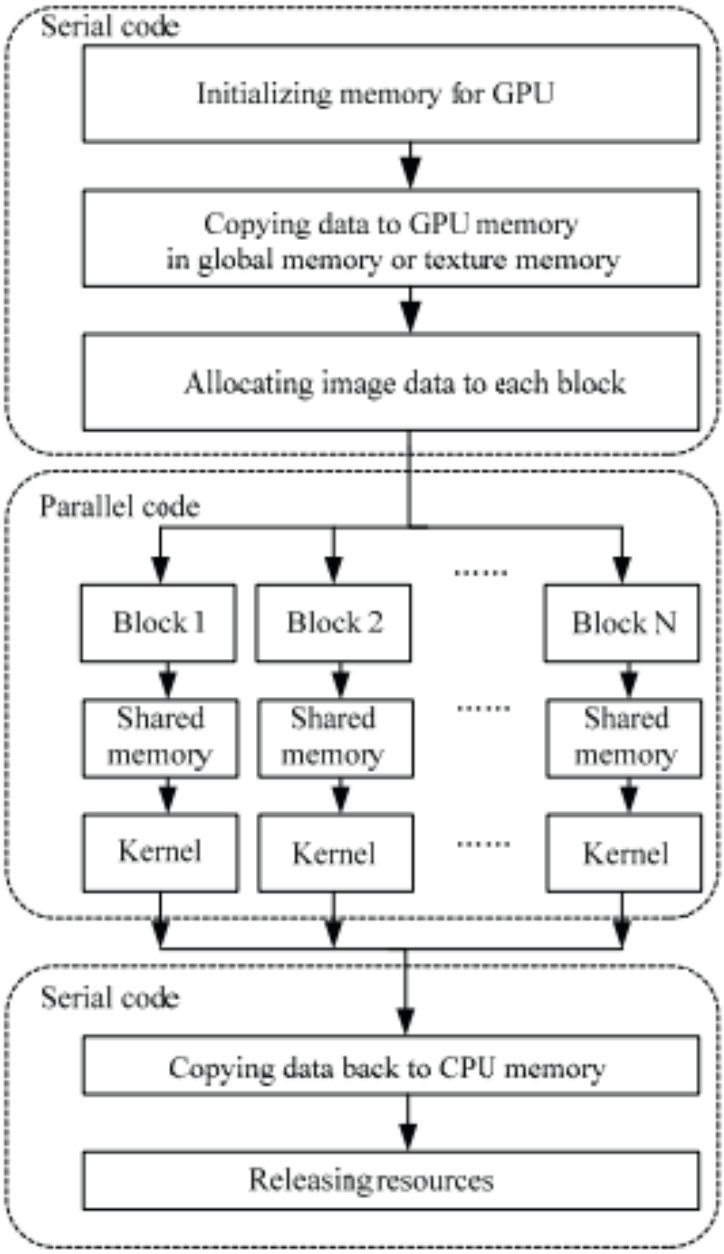


Figura 2.7 - Fluxograma de execução de um processamento de uma imagem com o uso de CUDA.

Fonte: Adaptada de (SI; ZHENG, 2010)

ção e definição da configuração para a execução dos kernels. Começa com a definição de quantos threads por blocos serão utilizados, e quantos blocos irão compor o Grid. Essa definição está intimamente ligada ao problema que está sendo tratado, no caso do processamento de imagem o tamanho dela.

2.2 Pocessamento de Digitais de Imagens

A área de processamento digitais de imagens considera que uma imagem digital possui as coordenadas x e y e a amplitude f da intensidade de tons de cinza finitas e discretas. O objetivo de se utilizar o processamento digital de imagens é melhorar características visuais de certos aspectos estruturais da imagem para facilitar a análise e interpretação humana (GONZALEZ; WOODS, 2010). Atualmente, o processamento de imagens é utilizado pelas mais diversas área de conhecimento e dentre elas está o processamento digital de imagens de satélites. Nessa área, as imagens de satélites são recebidas de forma bruta e processadas utilizando as técnicas de PDI para que informações sejam realçadas e classificadas para que o homem possa realizar sua análise e gerar resultados úteis para a sociedade. A Figura 2.2 apresenta uma possível sequência de etapas para o processamento digital de imagens de satélites.

A PDI apresenta diversas técnicas de manipulação de análise de imagens e dentre elas se encontra o realce de imagens utilizando operadores de filtragem. Existem diversos tipos de operadores cada um com suas características ideais para aplicação em áreas específicas. Um desses é o filtro Sobel que permite a detecção de contornos de bordas na imagem. O Sobel é detalhado na seção 2.2.1.

2.2.1 Filtro Sobel

O Sobel é um filtro de detecção de bordas horizontais e verticais em escalas de cinzas. O operador gradiente de Sobel tem a propriedade de realçar linhas verticais e horizontais mais escuras que o fundo, sem realçar pontos isolados. Esse filtro é dado pela seguinte equação

$$g(x, y) = Gx^2 + Gy^2 \tag{2.1}$$

Sendo que Gx e Gy são representados pelas máscaras $3x3$ a seguir:

$$[Gx] = \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix} \tag{2.2}$$

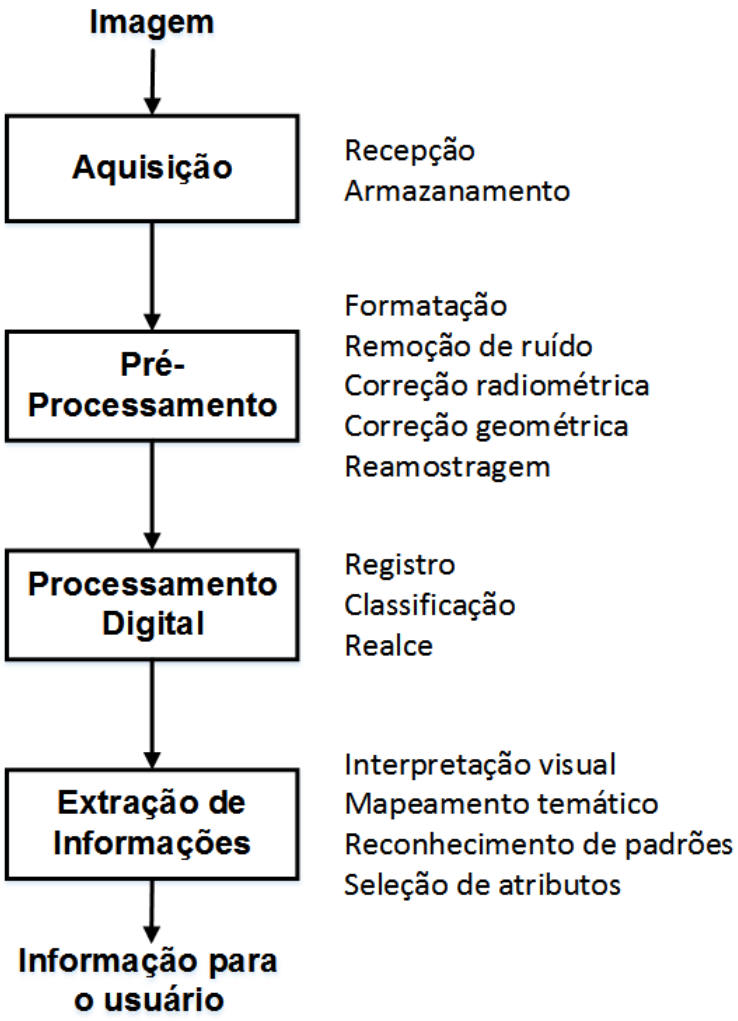


Figura 2.8 - Sequências de etapas de Processamento Digital de Imagens de Satélite. Fonte:(VIANA, 2016)

$$[Gy] = \begin{bmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{bmatrix} \tag{2.3}$$

A literatura (GONZALEZ; WOODS, 2010) apresenta algumas características importantes para o filtro de Sobel, como descritas a seguir:

- A detecção das bordas é obtida pela limiarização da magnitude dos gradientes
- A variação dos valores do limiar gera identificação de conjunto de bordas

diferentes

- Limiares muito baixo resultam em muitos pontos de bordas grossas e pontos isolados
- Limiares muito altos resultam em bordas muito finas e descontinuas

2.2.2 Imagens de Satélites

As imagens utilizadas no projeto são imagens geradas pelo satélite LandSat 8 lançado em 2013 que faz parte da série de satélites do programa LandaSat iniciado em 1972 pelos Estados Unidos.

O LandSat 8 possui as seguintes características

- órbita circular
- resolução espacial pancromática de $15m$
- resolução multiespectral de $30m$
- largura de faixa imageada $185km$
- resolução radiométrica de $16bits$
- frequência de revisita de 16 dias

2.2.3 OpenCV

O OpenCV é uma biblioteca *Open Source*, desenvolvida inicialmente pela Intel (INTEL, 2010), voltada para visão computacional e aprendizagem de máquina, com interface para C, C++, Python, Java e Matlab, e suporte para Windows, Linux, Android e Mac OS(BRADSKI; KAEHLER, 2008). A biblioteca é dividida em vários módulos, dentre eles, faz uso de bibliotecas para trabalhar com PAD, com o intuito de acelerar as funções do OpenCV, como o módulo de GPU, que explora a plataforma CUDA, desenvolvida pela NVIDIA, o módulo de OpenCL e configurar para trabalhar com OpenMP (TEAM, 2016).

