Victor Matheus de Araujo Oliveira

Uma Coleção de Estudos de Caso sobre o Uso da Linguagem Halide de Domínio-Específico em Processamento de Imagens e Arquiteturas Paralelas

Campinas
2013

Universidade Estadual de Campinas
Faculdade de Engenharia Elétrica e de Computação

Victor Matheus de Araujo Oliveira

Uma Coleção de Estudos de Caso sobre o Uso da Linguagem Halide de Domínio-Específico em Processamento de Imagens e Arquiteturas Paralelas

Dissertação de mestrado apresentada ao Programa de Pós-Graduação em Engenharia Elétrica da Faculdade de Engenharia Elétrica e de Computação da Universidade Estadual de Campinas para obtenção do título de Mestre em Engenharia Elétrica, na área de Engenharia de Computação.

Orientador: Prof. Dr. Roberto de Alencar Lotufo

Este exemplar corresponde à versão final da dissertação defendida pelo aluno, e orientada pelo Prof. Dr. Roberto de Alencar Lotufo

---

Campinas
2013

Ficha catalográfica
Universidade Estadual de Campinas
Biblioteca da Área de Engenharia e Arquitetura
Rose Meire da Silva - CRB 8/5974

OL4c

Oliveira, Victor Matheus de Araujo, 1988-
Uma coleção de estudos de caso sobre o uso da linguagem Halide de domínio-específico em processamento de imagens e arquiteturas paralelas / Victor Matheus de Araujo Oliveira. – Campinas, SP : [s.n.], 2013.

Orientador: Roberto de Alencar Lotufo.
Dissertação (mestrado) – Universidade Estadual de Campinas, Faculdade de Engenharia Elétrica e de Computação.

1. Processamento de imagens. 2. Compiladores. I. Lotufo, Roberto de Alencar,1955-. II. Universidade Estadual de Campinas. Faculdade de Engenharia Elétrica e de Computação. III. Título.

Informações para Biblioteca Digital

Título em outro idioma: A collection of case studies of using the Halide domain-specific language for image processing tasks in parallel architectures
Palavras-chave em inglês:
Image processing
Compilers
Área de concentração: Engenharia de Computação
Titulação: Mestre em Engenharia Elétrica
Banca examinadora:
Roberto de Alencar Lotufo [Orientador]
Helio Pedrini
José Mario De Martino
Data de defesa: 31-07-2013
Programa de Pós-Graduação: Engenharia Elétrica

COMISSÃO JULGADORA - TESE DE MESTRADO

Candidato: Victor Matheus de Araujo Oliveira

Data da Defesa: 31 de julho de 2013

Título da Tese: "Uma Coleção de Estudos de Caso sobre o Uso da Linguagem Halide de Domínio-Específico em Processamento de Imagens e Arquiteturas Paralelas"

Prof. Dr. Roberto de Alencar Lotufo (Presidente): _____

Prof. Dr. Helio Pedrini: Hélio Pedrini

Prof. Dr. José Mario De Martino: _____

Para Erika.

Agradecimentos

Agradeço,

ao Prof. Lotufo pela orientação e por me apoiar durante este trabalho mesmo nos momentos difíceis não só profissionalmente, como pessoalmente.

à minha namorada Erika, que tornou este período o melhor da minha vida.
aos meus pais, Jemoel e Laura, por terem acreditado em mim desde o início.
às pessoas do laboratório LCA pela ótima convivência.
à agência CAPES o apoio financeiro concedido durante todo o período do mestrado.

Lentamente desabrochava e amadurecia no espírito de Sidarta a percepção, o conhecimento daquilo que na verdade significava sabedoria e devia ser a meta das suas buscas prolongadas. Nada era a não ser uma predisposição da alma, a faculdade, a arte secreta de conceber, a cada instante, em plena vida, a idéia da unidade, de sentir a unidade, de encher dela os pulmões. Pouco a pouco, essa certeza crescia nele e seu reflexo aparecia no rosto velho e todavia infantil, de Vasudeva, revelando harmonia, ciência da eterna perfeição do cosmo, sorriso, unidade.

Herman Hesse — Sidarta

Resumo

Um novo desenvolvimento no campo de Linguagens de Domínio-Específico são linguagens de programação que podem funcionar tanto em CPUs multi-núcleo quanto em GPUs.

Nesta dissertação, avaliamos Halide, uma Linguagem de Domínio Específico (DSL) para processamento de imagens. Halide funciona tanto em CPUs como em GPUs e almeja ser uma forma mais simples e eficiente, em termos de desempenho, de expressar algoritmos da área do que as alternativas tradicionais.

Para mostrar o potencial e as limitações da linguagem Halide, fazemos nesta dissertação alguns estudos de caso com algoritmos que acreditamos ser bons exemplos de categorias-chave em Processamento de Imagens, especialmente em manipulação e edição de Imagens. Comparamos o desempenho e simplicidade de implementação desses problemas com implementações em C++ usando *threads* e vetorização, para arquiteturas CPU multi-núcleo, e OpenCL, para CPUs e GPUs.

Mostramos que há problemas na implementação atual de Halide e que alguns tipos de algoritmos da área não podem ser bem expressos na linguagem, o que limita a sua aplicabilidade prática. Entretanto, onde isso é possível, Halide tem performance similar à implementações em OpenCL, onde vemos que há de fato ganho em termos de produtividade do programador.

Halide é, portanto, apropriado para um grande conjunto de algoritmos usados em imagens e é um passo na direção certa para um modo de desenvolvimento mais fácil para aplicações de alto desempenho na área.

Palavras-chave: Processamento de Imagens. Compiladores.

Abstract

A development in the field of Domain-Specific Languages (DSL) are programming languages that can target both Multi-Core CPUs and accelerators like GPUs.

We use Halide, a Domain-Specific Language that is suited for Image Processing tasks and that claims to be a more simple and efficient (performance-wise) way of expressing imaging algorithms than traditional alternatives.

In order to show both potential and limitations of the Halide language, we do several case studies with algorithms we believe are representatives of key categories in today's Image Processing, specially in the area of Image Manipulation and Editing. We compare performance and simplicity of Halide implementations with multi-threaded C++ (for multi-core architectures) and OpenCL (for CPU and GPUs).

We show that there are problems in the current implementation of the DSL and that many imaging algorithms cannot be efficiently expressed in the language, which limits its practical application; Nevertheless, in the cases where it is possible, Halide has similar performance to OpenCL and is much more simple to develop for.

So we find that Halide is appropriate for a big class of image manipulation algorithms and is a step in the right direction for an easier way to use GPUs in imaging applications.

Key-words: Image Processing. Compilers.

Lista de Figuras

Lista de Tabelas

Sumário

Capítulo 1

Introdução

Processamento de Imagens é um campo em que o paralelismo tem se tornado cada vez mais importante. A grande quantidade de dados e a estrutura regular de muitos algoritmos da área, com poucas dependências de dados, facilitam o uso de múltiplos processadores simultaneamente.

Recentemente, Unidades de Processamento Gráfico (GPUs) se tornaram um recurso de computação bastante interessante para processamento de imagens. GPUs são co-processadores que possuem em geral muito mais banda de memória que CPUs e maior poder de processamento em ponto-flutuante.

Como motivação para este trabalho, temos o software de manipulação de imagens GIMP (Bunks 2000) e um problema de engenharia de software. Recentemente, o autor dessa dissertação implementou suporte à aceleração usando GPU no GIMP em um trabalho independente a esta dissertação. Como podemos ver na Figura 1.1, GPUs melhoraram consideravelmente o desempenho de vários filtros.

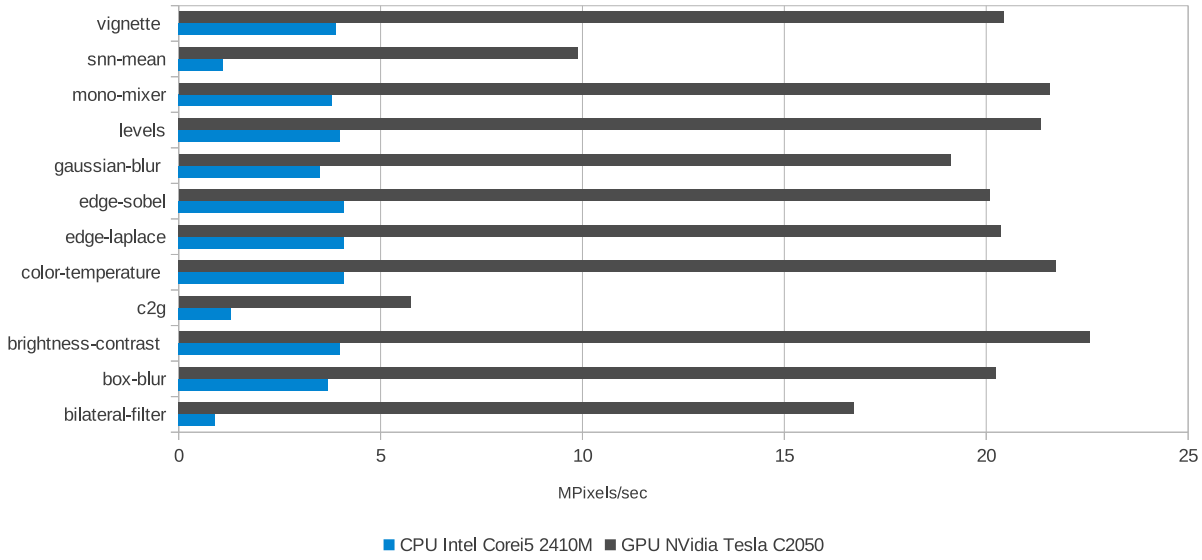


Figura 1.1: Desempenho de vários filtros em CPU e GPU no GIMP.

Felizmente, em um software de manipulação e edição de imagens, filtros geralmente são relativamente simples, pois cada pixel pode ser processado independentemente e seu valor final depende de uma janela em torno do pixel. A complexidade vem das várias combinações de filtros em um *pipeline* e do crescimento contínuo da resolução de fotografias. É esperado que o software processe imagens de mais de 15 Megapixels em tempo real, logo desempenho é essencial.

A estagnação nas frequências de *clock* em CPUs na última década fez com que avanços de desempenho nas arquiteturas de microprocessadores se dessem principalmente através de múltiplos núcleos (paralelismo) e vetorização. Temos também as GPUs, que costumavam ter muito pouca flexibilidade mas que têm aumentado gradativamente sua programabilidade (Owens, Luebke, Govindaraju, Harris, Krüger, Lefohn & Purcell 2007).

Uma das maiores consequências desta mudança é que o software tem de ser modificado para tirar proveito destas novas arquiteturas, ao contrário do passado, quando se era esperado que o desempenho de aplicações aumentaria automaticamente com o aumento das frequências de *clock* em CPUs.

Na Taxonomia de Flynn (Flynn 1972), temos que as arquiteturas GPU mais se aproximam do modelo SIMD (*Single Instruction - Multiple Data*), ou seja, temos centenas de processadores executando as mesmas instruções sobre elementos diferentes em um conjunto de dados. Em contraste, temos as arquiteturas multi-núcleo tradicionais que estão no modelo MIMD (*Multiple Instruction - Multiple Data*), em que cada núcleo pode executar instruções independentemente entre si em diferentes partes do conjunto de dados.

Temos, portanto, uma grande mudança de paradigma em termos do modelo de programação. A arquitetura massivamente paralela de GPUs faz com que algoritmos tenham de ser repensados para evitar dependências de dados e explorar paralelismo no nível do pixel.

O paralelismo massivo, o uso do modelo não-convencional SIMD, acesso não-uniforme à memória e limitações de programabilidade tornam a programação de GPUs bastante complexa.

Para utilizar GPUs em tarefas de propósito-geral, as duas linguagens mais usadas atualmente são CUDA (NVIDIA 2011) e OpenCL (Munshi, Gaster, Mattson, Fung & Ginsburg 2011). A diferença entre as duas é que a primeira é suportada apenas em GPUs da fabricante Nvidia, enquanto que a segunda é uma especificação aberta assim como o OpenGL e é suportado por diversas fabricantes; portanto, usaremos OpenCL neste trabalho sempre que possível. OpenCL começou a ser utilizado em outros tipos de dispositivos como CPUs multi-núcleo e FPGAs.

No GIMP, não podemos assumir que há uma implementação OpenCL nas máquinas de todos os usuários. A tecnologia é muito recente e nem todo hardware é suportado pela linguagem. Cada filtro possui uma implementação em C e uma em OpenCL torna-se um problema, pois temos que ter certeza que ambas implementações produzem o mesmo resultado. Manter o controle de que mudanças em um filtro são sempre reproduzidas em diferentes implementações é trabalhoso, especialmente em um projeto de software livre sem uma autoridade central, como é o caso.

Como veremos na Seção 2.2, a grande complexidade da arquitetura de GPUs é exposta em detalhes para o programador pela linguagem OpenCL. Seria possível um modo mais fácil de programar GPUs para o caso específico de filtros de processamento de imagens?

O meio que vamos analisar nesta dissertação para resolver o problema da duplicação de

código, facilidade de programação e desempenho é o uso de Linguagens de Domínio-Específico (DSLs).

A maiores vantagens potenciais de DSLs é que, como elas têm um escopo limitado a um certo tipo de problema, podem assumir hipóteses que não seriam possíveis em uma linguagem de propósito-geral. Esse conhecimento extra pode ser utilizado para efetuar otimizações mais agressivas na compilação, melhorar a portabilidade ou simplificar a tarefa de programação. Mas isso tem um preço. Esta limitação no escopo da linguagem inerente às DSLs podem fazer com que determinados algoritmos sejam difíceis – ou mesmo impossíveis – de serem implementados. Esta tensão é uma das principais escolhas de projeto no desenvolvimento de uma DSL.

Existem algumas DSLs para processamento de imagens na literatura que podem ser executadas em CPUs multi-núcleo e GPUs. Vemos no Capítulo 3 que elas possuem muitas semelhanças entre si.

