

Análise Comparativa do Desempenho de Unidades de Processamento Gráfico Nvidia e ATI através de filtros digitais de imagens

Darlisson Marinho de Jesus¹, Raimundo Correa Oliveira¹

¹ Universidade do Estado do Amazonas – UEA

Av. Darcy Vargas, 1200, Parque 10 de Novembro – 69.065.020 – Manaus – AM – Brasil

{darlisson.jesus@yahoo.com, rcorrea.oliveira@gmail.com}

Abstract. *Graphics Processing Unit (GPU) are high performance co-processors intendend, originally, to improve the use and quality of computer graphics applications. However, the GPU applications has been extended to other fields to general purpose, such as digital image processing. In this study, we compared the performance of some models of GPU from Nvidia and ATI manufacturers, through the implementation of the Sobel filter to edge detection and lowpass filter with the OpenCL language. The digital filters implemented in the models GPU of Nvidia, presented the best run time and data transfer rates of memory.*

Resumo. *Unidades de processamento gráfico ou GPU (do inglês, Graphics Processing Unit) são co-processadores de alto desempenho destinados inicialmente a melhorar ou prover de capacidade gráfica a um computador. Porém, as aplicações para GPUs tem sido expandida à outras áreas para fins gerais, como o processamento digital de imagens. Neste trabalho, foi comparado o desempenho da de alguns modelos de GPU dos fabricantes Nvidia e ATI, através da implementação do filtro de detecção de borda Sobel e o filtro Passa-baixa na linguagem OpenCL (do inglês, Open Computing Language). Os filtros digitais executados nos modelos de GPU da Nvidia apresentaram os melhores de tempo de execução e taxas de transferência de dados da memória.*

1. Introdução

Segundo Zhang, [17], nos últimos anos, modernas unidades de processamento gráfico têm sido amplamente adotadas em áreas de computação de alto desempenho para resolver problemas de cálculo em grande escala. Tais problemas são encontrados na área de processamento de digital de imagem, onde muitos algoritmos são computacionalmente caros, mas paralelizáveis [4]. Além disso, os métodos tradicionais de processamento não conseguem satisfazer a exigência de tempo real para o processamento de imagem com grandes dimensões. Uma possibilidade para otimizar esse tempo é utilizar a programação para GPU, denominado GPGPU (do inglês, *General Purpose Graphics Processing Unit*). No trabalho de Nan Zhang, [16], os resultados mostraram que algoritmos paralelos para o processamento de imagens implementados em GPU, podem alcançar notável aumento de velocidade, em comparação com métodos sequenciais baseados na CPU (do inglês, *Central Processing Unit*).

Além disso, como resultado da recente evolução dos processadores gráficos e ferramentas para sua programação, pesquisadores, cientistas e desenvolvedores, tornaram-se os maiores interessados em usufruir do poder destes processadores, principalmente no contexto da computação científica. Contudo, surge a necessidade de comparar o desempenho computacional das GPUs dos diferentes fabricantes. Uma vez que, tais informações podem auxiliar engenheiros e cientistas na busca do melhor desempenho de suas soluções, aliado ao menor custo das tecnologias com o passar do tempo. A Nvidia e a ATI, são as duas principais fabricantes das GPUs modernas. Essas empresas desenvolvem também, as tecnologias utilizadas para atender a demanda de desenvolvimento de aplicações de propósito geral com as GPUs.

A primeira, o CUDA (do inglês, *Compute Unified Device Architecture*) [12], foi desenvolvida pela pioneira NVIDIA e permite aumentos significativos de performance computacional ao aproveitar a potência da unidade de processamento gráfico (GPU). Mas, apesar de ser a tecnologia mais utilizada no mercado, ela é restrita para as placas gráficas fabricadas pela Nvidia, não permitindo a portabilidade das aplicações para GPUs de outros fabricantes.

A outra tecnologia é o OpenCL [6], que é um padrão aberto para programação paralela de propósito geral em CPUs, GPUs e outros processadores. Ele consiste de uma API para coordenar computação paralela entre processadores heterogêneos e uma linguagem de programação multi-plataforma baseada na C99. O OpenCL é gerido pelo consórcio tecnológico Khronos Group e entre os desenvolvedores estão a Intel, IBM, Apple, AMD, Nvidia etc. A portabilidade da aplicação é um dos requisitos obrigatórios para que possamos comparar o desempenho de GPUs de diferentes fabricantes, por isso, usaremos o OpenCL como recurso para implementar o filtro detector de bordas Sobel e o filtro Passa-baixa, pois com o OpenCL é possível escrever programas para dispositivos diferentes. O principal objetivo deste trabalho é comparar o desempenho das Unidades de Processamento Gráfico das fabricantes NVIDIA e ATI, através do Processamento Digital de Imagens com os filtros Passa-baixa e o filtro Sobel para detecção de borda implementados na linguagem OpenCL. Assim, identificaremos a GPU que apresenta o melhor desempenho quando se utiliza a linguagem OpenCL como recurso computacional.

Nosso trabalho faz contribuições interessantes, pois até o presente momento não encontramos registros de trabalhos que fazem uma análise comparativa do desempenho das placas gráficas da Nvidia e ATI, usando filtros para o processamento digital de imagens, escritos com o OpenCL. Além disso, este trabalho contém implementações de filtros digitais para imagens em GPU usando o OpenCL.

Este trabalho está organizado nas seguintes seções: *Trabalhos Relacionados*, onde apresentamos os trabalhos que estão relacionados ao nosso, enfatizando os resultados e metodologia de cada um. Na seção *Arquitetura de uma GPU*, fazemos uma breve descrição da arquitetura de uma GPU moderna, apresentamos as diferenças de projetos entre uma CPU e apresentamos as arquiteturas das GPUs utilizadas neste trabalho. A linguagem OpenCL é descrita na seção *OpenCL*, onde explicamos o seu propósito como uma linguagem para programação paralela. A definição do filtro Sobel e o filtro Passa-baixa, implementados neste trabalho, é feita na seção *Aplicações*. Na seção *Metodologia* é apresentada os métodos utilizados para atingir

nossos resultados, assim como, as ferramentas para isso. Por fim, na seção *Resultados e Discussões* apresentamos os resultados alcançados neste trabalho.

2. Trabalhos Relacionados

Purcell [14] foi quem realizou as primeiras experiências envolvendo GPUs programáveis. Seu trabalho demonstrou que o algoritmo (um algoritmo de ray tracing) escrito em assembly para a NVIDIA GeForce 3, no processador de fragmentos, conseguia ter um desempenho maior em GPU do que na CPU. Este trabalho publicado nos primórdios da era GPGPU (do inglês, *General Purpose Graphics Processing Unit*) demonstrou o potencial do hardware gráfico programável como um dispositivo para computação genérica e paralela.