3 Resultados

Os resultados da aplicação das tecnologias de processamento de alto desempenho, programação C com OpenCV, OpenMP, CUDA e OpenCL, foram obtidos utilizando o filtro Sobel no processamento de uma imagem retirada do satélite LandSat 8, com as bandas 8, 4, 3 e 2 da região do Vale do Paraíba, SP. No processamento da imagem foi utilizado o algoritmo do filtro disponibilizado nas respectivas bibliotecas de cada tecnologia. Para ampliar o caráter exploratório do trabalho, foram também utilizados a aplicação desse(s) filtros em código C sequencial, de forma a comparar os resultados e os tempos de processamento. As Figuras 3.1, 3.2, 3.3 e 3.4 mostram as imagens LandSat 8 utilizadas na aplicação das tecnologias de PAD.

Figura 3.1 - Imagem LandSat 8 utilizada na aplicação das tecnologias de PAD.

Características da Figura 3.1:

- Descrição: Recorte da região do aeroporto de São José dos Campos, SP;
- Banda: 8 Pancromática;
- Resolução Espacial: 15 metros;
- Resolução da Imagem: 400x400 pixels.

Características da Figura 3.2:

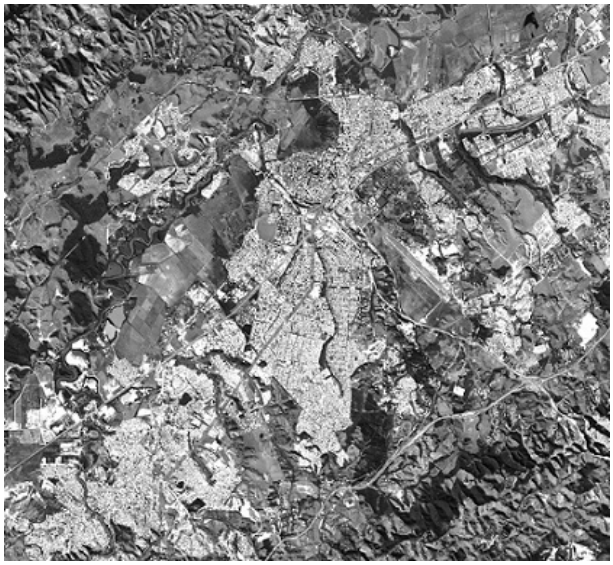


Figura 3.2 - Imagem LandSat 8 utilizada na aplicação das tecnologias de PAD.

- Descrição: Recorte da região da cidade de São José dos Campos, SP;
- Banda: 8 Pancromática;
- Resolução Espacial: 15 metros;
- Resolução da Imagem: 1449x1328 pixels.

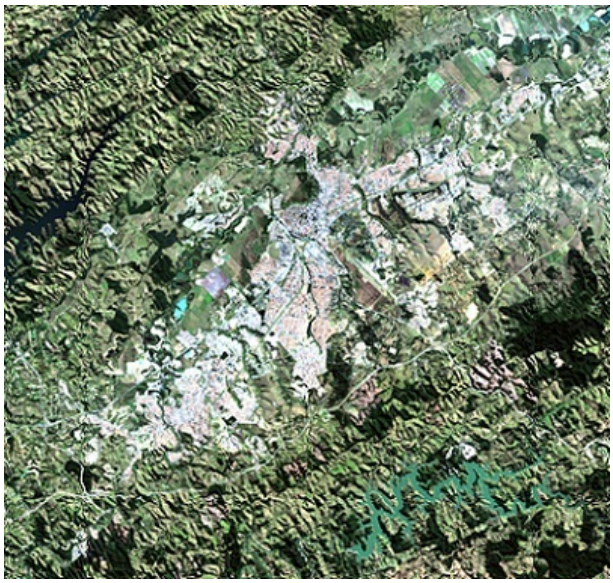


Figura 3.3 - Imagem LandSat 8 utilizada na aplicação das tecnologias de PAD.

Características da Figura 3.3:

- Descrição: Recorte da região da cidade de São José dos Campos, SP;
- Banda: Composição da 4 (Vermelho), 3 (Verde) e 2 (Azul);
- Resolução Espacial: 30 metros;
- Resolução da Imagem: 1177x1117 pixels.

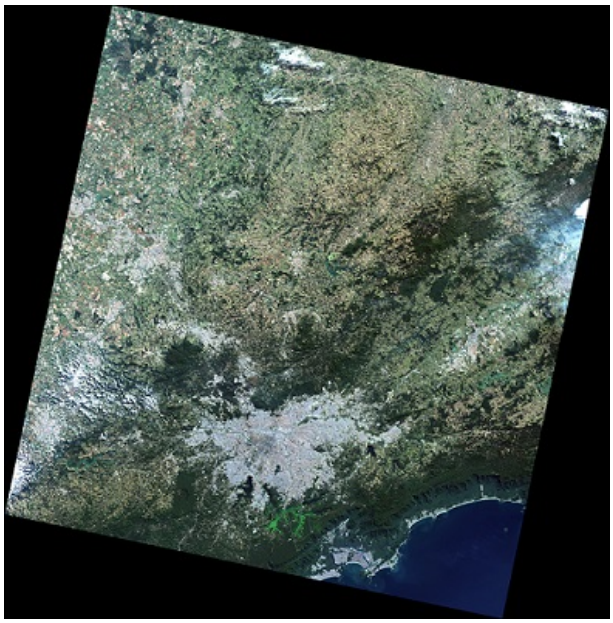


Figura 3.4 - Imagem LandSat 8 utilizada na aplicação das tecnologias de PAD.

Características da Figura 3.4:

- Descrição: Região do Vale do Paraíba, SP;
- Banda: Banda: Composição da 4 (Vermelho), 3 (Verde) e 2 (Azul);
- Resolução Espacial: 30 metros;
- Resolução da Imagem: 7621x7731 pixels;

As próximas seções apresentam o resultados obtidos com o processamento utilizando algoritmos em C padrão, e as tecnologias PAD estudadas no projeto.

3.1 Otimização do Compilador GCC

Como forma de ratificar os resultados alcançados pela aplicação dos filtros Sobel e Canny sobre as imagens do satélite Landsat 8, utilizando diversas técnicas de PAD. E como o trabalho possui caráter exploratório, foram então adaptadas para esses testes, versões desses filtros baseadas em *A*merican National Standards Institute (ANSI) C. Desta forma é possível fazer comparações com as versões desenvolvidas utilizando além do processamento paralelo, também bibliotecas com elevado grau de sofisticação como a OpenCV e Open MPI.

Os testes aqui descritos foram realizados em um computador com as seguintes especificações:

- Sistema Operacional: Ubuntu 16.04 64Bits;
- Memória RAM: 8GB;
- CPU: Intel(R) Core(TM) i7-4790 CPU @ 3.60GHz.

A versão ANSI C do filtro Sobel utilizada para esse trabalho, foi baseada no projeto do professor Wakahara[1]. Foram necessárias algumas modificações como: inclusão da detecção de bordas na direção vertical através da operação de convolução da matriz 3x3 descrita pela Equação 2.3, aplicada a imagem original; inclusão das bibliotecas time.h e float.h para o registros dos tempos de processamento.

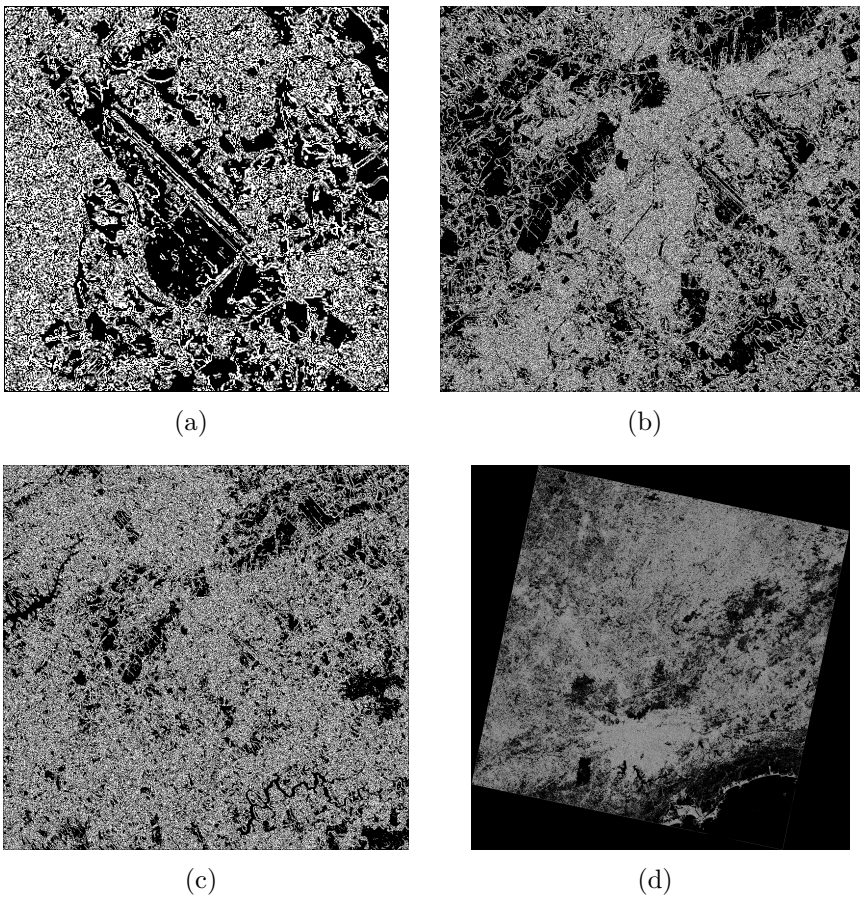
A tabela 3.1 mostra os tempos alcançados para o processamento das Figuras 3.1, 3.2, 3.3 e 3.4. Foram utilizadas a seguintes opções de compilação para definição dos níveis de otimização $O1$ e $O3$, a otimização de nível 1 entre outras coisas habilita o uso dos registradores do processador ao invés de escrever os dados na memória principal do computador. Já a otimização de nível 3 realiza a transformação do código fonte a fim de retirar todas as situações de desvios *b*ranchs, o que diminui o conflito com a arquitetura pipeline do processador. Nem sempre o maior nível de otimização reflete um maior desempenho na execução do algoritmo (SANTOSA, 2007).