Neste trabalho, com o objetivo de verificar se o uso de DSLs em processamento de imagens vale a pena, utilizamos a DSL Halide (Ragan-Kelley, Adams, Paris, Levoy, Amarasinghe & Durand 2012), que pode ser executada em CPUs e GPUs. Portanto, Halide poderia resolver os problemas mencionados anteriormente no software GIMP, entre outros.

Halide é uma linguagem funcional e sua sintaxe é projetada de modo que o paralelismo fica evidente. Ao possuir apenas um conjunto de primitivas que se dão ao paralelismo, Halide obriga o programador a escrever um código que possa ser mapeado eficientemente nas arquiteturas multi-núcleo e GPU, mas isso não elimina o esforço necessário de reorganizar um algoritmo serial para fazer uso destas primitivas.

A principal característica da linguagem Halide é que existe uma separação entre a especificação do algoritmo, que é independente da arquitetura subjacente, e a configuração de execução, que, ao contrário do anterior, depende da máquina em que o código é executado. Esta configuração de execução é chamada de *schedule*.

O schedule seria uma boa forma de rapidamente verificar diferentes possibilidades de mapeamento de um algoritmo em um determinado hardware, além de fazer uma boa separação, do ponto de vista de engenharia de software, entre a *especificação* de um algoritmo e sua *implementação*.

O artigo em que Halide foi introduzido (Ragan-Kelley et al. 2012) reinvidica que a linguagem oferece ganhos substanciais de desempenho e facilidade de uso sobre implementações tradicionais de filtros em processamento de imagens.

Pretendemos neste trabalho fazer uma análise independente da facilidade de uso e do quão rápido (ou lento) Halide é comparado com implementações cuidadosamente escritas com as ferramentas tradicionalmente usadas por programadores da área hoje em dia. Além disso, fazemos uma discussão sobre que tipo de problema é apropriado para a linguagem.

Vamos fazer uma série de estudos de caso com diferentes filtros e comparar implementações em Halide com códigos escritos em C++ usando multi-threading e tendo em mente a auto-vetorização do compilador em CPUs e OpenCL (em CPUs e GPUs).

Os seguintes estudos de caso são filtros normalmente utilizados em edição de imagem:

- Conversão de espaço de cor de RGB para CIELAB
- Motion-Blur

- Unsharped Masking
- Filtro Bilateral (Paris, Kornprobst & Tumblin 2009)

Implementamos também os seguintes algoritmos para detecção de pontos de interesse:

- Detector de Cantos de Harris & Stephens (Harris & Stephens 1988)
- SIFT (Lowe 1999)

Com estes estudos de caso, pretendemos avaliar o uso de Halide em diferentes sub-áreas de processamento de imagens e testar os limites e problemas da implementação atual do compilador Halide.

Comparações entre linguagens são um tópico subjetivo e que depende bastante da qualidade das implementações envolvidas. Nosso esforço neste trabalho é que nossas implementações sejam escritas do modo a serem eficientes mas sem que haja perda de legibilidade. Todas as implementações desta dissertação estão disponíveis como código-aberto para consulta em https://github.com/victormatheus/halide-casestudies.

Vemos em nossos estudos de caso que nem todo algoritmo de imagens pode ser implementado eficientemente em Halide e que há alguns problemas de desempenho e uso de memória na implementação atual da linguagem.

Nossos experimentos também mostram que Halide na CPU e na GPU possui desempenho similar ao OpenCL, mas que ambos foram bem mais rápidos que as implementações em C++ com *multi-threading* e auto-vetorização.

Vemos também que a quantidade de código necessária e a complexidade das implementações em Halide são inferiores às implementações em OpenCL.

Para um programador sem conhecimento prévio de desenvolvimento em GPUs, o uso de Halide e DSLs similares é capaz de facilitar o uso destes dispositivos em processamento de imagens, pois Halide abstrai muitos dos detalhes de gerenciamento de memória, configuração de execução, compilação de kernels, etc. Porém, devemos ter em mente que não há um modo automático de transformar conceitualmente um algoritmo para uma forma com paralelismo, então a maior vantagem de Halide do ponto de vista de usabilidade é permitir ao programador se concentrar mais neste tipo de tarefa algorítmica e menos em detalhes de arquitetura.

Esta dissertação está organizada do seguinte modo: no Capítulo 2 explicamos como é o modelo de programação de GPUs e a linguagem OpenCL; no Capítulo 3 fazemos uma explicação detalhada de como Halide funciona, da implementação do compilador e damos exemplos de uso da linguagem; no Capítulo 4 mostramos nossos estudo de caso e problemas encontrados nas diferentes implementações feitas; no Capítulo 5 temos os resultados de desempenho das diferentes implementações, analisamos o porque dos resultados e abordamos a complexidade das diferentes implementações; no Capítulo 6 temos nossa conclusão, em que comentamos que tipo de aplicação pode se beneficiar do uso de Halide. No apêndice A temos um exemplo de código-fonte em Halide, C++ e OpenCL.

Capítulo 2

Linguagens para programação paralela

Usaremos ao longo deste trabalho o operador de Sobel para demonstrar implementações nas diversas linguagens apresentadas. O operador de Sobel é normalmente encontrado como parte de algoritmos para cálculo de bordas em imagens pois ele computa uma aproximação da função de gradiente na imagem.

O filtro pode ser colocado na forma de uma convolução com duas máscaras $\mathbf{S}_x$ e $\mathbf{S}_y$, em que a primeira é o gradiente na direção horizontal e o segundo, na vertical.

$$\mathbf{S}_x = \begin{bmatrix} +1 & 0 & -1 \\ +2 & 0 & -2 \\ +1 & 0 & -1 \end{bmatrix} * \mathbf{I} \quad \text{e} \quad \mathbf{S}_y = \begin{bmatrix} +1 & +2 & +1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix} * \mathbf{I} \tag{2.1}$$

$\mathbf{S}_x$ e $\mathbf{S}_y$ são filtros separáveis e podem também ser colocados na seguinte forma:

$$\mathbf{S}_x = \begin{bmatrix} 1 \\ 2 \\ 1 \end{bmatrix} * \left(\begin{bmatrix} 1 & 0 & -1 \end{bmatrix} * \mathbf{I} \right) \quad \text{e} \quad \mathbf{S}_y = \begin{bmatrix} 1 \\ 0 \\ -1 \end{bmatrix} * \left(\begin{bmatrix} 1 & 2 & 1 \end{bmatrix} * \mathbf{I} \right) \tag{2.2}$$

Para encontrar a direção do vetor gradiente, basta fazermos:

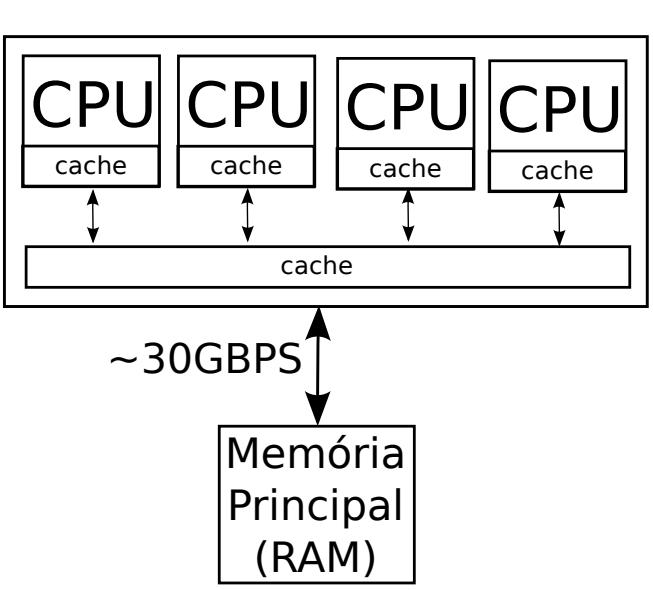
$$\langle \mathbf{S} \rangle = arctan(\mathbf{S}_x / \mathbf{S}_y) \tag{2.3}$$

E seu módulo:

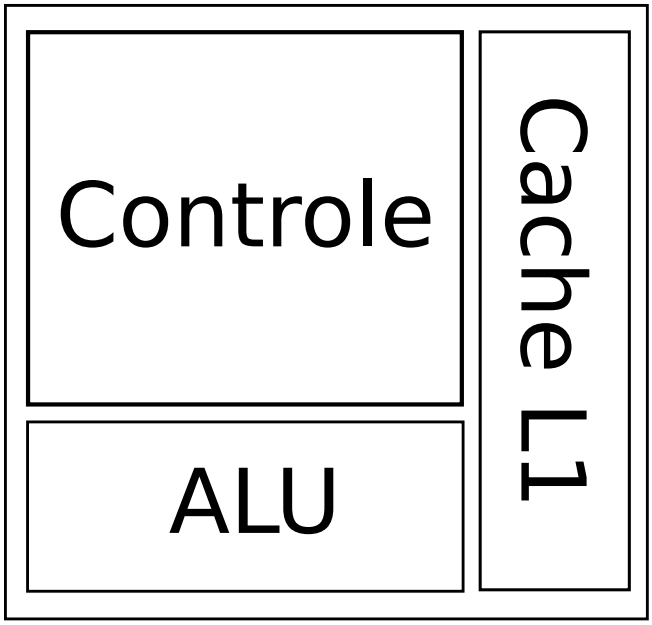
$$|\mathbf{S}|^2 = \mathbf{S}_x^2 + \mathbf{S}_y^2 \tag{2.4}$$

2.1 OpenMP e auto-vetorização

Limitações físicas têm impedido na última década que o crescimento das frequências de *clock* se dê na mesma taxa que no passado. A solução encontrada para continuar aumentando o desempenho de aplicações é através da exploração do paralelismo encontrado nelas.



(a) Diagrama simplificado de uma CPU multi-núcleo.



(b) Diagrama simplificado de um núcleo.

Figura 2.1: Diagramas de uma CPU

Podemos ver na Figura 2.1a um diagrama de uma CPU multi-núcleo com 4 núcleos. Cada um destes núcleos possui seu próprio cache, unidades aritméticas, unidades de controle e tudo o que existe em uma CPU de um único núcleo. Os núcleos são conectados entre si por um nível de cache e a memória. Eles podem, portanto, executar tarefas simultamente sobre o mesmo conjunto de dados na memória.

Em arquiteturas CPU multi-núcleo existem duas formas de paralelismo: *multi-threading* e vetorização.

No contexto de processamento paralelo de imagens, podemos com *multi-threading* criar diferentes versões do nosso programa, em que cada uma delas executa uma parte do trabalho total de processamento da imagem de entrada em núcleos diferentes simultaneamente, diminuindo o tempo total de execução.

Vetorização é o uso de instruções específicas do processador que permitem operações em mais de um elemento de dados por vez. Por exemplo: através do uso de SSE e AVX em CPUs x86, podemos executar operações como adição e multiplicação em 4 ou 8 números em ponto-flutuante com precisão simples na mesma quantidade de tempo de uma instrução escalar, que processaria apenas um elemento por vez.

Além disso, CPUs x86 possuem ordenamento de memória forte (*strongly-ordered memory*), isso significa que uma alteração na memória feita por um dos núcleos é automaticamente visível para todos os outros núcleos. Vemos na Seção 2.2 que o mesmo não acontece em GPUs.

A Figura 2.1b mostra um diagrama simples de um núcleo, ele representa que em CPUs existe uma grande quantidade de silício dedicada à logica de controle em relação ao espaço dedicado à operações aritméticas. CPUs são otimizadas para minimizar a latência de uma única *thread* e para aplicações com fluxo de controle complexo.

Para utilizar *multi-threading* em C++ usamos OpenMP, que é uma API usada extensiva-

mente nas linguagens C, C++ e Fortran para programação paralela usando anotações no código.

No Código 1 temos uma implementação em OpenMP do operador de Sobel:

```
#pragma parallel for
for (y=1; y<height-1; y++)
  for (x=1; x<width-1; x++)
  grad_x_h[y*width+x] = input[y*width+(x-1)]
                      - input[y*width+(x+1)];

#pragma parallel for
for (y=1; y<height-1; y++)
  for (x=1; x<width-1; x++)
  grad_x_v[y*width+x] = (grad_x_h[(y-1)*width+x]
                       + grad_x_h[ y   *width+x] * 2.0f
                       + grad_x_h[(y+1)*width+x]);

// (o mesmo para grad_y ...)

#pragma parallel for
for (y=1; y<height-1; y++)
  for (x=1; x<width-1; x++)
    magnitude[y*width+x] = grad_x_v[y*width+x] * grad_x_v[y*width+x]
                         + grad_y_h[y*width+x] * grad_y_h[y*width+x];
```

Código 1: Operador de Sobel usando OpenMP

Vemos que a utilização de OpenMP neste caso é extremamente simples, basta adicionar o pragma "parallel for" para que o laço na direção vertical da imagem seja quebrado e executado em várias *threads*. Por exemplo, se temos 4 processadores e o laço em y possui 1024 iterações, este espaço vai ser dividido em 4 partes e cada uma delas será executada em uma *thread*.

É função do programador verificar que o código dentro do laço que está sendo paralelizado não possui condições de corrida, i.e. quando o resultado de uma operação não é determinístico porque mais de uma *thread* está modificando e lendo os mesmos dados da memória sem sincronização (problema da região crítica).

Vetorização em x86 depende que os dados na memória estejam alinhados em 16-bytes (SSE) ou mesmo 32-bytes (AVX), ou seja, o endereço de memória do primeiro elemento do vetor tem que ser múltiplo de 16/32-bits. Portanto, é necessário alocar memória através de funções especiais como *posix_memalign*.

É possível usar vetorização diretamente através de *intrisics*, que são mapeadas diretamente nas instruções SSE/AVX. Porém, o uso destas dificulta extremamente a escrita e leitura do código, além de limitar a portabilidade para uma arquitetura que possua outro modo de vetorização.

Um modo mais comum, e que usaremos neste trabalho, é confiar no compilador GCC para automaticamente detectar padrões no código e vetorizá-los, especialmente em laços. Um problema desta abordagem é que não é claro se o compilador vai ou não ser capaz de vetorizar um laço antes da compilação, porém é possível analisar os logs do GCC para saber se isso foi ou não possível.