Uma comparação do desempenho do CUDA em contraste com o OpenCL foi feita por Kamrin [5], através da implementação de uma aplicação científica com elevado custo computacional, simulação Monte Carlo de um sistema de spin, para medir e compararem o tempo de transferência de dados da GPU para a CPU e vice-versa. Os tempos de execução dos kernels e os tempos de execução do ciclo de vida completo das execuções, tanto para o CUDA quanto para o OpenCL. Neste trabalho os testes restringiram-se a uma GPU da própria Nvidia e os autores concluíram que o CUDA era a melhor escolha para aplicações de alta performance.

Em [2], os autores propuseram algoritmos eficientes para o processamento linear de imagem, explorando as extensões SIMD (do inglês, *Single Instruction, Multiple Data*) fornecidas em processadores AMD e Intel. O trabalho experimental e os resultados obtidos sugeriram que implementações baseadas em OpenCL proporcionaram um rendimento mais baixo, uma média de 1,8 vezes, do que implementações equivalentes que utilizam diretamente o SIMD intrínsecos suportado pelo compilador Intel. O interessante neste trabalho é a aplicação heterogênea que se pode fazer com o OpenCL, que pode ser usado tanto para GPUs quanto para processadores multi-cores.

Uma comparação entre CPU e GPU com o filtro de detecção de borda Sobel foi feita por Zhang [16], onde os resultados experimentais indicaram que otimizações no tempo de transferência de dados entre a memória do computador e a memória da placa gráfica, podem melhorar em até 25 vezes o desempenho da implementação desse filtro para GPU.

3. Arquitetura de uma GPU

As GPUs são compostas de centenas de núcleos escalares, (cores), que executam o mesmo código através de centenas a milhares de threads concorrentes. Esta abordagem se opõe ao modelo tradicional de processadores multicore, onde algumas unidades de núcleos completos e independentes são capazes de processar threads ou processos. Estes núcleos completos, as CPUs, possuem poderosas unidades de controle e de execução capazes de executar instruções paralelas e fora de ordem, além de contarem com uma poderosa hierarquia de cache. Já as GPUs contam com unidades de controle e de execução mais simples, onde uma unidade de despacho envia apenas uma instrução para um conjunto de núcleos que a executarão em ordem.

Na Figura 1, podemos observar as diferenças entre as filosofias de projeto fundamentais nos dois tipos de processadores. Segundo David Kirk, [3], a arquitetura da CPU é otimizada para o desempenho de código sequencial. Ela utiliza uma lógica sofisticada para permitir que instruções de uma única thread de execução sejam executadas em paralelo ou mesmo fora de sua ordem sequencial. Enquanto que, a filosofia de projeto das GPUs é modelada pela crescente indústria de videogames que exerce uma tremenda pressão econômica para a capacidade de realizar um número maciço de cálculos de ponto flutuante por quadro. A solução que prevalece até o momento é otimizar para a vazão de execução do número maciço de threads. Outra característica importante é a hierarquia de memória. As GPUs, possuem memória global que pode ser acessada por todas as threads, porém as mais modernas já contam com caches de nível 1 e de nível 2. Além disso, outras memórias especializadas também podem ser usadas para acelerar o processamento.



Figura 1. CPUs e GPUs têm filosofias de projeto fundamentalmente diferentes.
Fonte: [3].

3.1. Arquitetura Nvidia Tesla

A arquitetura Nvidia Tesla baseia-se em uma matriz de processadores escaláveis. A Figura 2 mostra um diagrama de blocos da GPU GeForce 8800, GPU que introduziu a arquitetura Tesla em novembro de 2006, com 128 *stream processors* (SP) organizados em 16 multiprocessadores de stream (SMs), em oito unidades de processamento independentes chamado TPCs (*texture/processors cluster*). Cada SP tem uma unidade de multiplicação-adição (MAD) e uma unidade de multiplicação. Com os 128 SPS, têm-se um total de 500 gigaflops. Cada GPU atualmente possui memória do tipo DRAM GDDR (*Graphics Double Data Rate*), chamado *memória global* na Figura 2. Essas DRAMs GDDR diferem das DRAMs do sistema na placa-mãe da CPU porque são basicamente a memória do frame buffer que é usada para os gráficos. A arquitetura Tesla foi a primeira plataforma de supercomputação onipresente. A NVIDIA vendeu mais de 50 milhões de GPUs baseados em Tesla.

A GPU Nvidia GeForce 210 foi construída com essa arquitetura e é usada neste trabalho. Ela possui 2 multiprocessadores de streaming com 8 SP cada, totalizando 16 processadores de streaming.

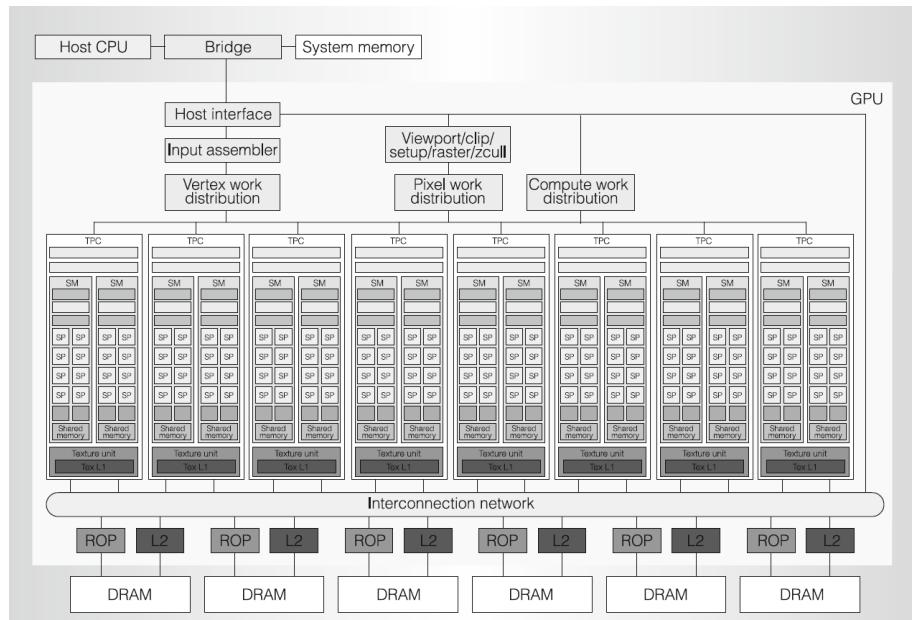


Figura 2. Arquitetura Nvidia Tesla. TPC:texture/processor cluster; SM: streaming multiprocessor; SP: stream processor; Tex: texture, ROP: raster operation processor. Fonte:([7])