A ferramenta Gprof permite a realização da operação de *p*rofiling ou seja análise do código binário do programa a fim de coletar informações de forma a identificar os tempos para cada função do programa e o número de requisições dessas funções de forma de avaliar melhor os resultados obtidos (JOAO, 2012). A Figura 3.6 apresenta

[1] Wakahara Project - Disponível em `http://cis.k.hosei.ac.jp/~wakahara/`, acesso set. 2016

Tabela 3.1 - Tempo de Processamento de CPU do filtro Sobel utilizando código do Anexo A.

	Tempos com otimização de CPU (s)			
Imagem	**O0**	**O1**	**O2**	**O3**
Figura 3.1	0,015	0,004	0,003	0,002
Figura 3.2	0,185	0,057	0,050	0,023
Figura 3.3	0,123	0,039	0,038	0,016
Figura 3.4	5,453	1,703	1,475	0,708



(a) (b)

(c) (d)

Figura 3.5 - Resultado do processamento das imagens do LandSat 8 aplicando o filtro Sobel do Anexo A. (a) Figura 3.1, (b) Figura 3.2, (c) Figura 3.3 e (d) Figura 3.4

o resultado do *p*rofiling, para o processamento das imagens com o filtro Sobel com nível 1 de otimização.

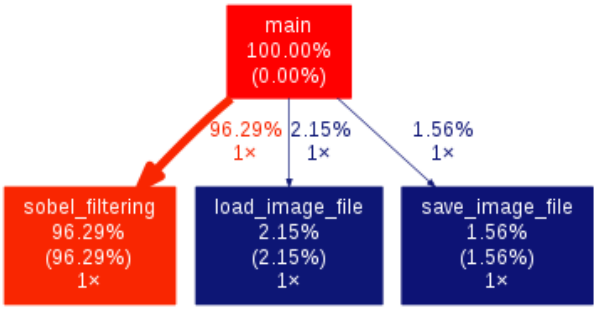


Figura 3.6 - *P*rofiling do algoritmo Sobel do Anexo A para processamento da Figura 3.4.

3.2 OpenMP

Para testar a eficiência do OpenMP com o Filtro de Sobel, faz-se uso do Speedup, que pode ser definido como a razão do tempo T gasto para executar uma tarefa em um processador e entre N processadores, como mostrado na equação:

$$S = \frac{T(1)}{T(N)}$$

Os testes foram executados em uma máquina com as seguintes especificações:

- Sistema Operacional: Ubuntu 14.04 64Bits;
- Memória RAM: 3GB;
- CPU: Intel Core i3 M370 2.4Hz.

Neste caso, a máquina possui uma CPU com 4 núcleos, que serão explorados com o OpenMP para execução do filtro em paralelo, onde a *Master Thread* irá dividir as tarefas entre as 4 *threads*.

Na Tabela 3.2 é apresentado os Speedup e o tempo médio resultante de 10 execuções em sequencial e paralelo com uso de OpenMP.

Tabela 3.2 - Tempo de processamento com uso de OpenMP

Imagem	**Tempo Sequencial (s)**	**Tempo Paralelo (s)**	**Speedup**
Figura 3.1	0,003	0,003	1,00
Figura 3.2	0,037	0,016	2,34

A implementação se encontra no Anexo B e o resultado da aplicação do filtro na

imagem do LandSat 8 é apresentado nas Figuras 3.7 e 3.8.

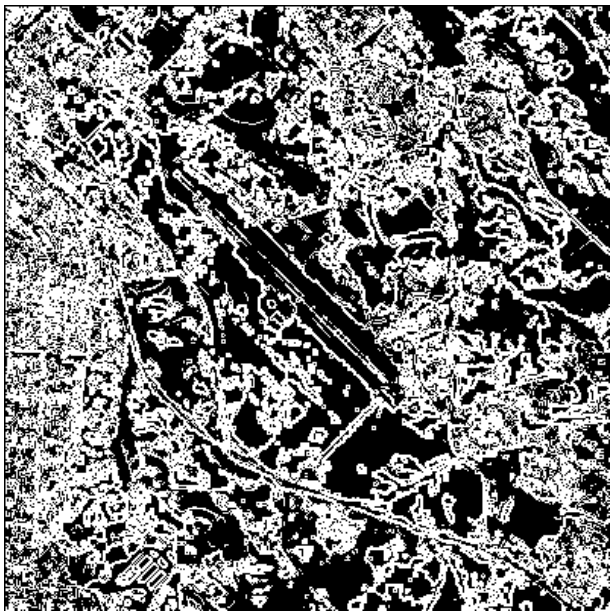


Figura 3.7 - Resultado do Filtro Sobel com OpemMP aplicado na Figura 3.1

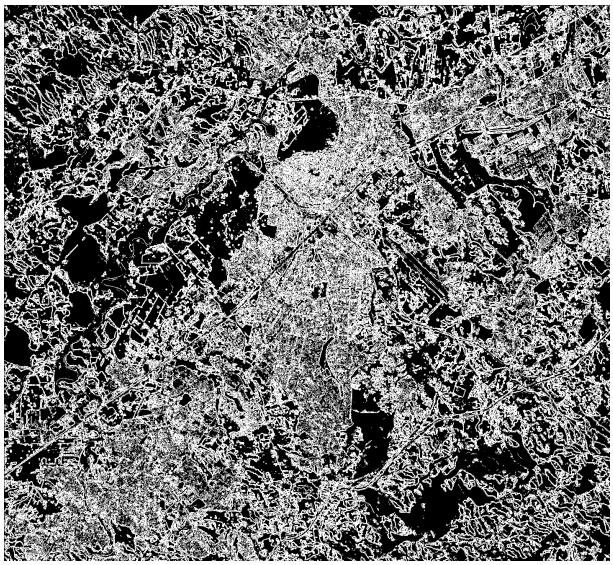


Figura 3.8 - Resultado do Filtro Sobel com OpemMP aplicado na Figura 3.2

3.3 OpenCL

Os resultados obtidos com a aplicação do OpenCL no processamento de imagens foram alcançados utilizando o filtro Sobel da biblioteca AMDAPPSDK Opencl disponibilizada de forma OpenSource pela AMD Radeon (AMD, 2016) e desenvolvido na linguagem C++. Esse filtro possibilita a aplicação de imagens de 8 bits do tipo BitMap(.bmp). Assim, como as imagens do Landsat 8 são imagens de 16 bits e do tipo Geotiff (.tif), foi realizado a conversão do número de bits dos pixels de 16 bits para 8 bits e do tipo de imagem de .tif para .bmp. Para essa conversão foi utilizado o comando *convert* do pacote de bibliotecas para manipulação de imagens ImageMagick. O comando para realizar as conversões das imagens é apesentado a seguir.

```
convert -depth 8 LO82190762016189CUB00_
B8sjc_aeroporto. tif LO82190762016189CUB00_B8sjc_aeroporto.bmp
```

O processamento da imagem foi realizado em uma máquina com as características a seguir:

- Sistema Operacional: Fedora 24
- Memória RAM: 6GB
- CPU: AMD A6-3420M 1.5GHz
- Placa de Vídeo: Radeon Graphics HD 6720G2 2GB

A execução do framework OpenCL identificou as seguintes configurações da placa de Vídeo no computador

- Platform 0 : Mesa (APU)
- Platform 1 : Advanced Micro Devices, Inc. (GPU)
- Platform 2 : Advanced Micro Devices, Inc. (CPU)
- Selected Platform Vendor : Advanced Micro Devices, Inc.
- Device 0 : AMD A6-3420M APU with Radeon(tm) HD Graphics Device ID is 0x208e160
- Executing kernel for 1 iterations

O tempo de processamento foi calculado utilizando a biblioteca time.h da biblioteca GCC com 2 núcleos da GPU. A quantidade de 2 núcleos refere-se ao máximo de processamento que a placa de vídeo suporta. Os resultados podem ser observado na tabela 3.3

Tabela 3.3 - Tempo de Processamento do filtro Sobel utilizando o CPU e a GPU para OpenCL.