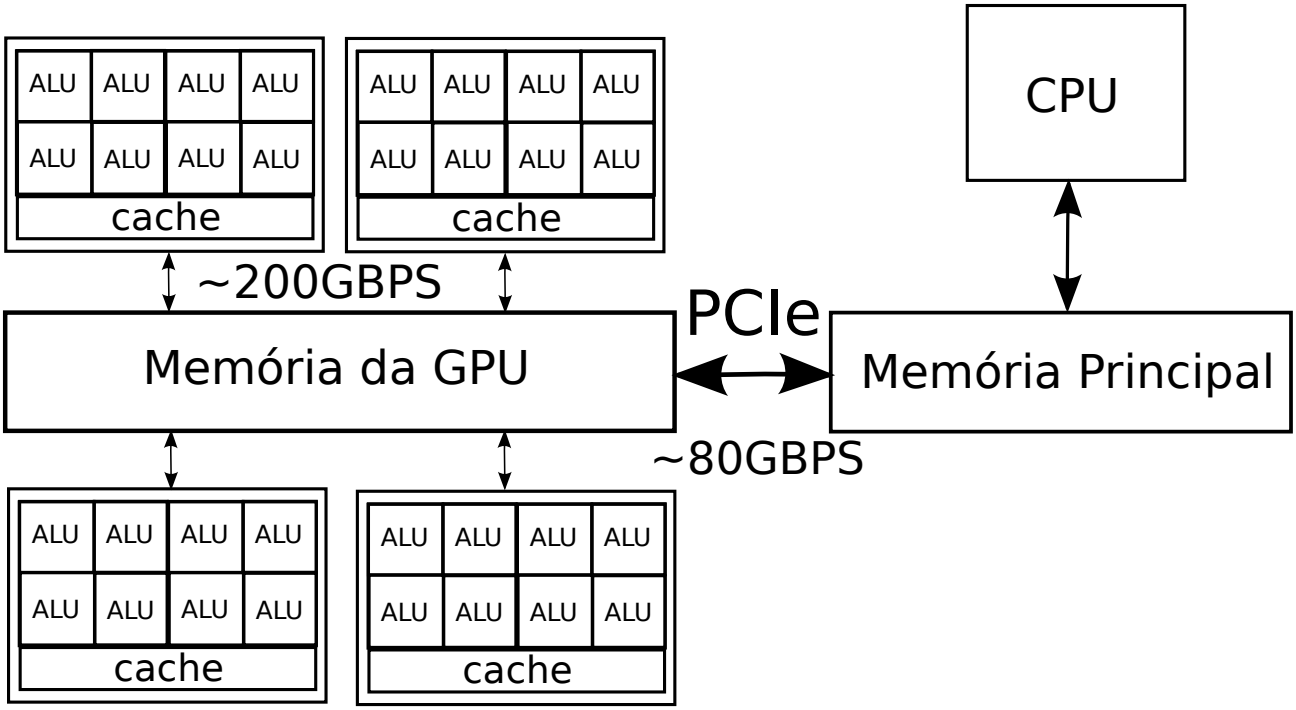


Figura 2.2: Diagrama simplificado de uma GPU.

Um outro problema é que o compilador GCC possui um tempo limite para executar cada passo de otimização, fazendo com que alguns laços complexos não sejam vetorizados, apesar disso ser possível.

2.2 Arquitetura GPU e OpenCL

GPUs são hardware especializados em tarefas de computação gráfica como renderização, operações em malhas. GPUs têm se tornado gradativamente mais programáveis e estão sendo usadas com sucesso em várias áreas que necessitam de bastante processamento em ponto-flutuante, como biotecnologia, simulação de fluidos, processamento de imagens, etc. Esta nova arquitetura GPU, agora totalmente programável, também é chamada de GPGPU (GPU de propósito-geral).

Podemos ver na Figura 2.2 um diagrama simplificado de uma GPU. GPUs possuem um grande número de processadores simples com menos espaço dedicado ao controle e cache, mas uma grande quantidade de hardware para operações numéricas e um canal de memória otimizado para banda, pois é necessário servir um grande número de ALUs. A maior banda vem a um custo de latência. GPUs, portanto, não são adequadas para algoritmos com muita diversidade de fluxo de dados e acessos não-sequenciais à memória.

GPUs têm uma arquitetura massivamente paralela, com centenas de processadores simples trabalhando simultaneamente. GPUs exploram o *paralelismo de dados*, ou seja, cada processador executa o mesmo programa em diferentes partes do conjunto de dados. Se temos, por exemplo, uma imagem em que queremos executar uma convolução, cada processador pode ser responsável por avaliar um pixel da saída. Este modelo faz oposição ao *paralelismo de tarefa*, que é o modelo

mais usado em CPUs, em que cada processador executa em paralelo tarefas completamente diferentes.

GPUs também possuem um modelo fraco de ordenamento da memória, portanto, uma alteração feita por um processador à memória não é automaticamente visível para todos os outros processadores. Este modelo de memória é mais escalável do ponto de vista do hardware mas introduz grandes dificuldades e erros sutis na programação de GPUs.

Além disso, GPUs atualmente funcionam como co-processadores. O que significa que é necessário programar a máquina hospedeira para controlar o dispositivo.

Neste trabalho, usamos uma GPU Nvidia e as linguagens OpenCL (Stone, Gohara & Guochun 2010) e CUDA (NVIDIA 2011).

Vamos brevemente definir alguns conceitos da linguagem OpenCL/CUDA que serão utilizados ao longo do trabalho:

- Kernel: uma função que é executada em paralelo na GPU.
- WorkItem/Thread: contexto (valor atual dos registradores) e execução de instruções em série em um processador da GPU.
- WorkGroup/Bloco: grupo de *WorkItems*/*Threads* que são executadas simultaneamente (SIMD) em um processador da GPU e podem compartilhar memória através de primitivas de sincronização. Normalmente são na ordem de centenas de *threads*.
- NDRange/Grid: número de *WorkGroups*/Blocos sendo executados ao todo na GPU para um kernel.

Diferente de arquiteturas x86, o acesso à memória de GPUs não é uniforme, ou seja, existem diferentes espaços de endereçamento dentro de um kernel que possuem diferentes desempenhos em latência e banda de memória:

- Memória Global: memória RAM (usualmente no mesmo pacote da GPU) que a GPU pode acessar dentro de um kernel e que é usada para ler e gravar dados que podem ser posteriormente copiados para a memória principal e lidos pela CPU. É da ordem de Gigabytes em GPUs modernas.
- Memória Compartilhada: memória que somente os *threads* em um mesmo bloco podem acessar e que possui latência extremamente baixa, da ordem de Kilobytes.
- Memória Privada: arquivo de registradores de cada *thread*, na ordem também de Kilobytes.
- Memória de Textura: cache gerenciado automaticamente que é otimizado para acessos bidimensionais a imagens. Na prática, ele torna mais rápidas operações como projeções e convoluções.

Em relação à arquitetura de GPUs, na Figura 2.2 temos que o cache de cada processador é onde residem a memória compartilhada e de textura. Como vemos, diferentemente de CPUs,

não há sincronização entre caches de diferentes processadores, portanto estas memórias têm validade apenas no contexto de um mesmo bloco.

Nesta hierarquia de memória, como esperado, as memórias privada, compartilhada e de textura são as mais rápidas, porém possuem tamanho bastante limitado comparado à memória global e à memória da máquina hospedeira.

Além disso, GPUs discretas como a que estamos usando neste trabalho não se comunicam diretamente com a CPU. É necessário transferir os dados que serão processados para a memória da GPU antes da execução de um kernel e, ao fim da execução, transferir de volta os resultados. Isso é feito através da porta PCIe, que possui grande latência comparada a um acesso simples à memória pela CPU. Portanto, o uso de GPUs envolve uma sobrecarga que deve ser analisado para cada problema, o que pode fazer com que um algoritmo seja mais rápido na CPU dependendo do quanto é possível explorar o paralelismo de dados no problema e a quantidade de transferências de memória. Além disso, como a banda da porta PCIe é bem menor do que a da memória da GPU, não é incomum que aplicações que necessitam de muita comunicação com a CPU sejam na verdade limitadas por este fator e não pelo desempenho da GPU.

Com relação à arquitetura CUDA, existem alguns compromissos que são feitos com relação ao tamanho do bloco. Basicamente, na arquitetura Nvidia (e GPUs similares de outros fabricantes), a GPU é composta de processadores que ficam responsáveis por executar vários blocos. Todos os blocos em um processador compartilham o arquivo de registradores e a memória compartilhada do mesmo, embora apenas um bloco seja executado por vez. É importante que tenhamos vários blocos disponíveis para execução porque, embora GPUs tenham uma grande quantidade de banda de memória em relação à GPU, há também uma grande latência. A GPU mascara esta latência intercalando a execução de vários blocos, ou seja, enquanto um bloco espera a execução de uma operação na memória, outro bloco pode executar instruções aritméticas. Assim, o parâmetro do tamanho do bloco que será utilizado na execução de um *kernel* é extremamente importante para um bom desempenho.

2.2.1 Exemplo de uso de OpenCL

Temos no Código 2 um exemplo de código OpenCL que executa o filtro de Sobel (Equação 2.2) para computar o gradiente da imagem,. Este código possui 3 kernels que têm de ser configurados e executados pela GPU.

Para lançar os kernels do operador de Sobel, executamos na máquina hospedeira o Código 3.

Podemos ver que através da função *get_global_id* temos um algoritmo que funciona para um pixel de entrada/saída, o que é um exemplo de paralelismo de dados. Detalhes de agrupamento de *WorkItems* e gerenciamento de *hardware* durante a execução são em maior parte abstraídos pela linguagem.

O domínio de um *kernel* (os índices retornados por *get_global_id*) são um parâmetro de configuração de execução do mesmo. Em processamento de imagens, normalmente temos que o domínio é o tamanho da imagem de saída e que cada *thread* escreve o valor de apenas um dos pixels da saída.

```
kernel void kernel_x_h(global float *input,
                       global float *grad_x_h,
                       int width)
{
  int x = get_global_id(0); // cada thread tem um indice de 1 a W-1
  int y = get_global_id(1); // cada thread tem um indice de 0 a H

  grad_x_h[y*width+x] = input[y*width+(x-1)]
                      - input[y*width+(x+1)];
}

// (o mesmo para kernel_y_h)

kernel void kernel_magnitude(global float *grad_x_h,
                             global float *grad_y_h,
                             global float *magnitude,
                             global float *angle,
                             int width)
{
  int x = get_global_id(0); // cada thread tem um indice de 1 a W-1
  int y = get_global_id(1); // cada thread tem um indice de 1 a H-1

  float grad_x_v = (grad_x_h[(y-1)*width+x]
                 + grad_x_h[ y   *width+x] * 2.0f
                 + grad_x_h[(y+1)*width+x]);

  // (o mesmo para grad_y_v)

  magnitude[y*width+x] = grad_x_v * grad_x_v
                       + grad_y_h * grad_y_h;

  angle[y*width+x] = arctan2(grad_y_v, grad_x_h);
}
```

Código 2: Operador de Sobel usando OpenCL.

Neste caso, a diferença entre o código em C++ e em OpenCL é que os dois laços mais externos, em x e em y, ficam implícitos no *kernel*, mas que ambos são bastante similares.

Por GPUs serem co-processadores é necessário executar o kernel *edge_sobel* na máquina hospedeira que comande na GPU. Os seguintes passos são necessários antes de ser possível de fato executar um Kernel:

- Detectar e selecionar qual dispositivo OpenCL que será utilizado. Neste caso, uma GPU.
- Criar o contexto para alocação de memória na GPU.
- Criar a fila de comandos para enviar comandos à GPU.
- Compilar o kernel para a GPU apropriada.
- Alocar memória na GPU para os buffers “input”, “angle” e “magnitude”.

```
// iteracao de [1,0] a [W-1,H]
size_t offset[3] = {1,0,0};
size_t ndrange_size[3] = {W-2,H,1};

clSetKernelArg(kernel_x_h, 0, sizeof(cl_mem), &input);
clSetKernelArg(kernel_x_h, 1, sizeof(cl_mem), &grad_x_h);
clSetKernelArg(kernel_x_h, 2, sizeof(cl_int), &width);

clEnqueueNDRangeKernel(command_queue,
                       kernel_x_h, 2,
                       offset, ndrange_size, NULL,
                       0, NULL, NULL);

// (o mesmo para kernel_y_h ...)

// iteracao de [1,1] a [W-1,H-1]
size_t offset2[3] = {1,1,0};
size_t ndrange_size2[3] = {W-2,H-2,1};

clSetKernelArg(kernel_magnitude, 0, sizeof(cl_mem), &grad_x_h);
clSetKernelArg(kernel_magnitude, 1, sizeof(cl_mem), &grad_y_h);
clSetKernelArg(kernel_magnitude, 2, sizeof(cl_mem), &magnitude);
clSetKernelArg(kernel_magnitude, 3, sizeof(cl_mem), &angle);
clSetKernelArg(kernel_magnitude, 4, sizeof(cl_int), &width);

clEnqueueNDRangeKernel(command_queue,
                       kernel_magnitude, 2,
                       offset2, ndrange_size2, NULL,
                       0, NULL, NULL);
```

Código 3: Código em C para executar 2 *kernels* OpenCL.

- Especificar parâmetros do kernels.
- Colocar na fila de comandos a transferência para a GPU (via a porta PCIe) da imagem de entrada no buffer “input”.
- Lançar o kernel na fila de comandos com domínio $largura \times altura$.
- Sincronizar a máquina hospedeira, para esperar a execução do kernel.
- Transferência de memória para a máquina hospedeira de “angle” e “magnitude”.