3.2. Arquitetura Nvidia Fermi

Lançada em abril de 2010, esta arquitetura trouxe suporte para novas instruções para programas em C++, como alocação dinâmica de objeto e tratamento de exceções em operações de try e catch. Cada SM de um processador Fermi possui 32 CUDA cores. Até 16 operações de precisão dupla por SM podem ser executadas em cada ciclo de clock [11]. Além disso, cada SM possui:

- Dezesseis unidades de load e store, possibilitando que o endereço de fonte e destino possam ser calculados para dezesseis threads por clock.
 - Quatro Special Function Units (SFUs), que executam instruções transcendentais, como seno, cosseno, raiz quadrada, etc. Cada SFU executa uma instruções por thread por ciclo. O pipeline da SFU é desacoplado da dispatch unit, permitindo que esta possa realizar o issue (despacho) de outra instrução enquanto a SFU está ocupada.

A Figura 3 apresenta a visão geral da arquitetura Fermi. A GPU Nvidia GeForce GT 520, que é usada neste trabalho, é fabricada com a segunda geração dessa arquitetura e ela possui um total de 48 processadores de streammimg em um único multiprocessador de streaming SM.

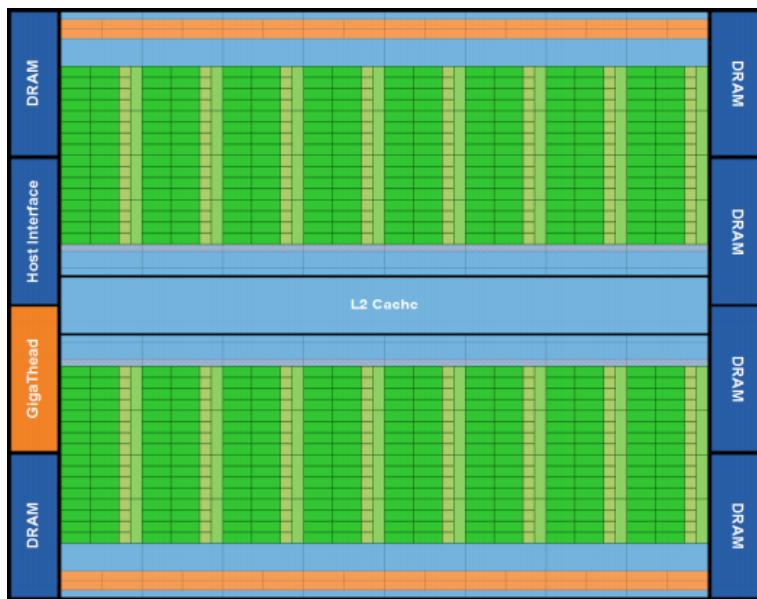


Figura 3. Visão geral da arquitetura Nvidia Fermi

3.3. Arquitetura ATI Caicos

De codinome Caicos, esta arquitetura foi lançada em 7 de fevereiro de 2011 com um único produto, a GPU Radeon HD 6450. A GPU ATI Radeon HD 6450, foi usada neste trabalho, ela possui 160 *stream cores* e 2 mecanismos SIMDs (parecido com os multiprocessadores de streaming (SM) da Nvidia), cada um com 80 stream core. Os streams cores usam a tecnologia VLIW (do inglês, *Very Long Instruction Word*), o que permite a execução de um grupo de instruções ao mesmo tempo.

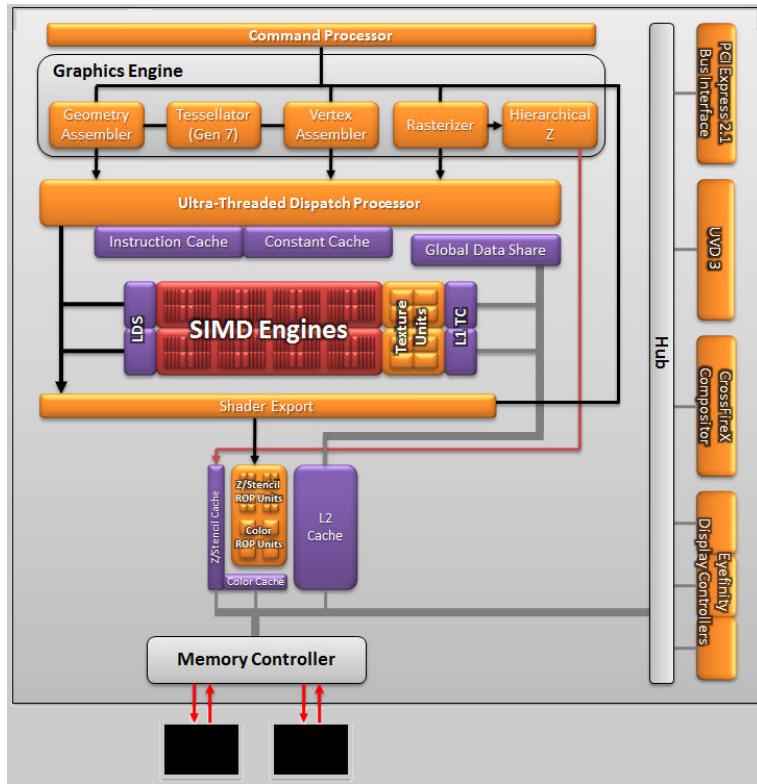


Figura 4. Visão geral da arquitetura ATI Caicos

4. OpenCL

A linguagem OpenCL é uma API (do inglês, *Application Programming Interface*) padrão para programação de computadores compostos de uma combinação de CPUs, GPUs e outros processadores. Estruturas como essas são conhecidas como sistemas heterogêneos. O OpenCL é um padrão aberto mantido pelo Khronos group e voltado para o desenvolvimento paralelo em sistemas heterogêneos contendo CPUs, GPUs e outros aceleradores [6]. O OpenCL é baseada em um subconjunto estendido do padrão ISO C99 e, portanto, é muitas vezes referido como OpenCL C.