Imagem	**Tempo CPU(s)**	**Tempo GPU(s)**	**Speedup**
Figura 3.1	0.035	0.007	5,33
Figura 3.2	0.271	0.057	4,77
Figura 3.4	0.374	0.080	4.65

Os resultados da Tabela 3.3 mostram que o processamento das imagens direto na GPU alacançam um desempenho bem mais relevante do que o processamento somente na CPU.

O resultado da aplicação do filtro na imagem LandSat 8 é apresentado na Figura 3.9

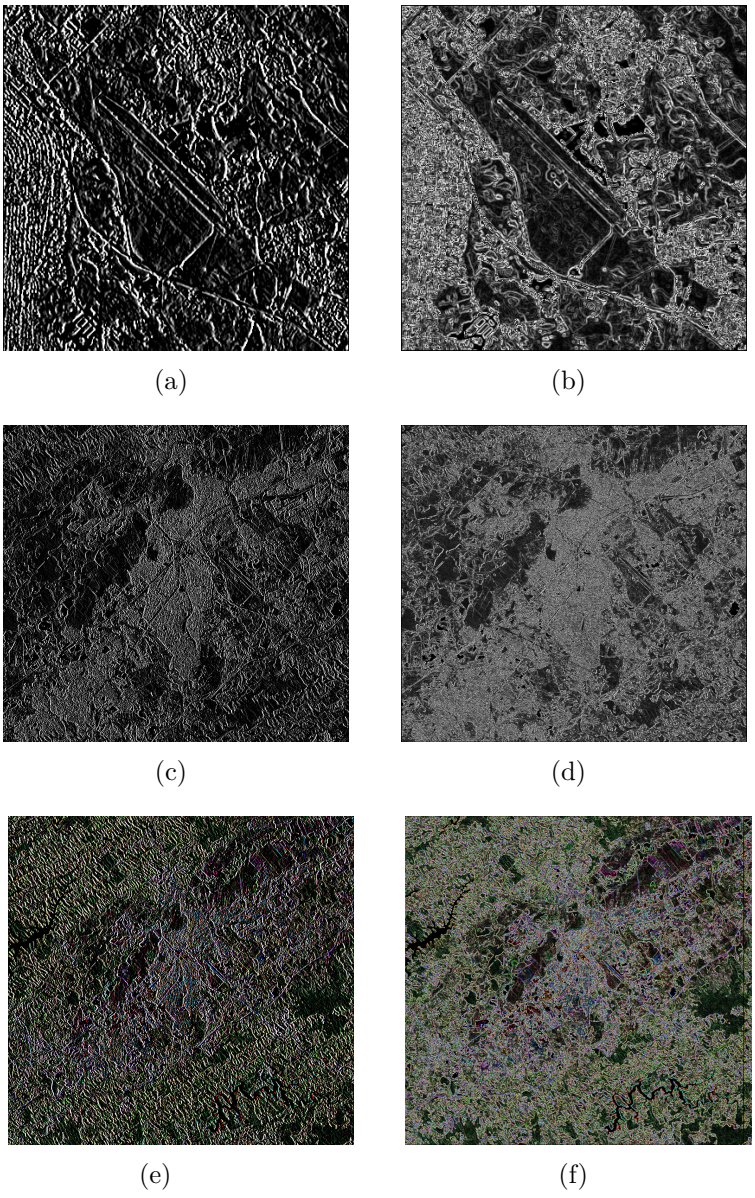


Figura 3.9 - Resultado do Processamento da LandSat 8 aplicando o filtro Sobel em (a)CPU - OpenCV e (b)GPU - OpenCL da região do aeroporto de SJC; (c)CPU - OpenCV e (c)GPU - OpenCL de SJC e (e)CPU - OpenCV e (f)GPU - OpenCL de SJC colorida

3.4 CUDA

Para os testes com CUDA, foi utilizada a biblioteca OpenCV, que disponibiliza um módulo com um conjunto de funções para utilizar a GPU como componente principal no processamento e execução do algoritmo. Uma das vantagens de utilizar o módulo GPU do OpenCV é não precisar de conhecimento na linguagem CUDA, ou seja, as funções são chamadas direto do código em C++ e a implementação interna

das funções já contém as diretivas responsáveis pela comunicação com a GPU.

Para implementar o algoritmo de testes, a IDE utilizada foi o Microsoft Visual Studio 2013 pela facilidade em reconhecer o CUDA Toolkit, disponibilizado pela NVIDIA. Para liberar os núcleos CUDA nas funções do OpenCV, foi utilizado o CMake com a opção WITH_CUDA = ON. Algumas configurações no projeto são necessárias para que o Visual Studio reconheça o OpenCV, como apontar o diretório das bibliotecas e adicionar os parâmetros no compilador.

Os testes foram executados em uma máquina com as seguintes especificações:

- Sistema Operacional: Microsoft Windows 10 64Bits
- Memória RAM: 8GB
- CPU: Intel Core i5 4690K 3.5GHz
- Placa de Vídeo: NVIDIA Geforce GTX 950 2GB GDDR5

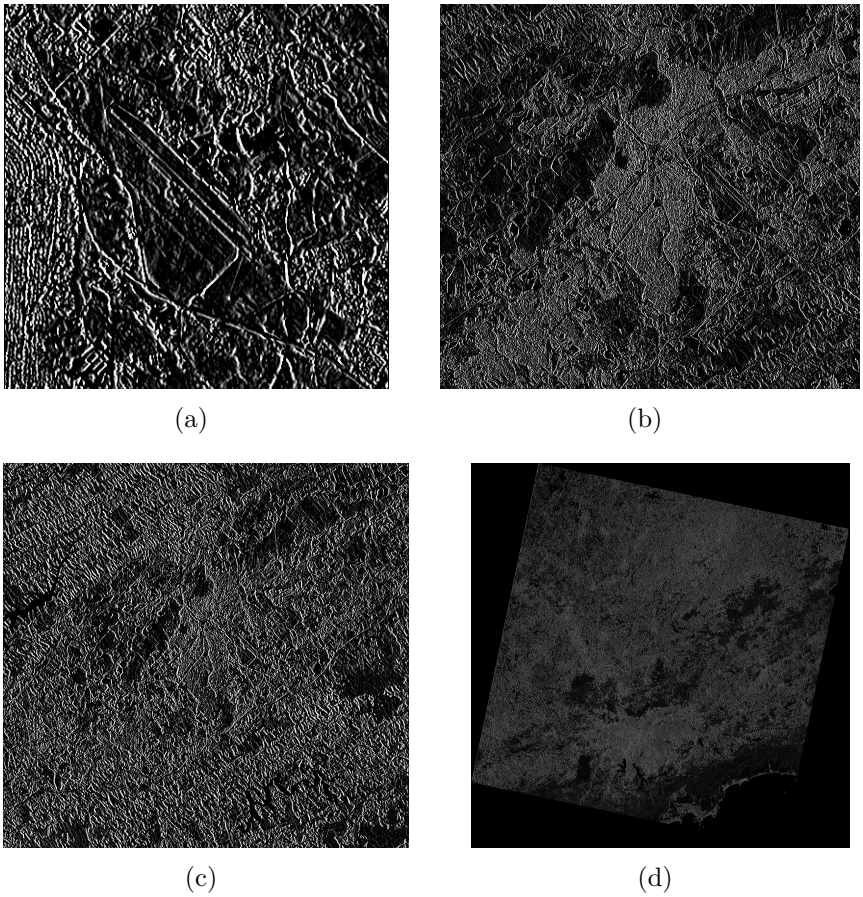
A GPU em questão possui 2GB de memória GDDR5 e 768 núcleos CUDA que trabalham em paralelo quando o algoritmo é executado. Foram testados os filtros Sobel e Canny do OpenCV, tanto utilizando a GPU quanto o CPU, nas imagens do Landsat 8 de bandas 8 e 432. A Tabela 3.4 mostra os tempos obtidos no processamento das imagens utilizando o filtro Sobel, mas devido ao desempenho satisfatório do CPU, a diferença nos tempos de execução foram pequenas. Por isso, foi executado também o filtro Canny, que é basicamente o filtro Sobel com algumas melhorias como o afinamento e rastreamento de borda. Devido a complexidade maior do Canny, ele acaba ilustrando melhor a superioridade da GPU pois exige mais desempenho do hardware, a Tabela 3.5 mostra os resultados de tempo obtidos com o filtro Canny. O tempo exibido foi o tempo médio resultante de 100 execuções para cada filtro em cada imagem, para adquirir os tempos foi utilizada a biblioteca 'time.h'.

Tabela 3.4 - Tempo de Processamento do filtro Sobel utilizando o CPU e a GPU para CUDA.

Imagem	**Tempo CPU(s)**	**Tempo GPU(s)**	**Speedup**
Figura 3.1	0,001	0,001	1,00
Figura 3.2	0,013	0,004	3,25
Figura 3.3	0,009	0,003	3,00
Figura 3.4	0,350	0,098	3,57

Tabela 3.5 - Tempo de Processamento do filtro Canny utilizando o CPU e a GPU.