Vemos que a função *clSetKernelArg* determina qual será o valor de um dos parâmetro do kernel, portanto, chamamo-las 3 vezes para o kernel *kernel_x_h* para selecionar a entrada, saída e largura das imagens. *clEnqueueNDRangeKernel* executa o kernel na GPU com os parâmetros selecionados previamente e com o domínio de execução: o intervalo de valores que *get_global_id(0)* e *get_global_id(1)* vão assumir.

Como veremos na Seção 5.2, é necessária uma grande quantidade de código na máquina hospedeira para o gerenciamento da GPU.

Enquanto que Halide gera código de acordo com o modelo CUDA, neste trabalho todas as implementações para GPU são escritas em OpenCL (Khronos Group 2010). OpenCL possui um modelo de programação bastante parecido com CUDA, mas com a vantagem de que pode ser executado em diversos tipos de hardware como GPUs, CPUs x86, FPGAs e DSPs. Porém, enquanto que OpenCL garante a compatibilidade de um código entre diferentes tipos de hardware, a linguagem não faz garantias sobre desempenho.

Portanto, um dos pontos que é avaliado neste trabalho é o comportamento de diferentes implementações OpenCL em CPUs e GPUs.

Para uma descrição mais profunda do modelo de programação de GPUs, pode-se consultar os trabalhos desenvolvidos por (Kirk & Hwu 2010) e (Munshi et al. 2011).

Capítulo 3

A Linguagem Halide

Halide (Ragan-Kelley et al. 2012) é uma linguagem de domínio-específico para aplicações em processamento de imagens.

Halide também é uma DSL embarcada (*embedded DSL*), no sentido que ela é usada dentro de uma linguagem de propósito-geral (C++), desse modo é relativamente fácil substituir partes de uma base de código já existente por um equivalente em Halide.

DSLs para processamento de imagens não são um desenvolvimento recente. Focando em DSLs que suportam CPU e GPU, temos o Apple CoreImage (Apple 2013) e o Adobe PixelBender (Adobe 2013). Na literatura acadêmica, temos Neon (Guenter & Nehab 2010), que é uma linguagem embarcada em C#, Nikola (Mainland & Morrisett 2010), que é embarcada em Haskell.

Estas linguagens basicamente especificam algoritmos pontuais em imagens, mas não permitem facilmente configurar em detalhes como será dada esta execução em um hardware, como veremos que pode ser feito em Halide. Enquanto que estas linguagens fazem otimizações dentro de um kernel, não é possível fazer otimizações entre diferentes kernels conectados por dependências de dados, o que é possível de ser especificado em Halide.

Além disso, DSLs possuem problemas de expressividade, i.e. nem todo algoritmo de processamento de imagens pode ser colocado nas linguagens. Halide reivindica ser a melhor linguagem até o momento nesse quesito, e de fato há muitas primitivas em Halide que podem ser usadas para implementar algoritmo da área, como vemos no Capítulo 4.

Todas as DSLs mencionadas, incluindo Halide, são baseadas em (Shantzis 1994), que propõe um sistema em que a renderização de um *pipeline* é feita sob demanda. O grafo de dependências de dados de um *pipeline* é avaliado da saída para a entrada, e nisso são calculadas o tamanho da região em cada etapa do pipeline que deve ser processada. Na prática, isso elimina computação desnecessária de regiões que nunca serão utilizadas mas faz com que cada pipeline só possa ter como resultado apenas uma única imagem. Veremos que certos algoritmos necessitam de mais de uma saída.

Existe uma grande sobreposição na literatura entre o trabalho feito por estas linguagens. NEON, por exemplo, implementa muitas das mesmas otimizações para vetorização usadas em Halide. Além disso, todas as linguagens possuem foco em especificar o resultado pixel-a-pixel, para que o paralelismo de dados fique evidente.

Nesse trabalho usamos Halide pois a linguagem é bastante recente, possui uma crescente

base de usuários e implementação aberta. Comparamos Halide com C++ e OpenCL. Devemos ter em mente que muitos dos resultados encontrados aqui podem ser generalizados para outras DSLs similares.

Como curiosidade, a mesma abordagem do uso de DSLs para diferentes arquiteturas está sendo tomada em outros campos da computação: na área de algoritmos evolutivos temos ESOL (Dower 2012); em simulação de fluidos, Jet (Bailey, Masters & Warner 2011). Então vemos que esta é uma tendência não só na comunidade de processamento de imagens.

Um código em Halide possui sempre duas partes: uma especificação do algoritmo numa linguagem funcional pura e livre de efeitos colaterais (*side-effects*) que é independente de arquitetura e uma configuração de execução que diz como será mapeada a execução do algoritmo no hardware.

No Código 4 temos um pequeno exemplo de um programa em Halide que computa o operador de Sobel em uma imagem de acordo com a Equação 2.2. Como o filtro é separável, podemos executar dois passos na imagem para encontrar $\mathbf{S}_x$ e $\mathbf{S}_y$.

```
1  UniformImage input(Float(32), 2);
2  Var x,y;
3  Func grad_x_h, grad_x_v, grad_y_h, grad_y_v, magnitude;
4
5  grad_x_h(x,y) = (input(x  , y+1)
6                 - input(x+2, y+1));
7
8  grad_x_v(x,y) = (grad_x_h(x+1, y  )
9                 + grad_x_h(x+1, y+1) * 2.0f
10                + grad_x_h(x+1, y+2));
11
12 // (o mesmo para grad_y ...)
13
14 magnitude(x,y) = (grad_x_v(x,y) * grad_x_v(x,y)
15                  + grad_y_v(x,y) * grad_y_v(x,y));
```

Código 4: Código em Halide para o operador de Sobel.

Halide verifica em tempo de compilação que não há possibilidade de haver acessos de memória fora da imagem. Note que em Halide não há um modo de especificar como se comportam as coordenadas da imagem na saída, portanto o único modo de evitar acessos fora da imagem é usar coordenadas centradas no canto superior esquerdo. Isso explica a diferença entre as coordenadas usadas e as do Capítulo 2.

Para executar o código Halide, basta fazermos:

```
1  Image<float> out (input.width()-2, input.height()-2, 1);
2  magnitude.realize(out);
```

3.1 Sintonizando um schedule

Um *schedule* em Halide é uma declaração de como uma função vai se comportar em termos de localidade de memória, vetorização, *multi-threading* e uso de GPU.

As duas principais políticas de *schedule* são *root* e *inline*. Em *root*, os valores da função são computados e guardados na memória, cada utilização subsequente desses valores vai simplesmente referenciar esta imagem na memória. Em *inline*, temos o inverso, todos os usos da função vão computar e recomputar os cálculos necessários da função, portanto não há uso de memória extra para guardar os valores da função.

Temos através destes dois schedules o compromisso entre recomputação e acessos à memória.

Podemos ver que existem 4 funções em nosso filtro: magnitude, grad_y_v, grad_x_h, grad_y_h, grad_x_v, onde as dependências de dados estão na mesma ordem, respectivamente. Se não especificamos um *schedule*, Halide assume a política *inline*. Então, com esta política teríamos que cada pixel na saída recomputa os valores adjacentes de grad_x_h e grad_y_v, o que resulta em uma grande quantidade de recomputação e acessos desnecessários à memória, levando ao primeiro *schedule*.

3.1.1 Schedule 1

Primeiro, vamos tentar guardar na memória os valores necessários de grad_x_h e grad_y_v, assim evitamos a recomputação de valores para diferentes pixels na saída. Para isso, usamos a política *root*:

```
1  grad_x_h .root();
2  grad_x_v .root();
3  grad_y_v .root();
4  grad_y_h .root();
5  magnitude.root();
```

Schedules em Halide funcionam basicamente aplicando o que se chama de *Loop Scheduling Transformations* (Allen & Kennedy 2002). Temos no Código 5 a combinação do operador de Sobel com o *schedule* anterior em uma linguagem imperativa.

3.1.2 Schedule 2

Podemos usar múltiplas *threads* para computar cada estágio em paralelo:

```
1  grad_x_h .root().parallel(y);
2  grad_x_v .root().parallel(y);
3  grad_y_v .root().parallel(y);
4  grad_y_h .root().parallel(y);
5  magnitude.root().parallel(y);
```

Com isso, cada laço é executado em paralelo para a imagem toda. Podemos ver na Figura 3.1 que isso leva a um ganho de quase 4 vezes de desempenho, como esperado em uma CPU com 4 núcleos.

```
for (y=0; y<H; y++)
  for (x=0; x<W+1; x++)
    grad_x_h(x,y) = (input(x  , y+1)
                     - input(x+2, y+1));

for (y=0; y<H; y++)
  for (x=0; x<W; x++)
    grad_x_v(x,y) = (grad_x_h(x+1, y  )
                     + grad_x_h(x+1, y+1) * 2.0f
                     + grad_x_h(x+1, y+2));

// (o mesmo para grad_y ...)

for (y=0; y<H; y++)
  for (x=0; x<W; x++)
    magnitude(x,y) =   grad_x_v(x, y) * grad_x_v(x, y)
                     + grad_y_h(x, y) * grad_y_h(x, y);
```

Código 5: Equivalente em C do operador de Sobel com um *schedule* em que as funções são completamente avaliadas e guardadas na memória para todos os pixels.

3.1.3 Schedule 3

Os *schedules* apresentados até agora não têm feito bom uso do cache da CPU: nos *schedules* anteriores computamos todos os valores de grad_x_h e grad_y_v antes de grad_x_v e grad_y_v, portanto, os valores em grad_x_h necessários para computar um elemento em grad_x_v dificilmente estarão no cache da CPU, melhor seria um modo de intercalar a execução de elementos de grad_x_h com elementos de grad_x_v.

Vamos usar a política *inline* para as funções grad_y_h, grad_x_v, grad_y_v e grad_x_h. Isso significa que estas funções serão computadas sob demanda. Por isso, elementos em grad_x_h e grad_y_h serão avaliados 3 vezes:

```
grad_x_h .inline();
grad_x_v .inline();
grad_y_v .inline();
grad_y_h .inline();
magnitude.root().parallel(y);
```

Podemos ver na Figura 3.1 que houve um grande aumento de desempenho com esse *schedule*, então surpreendentemente, a melhor escolha foi não utilizar a propriedade de separabilidade do filtro e manter a localidade de memória, a um custo de um maior número de operações.

Este *schedule* seria equivalente ao seguinte código:

```
parallel for (y=0; y<H; y++)
  for (x=0; x<W; x++)
    grad_x_h[0] = (input(x  , y)
                   - input(x+2, y));

```

```
    grad_x_h[1] = (input(x  , y+1)
                 - input(x+2, y+1));

    grad_x_h[2] = (input(x  , y+2)
                 - input(x+2, y+2));

    grad_x_v = grad_x_h[0] + 2.0f * grad_x_h[1] + grad_x_h[2];

    // (o mesmo para grad_y ...)

    magnitude(x,y) =   grad_x_v(x, y) * grad_x_v(x, y)
                     + grad_y_h(x, y) * grad_y_h(x, y);
```

3.1.4 Schedule 4

Nosso último *schedule* na CPU utiliza instruções vetoriais (SSE e SSE2) para processar 4 elementos de uma só vez:

```
grad_x_h .inline().vectorize(x, 4);
grad_x_v .inline().vectorize(x, 4);
grad_y_v .inline().vectorize(x, 4);
grad_y_h .inline().vectorize(x, 4);
magnitude.root().parallel(y).vectorize(x, 4);
```

3.1.5 Schedule 5

Para executar o operador de Sobel na GPU, basta usar o seguinte *schedule*:

```
magnitude.root().cudaTile(x, y, 16, 16)
```

Podemos ver na Figura 3.1 que a escolha do *schedule* é essencial para um bom desempenho. *Schedules* para diferentes arquiteturas são naturalmente distintos e separam os detalhes relacionados à arquitetura da especificação do algoritmo, mitigando o problema de se ter que escrever várias versões do mesmo algoritmo, pois apenas o schedule tem de ser modificado em uma nova arquitetura e fazendo com que seja mais fácil descobrir possíveis otimizações através de tentativa e erro.

Tivemos que escrever 15 diferentes *schedules* para o operador de Sobel para encontrar qual forma de execução possui melhor desempenho. Podemos ver que a interação entre elementos do hardware (como o cache, memória, etc) fazem com que esta escolha seja um problema complexo. O uso de *schedules* permite uma avaliaçao rápida de diferentes modos de execução do algoritmo.

Pode-se ver também na Figura 3.1 que o uso de GPUs não necessariamente melhora o desempenho de um filtro. No caso, a sobrecarga das transferências de memória para a GPU usando a porta PCIe corresponde a 60% do tempo total de execução do filtro.

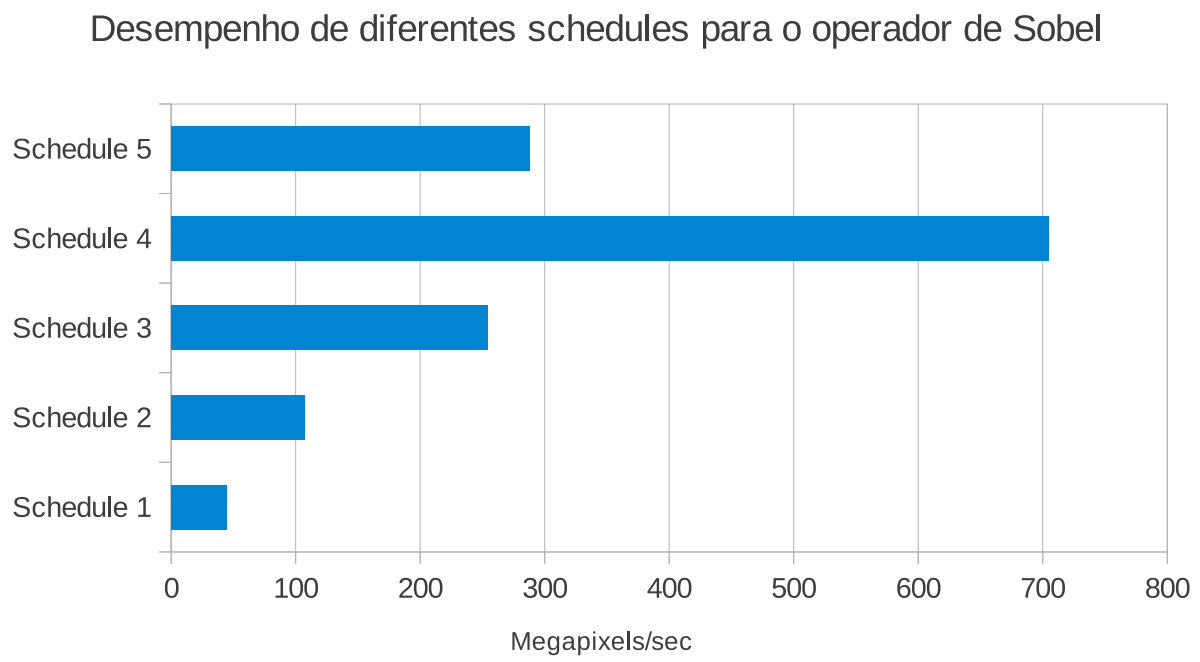


Figura 3.1: Desempenho do operador de Sobel para diferentes *schedules* (quanto maior, melhor).