O modelo de execução proposto pelo OpenCL se baseia em dispositivos contendo diversas unidades computacionais (*compute unit*) capazes de gerenciar grupos de threads chamados *workgroups*. Esses grupos de threads executam uma mesma rotina, chamada de *Kernel*, sobre dados diferentes, sendo um modelo de programação totalmente compatível com implementações paralelas de algoritmos para o processamento de imagens. Ao contrário de CPUs que contam com uma hierarquia multinível de cache, algumas GPUs possuem apenas uma cache de instruções e uma memória local de baixa latência compartilhada entre elementos do workgroup, que deve ser explicitamente manipulada pelo algoritmo. Por ser um framework aberto e padronizado, independente de fabricante, o OpenCL aparece como uma alternativa conveniente para sistemas computacionais que demandem desempenho e portabilidade.

Uma plataforma OpenCL é composta por um ou vários dispositivos computacionais ou (*Devices*). Um *device* pode ser uma GPU, por exemplo. Cada *device* possui

diversas unidades computacionais (*comput units*), que possuem diversos elementos de processamento (*processing elements*) (ALU, cache, memória compartilhada). Já um *kernel* é um programa escrito na linguagem OpenCL que é executado por todas as unidades computacionais ao mesmo tempo, que trata o fluxo de dados multiprocessado no modelo SIMD (do inglês, *Single Instruction, Multiple Data*). O sistema computacional responsável pela agregação, inicialização e submissão de tarefas a diferentes devices OpenCL é denominado de *Host*. *Host*, *devices*, *compute units* e *processing elements* compõem o modelo de plataforma da arquitetura OpenCL, como mostrado na Figura 5.

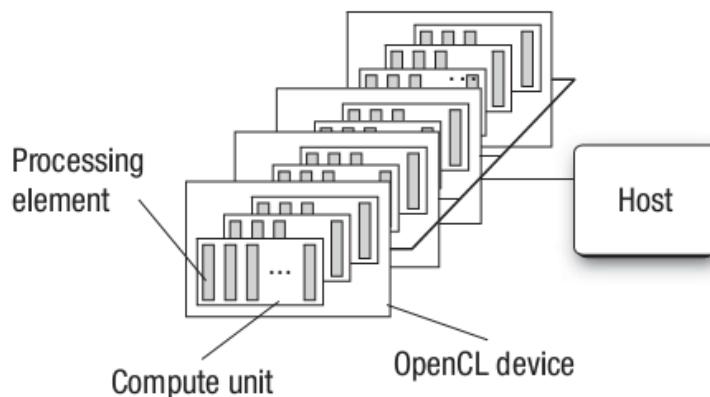


Figura 5. O modelo de plataforma OpenCL com um host e um ou mais dispositivos OpenCL. Cada dispositivo OpenCL tem uma ou mais unidades de cálculo, cada um dos quais tem um ou mais elementos de processamento [10]

5. Aplicações

5.1. Filtro Sobel para detecção de borda

O filtro Sobel é uma operação utilizada em processamento de imagem, aplicada sobretudo em algoritmos de detecção de borda, e foi proposto por Irwin Sobel em [15]. Em termos técnicos, consiste num operador que calcula diferenças finitas, dando uma aproximação do gradiente da intensidade dos pixels da imagem. O filtro Sobel calcula o gradiente da intensidade da imagem em cada ponto, dando a direcção da maior variação de claro para escuro e a quantidade de variação nessa direcção. Assim, obtém-se uma noção de como varia a luminosidade em cada ponto, de forma mais suave ou abrupta. Com isto consegue-se estimar a presença de uma transição claro-escuro e de qual a orientação desta. Como as variações claro-escuro intensas correspondem a fronteiras bem definidas entre objectos, consegue-se fazer a detecção de borda [4].

Matematicamente este operador utiliza duas matrizes 3x3 que são convoluídas com a imagem original para calcular aproximações das derivadas - uma para as variações horizontais G_x e uma para as verticais G_y . Sendo A a imagem inicial então, G_x e G_y serão duas imagens que em cada ponto contêm uma aproximação

às derivadas horizontal e vertical de A .

$$Gx = \begin{bmatrix} -1 & 0 & +1 \\ -2 & 0 & +2 \\ -1 & 0 & +1 \end{bmatrix} * A \quad Gy = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ +1 & +2 & +1 \end{bmatrix} * A$$

A magnitude do gradiente é dado por:

$$|G| = \sqrt{Gx^2 + Gy^2}$$

O código fonte em OpenCL encontra-se no Apêndice A.

5.2. Filtro Passa-baixa

Um filtro do tipo passa-baixa deixa passar as baixas frequências e elimina os valores relacionados às altas frequências. Portanto, o efeito deste filtro é o de suavizar a imagem, uma vez que as altas frequências que correspondem às transições abruptas são atenuadas. A suavização tende pelo mesmo motivo, diminuir o ruído em imagens. [4]

Neste trabalho foi implementado o filtro passa-baixa ideal que opera no domínio da frequência. E para isso, foi necessário implementar o algoritmo que computa a Transformada Rápida de Fourier, uma vez que a idéia básica dos filtros no domínio da frequência está em computar a Tranformada de Fourier da imagem a ser filtrada, multiplicar este resultado pela função de transferência do filtro e extrair a Inversa da Transformada de Fourier do resultado

De acordo com Vieira, [8], sendo $F(u, v)$ a transformada de Fourier da imagem a ser processada e sendo $G(u, v)$ a transformada de Fourier da imagem que se deseja obter à saída (com os componentes de alta frequência atenuados), a filtragem passa-baixas consiste em encontrar um $H(u, v)$ tal que: $G(u, v) = F(u, v)H(u, v)$

Filtro passa-baixas ideal - Segundo [4], um filtro passa-baixas 2-D ideal é aquele cuja função de transferência satisfaz a relação:

$$H(u, v) = \begin{cases} 1, & \text{se } D(u, v) \leq Do, \\ 0, & \text{se } D(u, v) > Do. \end{cases}$$

onde Do é um valor constante positivo e $D(u, v)$ é a distância do ponto (u, v) à origem do plano de frequência, que é obtido por:

$$D(u, v) = \sqrt{(u - P/2)^2 + (v - Q/2)^2}$$

onde, P e Q são, respectivamente, a largura e a altura da imagem. Quanto menor o raio Do , menor a frequência de corte e, portanto, maior o grau de borrimento da imagem resultante. Em nosso trabalho definimos Do como sendo $Do = DimensoDaImagen/8$, esse valor foi o que melhor se encaixou em nossos testes. O código fonte em OpenCL encontra-se no Apêndice A.

6. Objetivo Geral

Comparar o desempenho das Unidades de Processamento Gráfico das fabricantes NVIDIA e ATI, através do Processamento Digital de Imagens com os filtros Passa-baixa e o filtro Sobel para detecção de borda implementados na linguagem OpenCL. Assim, identificaremos qual GPU apresenta o melhor desempenho quando se utiliza a linguagem OpenCL como recurso computacional para a implementação destes filtros.