Imagem	Tempo CPU(s)	Tempo GPU(s)	Speedup
Figura 3.1	0,054	0,012	4,50
Figura 3.2	0,422	0,076	5,50
Figura 3.3	0,273	0,057	4,79
Figura 3.4	10,240	1,850	5,53

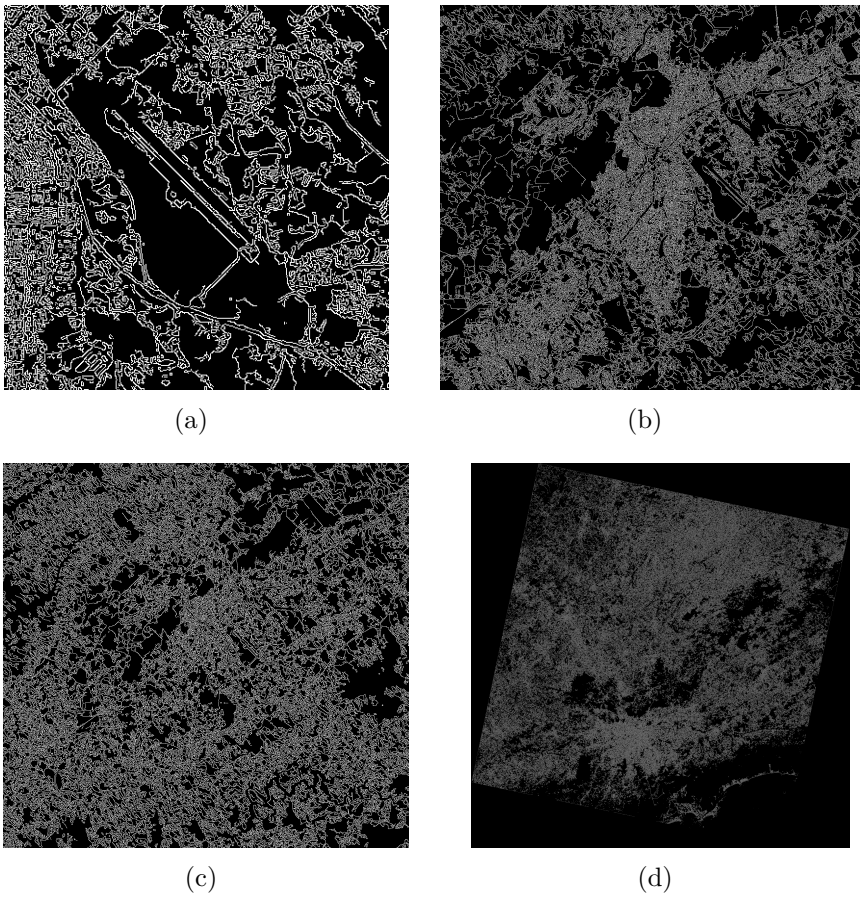


(a) (b) (c) (d)

Figura 3.10 - Resultado do processamento das imagens do LandSat 8 aplicando o filtro Sobel em GPU. (a) Figura 3.1, (b) Figura 3.2, (c) Figura 3.3 e (d) Figura 3.4

A Figura 3.10 ilustra os resultados do filtro Sobel nas imagens do LandSat 8 utilizando a GPU para fazer o processamento, enquando a Figura 3.11 mostra as mesmas imagens com o filtro Canny aplicado, também processadas pela GPU.

Os resultados mostraram um ganho de desempenho razoável quando as imagens são processadas utilizando a GPU, o que torna as GPUs opções interessantes para realizar pré-processamentos nas imagens obtidas por satélites. A grande quantidade



(a) (b) (c) (d)

Figura 3.11 - Resultado do processamento das imagens do LandSat 8 aplicando o filtro Canny em GPU. (a) Figura 3.1, (b) Figura 3.2, (c) Figura 3.3 e (d) Figura 3.4

de núcleos CUDA não demonstra inserir *overheads* de desempenho, pelo contrário, tornam a execução mais eficiente principalmente em aplicações gráficas. As GPUs atuais possuem um desempenho suficiente para aplicações em tempo real à bordo de satélites para executar processamentos dos dados antes de enviá-las para um estação em solo, diminuindo o gasto com armazenamento e transmissão dos dados.

4 Trabalhos Relacionados

Além das ferramentas citadas anteriormente, existe uma outra gama de dispositivos utilizados para garantir alto desempenho em diversas aplicações. Estes dispositivos se inserem no contexto de Computação Reconfigurável (BUELL et al., 2007), e exploram o paralelismo através de pequenos componentes eletrônicos que podem ser utilizados à bordo de missões espaciais. Esta seção trata do estado da arte envolvendo PLDs (*Programmable Logic Devices*), mais especificamente FPGAs (*Field-Programmable Gate Arrays*) e seu uso em processamento digital de imagens para propósito espacial.

4.1 *Field-Programmable Gate Array* (FPGA)

Dentro da computação reconfigurável, os PLDs de maior capacidade de processamento são as FPGAs (SKLIAROVA; FERRARI, 2003). Após diversas evoluções na tecnologia dos circuitos integrados, surgiu a necessidade de dispositivos capazes de processar e armazenar informações dentro de um mesmo *chip*. Com o surgimento dos microprocessadores e das memórias DRAM (*Dynamic Random Access Memory*) e SRAM (*Static Random Access Memory*) essa tecnologia se tornou viável e surgiram então os primeiros PLDs. Hoje, os PLDs são classificados pela arquitetura interna dos circuitos lógicos e pelo tipo de memória que eles possuem.

Em contrapartida aos PLDs, existiam também os ASICs (Application Specific Integrated Circuits), que eram utilizados pelo poder de processamento superior aos PLDs da época, mas não possuíam a capacidade de reprogramação dos PLDs, ou seja, uma vez programados nada poderia ser alterado ou adicionado posteriormente.

Para suprir esse *gap*, em 1985 a Xilinx desenvolveu a primeira FPGA, que possuía a flexibilidade de programação dos PLDs e a robustez de desempenho dos ASICs. A estrutura da FPGA consiste em um grande arranjo de células configuráveis, e cada célula tem a capacidade computacional para implementar diferentes funções lógicas, a Figura 4.1 ilustra os componentes que compõem a FPGA: Os blocos lógicos configuráveis (CLBs) que armazenam toda a lógica programável, os blocos de entrada e saída que possuem portas para o fluxo dos dados e as interconexões programáveis que relacionam os CLBs com os blocos de entrada e saída e com todos os outros CLBs.

Com os modelos que possuem memória SRAM, é possível reconfigurar a FPGA quantas vezes necessário, essa flexibilidade unida ao alto poder de processamento

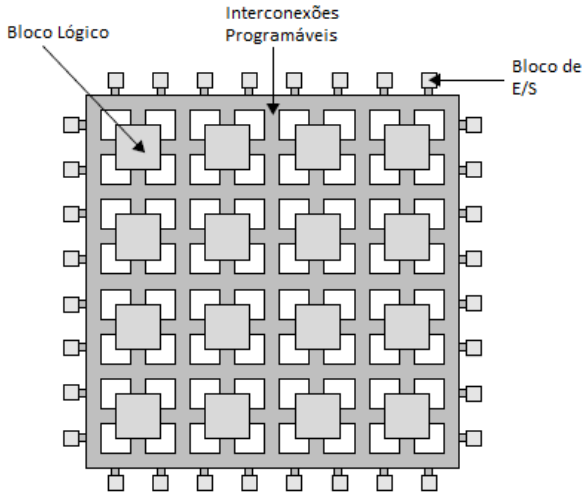


Figura 4.1 - Estrutura básica de uma FPGA.
Fonte: (ZEIDMAN, 2002)

paralelo e um custo acessível causou um grande número de vendas e uma evolução rápida das tecnologias aplicadas nas FPGAs, hoje os modelos encontrados no mercado possuem memória e núcleos de processamento embarcados em cada bloco, além de alta velocidade de entrada e saída.

Devido à todas essas características, as FPGAs são dispositivos confiáveis até para o uso em aplicações críticas, seja no ramo militar, hospitalar ou espacial. Alguns dos trabalhos citados abaixo ilustram o uso das FPGAs no contexto de processamento digital de imagens, e muitos deles para fins espaciais.

4.1.1 FPGAs no contexto de Processamento Digital de Imagens

Esta subseção resume os artigos que se destacaram pela relevância dos resultados obtidos utilizando FPGAs em diversas aplicações envolvendo processamento de imagens, incluindo filtros para detecção de bordas, análises de imagens hiperespectrais e compressão.

Em (SUDEEP; MAJUMDAR, 2011) os autores utilizam uma FPGA para implementar algoritmos de detecção de borda como Sobel, Prewitt, Robert e Compass. A FPGA recebe a imagem de entrada, aplica os métodos de detecção e gera uma imagem de resultado em tempo real dentro de uma taxa de 60 frames por segundo. O artigo descreve as características de cada método de detecção e mostra a organização dos blocos lógicos para cada implementação. Apesar de não se tratarem de imagens espaciais, este trabalho mostra o poder de processamento de uma FPGA.