3.2 Reduções em Halide

Reduções são uma construção da linguagem Halide em que podemos expressar muitos dos padrões de programação encontrados em processamento de imagens. Até o momento, não há como expressar convoluções gerais em que o tamanho da máscara só é conhecido em tempo de execução e operações recursivas, i.e., quando o valor de um pixel na imagem depende do valor de outros pixels na mesma imagem.

Isso pode ser feito em Halide através de *Reduções*. Reduções são compostas por duas funções: a primeira é a inicialização da imagem; a segunda especifica passos incrementais que serão feitos sobre a mesma imagem em um intervalo chamado de Domínio de Redução. Portanto, reduções são o único modo de escrever funções não-puras em Halide (funções que alteram permanentemente seus parâmetros).

Aqui está um exemplo de redução onde fazemos a convolução de uma imagem por uma máscara gaussiana cujo desvio padrão é um argumento. Halide detecta se é possível haver acessos fora das imagens de entrada, portanto temos que colocar guardas na forma da operação *clamp* em *blurx* e *blury*:

```
1  // desvio padrao da gaussiana
2  Uniform<float> sigma;
3
4  // funcao gaussiana
5  Func gaussian;
6  gaussian(x) = exp(-(x/sigma)*(x/sigma)*0.5f);
7
```

```
8  // truncar em 3 sigma
9  Expr radius = cast<int>(3*sigma + 1.0f);
10
11 // dominio de reducao
12 RDom r(-radius, 2*radius+1);
13
14 // mascara gaussiana normalizada
15 Func ngaussian;
16 ngaussian(x) = gaussian(x) / sum(gaussian(r));
17
18 // Convolucao da entrada na horizontal e vertical
19 Func blurx, blury;
20 blurx(x, y) += input(clamp(x+r, 0, W-1), y) * ngaussian(r);
21 blury(x, y) += blurx(x, clamp(y+r, 0, H-1)) * ngaussian(r);
```

Com reduções, laços ficam implícitos através do uso da variável r. Na linha 2 temos um dos parâmetros do algoritmo, o desvio padrão do filtro gaussiano. Na linha 9 truncamos o raio da máscara em 3 σ. Na linha 16 temos uma redução implícita através do uso da função *sum* para acumular todos os valores da máscara. Nas linhas 20 e 21 temos duas reduções: como o filtro gaussiano usado é simétrico e separável, podemos primeiro calcular a filtragem na horizontal e depois na vertical, usamos a função *clamp* para não permitir acessos fora da imagem de entrada.

Por exemplo, para cada pixel de blurx é acumulado o valor dos pixels $[-3\sigma, +3\sigma]$ centrados em input(x,y) ponderados pela máscara gaussiana, que é uma convolução:

$$g(x) = e^{\frac{-(x/\sigma)^2}{2}} \tag{3.1}$$

$$\bar{g}(x) = \frac{g(x)}{\sum_{r=-3\sigma}^{3\sigma} g(r)} \tag{3.2}$$

$$G_x(x,y) = \sum_{r=-3\sigma}^{+3\sigma} I(x+r,y) * \bar{g}(r) \tag{3.3}$$

Reduções são relacionadas ao conceito de *estêncil*. Um estêncil define a computação de um elemento em um grade espacial n-dimensional como uma função de elementos vizinhos na grade ao longo de várias iterações. Convoluções, subamostragens, projeções, derivadas na imagem, e outras operações podem ser colocadas na forma de estêncils. Basicamente, um algoritmo pode ser mapeado facilmente em Halide se ele puder ser expressado em uma sequência de estêncils.

Podemos executar reduções na GPU. Para o filtro gaussiano, basta usar o seguinte *schedule*:

```
1 blurx.root().cudaTile(x, y, 16, 16);
2 blurx.update().root()
3               .cudaTile(x, y, 16, 16);
4
5 blury.root().cudaTile(x, y, 16, 16);
6 blury.update().root()
7               .cudaTile(x, y, 16, 16);
```

Vemos que como pode existir em reduções dependências de dados entre elementos na saída, a paralelização se torna mais difícil. Temos o exemplo de redução usado para equalização de histograma mostrado em (Ragan-Kelley et al. 2012):

```
1  UniformImage in(UInt(8), 2);
2  Func histogram, cdf, out;
3  RDom r(0, in.width(), 0, in.height()), ri(0, 255);
4  Var x, y, i;
5
6  // condicao de corrida
7  histogram(in(r.x, r.y))++;
8
9  cdf(i) = 0;
10
11 // computacao serial
12 cdf(ri) = cdf(ri-1) + histogram(ri);
13
14 out(x, y) = cdf(in(x, y));
```

A redução da linha 7 corresponde à seguinte equação:

$$H(i) = \sum_{p} I(p) | I(p) = i \tag{3.4}$$

Se tentássemos executar a função *histogram* em paralelo, teríamos diversos acessos concorrentes na imagem e uma condição de corrida, que produziria um resultado inválido. Portanto não podemos usar as políticas *cudaTile* e *parallel* para esta função.

Além disso, a computação da função *cdf* é totalmente serial, devido às dependências de dados em todos os elementos.

Como ambos os problemas são limitações do próprio algoritmo, não há como Halide paralelizar estas funções. Portanto, é necessário cautela durante a escrita de reduções para que não haja esse tipo de dependência de dados com um schedule que especifica paralelismo na CPU ou GPU.

Seria possível a computação em GPU da função *histogram* através do uso de operações atômicas, mas não há como utilizar estas intruções em Halide. Também, a serialização devido ao congestionamento na memória de tantas *threads* escrevendo no mesmo endereço faz com que esta implementação seja bastante lenta.

3.3 Funcionamento Interno da Linguagem Halide

Halide usa a infra estrutura de compiladores LLVM (Lattner & V. 2004) na geração de código para diferentes arquiteturas, que usa uma representação intermediária chamada de LLVM IR que é relativamente independente de arquitetura.

Halide, de acordo com o modelo proposto em (Shantzis 1994), precisa encontrar a região de cada imagem do pipeline que é avaliada, dependendo das suas dependências de dados. Isso é

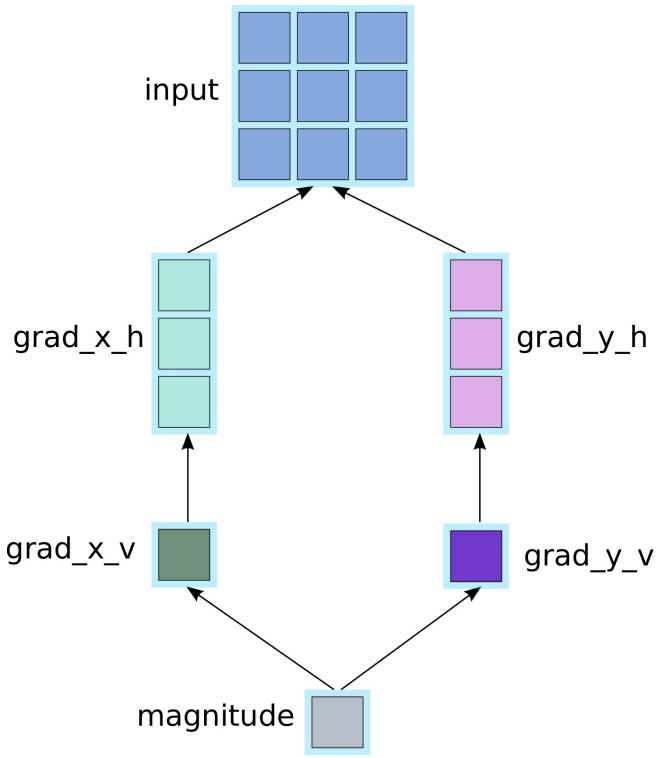


Figura 3.2: Dependências de dados no operador de Sobel.

feito através de aritmética de intervalos. Portanto, podemos ver na Figura 3.2 que cada pixel na saída depende de 1 pixel em grad_x_v, 3 pixels em grad_x_h e 9 pixels na entrada, o mesmo para grad_y_v e grad_y_h.

Halide possui uma representação intermediária (que chamaremos de Halide IR) do código que é onde as transformações de *schedule* são aplicadas.

Temos no Código 6 a representação em Halide IR do operador de Sobel usando o seguinte *schedule*:

```
1 grad_x_h.root().cudaTile(x,y,16,16);
2 grad_x_v.root().cudaTile(x,y,16,16);
3
4 grad_y_h.root().cudaTile(x,y,16,16);
5 grad_y_v.root().cudaTile(x,y,16,16);
6
7 magnitude.root().cudaTile(x,y,16,16);
```

Vemos que Halide IR é uma linguagem funcional pura, similar à LISP, e que as operações na memória e alocações de memória são explícitas.

Como o código será executado na GPU, temos que cada *schedule* com política *root* é mapeado em um único kernel (portanto, temos 5 kernels nesse exemplo). Halide também precisa quebrar o problema em regiões de 16x16 pixels, que são mapeadas em um único bloco. Então os 4 primeiros laços de cada kernel são mapeados nas dimensões do bloco e grade no modelo de execução CUDA.

Aqui estão as etapas executadas pelo compilador Halide até a geração do código final:

1. Código em C++ do usuário gera uma representação intermediária em Halide.
2. Através de aritmética de intervalos, encontrar a extensão do domínio de cada função que deve ser avaliado.
3. Halide aplica os *schedules*, que quebram e reordenam laços sem alterar a semântica do programa.
4. Algumas otimizações relativas à arquitetura de destino (e.g. vetorização) são executadas.
5. Halide IR é transformada em LLVM IR.
6. Compilador LLVM executa otimizações sobre LLVM IR.
7. LLVM transforma LLVM IR para a linguagem de máquina da arquitetura de destino.
8. LLVM executa mais otimizações de baixo-nível e produz o binário final que será executado.

3.4 Limitações

Existem algumas limitações inerentes à linguagem Halide. A mais óbvia é que o uso de uma linguagem funcional pura complica (ou até mesmo impede) a implementação de certos algoritmos que são mais facilmente descritos com o uso de efeitos colaterais.

Por exemplo, em um algoritmo simples de rotulação de componentes conexos baseado em autômatos celulares (Algoritmo 1), temos que a execução do seguinte algoritmo eventualmente converge para a solução final. A variável *acabou* indica que não temos mais que executar iterações do algoritmo. Porém, sem a habilidade de ter efeitos colaterais, não podemos retornar esse valor. Portanto, esse algoritmo não pode ser expresso em Halide.

A proibição de efeitos colaterais se deve ao fato de que permitir que uma função altere seus parâmetros complica consideravelmente o projeto do compilador e suas otimizações.

Outra limitação da linguagem é que só é possível retornar uma imagem por função Halide. Temos, por exemplo, o filtro de Sobel, que é utilizado para achar o gradiente da imagem em cada ponto. Para expressar o gradiente, precisamos do ângulo do vetor (em radianos) e o módulo, portanto, 2 imagens são necessárias. Então, para computar o gradiente, teríamos que executar a maior parte do filtro duas vezes.

Poderíamos também utilizar uma imagem com dois canais para armazenar esse resultado, porém isso aumenta bastante o código Halide e causa ineficiências na GPU. Este método não poderia ser utilizado se as duas imagens de saída fossem de tamanhos diferentes.

A limitação de apenas uma saída por função é devido ao fato de que Halide computa dinamicamente os limites de cada função baseado em suas dependências de dados a partir do tamanho da imagem de saída, que é o modelo de execução apresentado em (Shantzis 1994).

A última limitação que vamos mencionar é que não há no momento um modo em Halide de se utilizar a memória de textura e memória compartilhada. Enquanto que o uso de texturas

é relativamente direto em OpenCL, o uso eficiente da memória compartilhada é extremamente difícil, pois ela é bastante limitada e na prática funciona como um cache gerenciado pelo usuário. Para isso é necessário o uso primitivas de sincronização entre *threads*, portanto os autores consideram difícil que a memória compartilhada possa ser usada por DSLs como Halide. Alguns de nossos experimentos no Capítulo 4 fazem uso da memória compartilhada em OpenCL, portanto veremos os efeitos desta limitação no contexto de processamento de imagens.

```
magnitude(input, output) =
; novo kernel - grad_x_h
 let grad_x_h.alloc_size = (W+1)*H
 allocate grad_x_h[grad_x_h.alloc_size]          ; alocacao de memoria na GPU
  produce grad_x_h
   parallel (bid.y: 0, (H/16))                   ; mapeado na dimensao y da grade
    parallel (bid.x: 0, (W/16+1))                ; mapeado na dimensao x da grade
     parallel (tid.y: 0, 16)                     ; mapeado na dimensao y do bloco
      parallel (tid.x: 0, 16)                    ; mapeado na dimensao x do bloco
       let y = bid.y * 16 + tid.y
       let x = bid.x * 16 + tid.x
       grad_x_h(x,y) = input(x,   y+1)
                     - input(x+2, y+1);

; novo kernel - grad_x_v
  let grad_x_v.alloc_size = W*H
  allocate grad_x_v[grad_x_v.alloc_size]          ; alocacao de memoria na GPU
   produce grad_x_v
    parallel (bid.y: 0, (H/16))                  ; mapeado na dimensao y da grade
     parallel (bid.x: 0, (W/16))                 ; mapeado na dimensao x da grade
      parallel (tid.y: 0, 16)                    ; mapeado na dimensao y do bloco
       parallel (tid.x: 0, 16)                   ; mapeado na dimensao x do bloco
        let y = bid.y * 16 + tid.y
        let x = bid.x * 16 + tid.x
        grad_x_v(x,y) = grad_x_h(x+1, y  )
                      + grad_x_h(x+1, y+1) * 2.0f
                      + grad_x_h(x+1, y+2);

; (o mesmo para grad_y ...)

; novo kernel - magnitude
  parallel (bid.y: 0, (H/16))                    ; mapeado na dimensao y da grade
   parallel (bid.x: 0, (W/16))                   ; mapeado na dimensao x da grade
    parallel (tid.y: 0, 16)                      ; mapeado na dimensao y do bloco
     parallel (tid.x: 0, 16)                     ; mapeado na dimensao x do bloco
      let y = bid.y * 16 + tid.y
      let x = bid.x * 16 + tid.x
      output(x,y) = (grad_x_v(x,y)*grad_x_v(x,y)
                  + (grad_y_v(x,y)*grad_y_v(x,y)));
```

Código 6: Representação interna de Halide para o operador de Sobel.