6.1. Objetivos Específicos

- Determinar os indicadores de desempenho que permitam avaliar as rotinas destes filtros no contexto das Unidades de Processamento Gráfico;
- Avaliar a arquitetura das GPUs da Nvidia e ATI, buscando identificar as diferenças que podem afetar no desempenho das implementações;
- Implementar o algoritmo do filtro Passa-baixa na linguagem OpenCL e obter os dados de desempenho das GPUs Nvidia e ATI;
- Implementar o algoritmo do filtro para detecção de borda Sobel na linguagem OpenCL e obter dados de desempenho das GPU Nvidia e ATI;

7. Metodologia

Nesta seção será apresentada a metodologia adotada nesse trabalho para obter os resultados finais, isto é, o resultado da avaliação de desempenho das Unidades de Processamento Gráfico Nvidia e ATI.

7.1. Métodos

A pesquisa, quanto aos objetivos, conduz a uma experimentação, desta forma, caracterizando-se como pesquisa experimental, assim, foi realizado um estudo da especificação do OpenCL 1.1 [6] e dos principais conceitos de filtragem digital, tanto no domínio do tempo, quanto no domínio da frequência. Os filtros implementados neste trabalho, o filtro Passa-baixa, o filtro Sobel para detecção de borda e a rotina da Transformada Rápida de Fourier, foram embassados no livro do Rafael Gonzalez [4].

A etapa seguinte, consistiu em montar o hardware necessário, sobre o qual foram realizados os testes de desempenho. Foram utilizados 3 computadores, com as mesmas configurações de processador, memória, placa-mãe, etc; e cada um, desses computadores, foi equipado com as placas gráficas: Nvidia Geforce GT 520, Nvidia Geforce 210 e ATI Radeon HD 6450. As aplicações foram implementados na linguagem C e OpenCL, com o Microsoft Visual Studio 2010 professional e o kit de desenvolvimento disponibilizadas por cada um dos fabricantes das placas.

7.1.1. Dados de entrada

Nossos dados de entrada consistiram de um total de 32 imagens no formato *PGM* (do inglês, *Portable Gray Map*), divididas em 8 amostras para cada uma das seguintes dimensões (em pixels): 256x256, 512x512, 1024x1024 e 2048x2048. Estas imagens foram obtidas no banco de dados de imagens proposto em [9]. E teve como objetivo avaliar o desempenho das GPUs com os filtros propostos na seção 5.1 e 5.2, sob o ponto de vista do tempo médio de execução do *kernel* (programa executado exclusivamente pela GPU) e as taxas média de transferências da memória, tanto da memória do Host para a memória do device quanto da memória do device para a memória do Host.

7.1.2. Coleta dos dados

A coleta de dados consistiu em submeter as amostras aos Filtros Sobel e Passa-baixa, em cada uma das placas gráfica utilizadas neste trabalho e coletar essas métricas com as ferramentas descritas abaixo:

Nvidia Nsight 3.0 - De acordo com o manual [13], trata-se de um ambiente de desenvolvimento para aplicações CUDA e aplicações gráficas que são executadas em GPUs da NVIDIA. O Nvidia Nsight 3.0, permite também, a depuração e análise de desempenho da aplicações escritas em OpenCL.

Os dados coletados para este trabalho, foram obtidos a partir de relatórios gerados manualmente, no formato CSV (do inglês, *Comma Separated Values*), uma vez que o Nvidia Nsight somente gera relatórios automáticos em um formato proprietário que somente podem ser lidos com Visual Studio. O Nvidia Nsight está na versão 3.0 e está disponível para o Windows como uma extensão para o Visual Studio no Windows e para o Linux, como um plugin para o Eclipse IDE.

AMD CodeXL 1.2 - O manual do AMD CodeXL, o descreve como um conjunto de ferramentas que permite aos desenvolvedores aproveitarem os benefícios das CPUs, GPUs e APUs (do inglês, *Accelerated processing unit*) da AMD, ajudando-os a identificar erros de programação e problemas de desempenho em sua aplicação de forma rápida e fácil. Permite aos desenvolvedores, depurar, realizar profile e análise estática dos códigos de suas aplicações sobre GPU, CPU e APU da AMD.

Neste trabalho, o profiler de GPU foi a principal funcionalidade usada. Esse profiler coleta e exporta automaticamente os dados de desempenho da aplicação OpenCL para relatórios em HTML (do inglês, *HyperText Markup Language*). Apesar os relatórios de tempo de execução e taxas de transferência da memória foram analisados. O AMD CodeXL está na versão 1.2 e está disponível como uma extensão para o Visual Studio e como um aplicativo com interface de usuário para Windows e Linux. O manual é disponibilizado por [1].

7.1.3. Análise dos dados

O formato dos relatórios gerados pelas ferramentas de profile dificultavam a recuperação e organização dos dados, e devido a isso, foi desenvolvido uma ferramenta para automatizar a recuperação e consolidação desses dados. Os dados das métricas de desempenho, tempo de execução e taxas de transferência da memória, foram agrupados pela dimensão das amostras (256x256, 512x512, 1024x1024 e 2048x2048, em pixels), ou seja, foram 8 valores de tempo execução para as imagens de 256x256 pixels e assim por diante para as outras dimensões e métricas.

Tempo Médio de Execução do Kernel - O tempo de execução do kernel da aplicação, corresponde ao tempo, em milisegundos, em que essa rotina permanece em execução pela GPU. No caso da aplicação do filtro Passa-baixa, que é formado por vários kernels, o tempo de execução corresponde a soma dos tempos de execução individual de cada kernel.

O tempo médio de execução foi calculado aplicando-se o cálculo da média aritmética sobre o conjunto de dados de tempo de execução, obtidos pelas ferramentas de profile e descritos na [seção 7.1.2](#).

Taxa Média de Transferência Dados da Memória - A taxa de transferência de dados corresponde ao número de bytes por unidade de tempo transmitidos entre a memória do computador e a memória da placa gráfica de forma bi-direcional. A amostra com o maior tamanho, possuía 4,194,344 bytes, e a menor, 65,551 bytes. A taxa média de transferência de dados da memória foi calculado aplicando-se o cálculo da média aritmética sobre o conjunto de dados de taxas de transferência, obtidos pelas ferramentas de profile descritas na [seção 7.1.2](#).