Para resolver os problemas de transmissão e armazenamento dos conjuntos de imagens de satélite multi-espectrais, (FLEURY et al., 2005) implementam a transformada de Karhunen-Loève (KLT) em FPGA. O primeiro passo é utilizar a KLT para comprimir os dados com perdas, guardando apenas as imagens de ordem superior, pois nelas estão contidas as características significantes. Depois, se a imagem contém construções, é utilizada a KLT para destacar essas características. A FPGA é utilizada pois apresenta processamento paralelo de granularidade fina, adequado para processamento de imagens utilizando a KLT, enquanto processadores convencionais utilizam paralelismo de granularidade média, o que causa um gargalo computacional quando uma grande quantidade de imagens devem ser processadas. Baseado nisso, os autores demonstram que é possível utilizar essa plataforma para obter conjuntos de imagens de satélite processadas em tempo real para compressão ou extração de recursos com a KLT. Explorando a escalabilidade, é possível combinar mais FPGAs e obter um desempenho maior no processamento da transformada.

Neste trabalho, (GONZÁLEZ et al., 2012) trata do problema existente na transmissão e processamento de dados em tempo real enviados pelos satélites de sensoreamento remoto hiperespectrais. A quantidade alta de dados gerados por esses satélites gera um gargalo na largura de banda entre o satélite e a estação receptora desses dados, isso ocasiona uma limitação drástica na quantidade de dados que pode ser enviada em tempo real, que consequentemente gera uma perda de recursos e de informações relevantes. Uma das soluções trazida pelos autores é incluir algum recurso computacional à bordo do satélite para pré-processar os dados e reduzir o gargalo existente na transmissão, para isso, implementaram o algoritmo de análise de imagens hiperespectrais N-FINDR em uma FPGA, que possui capacidade computacional e flexibilidade para fazer o pré-processamento à bordo do satélite. Com a reprogramabilidade da FPGA, é possível utilizá-la para mais de um propósito dentro do satélite, reduzindo o número de recursos computacionais necessários. A Figura 4.2 ilustra as possíveis vantagens em se utilizar FPGAs à bordo de um satélite.

Após descrever o funcionamento do algoritmo N-FINDR e como ele foi implementado na FPGA, os autores concluíram que o desempenho em hardware pode superar significativamente uma versão equivalente em software além de prover resultados mais precisos com tamanhos menores. Apesar do tempo de processamento ainda estar distante de ter um desempenho de tempo real, o processamento de dado hiperespectral implementado em FPGA se mostrou promissor.

Nos trabalhos (DAWOOD et al., 2002a) (DAWOOD et al., 2002b) (WILLIAMS et al.,

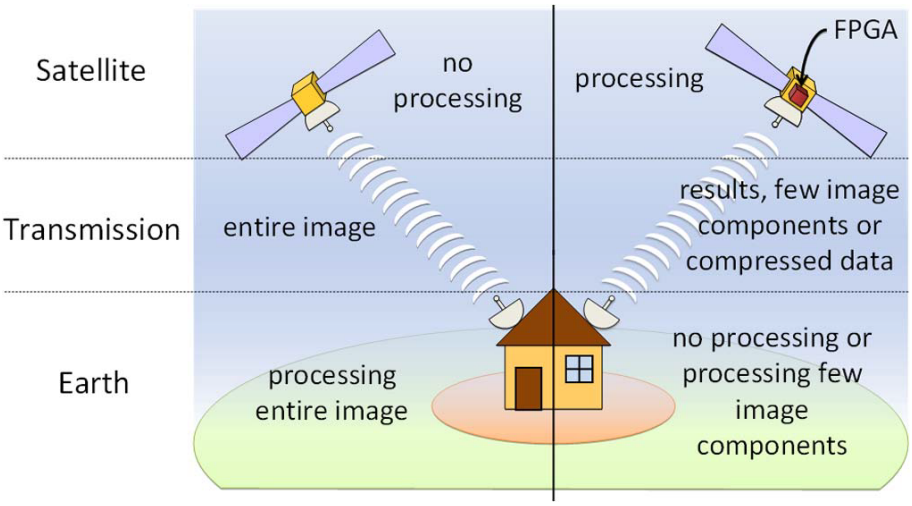


Figura 4.2 - Vantagens de usar hardware reconfigurável em processamento de dados de sensoreamento remoto.
Fonte: (GONZÁLEZ et al., 2012)

2002) os autores mostram a flexibilidade e desempenho da FPGA utilizando-a para três aplicações diferentes dentro do mesmo contexto de processamento paralelo de imagens para aplicações espaciais. A carga útil do *High Performance Computing* (HPC-I) para o satélite científico australiano FedSat foi projetada e fabricada para validar o uso de FPGAs à bordo dos satélites para diversas aplicações. A FPGA foi a melhor opção para os trabalhos devido aos fatores citados anteriormente, como alta capacidade de processamento, paralelismo nativo, flexibilidade, reconfiguração dinâmica, baixo custo entre outros. Para garantir a segurança do dispositivo e dos dados, o modelo de FPGA escolhido é *radiation-hardened*, ou seja, possui uma proteção contra radiações que possam atingir componentes eletrônicos utilizados em aplicações espaciais.

No primeiro artigo, (DAWOOD et al., 2002a) utilizam a FPGA para implementar um filtro Gaussiano e posteriormente estendê-lo para um mecanismo de convolução. As vantagens de utilizar FPGAs para este propósito vêm da facilidade de modificar parâmetros do filtro Gaussiano como o tamanho e peso mesmo após finalizar a implementação. Depois de explicar o processo de convolução, os autores avaliam o desempenho e chegam à conclusão de que a aplicação desenvolvida na FPGA possui potencial para grandes taxas de processamento, a flexibilidade e o poder de processamento de imagens das FPGAs mostram vantagens significantes em relação aos dispositivos sequenciais convencionais.

No trabalho seguinte, (DAWOOD et al., 2002b) mostram as vantagens de se comprimir

as imagens à bordo do satélite antes de enviá-las para as estações no solo. Como as imagens sensoriais geram um volume alto de dados e a capacidade de armazenamento e de comunicação são recursos caros em um satélite, a solução encontrada pelos autores foi comprimir essas imagens utilizando uma FPGA assim que adquiridas, gerando uma economia destes recursos para que possam ser usados em outras aplicações. Após definir os algoritmos de compressão e a forma com que os mesmos foram implementados na FPGA, os resultados mostraram uma boa relação entre desempenho e complexidade dos algoritmos de compressão.

Na terceira aplicação, (WILLIAMS et al., 2002) projetam e implementam um sistema de detecção de nuvens em tempo real à bordo do satélite FedSat, para reduzir o atraso existente entre a captura, análise, armazenamento e transmissão da imagem. A detecção de nuvens pode auxiliar essa redução no atraso, pois os *pixels* que forem detectados com nuvens podem ser rejeitados, ou até mesmo comprimidos pois não possuem informação relevante, tornando a quantidade de dados menor para o armazenamento e para a transmissão. Após a definição da teoria de detecção de nuvens e sua implementação em FPGA, os autores compararam os resultados obtidos entre FPGA e um microprocessador equivalente, e a conclusão foi de que mesmo utilizando um modelo de FPGA limitado, ela foi 15 vezes mais rápida do que o microprocessador, o que torna viável a aplicação de detecção de bordas em tempo real à bordo do satélite. Em (SHAN et al., 2009) os autores obtiveram resultados semelhantes no satélite CBERS-2B, que possui um módulo de detecção de nuvens projetado com FPGA.

5 Conclusões

Com o crescimento do uso de processamento de imagens para aplicações críticas, como nas áreas espacial, militar e hospitalar, o uso das tecnologias de alto desempenho permitiram expandir a demanda das aplicações e fomentar o uso das técnicas de paralelismo devido aos resultados positivos reportados pela comunidade científica.

O uso dos dispositivos de hardware de alto desempenho aliado às APIs e bibliotecas de processamento de imagens de satélite, tanto embarcado quanto em solo têm trazido resultados relevantes para as aplicações, permitindo até processamentos em tempo real.

Neste projeto foi apresentado uma aplicação da metodologia de realce, com uso do filtro Sobel para análise de imagens do satélite LandSat 8 com a utilização das API's OpenMP, CUDA, OpenCL e na programação direta na CPU. O desenvolvimento do trabalho também possibilitou um maior entendimento dos conceitos teóricos e práticos das tecnologias de PAD aplicadas na área de processamento de imagens.

Com os resultados obtidos, foi identificado um ganho de desempenho significativo entre o processamento de imagens realizado em sequencial e o obtido utilizando as tecnologias de processamento paralelo apresentadas.