Algoritmo 1 Uma iteração do algoritmo de rotulação de componentes conexos.

```
procedimento CCL-ITERAÇÃO(int[] rotulo, bool acabou)
    acabou ← verdadeiro
    para todo pixel p na imagem faça
        rotulo[p] ← menor rotulo na vizinhança de p
        se houve mudança em rotulo[p] então
            acabou ← falso
        fim se
    fim para
     retorna rotulo
fim procedimento
```

Capítulo 4

Estudos de Caso

Para avaliar o desempenho de Halide e sua facilidade de uso, decidimos implementar em C++ com OpenMP, OpenCL e Halide, filtros com diferentes padrões normalmente encontrados em processamento de imagens, para que possamos generalizar nossos resultados tanto quanto seja possível.

Nosso objetivo neste trabalho é também verificar se Halide pode ser utilizado em um programa de manipulação e edição de imagem, como o GIMP. Portanto muitos dos filtros avaliados neste trabalho pertencem à categoria de manipulação de imagens:

- Conversão de espaço de cores: RGB para CIELAB
- Unsharp Masking
- Motion-Blur
- Filtro Bilateral (Paris et al. 2009)

Além disso, implementamos os seguintes 2 filtros para detecção de pontos de interesse em imagens:

- Detector de Cantos de Harris & Stephens (Harris & Stephens 1988)
- SIFT (Lowe 1999)

Ambos os algoritmos podem ser usados em aplicações como: rastreamento de vídeo, casamento e registro de imagens, descritores visuais, etc.

Deste modo, avaliamos a possibilidade do uso de Halide em diferentes áreas de processamento de imagens. Preocupamo-nos em ter um conjunto diverso de filtros, com diferentes padrões de implementação para que as conclusões retiradas do trabalho possam ser estendidas para o maior número possível de outras aplicações em imagens. Filtros complexos como o Filtro Bilateral e SIFT têm o objetivo de verificar quais são as limitações da linguagem e sua atual implementação.

Verificamos o desempenho e facilidade de implementação destes filtros em CPU (para C++ e OpenMP, OpenCL e Halide) e GPU (para OpenCL e Halide).

4.1 Conversão de RGB para CIELAB

O espaço de cor CIELAB tem como principal vantagem ser perceptualmente uniforme, i.e. a diferença numérica entre os valores de cor corresponde aproximadamente à diferença sentida pelo olho humano.

Para converter do espaço sRGB (*standard RGB color space*) para o CIELAB, são necessários os seguintes passos para cada pixel:

4.1.1 Reversão da Correção de Gamma

O espaço sRGB assume que suas imagens serão exibidas em um monitor de raios catódicos. Como este tipo de equipamento exibe uma curva de potência em relação à luminância, temos que aplicar a operação inversa para trazer os valores a um espaço independente de dispositivo. A seguinte transformação é aplicada para os canais R,G e B:

$$C_{\text{linear}} = \begin{cases} \frac{C_{\text{srgb}}}{12.92}, & C_{\text{srgb}} \leq 0.04045 \\ \left(\frac{C_{\text{srgb}}+a}{1+a}\right)^{\gamma}, & \text{caso contrário} \end{cases} \tag{4.1}$$

Em que:

$$\begin{aligned} &a = 0.055, \\ &\gamma = 2.4, \\ &C_{\text{linear}} = R, G, B_{\text{linear}} \end{aligned} \tag{4.2}$$

4.1.2 Conversão para o espaço CIEXYZ

Fazemos agora a conversão para o espaço CIEXYZ, que serve como passo intermediário para o CIELAB, que é uma operação linear:

$$\begin{bmatrix} X \\ Y \\ Z \end{bmatrix} = \begin{bmatrix} 0.4124 & 0.3576 & 0.1805 \\ 0.2126 & 0.7152 & 0.0722 \\ 0.0193 & 0.1192 & 0.9505 \end{bmatrix} \begin{bmatrix} R_{\text{linear}} \\ G_{\text{linear}} \\ B_{\text{linear}} \end{bmatrix} \tag{4.3}$$

4.1.3 Conversão para CIELAB

Como dissemos, CIELAB é perceptualmente uniforme, ou seja, uma mudança de valor nos valores de cor devem corresponder a uma mudança correspondente em importância visual.

O espaço CIELAB, assim como o CIEXYZ, inclui todas as cores perceptíveis pelo olho humano, então também é bastante usado para conversão entre diferentes formatos de cor.

$$\begin{aligned} L^{\star} &= 116 f(Y/Y_n) - 16 \\ a^{\star} &= 500(f(X/X_n) - f(Y/Y_n)) \\ b^{\star} &= 200(f(Y/Y_n) - f(Z/Z_n)) \end{aligned} \tag{4.4}$$

Em que:

$$
\begin{aligned}
f(t) &= \begin{cases} t^{1/3}, & \text{se } t > (\frac{6}{29})^3 \\ \frac{1}{3}\left(\frac{29}{6}\right)^2 t + \frac{4}{29}, & \text{caso contrário} \end{cases} \\
X_n &= 95.047 \\
Y_n &= 100 \\
Z_n &= 100.883
\end{aligned}
\tag{4.5}
$$

Como podemos ver, a conversão de RGB para CIELAB é uma operação independepente para cada pixel mas que é bastante cara computacionalmente, pois a conversão para o espaço CIEXYZ envolve uma exponenciação e a conversão final para CIELAB possui uma raiz cúbica e diversas multiplicações em ponto-flutuante.

Como GPUs possuem hardware específico para computar funções transcedentais (como exponenciação, raízes e logaritmos), o algoritmo pode fazer bom uso deste hardware. Além disso, por ser uma operação pontual, sem dependência de dados entre pixels, é uma ótima candidata para paralelização.

Além disso, o algoritmo possui poucos requisitos de banda de memória. Portanto, podemos esperar que este filtro tenha seu desempenho limitado pela porta PCIe, pois a transferência de dados entre CPU e GPU é o passo que consome mais tempo em filtros simples, como vimos na Figura 2.2.

Mesmo assim, tivemos alguns problemas para portar a conversão RGB para CIELAB em Halide. Não havia a implementação de algumas funções transcedentais (como exponenciação) no compilador Halide, portanto implementamos estas funções usando a linguagem Assembly de GPUs Nvidia: PTX (NVIDIA 2008).

4.2 Unsharp Masking

Unsharp Masking é uma técnica bastante conhecida para aumentar a percepção de detalhe em uma imagem. O filtro funciona sobrepondo a imagem original com a diferença entre a imagem borrada e ela mesma. Como o filtro gaussiano é um filtro passa-baixa, amplificamos assim as altas frequências da imagem de entrada, fazendo com que o constraste entre as cores da imagem fiquem mais evidentes. Podemos ver na Figura 4.1 um diagrama do filtro e na Figura 4.2, uma demonstração de sua execução.

Existem várias possíveis implementações para o filtro Unsharp Masking, neste trabalho usaremos a seguinte definição, usando $\bar{g}$ da Equação 3.2:

$$
\begin{aligned}
G(x,y) &= \sum_{r_y=-3\sigma}^{3\sigma} \sum_{r_x=-3\sigma}^{3\sigma} I(x+r_x, y+r_y) \cdot \bar{g}(r_x) \cdot \bar{g}(r_y) \\
D(x,y) &= G(x,y) - I(x,y) \\
U_m(x,y) &= I(x,y) + \kappa
\end{aligned}
\tag{4.6}
$$

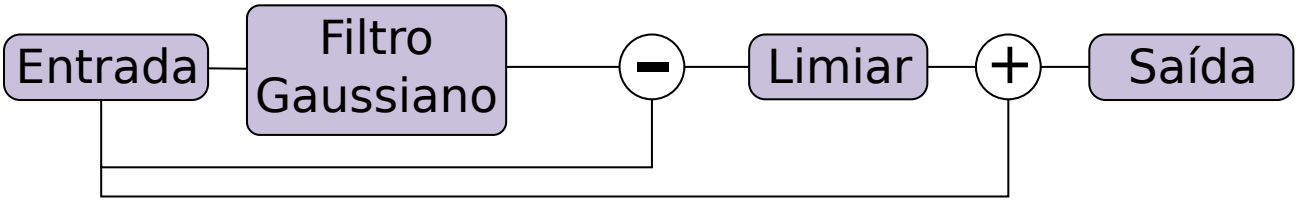


Figura 4.1: Diagrama do filtro Unsharp Masking.

(a) Imagem Original. (b) Após Unsharp Masking.

Figura 4.2: Demonstração do filtro Unsharp Masking.

Em que:

$$\kappa = \begin{cases} S(D(x,y) - T) & \text{se } D(x,y) > 0 \text{ e } D(x,y) > T \\ S(D(x,y) + T) & \text{se } D(x,y) < 0 \text{ e } D(x,y) < -T \\ 0 & \text{caso contrário} \end{cases} \tag{4.7}$$

Portanto, temos que os parâmetros do algoritmo são a imagem de entrada I; σ, o tamanho do filtro passa-baixa; o limiar T, que é ajustado para ajustar a quantidade de ruído na imagem final e S que controla a amplificação das altas frequências. A função U_m nos retorna a imagem de saída.

Vamos verificar neste filtro o desempenho de convoluções em Halide, pois a etapa mais longa do Unsharp Masking é o filtro gaussiano. Note que, como vimos na Seção 3.2, para implementar o filtro gaussiano em Halide precisamos usar reduções, enquanto que as outras operações do Unsharp Masking são pontuais.

Nossa implementação em OpenCL usa a memória de textura para imagens, pois com isso temos suporte em hardware para manter os acessos de memória dentro dos limites da imagem durante a convolução e um cache espacial bidimensional. Como Halide no momento não tem acesso a estes recursos, o experimento também vai mostrar o quanto isso afeta o desempenho de ambas as implementações.

(a) Imagem Original. (b) Após o Motion-Blur.

Figura 4.3: Demonstração do filtro Motion-Blur.

4.3 Motion-Blur

Motion-Blur ocorre naturalmente em fotografia quando tiramos uma foto e durante o tempo de exposição do filme, a câmera se move. O que acontece é que a imagem final é a integração de múltiplas imagens no fotossensor da câmera. Temos na Figura 4.3 uma demonstração do filtro.

Poderíamos, assim como no Unsharp Masking, usar uma convolução tradicional para o Motion-Blur usando máscaras com '1's seguindo uma linha reta (Equação 4.8), porém este método é bastante ineficiente devido à esparcidade das matrizes de convolução.

$$M = \frac{1}{5} \cdot \begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix} \tag{4.8}$$

Como podemos ver, esta matriz pode ser parametrizada como uma linha. Colocando esta informação na equação tradicional de convolução 2D, temos a seguinte fórmula para o filtro Motion-Blur:

$$\begin{aligned} \hat{I}(x', y') &= bilinear(I(\lfloor x' \rfloor, \lfloor y' \rfloor), I(\lfloor x' \rfloor, \lceil y' \rceil), I(\lceil x' \rceil, \lfloor y' \rfloor), I(\lceil x' \rceil, \lceil y' \rceil)) \\ M_b(x, y) &= \sum_{i=0}^{N-1} \frac{\hat{I}(x + t \cdot o_x, y + t \cdot o_y)}{N} \end{aligned} \tag{4.9}$$

Em que:

$$\begin{aligned} t &= \frac{i}{N-1}, \\ o_x &= N \cdot \cos(\theta), \\ o_y &= N \cdot \sin(\theta) \end{aligned} \tag{4.10}$$

(a) Imagem Original. (b) Após o Filtro Bilateral.

Figura 4.4: Demonstração do Filtro Bilateral.

N e θ são os parâmetros do filtro e descrevem a quantidade de blur e o ângulo de incidência do blur na imagem, respectivamente. Além disso, como o acesso à imagem de entrada é feito com coordenadas fracionárias, fazemos interpolação bilinear e clamping para manter os acessos à memória dentro da imagem. Nossa implementação em Halide usa uma redução no domínio $[0, \cdots, N-1]$.

4.4 Filtro Bilateral

Em (Paris et al. 2009), temos a seguinte explicação do Filtro Bilateral:

> The Bilateral Filter is a non-linear technique that can blur an image while respecting strong edges. Its ability to decompose an image into different scales without causing haloes after modification has made it ubiquitous in computational photography applications such as tone mapping, style transfer, relighting, and denoising.