7.2. Materiais

Os seguintes equipamentos foram utilizados para execução dos programas em nossos experimentos:

- Três PC/x64 com Sistema Operacional Microsoft Windows 7 Professional 64 Bits, Processador *Intel® Core™ 2 Duo E7400* 2.80GHz e 4 GB de memória RAM.
- Três Unidades de processamento gráfico. A Tabela 1 apresenta as especificações de hardware detalhados dessas GPUs.

As seguintes ferramentas foram utilizadas para compilação dos programas:

- Microsoft Visual Studio 2010 versão 10.0.30319.1 RTMRel
- Microsoft .NET Framework versão 4.5.50709 RTMRel

Modelo	Geforce GT 520	Geforce 210	Radeon HD 6450
Processadores de Stream	48	16	160
Clock do processador	810 MHz	589 MHz	750 MHz
Arquitetura da GPU	Tesla	Fermi	Caicos
Memória	—	—	—
Clock da memória	900 MHz	533 MHz	1066 MHz
Tamanho da memória	1024 MB	512 MB	1024 MB
Interface da memória	64-bit	64-bit	64-bit
Largura de Banda (GB/sec)	14.4	8.0	8.5
Tipo de memória	DDR3	DDR3	DDR3

Tabela 1. Resumo das especificações das GPUs Nvidia e ATI

- OpenCL-GPU: Pacote *AMD APP SDK* versão 2.8.1 e driver de vídeo *ATI Catalyst* versão 12.104 no Windows 7 32 bits.
- OpenCL-GPU: Pacote *Nvidia Cuda Toolkit* versão 5.0 e driver de vídeo versão 320.18 no windows 7 64 bits.

As seguintes ferramentas foram utilizadas para coleta das métricas de desempenho dos programas e análise:

- Nvidia Nsight Visual Studio Edition versão 3.0
- AMD CodeXL versão 1.2
- Software R versão 3.0.1

8. Resultados e Discussões

8.1. Filtro Sobel

Após aplicar o filtro Sobel sobre as amostras, observamos, por meio da análise visual das imagens, que não existiu diferença nos resultados gerados pelas diferentes GPUs. Na Figura (b) podemos observar as bordas da imagem como resultado da aplicação do filtro Sobel sobre a imagem da Figura (a).



(a) Imagem original

(b) Imagem após a aplicação do filtro Sobel

Figura 6. Resultado da aplicação filtro Sobel para detecção de borda

8.1.1. Tempo médio de execução

O gráfico da Figura 7 apresenta os resultados obtidos dos tempos médios de execução da aplicação do Filtro Sobel em função das dimensões das imagens.

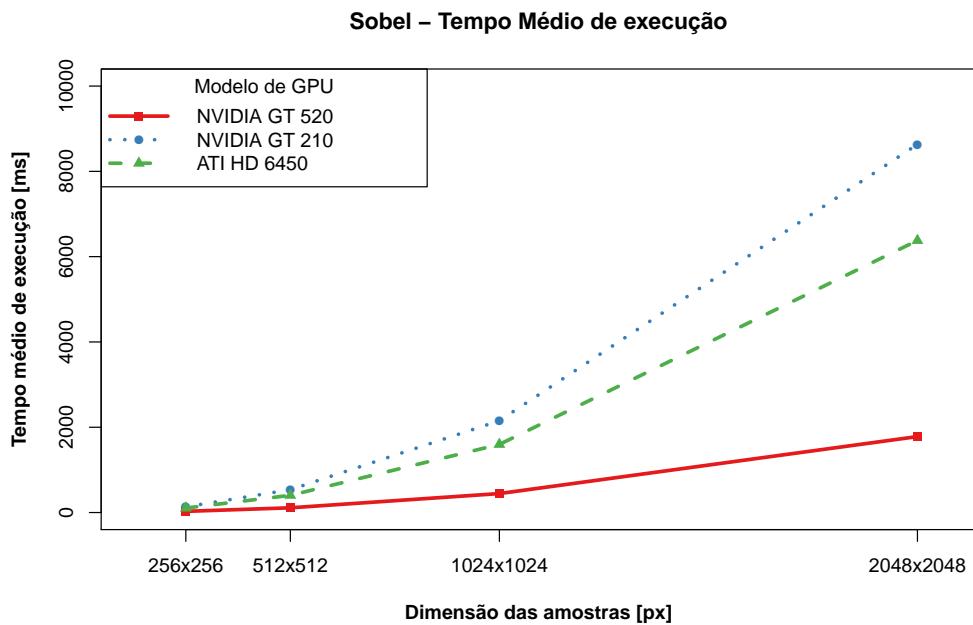
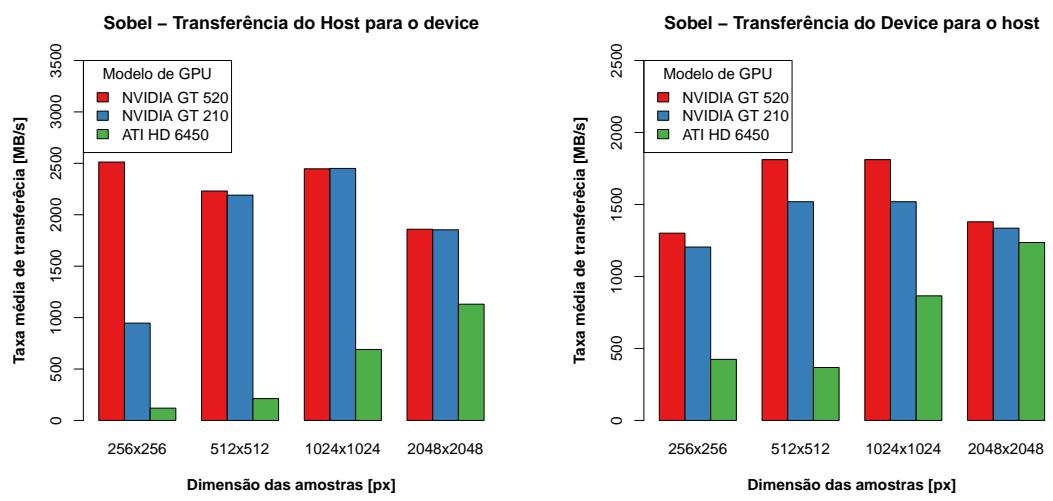


Figura 7. Sobel - Tempo médio de execução

A Nvidia Geforce GT 520 alcançou os menores tempos médios de execução e obteve o melhor desempenho. Seguido da ATI Radeon HD 6450 e da Nvidia Geforce 210.