REFERÊNCIAS BIBLIOGRÁFICAS

AMD. **Open Standard for Prallel Programming**. 2016. Acessado em 28 de julho de 2016. Available from: <$http://www.amd.com/pt-br/solutions/professional/hpc/opencl$>. 7, 22

BOARD, O. Openmp application program interface version 3.0. In: **The OpenMP Forum, Tech. Rep**. [S.l.: s.n.], 2008. 3

BRADSKI, G.; KAEHLER, A. **Learning OpenCV: Computer vision with the OpenCV library**. [S.l.]: "O'Reilly Media, Inc.", 2008. 13

BUELL, D.; EL-GHAZAWI, T.; GAJ, K.; KINDRATENKO, V. High-performance reconfigurable computing. **COMPUTER-IEEE COMPUTER SOCIETY-**, IEEE INSTITUTE OF ELECTRICAL AND ELECTRONICS, v. 40, n. 3, p. 23, 2007. 29

CHAPMAN, B.; JOST, G.; PAS, R. V. D. **Using OpenMP: portable shared memory parallel programming**. [S.l.]: MIT press, 2008. 3

DAWOOD, A. S.; VISSER, S.; WILLIAMS, J. Reconfigurable fpgas for real time image processing in space. In: IEEE. **Digital Signal Processing, 2002. DSP 2002. 2002 14th International Conference on**. [S.l.], 2002. v. 2, p. 845–848. 31, 32

DAWOOD, A. S.; WILLIAMS, J. A.; VISSER, S. J. On-board satellite image compression using reconfigurable fpgas. In: IEEE. **Field-Programmable Technology, 2002.(FPT). Proceedings. 2002 IEEE International Conference on**. [S.l.], 2002. p. 306–310. 31, 32

FLEURY, M.; SELF, R. P.; DOWNTON, A. C. Multi-spectral satellite image processing on a platform fpga engine. **Military and Aeronautics Logic Devices (MAPLD'05)**, p. 133, 2005. 31

GAO, W.; HUYEN, N. T. T.; LOI, H. S.; KEMAO, Q. Real-time 2D parallel windowed Fourier transform for fringe pattern analysis using Graphics Processing Unit. **Optics express**, v. 17, n. 25, p. 23147–23152, 2009. ISSN 1094-4087. Available from: <https://www.osapublishing.org/abstract.cfm?uri=oe-17-25-23147>. 8, 9

GONG, H. X.; HAO, L. Roberts edge detection algorithm based on GPU. **Journal of Chemical and Pharmaceutical Research**, v. 6, n. 7, p. 1308–1314, 2014. Available from: <http://jocpr.com/vol6-iss7.html>. 1, 5

GONZÁLEZ, C.; MOZOS, D.; RESANO, J.; PLAZA, A. Fpga implementation of the n-findr algorithm for remotely sensed hyperspectral image analysis. **IEEE transactions on geoscience and remote sensing**, IEEE, v. 50, n. 2, p. 374–388, 2012. 31, 32

GONZALEZ, R.; WOODS, R. **Processamento Digital de Imagens**. [S.l.]: Pearson, 2010. 1, 2, 11, 12

INTEL, O. **open source computer vision library, Intel Corporation**. 2010. 13

JOAO, R. S. **Introdução ao GPROF**. 2012. Available from: <http://www.ibm.com/developerworks/br/local/linux/gprof{_}introduction/>. 18

KRONOS. **Getting Familiar with GCC Parameters**. 2016. Acessado em 30 de julho de 2016. Available from: <$$https://www.khronos.org/opencl/$>. 6, 7

NERSC. **OpenMP Resources**. 2016. Acessado em 19 de agosto de 2016. Available from: <$http://www.nersc.gov/users/computational-systems/edison/programming/using-openmp/openmp-resources/$>. 4

PLAZA, A.; CHANG, C. **High Performance Computing in Remote Sensing**. [S.l.]: CRC Press, 2007. (Chapman & Hall/CRC Computer and Information Science Series). 3

SANTOSA, M. **Getting Familiar with GCC Parameters**. 2007. Available from: <http://www.onlamp.com/pub/a/onlamp/2007/04/03/getting-familiar-with-gcc-parameters.html?page=1>. 18

SHAN, N.; ZHENG, T. y.; WANG, Z. s. Onboard real-time cloud detection using reconfigurable fpgas for remote sensing. In: **2009 17th International Conference on Geoinformatics**. [S.l.: s.n.], 2009. 33

SI, X.; ZHENG, H. High Performance Remote Sensing Image Processing Using CUDA. In: **2010 Third International Symposium on Electronic Commerce and Security**. IEEE, 2010. p. 121–125. ISBN 978-1-4244-8231-3. Available from: <http:

//ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5557422>. 8, 9, 10

SKLIAROVA, I.; FERRARI, A. B. Introdução à computação reconfigurável. **Electrónica e Telecomunicações**, v. 4, n. 1, p. 103–119, 2003. 29

SUDEEP, K.; MAJUMDAR, J. A novel architecture for real time implementation of edge detectors on fpga. **International Journal of Computer Science Issues**, Citeseer, v. 8, n. 1, p. 193–202, 2011. 30

TEAM, O. D. **OpenCV 2.4. 13.0 documentation**. 2016. 13

TSE, J. **Image processing with cuda**. 1–3 p. Master Thesis (Mestrado) — University of Nevada, Las Vegas, 2012. Available from: <http://www.cs.kent.edu/{~}xchang/public{_}bak/paper/ImageProcessingwithCUDA.pdf>. 9

VIANA, J. **Processamento de Imagens - Uso de Imagens de Satélite**. 2016. Acessado em 05 de agosto de 2016. Available from: <http://www.ufrrj.br/institutos/it/de/acidentes/sr4.htm>. 12

WILLIAMS, J. A.; DAWOOD, A. S.; VISSER, S. J. Fpga-based cloud detection for real-time onboard remote sensing. In: IEEE. **Field-Programmable Technology, 2002.(FPT). Proceedings. 2002 IEEE International Conference on**. [S.l.], 2002. p. 110–116. 32, 33

YANG, L.; GUO, M. **High-Performance Computing: Paradigm and Infrastructure**. [S.l.]: Wiley, 2005. (Wiley Series on Parallel and Distributed Computing). 3

ZEIDMAN, B. **Designing with FPGAs and CPLDs**. [S.l.]: Taylor & Francis, 2002. (Computer engineering / Logic design). 30

ANEXO A - CÓDIGO FONTE - PROCESSAMENTO DO FILTRO SOBEL EM LINGUAGEM C

O algoritmo ANSI C para aplicação do filtro Sobel, foi baseado na versão disponibilizada pelo professor Wakahara (Wakahara Project - Disponível em http://cis.k.hosei.ac.jp/~wakahara/, acesso set. 2016). Para atender os objetivos previstos nesse trabalho, foi necessário refatorar o código original para a adição da operação de convolução da matriz 3x3 descrita pela Equação 2.3, o que possibilita a detecção de bordas na vertical. Além disso foi definido o valor em nível de cinza 110 como *t*hreshold para a binarização das imagens.

```
#include <stdio.h>
#include <stdlib.h>
#include <float.h>
#include <math.h>
#include<time.h>
#include "mypgm.h"

void sobel_filtering( )
     /* Spatial filtering of image data */
     /* Sobel filter (horizontal and vertical) differentiation */
     /* Input: image1[y][x] ---- Outout: image2[y][x] */
{
  /* Definition of Sobel filter in horizontal direction */
  int gx[3][3] = {{ -1,  0,  1 },
      { -2,  0,  2 },
      { -1,  0,  1 }};

  /* Definition of Sobel filter in vertical direction */
  int gy[3][3] = {{ -1,  -2,  -1 },
      { 0,  0,  0 },
      { 1,  2,  1 }};

  double px_value, py_value;
  double min, max;
  int x, y, i, j;  /* Loop variable */

  /* Maximum values calculation after filtering*/
```

```
printf("Now, filtering of input image is performed\n\n");
min = DBL_MAX;
max = -DBL_MAX;
for (y = 1; y < y_size1 - 1; y++) {
  for (x = 1; x < x_size1 - 1; x++) {
    px_value = 0.0;
    for (j = -1; j <= 1; j++) {
  for (i = -1; i <= 1; i++) {
    px_value += gx[j + 1][i + 1] * image1[y + j][x + i];
  }
    }
    if (px_value < min) min = px_value;
    if (px_value > max) max = px_value;
  }
}
if ((int)(max - min) == 0) {
  printf("Nothing exists!!!\n\n");
  exit(1);
}

/* Initialization of image2[y][x] */
x_size2 = x_size1;
y_size2 = y_size1;
for (y = 0; y < y_size2; y++) {
  for (x = 0; x < x_size2; x++) {
    image2[y][x] = 0;
  }
}
/* Generation of image2 after linear transformtion */
for (y = 1; y < y_size1 - 1; y++) {
  for (x = 1; x < x_size1 - 1; x++) {
    px_value = 0.0;
    py_value = 0.0;
    for (j = -1; j <= 1; j++) {
  for (i = -1; i <= 1; i++) {
    px_value += gx[j + 1][i + 1] * image1[y + j][x + i];
    py_value += gy[j + 1][i + 1] * image1[y + j][x + i];
```

```
    }
      }

      //Limiar:
      image2[y][x] = image2[y][x] > 110 ? 255 : 0;
    }
  }
}

int main( int argc, char *argv[ ] )
{
  if(argc < 2){
    load_image_data( );   /* Input of image1 */
  }
  else {
    load_image_file(argv[1]);   /* Input of image1 */
  }

  sobel_filtering( );   /* Sobel filter is applied to image1 */

  if(argc < 3){
    save_image_data( );   /* Output of image2 */
  }
  else {
    save_image_file(argv[2]);   /* Output of image2 */
  }
  return 0;
}
```