O Filtro Bilateral (Figura 4.4) pode ser visto como um filtro gaussiano no domínio espacial e no domínio de cor, portanto um pixel é borrado apenas com seus pixels vizinhos que possuem níveis de cinza similares. Mais formalmente:

$$BF(p) = \frac{1}{W_p} \sum_q G_{\sigma_s}(||p-q||) G_{\sigma_r}(||I(q)-I(p)||) I(q) \tag{4.11}$$

Entretanto, enquanto que é possível pré-computar os valores da normal espacial nesta fórmula, não é possível fazê-lo para a normal no domínio de cor. Portanto, teríamos que calcular várias exponenciais para cada pixel na saída, o que é bastante custoso computacionalmente.

Felizmente, temos em (Paris & Durand 2006) um aproximação rápida do Filtro Bilateral chamada de Bilateral Grid, que é o algoritmo que usamos aqui. O mesmo algoritmo foi implementado no artigo original de Halide (Ragan-Kelley et al. 2012), mas foi comparado com uma implementação serial e que não fazia uso de vetorização.

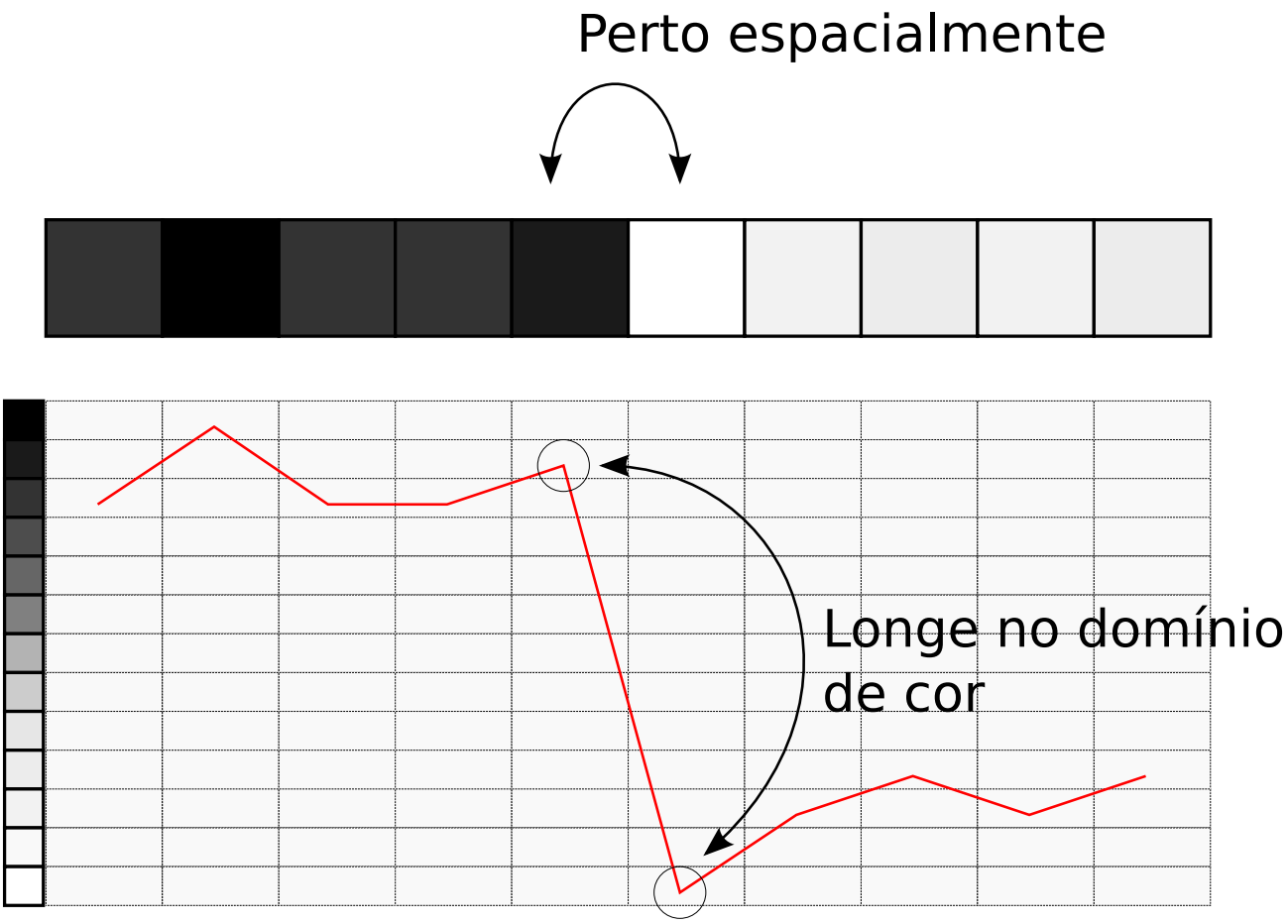


Figura 4.5: Primeiro passo do Bilateral Grid.

4.4.1 Bilateral Grid

A ideia básica por trás do Bilateral Grid é que se formos para um espaço de maior dimensionalidade, é possível executar o filtro com operações mais simples. Então temos um compromisso de maior uso de memória por menor quantidade de processamento, mas que geralmente vale a pena.

Na Figura 4.5 mostramos o primeiro passo do Bilateral Grid para um sinal unidimensional. Vemos que ao colocar o sinal em uma grade 2D, elementos com cores muito diferentes que estavam próximos espacialmente passam a estar distantes na grade. A intuição do funcionamento do algoritmo é que o filtro gaussiano borra um pixel apenas com os pixels vizinhos com nível de cinza similar.

A partir daí, são feitos os seguintes passos:

1. Subamostragem da grade no domínio espacial e de cor.
2. Filtragem gaussiana da grade subamostrado.
3. Interpolação trilinear da grade subamostrado para retornarmos às coordenadas da imagem original.

Este algoritmo é interessante pois exercita estruturas de dados em Halide diferentes da imagem bidimensional.

Neste trabalho estendemos o algoritmo para funcionar com 4 canais (RGBA) em vez de níveis de cinza e validamos a seu desempenho em Halide com uma implementação do filtro que usa *multi-threading* e vetorização na CPU.

Em Halide, temos que para executar o passo 2, temos que fazer uma subamostragem da imagem bidimensional com 4 canais em uma grade tridimensional:

```
1  RDom r(0, s_sigma, 0, s_sigma);
2
3  // subamostragem espacial
4  Expr val = clamped(x * sigma_s + r.x - sigma_s/2,
5                     y * sigma_s + r.y - sigma_s/2, c);
6
7  // subamostragem no dominio de cor
8  Expr zi = cast<int>(val * (1.0f/sigma_r) + 0.5f);
9
10 Func grid("grid");
11 grid(x, y, zi, c, k) += select(k == 0, val, 1.0f);
```

Fazendo apenas um canal e considerando $k = 0$, a redução da linha 11 é equivalente à seguinte equação:

$$grid(x, y, z) = \sum_{r_y=-\sigma_s/2}^{+\sigma_s/2} \sum_{r_x=-\sigma_s/2}^{+\sigma_s/2} V(x, y, r_x, r_y, z) \tag{4.12}$$

onde:

$$V(x, y, r_x, r_y, z) = \begin{cases} P(x, y, r_x, r_y) & \text{se } z = z_i \\ 0 & \text{caso contrário} \end{cases} \tag{4.13}$$

sendo:

$$\begin{aligned} P(x, y, r_x, r_y) &= I(x \cdot \sigma_s + r_x, y \cdot \sigma_s + r_y) \\ z_i &= \left\lceil \frac{P(x, y, r_x, r_y)}{\sigma_r} \right\rceil \end{aligned} \tag{4.14}$$

É bastante interessante que Halide tenha conseguido colocar toda a complexidade deste passo do filtro de um modo conciso. Existe uma relação entre a função "grid" e a função "histogram" da Seção 3.2: Enquanto que "histogram" é dependente dos valores de todos os pixels na imagem, temos aqui que o Bilateral Grid calcula um histograma na direção z para cada janela em torno de um ponto (x,y) na imagem de entrada. Como a computação de cada (x,y) em "grid" é independente, podemos utilizar o seguinte schedule na GPU:

```
1  grid.cudaTile(x,y,16,16);
```

Portanto, temos que, devido à estrutura particular deste algoritmo, em que os pixels na saída não possuem dependências de dados entre si, podemos paralelizá-lo facilmente. O mesmo não ocorre para a função "histogram" da Seção 3.2, como vimos.

Nossa implementação do Filtro Bilateral em OpenCL faz uso extensivo da memória local e de textura, que, como dissemos, são impossíveis de serem acessadas em Halide no momento.

4.5 Detector de Cantos de Harris

O Detector de Cantos de Harris (Harris & Stephens 1988) é um detector de pontos relevantes na imagem, i.e. pixels que se destacam em relação aos seus vizinhos e que podem ser usados em tarefas como registro de imagem e reconhecimento de objetos. O Detector de Harris é simples, mas produz bons resultados.

O algoritmo tem como entrada uma imagem em nivel de cinza e como saída uma imagem binária que diz se o detector considera aquele pixel na entrada relevante ou não. Como dissemos, podemos esperar que os pontos relevantes sejam relativamente diferentes da sua vizinhança e que um ponto correspondente será detectado se aplicarmos uma transformação afim na imagem ou em uma sequência de frames em um vídeo, por exemplo.

Como podemos ver, o Detector de Harris é um pipeline com muitos estágios intercalados. Em OpenCL, sempre tentamos fundir o máximo possível de estágios em um kernel, pois assim diminuímos a sobrecarga de leitura e escrita na memória da GPU, mas isso tem o efeito de aumentar a complexidade da implementação. Em Halide, podemos fazer cada estágio como uma função e então usar um *schedule* apropriado para fundí-los.

Neste algoritmo, vemos o comportamento de Halide com pipelines mais complexos formados por vários filtros. Na Figura 4.6 temos uma visão em alto-nível dos estágios que compõem o algoritmo.

Usamos o filtro de Sobel para calcular o gradiente em x e em y (Equação 2.2) e, a partir daí, a matriz hessiana para cada pixel:

$$H = \begin{bmatrix} S_x^2 & S_x S_y \\ S_x S_y & S_y^2 \end{bmatrix} \tag{4.15}$$

Para que a resposta do filtro seja isotrópica (i.e. o filtro seja invariante à direção), fazemos um filtro gaussiano nos valores de H_{xx}, H_{xy} e H_{yy} com uma vizinhança em torno de cada pixel, produzindo a matrix $\hat{H}$ para cada pixel.

A medida da resposta de cantos é uma função que a partir do determinante e traço da matriz hessiana, ela é derivada do fato que um ponto de interesse provavelmente deve possuir 2 grandes auto-valores em sua hessiana:

$$R = \det(\hat{H}) - \kappa \ \mathrm{trace}^2(\hat{H}) \tag{4.16}$$

O parâmetro κ ajusta a sensitividade do algoritmo. A partir daí, fazemos uma limiarização sobre a resposta dos cantos e uma supressão não-maximal, em que só os pontos com resposta

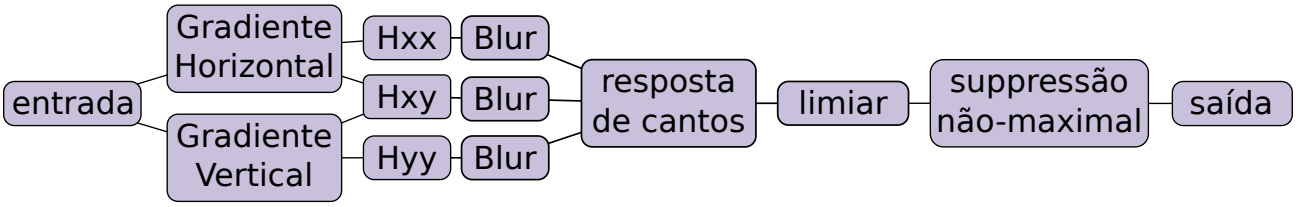


Figura 4.6: Estágios no Detector de Harris.

maior que seus vizinhos numa região 3x3 são considerados de interesse:

$$P(x,y) = \begin{cases} 1 & \text{se } R(x,y) > T \text{ e } R(x,y) \geq R(x+r_x, y+r_y) \ , \forall r_x, r_y \in [-1,+1] \times [-1,+1] \\ 0 & \text{caso contrário} \end{cases} \tag{4.17}$$

4.6 SIFT

Assim como o Detector de Harris, o SIFT (Scale-invariant feature transform) (Lowe 1999) resolve o problema de encontrar pontos relevantes numa imagem. O SIFT é um algoritmo muito utilizado no campo de processamento de imagens, portanto tê-lo funcionando em Halide é relevante.

Vemos na Figura 4.7 os estágios do SIFT. Podemos ver que é um pipeline bastante extenso e que as etapas do algoritmo possuem uma relação complexa de dependência. Mostramos na figura apenas uma *oitava* do algoritmo, geralmente o SIFT é utilizado com 4 oitavas ou mais, em que cada oitava é executada sobre a imagem de entrada subamostrada progressivamente (1x, 1/4x, 1/16x, 1/64x, e assim em diante).

Na Figura 4.7 temos que cada oitava do algoritmo começa com uma subamostragem que reduz pela metade a imagem na horizontal e vertical. Após isso, aplicamos uma série de filtros gaussianos na imagem com o objetivo de aplicar diferentes filtros passa-baixa na imagem. Na etapa de diferença de gaussianas (DdG), subtraímos uma imagem borrada da outra, que tem o efeito de um filtro passa-faixa, que é uma aproximação do laplaciano da imagem. Deste modo, capturamos pixels importantes em diferentes escalas da imagem.

O último passo que fazemos na GPU é encontrar os máximos e mínimos entre diferentes DdG. Estes pontos são considerados de interesse. A idéia é que deste modo, selecionamos apenas pontos que provavelmente serão também encontrados na mesma imagem de entrada submetida a mudanças de escala e transformações afins.

Temos então um *pipeline* composto de operações com diversos níveis de paralelismo que podem ser explorados. As etapas de subamostragem, criação da pirâmide gausiana, diferença entre gaussianas e a supressão não-maximal são operações facilmente executadas em GPU e que conseguimos expressar na linguagem OpenCL e Halide sem problemas.