8.1.2. Taxa média de transferência de dados da memória

Os gráficos das Figuras 8(a) e 8(b) mostram que, apesar da Nvidia Geforce 210 ser da arquitetura Tesla, ela apresentou um desempenho muito próximo ao da Nvidia Geforce GT 520, que possui é da arquitetura Fermi (sucessora da Tesla). A ATI Radeon HD 6450 foi a que teve as piores taxas de transferência, porém, é possível observar que, ao contrário das GPUs da Nvidia, as taxas de transferência da ATI Radeon HD 6450 foram aumentando a medida que o tamanho das imagens também aumentava. Isso sugere que **talvez o tamanho das amostras escolhidas não seja suficiente pra avaliar toda a capacidade dessa GPU.**



(a) Taxa média de transferência do Host para o device (b) Taxa média de transferência do Device para o host

Figura 8. Comparação do desempenho das taxas de transferência do filtro Sobel

8.2. Filtro Passa-baixa

Após aplicar o filtro Passa-baixa sobre as amostras, observamos, por meio da análise visual das imagens, que não existiu diferença nos resultados gerados pelas diferentes GPUs. Como esperado, ao se aplicar o filtro Passa-baixa sobre a imagem da Figura. 9(a) observamos na Figura 9(b) um aumento na luminosidade e a atenuação do contraste da imagem.



(a) Imagem original

(b) Imagem após a aplicação do filtro Passa-baixa

Figura 9. Resultado da aplicação do filtro Passa-baixa

8.2.1. Tempo Médio de Execução

A Figura 10 apresenta as curvas do tempo médio de execução para GPU utilizada neste trabalho. A Nvidia Geforce GT 520 obteve os melhores tempos médios de execução para o Filtro Passa-baixa. A ATI Radeon HD 6450 obteve o segundo melhor desempenho, seguido da Nvidia Geforce 210.

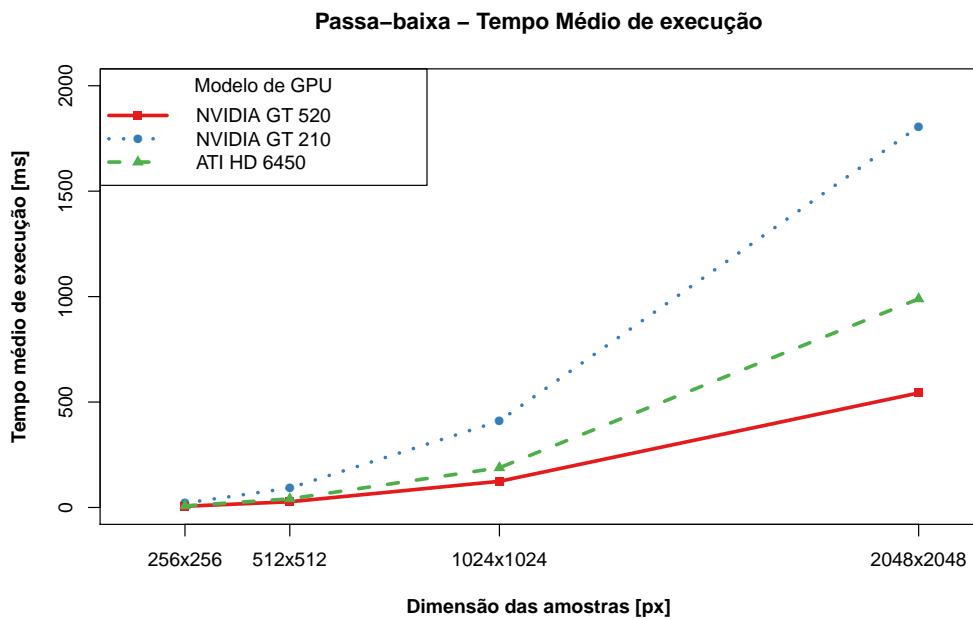
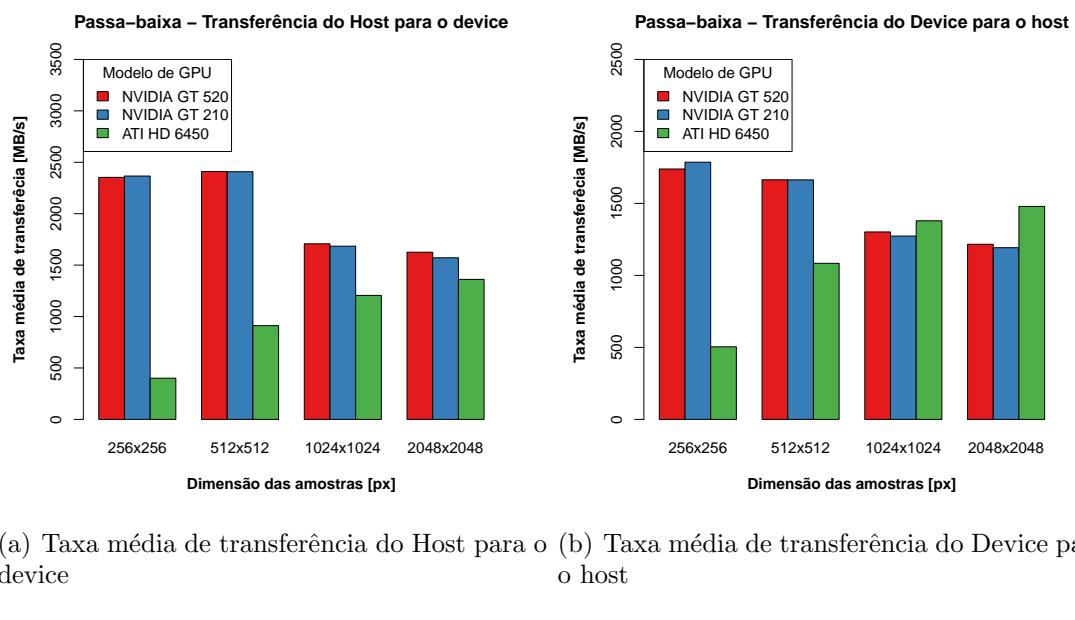


Figura 10. Passa-Baixa - Tempo médio de execução

8.2.2. Taxa média de transferência de dados da memória

Nas Figuras 11(a) e 11(b) é possível observar que as GPUs Nvidia mantiveram-se com as taxas médias de transferência muito próximas novamente. A GPU da ATI apresentou, novamente, um crescimento das taxas de transferência tão maior fosse a entrada. Sendo que na Figura 11(b) observou-se que a GPU da ATI superou as GPUs da Nvidia para as amostras de tamanho 1024 e 2096. Esse comportamento que GPU ATI apresenta, possuem similaridades com os resultados obtidos em [17], onde a GPU da ATI, obteve as melhores taxas somente para os maiores tamanhos das amostras.