ANEXO B - CÓDIGO FONTE - PROCESSAMENTO DO FILTRO SOBEL COM OPENMP

Código fonte na liguagem C++ para a execução do operador Sobel em sequencial e paralelo, faz uso da biblioteca OpenCV para ler as imagens e o OpenMP utilizado para processamento paralelo.

```
#include "opencv2/imgproc/imgproc.hpp"
#include "opencv2/highgui/highgui.hpp"
#include <stdlib.h>
#include <stdio.h>
#include <iostream>
#include <chrono>
using namespace cv;

int main()
{
    // Carregar a imagem
    Mat src;
    src = imread("build-Debug/LO8219076201618 9CUB00_B8sjc.tif", 0);

    Mat out1, out2;
    Mat out3(src.rows-1, src.cols-1, src.type());
    Mat out4(src.rows-1, src.cols-1, src.type());

    if( !src.data )
    { return -1; }

    int gx, gy, mag, i, j;
    auto t3 = std::chrono::high_resolution_clock::now();

    //Operador de sobel, deteccao de borda na horizontal e vertical
    for(i=1; i < src.rows-1; i++){
        for(j=1; j < src.cols-1; j++){
            //gradiente
            gx = (src.at<char>(i-1,j+1) + 2*src.at<char>(i,j+1) +
            src.at<char>(i+1,j+1)) - (src.at<char>(i-1,j-1) +
            2*src.at<char>(i,j-1) + src.at<char>(i+1,j-1));
```

```
            gy = (src.at<char>(i+1,j+1) + 2*src.at<char>(i+1,j) +
            src.at<char>(i+1,j-1)) - (src.at<char>(i-1,j+1) +
            2*src.at<char>(i-1,j) + src.at<char>(i-1,j-1));

            mag = sqrt(pow(gx,2)+pow(gy,2));

            //limiar
            if(mag>200)
                mag=255;
            else
                mag=0;
            out3.at<char>(i,j) = mag;
        }
    }
    auto t4 = std::chrono::high_resolution_clock::now();

    //Operador de sobel Paralelo
#pragma omp parallel for private(i, j, gx, gy, mag) shared(out4, src)
    for( i=1; i < src.rows-1; i++){
        for( j=1; j < src.cols-1; j++){
            gx = (src.at<char>(i-1,j+1) + 2*src.at<char>(i,j+1) +
            src.at<char>(i+1,j+1)) - (src.at<char>(i-1,j-1) +
            2*src.at<char>(i,j-1) + src.at<char>(i+1,j-1));

            gy = (src.at<char>(i+1,j+1) + 2*src.at<char>(i+1,j) +
            src.at<char>(i+1,j-1)) - (src.at<char>(i-1,j+1) +
            2*src.at<char>(i-1,j) + src.at<char>(i-1,j-1));

            mag = sqrt(pow(gx,2)+pow(gy,2));
            if(mag>200)
                mag=255;
            else
                mag=0;
            out4.at<char>(i,j) = mag;
        }
    }
```

```
    auto t5 = std::chrono::high_resolution_clock::now();

    std::cout << "Sobel Manual: " << std::chrono::duration_cast
    <std::chrono::milliseconds>(t4-t3).count()<< " milisegundos\n";
    std::cout << "Sobel Manual OpenMP: " << std::chrono::duration_cast
    <std::chrono::milliseconds>(t5-t4).count()<< " milisegundos\n";

    imwrite("sobelB8.tif", out3);
    imwrite("sobelOmp.tif", out4);

    waitKey(0);
    return 0;
}
```

ANEXO C - CÓDIGO FONTE - PROCESSAMENTO DO FILTRO SOBEL COM CUDA

O código-fonte em C++ abaixo utiliza as funções da biblioteca de processamento de imagens do OpenCV, incluindo as funções que utilizam a GPU para fazer o processamento de alto desempenho. Para comparação dos tempos de execução, os parâmetros das funções, tanto para a CPU quanto GPU foram os mesmos.

```
#include "opencv2/imgproc/imgproc.hpp"
#include "opencv2/highgui/highgui.hpp"
#include "opencv2/gpu/gpu.hpp"
#include <stdlib.h>
#include <stdio.h>
#include <time.h>
#include <iostream>

using namespace cv;

int main(){
    Mat src;
    src = imread("LO82190762016189CUB00_B8sjc.tif", 0);
    Mat out(src.rows - 1, src.cols - 1, src.type());

    if (!src.data){
        return -1;
    }

    gpu::GpuMat d_src(src);
    gpu::GpuMat d_out;
    clock_t tStart1 = clock();
        gpu::bilateralFilter(d_src, d_out, -1, 50, 7);
        gpu::Canny(d_out, d_out, 35, 200, 3);
        gpu::Sobel(d_src, d_out, CV_8UC1, 1, 0);
    clock_t tEnd1 = clock();
    Mat out1(d_out);

    clock_t tStart2 = clock();
        bilateralFilter(src, out, -1, 50, 7);
```

```
        Canny(out, out, 35, 200, 3);
        Sobel(src, out, CV_8UC1, 1, 0);
    clock_t tEnd2 = clock();

    imshow("Entrada", src);
    imshow("BordaCPU", out);
    imshow("BordaGPU", out1);

    printf("Time taken GPU: %.2fs\n", (double)(tEnd1 - tStart1) / CLOCKS_PER_SEC);
    printf("Time taken CPU: %.2fs\n", (double)(tEnd2 - tStart2) / CLOCKS_PER_SEC);

    waitKey(0);

    return 0;
}
```

ANEXO D - CÓDIGO FONTE - PROCESSAMENTO DO FILTRO SOBEL COM OPENCL

O código OpenCL na liguagem C++ utilizado para a execução do operador Sobel, para o realce da imagem LandSat 8 utilizada no projeto, foi disponibilizado junto com a APPSDK da AMD. A seguir é apresentado as partes de construção do Filtro Sobel e da construção do Kernel para o OpenCL presentes no código.

Construção do Filtro Sobel.

```
void
SobelFilter::sobelFilterCPUReference()
{
    // x-axis gradient mask
    const int kx[][3] =
    {
        { 1, 2, 1},
        { 0, 0, 0},
        { -1,-2,-1}
    };

    // y-axis gradient mask
    const int ky[][3] =
    {
        { 1, 0, -1},
        { 2, 0, -2},
        { 1, 0, -1}
    };

    int gx = 0;
    int gy = 0;

    // pointer to input image data
    cl_uchar *ptr = (cl_uchar*)malloc(width * height * pixelSize);
    memcpy(ptr, inputImageData, width * height * pixelSize);

    // each pixel has 4 uchar components
    int w = width * 4;
```

```
int k = 1;

// apply filter on each pixel (except boundary pixels)
for(int i = 0; i < (int)(w * (height - 1)) ; i++)
{
    if(i < (k+1)*w - 4 && i >= 4 + k*w)
    {
        gx =  kx[0][0] **(ptr + i - 4 - w)
              + kx[0][1] **(ptr + i - w)
              + kx[0][2] **(ptr + i + 4 - w)
              + kx[1][0] **(ptr + i - 4)
              + kx[1][1] **(ptr + i)
              + kx[1][2] **(ptr + i + 4)
              + kx[2][0] **(ptr + i - 4 + w)
              + kx[2][1] **(ptr + i + w)
              + kx[2][2] **(ptr + i + 4 + w);


        gy =  ky[0][0] **(ptr + i - 4 - w)
              + ky[0][1] **(ptr + i - w)
              + ky[0][2] **(ptr + i + 4 - w)
              + ky[1][0] **(ptr + i - 4)
              + ky[1][1] **(ptr + i)
              + ky[1][2] **(ptr + i + 4)
              + ky[2][0] **(ptr + i - 4 + w)
              + ky[2][1] **(ptr + i + w)
              + ky[2][2] **(ptr + i + 4 + w);

        float gx2 = pow((float)gx, 2);
        float gy2 = pow((float)gy, 2);


        *(verificationOutput + i) = (cl_uchar)(sqrt(gx2 + gy2) / 2.0);
    }

    // if reached at the end of its row then incr k
```

```
        if(i == (k + 1) * w - 5)
        {
            k++;
        }
    }

    free(ptr);
}
```

Criação do Kernel do OpenCl

```
__kernel void sobel_filter(__global uchar4* inputImage,
   __global uchar4* outputImage)
{
        uint x = get_global_id(0);
    uint y = get_global_id(1);

        uint width = get_global_size(0);
        uint height = get_global_size(1);

        float4 Gx = (float4)(0);
        float4 Gy = Gx;

        int c = x + y * width;


        /* Read each texel component and calculate the filtered
        value using neighbouring texel components */
        if( x >= 1 && x < (width-1) && y >= 1 && y < height - 1)
        {
                float4 i00 = convert_float4(inputImage[c - 1 - width]);
                float4 i10 = convert_float4(inputImage[c - width]);
                float4 i20 = convert_float4(inputImage[c + 1 - width]);
                float4 i01 = convert_float4(inputImage[c - 1]);
                float4 i11 = convert_float4(inputImage[c]);
                float4 i21 = convert_float4(inputImage[c + 1]);
```

```
                float4 i02 = convert_float4(inputImage[c - 1 + width]);
                float4 i12 = convert_float4(inputImage[c + width]);
                float4 i22 = convert_float4(inputImage[c + 1 + width]);

                Gx =   i00 + (float4)(2) * i10 + i20 - i02
                - (float4)(2) * i12 - i22;

                Gy =   i00 - i20  + (float4)(2)*i01
                - (float4)(2)*i21 + i02  -  i22;

                /* taking root of sums of squares of Gx and Gy */
                outputImage[c] = convert_uchar4(hypot(Gx, Gy)/(float4)(2));

        }

}
```