(a) Taxa média de transferência do Host para o device (b) Taxa média de transferência do Device para o host

Figura 11. Comparação do desempenho das taxas de transferências do filtro Passa-baixa

Referências

- [1] AMD Developer Tools Team. CodeXL Quick Start Guide Version 1.2. http://developer.amd.com.wordpress/media/2013/07/CodeXL_Quick_Start_Guide.pdf, 2013.
- [2] S. Antao and L. Sousa. Exploiting simd extensions for linear image processing with opencl. In *Computer Design (ICCD), 2010 IEEE International Conference on*, pages 425–430, 2010.
- [3] Wen-mei W.Hwu David B. Kirk. *Programando Para Processadores Paralelos: Uma abordagem prática à programação de GPU*. Elsevier Brasil, 2011.
- [4] Rafael C. Gonzalez and Richard E. Woods. *Digital Image Processing (3rd Edition)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 2006.
- [5] Kamran Karimi, Neil G. Dickson, and Firas Hamze. A performance comparison of cuda and opencl. *CoRR*, abs/1005.2581, 2010.

- [6] Khronos OpenCL Working Group. *The OpenCL Specification, version 1.0.29*, 8 December 2008.
- [7] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym. Nvidia tesla: A unified graphics and computing architecture. *Micro, IEEE*, 28(2):39–55, 2008.
- [8] Ogê Marques Filho and Hugo Vieira Neto. *Processamento Digital de Imagens*. Editora Brasport, Rio de Janeiro, Brazil, 1999.
- [9] D. Martin, C. Fowlkes, D. Tal, and J. Malik. A database of human segmented natural images and its application to evaluating segmentation algorithms and measuring ecological statistics. In *Proc. 8th Int'l Conf. Computer Vision*, volume 2, pages 416–423, July 2001.
- [10] A. Munshi, B. Gaster, T.G. Mattson, and D. Ginsburg. *OpenCL Programming Guide*. OpenGL. Pearson Education, 2011.
- [11] Nvidia Corporation. Whitepaper nvidia’s next generation cuda compute architecture: Fermi. http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf, 2010.
- [12] NVIDIA Corporation. CUDA C Programming Guide. http://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf, 2012.
- [13] NVIDIA Corporation. NVIDIA Nsight Visual Studio Edition 3.0 User Guide. http://http://developer.nvidia.com/NsightVisualStudio/3.0/Documentation/UserGuide/HTML/Nsight_Visual_Studio_Edition_User_Guide.htm, 2013.
- [14] Timothy J. Purcell, Ian Buck, William R. Mark, and Pat Hanrahan. Ray tracing on programmable graphics hardware. *ACM Trans. Graph.*, 21(3):703–712, July 2002.
- [15] Irwin Edward Sobel. *Camera models and machine perception*. PhD thesis, Stanford University, Stanford, CA, USA, 1970. AAI7102831.
- [16] Nan Zhang, Yun shan Chen, and Jian-Li Wang. Image parallel processing based on gpu. In *Advanced Computer Control (ICACC), 2010 2nd International Conference on*, volume 3, pages 367–370, 2010.
- [17] Ying Zhang, Lu Peng, Bin Li, Jih-Kwon Peir, and Jianmin Chen. Architecture comparisons between nvidia and ati gpus: Computation parallelism and data communications. *2012 IEEE International Symposium on Workload Characterization (IISWC)*, 0:205–215, 2011.

A. Códigos Fontes em OpenCL

A.1. Filtro Sobel

Abaixo é apresentado o código fonte do *kernel* do filtro Sobel, em OpenCL.

```
1  __kernel void sobel_kernel(
2      __global const char* inputImg ,
3      __global char * outputImg ,
4      int largura)
5  {
6
7      unsigned int i = get_global_id(0);
8      unsigned int j = get_global_id(1);
9      int cont = 0;
10     char gx, gy;
11     float result = 0.0;
12
13     for (i = 0; i < largura; i++){
14         for (j = 0; j < largura; j++){
15             if (j == largura-1 || j == 0
16                 || i == 0 || i == largura-1)
17             {
18                 outputImg[cont] = (char) 0;
19             } else{
20
21                 gx = ( inputImg[(i+1)*largura + (j-1)]
22                         + 2*inputImg[(i+1)*largura + j]
23                         + inputImg[(i + 1)*largura + (j+1)])
24                         - ( inputImg[(i-1)*largura + (j-1)]
25                         + 2*inputImg[(i-1)*largura + j]
26                         + inputImg[(i-1)*largura + (j+1)]);
27
28                 gy = ( inputImg[(i-1)*largura + (j+1)]
29                         + 2*inputImg[i*largura + (j+1)]
30                         + inputImg[(i+1)*largura + (j+1)])
31                         - ( inputImg[(i-1)*largura + (j-1)]
32                         + 2*inputImg[i*largura + (j-1)]
33                         + inputImg[(i+1)*largura + (j-1)]);
34
35                 result = gx*gx + gy*gy;
36                 outputImg[cont] = sqrt (result);
37             }
38             cont++;
39         }
40     }
41 }
```

sobel_kernel.cl

A.2. Filtro Passa-baixa

Abaixo, é apresentado o *kernel* do filtro passa baixa, em OpenCL.

```
1  __kernel void passa_baixa_kernel(
2      __global float2* imagem,
3      int imgTam,
4      int raio_d)
5  {
6      unsigned int u = get_global_id(0);
7      unsigned int v = get_global_id(1);
8
9      int2 imgTam_2 = (int2)(imgTam/2, imgTam/2);
10     /* P/2 e Q/2 */
11     int2 mask = (int2)(imgTam-1, imgTam-1); /* mask pra limitar os pixels
12        0-255*/
13     int2 uv = ((int2)(u, v) + imgTam_2) & mask;
14     int2 diff = uv - imgTam_2; /* (u - P/2) e (v - Q/2) */
15     int2 diff2 = diff * diff; /* (u - P/2)^2 e (v - Q/2)^2 */
16     int dist2 = diff2.x + diff2.y; /* D(u,v) = [(u - P/2)^2 + (v - Q/2
17        )^2] */
18     int2 H_uv; /* H(u,v) */
19
20     if (dist2 < raio_d) {
21         H_uv = (int2)(-1L, -1L); /* 1 se D(u,v) < Do */
22     } else {
23         H_uv = (int2)(0L, 0L); /* 1 se D(u,v) > Do*/
24     }
25     imagem[v*imgTam+u] = as_float2(as_int2(imagem[v*imgTam+u]) & H_uv);
26 }
```

passa_baixa.cl