Entretanto, na segunda parte do algoritmo temos uma lista de coordenadas (x,y) na imagem que corresponde aos pontos interessantes encontrados pela supressão não-maximal. A partir daí,

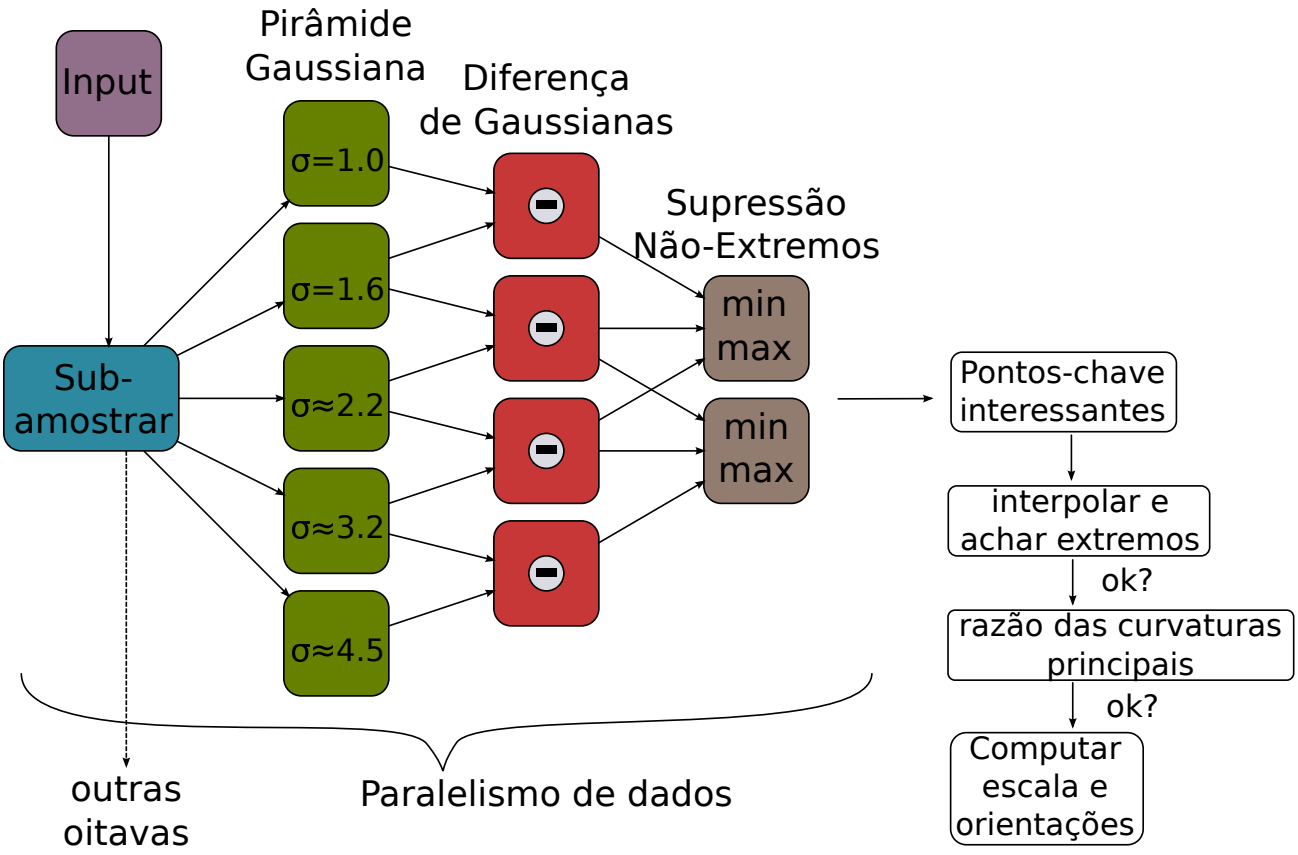


Figura 4.7: Estágios no algoritmo SIFT (uma oitava).

há vários testes para descartar pontos que não possuem boas características. Dada a natureza bastante serial desta operação, é impossível representá-la em Halide no momento. Não há como criar e processar listas de tamanho variável. Esta limitação pode ser enquadrada no princípio de que Halide é uma linguagem que tenta ser eficiente tanto na CPU quanto na GPU e que tal operação de manipulação de listas necessitaria do uso de operações atômicas em GPUs, que geralmente implicam uma grande perda de desempenho devido à serialização e complexidade no projeto do compilador.

Em nossas implementações, calculamos a interpolação de extremos e a razão das curvaturas principais para todos os pontos. Deste modo, seria possível computar o descritor SIFT com operações bastante simples após a execução das oitavas.

Portanto, em nosso estudo de caso vamos implementar os estágios que possuem paralelismo de dados e que, na realidade, constituem a maior parte do tempo de execução do algoritmo.

Podemos dizer que a grande complexidade do SIFT atingiu os limites de projeto da implementação atual da linguagem Halide. Tivemos diversos problemas com o tempo de compilação de Halide e com o uso de memória na GPU. Halide aloca a memória necessária para os resultados intermediários do pipeline e só a desaloca ao fim da execução, enquanto em nossa implementação OpenCL cada estágio do algoritmo desaloca os dados que não serão mais utilizados.

Outro ponto interessante que mostra a diferença entre nossa implementação OpenCL e Halide é que, enquanto na primeira temos um kernel para cada tipo de estágio (subamostragem, blur, etc), em Halide isso não acontece. Todas as função são *inline*, o compilador Halide substitui todas as chamadas de uma função pela sua definição e os parâmetros da determinada chamada.

Esta abordagem facilita a construção do compilador Halide e tem o potencial de fazer o código gerado ser mais rápido, pois otimizações subsequentes podem usar a informação contidas nos parâmetros. Por exemplo, uma multiplicação por um parâmetro da função que seja zero pode ser eliminada.

Porém, como o algoritmo SIFT é bastante complexo, com diversas chamadas de funções, esta abordagem leva a um grande tempo de compilação, pois é necessário aplicar otimizações subsequentes em várias cópias da mesma função e sobrecarga no driver da GPU, pois nas placas Nvidia há etapas de otimização de código que são feitas no *driver* e que dependem do modelo usado de GPU.

Capítulo 5

Resultados e Análise

Para verificar o desempenho da linguagem Halide, implementamos todos os filtros em C++, OpenCL e Halide.

A máquina utilizada para os testes tem uma CPU Intel i5 E5506 com 4 núcleos; uma GPU Nvidia Tesla C2050 com a porta PCIe 3.0 x16; compilador GCC 4.7 com a flag -Ofast e Linux Ubuntu 12.10.

Os filtros foram executados nas seguintes 5 configurações em uma imagem de 4 Megapixels (2048x2048 pixels), exceto para o SIFT, em que usamos uma imagem de 1280x1024 pixels devido aos problemas mencionados de alocação de memória em Halide.

1. **C++**: implementação tradicional em C++ usando OpenMP para multi-threading e cuidadosamente feita para utilizar a auto-vetorização do compilador C++.

2. **OpenCL (CPU)**: implementação em OpenCL executando na CPU através da biblioteca OpenCL da Intel.

3. **Halide (CPU)**: implementação em Halide com um *schedule* apropriado para a execução na CPU.

4. **OpenCL (GPU)**: mesmo código-fonte que o Item 2, mas executando na GPU com a biblioteca OpenCL e drivers da Nvidia.

5. **Halide (GPU)**: mesmo código-fonte que o Item 3, mas com um *schedule* apropriado para a GPU.

A Tabela 5.1 mostra os tempos de execução dos filtros e os *speedups* (ou *slow-downs*) sobre a implementação tradicional em C++. Os mesmos *speedups* são mostrados na Figura 5.1.

5.1 Análise

Podemos ver alguns padrões simples em nossos números de desempenho:

	C++ e OpenMP	CPU		GPU	
		OpenCL	Halide	OpenCL	Halide
CIELAB	0.6908s	0.3556s	1.0191s	0.0659s	0.0597s
Speedup	1	1.94	0.68	10.48	11.58
M. BLUR	1.3090s	0.3356s	0.5653s	0.0784s	0.0894s
Speedup	1	3.90	2.32	16.70	14.64
U. MASK.	0.7684s	0.3808s	0.4216s	0.0845s	0.0911s
Speedup	1	2.02	1.82	9.10	8.43
BIL. FILTER	0.5686s	0.5456s	0.5666s	0.0995s	0.1203s
Speedup	1	1.04	1.00	5.71	4.73
HARRIS	0.4142s	0.3779s	0.2447s	0.0880s	0.0750s
Speedup	1	1.10	1.69	4.70	5.52
SIFT	0.5942s	1.1225s	1.2976s	0.0711s	0.0913s
Speedup	1	0.53	0.46	8.36	6.51

Tabela 5.1: Comparação de desempenho entre implementações em C++, OpenCL e Halide com tempos de execução e *speedups*. Os valores correspondem ao menor tempo de execução entre 10 tentativas.

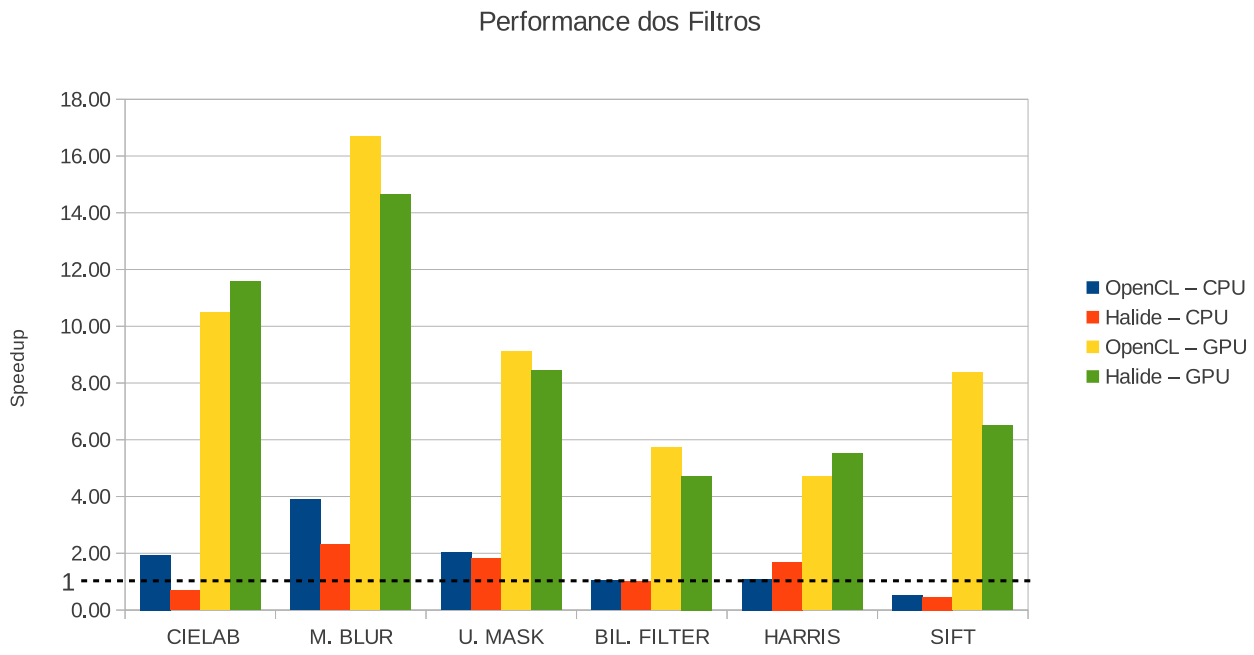


Figura 5.1: *Speedup* das implementações em OpenCL e Halide sobre a implementação em C++.

1. O uso de GPUs melhorou significativamente o desempenho em todos os nossos estudos de caso, com o *speedup* variando entre 4.7x e 16.7x quando comparado à implementação em OpenMP executando nos 4 núcleos da CPU.

2. Desempenho na GPU é similar em OpenCL e Halide.

3. Desempenho na CPU é similar em OpenCL e Halide, mas com maior variabilidade que na GPU.

4. Implementações tradicionais em C++ e OpenMP foram, em geral, mais lentas que Halide e OpenCL.

Argumentamos que a similaridade no tempo de execução entre as implementações em Halide e OpenCL na GPU podem ser explicadas pelos seguintes fatores:

- Algoritmos de processamento de imagens na GPU são geralmente limitados pela banda entre a memória da CPU para a memória da GPU através da porta PCIe.

- GPUs possuem grande capacidade de processamento em ponto-flutuante comparado à CPUs.

A razão por trás do primeiro argumento é que GPUs discretas são conectadas à placa-mãe de computadores através da porta PCI e esta possui grandes limitações de banda. Como em aplicações de processamento de imagens geralmente temos um fluxo de trabalho em que transferimos uma imagem para a memória da GPU e esta é usada apenas uma vez para seu processamento (em oposição, por exemplo, à computação gráfica, onde uma textura é utilizada diversas vezes em renderizações), essa etapa se torna o gargalo de muitos filtros. Por exemplo, mesmo usando a porta PCIe x16 3.0, temos que o Filtro Bilateral passa mais de 50% do seu tempo de execução em transferências de memória; para referência, o speedup da implementação em OpenCL do filtro sem essas transferências seria de 16.5x, em vez de 5.88x.

Se pudéssemos amortizar o *sobrecusto* dessas transferências de memória por um grande número de operações na GPU, teríamos melhores resultados. Um exemplo de fluxo de trabalho em que isso acontece é em um programa de edição de imagens onde o usuário tem que selecionar parâmetros para um filtro que é executado repetidamente sobre a mesma imagem. Porém, nessa dissertação não levaremos essa possibilidade em consideração em nossa análise.

Além disso, a grande quantidade de circuito dedicado a operações de ponto-flutuante faz com que sejam menos relevantes diferenças no código gerado pelos compiladores OpenCL e Halide.

Esses dois principais fatores equilibram o desempenho de implementações em OpenCL e Halide em GPU.

Sobre o uso de memória local e de textura, temos que o desempenho do Unsharp-Mask e do Filtro Bilateral foram praticamente similares em OpenCL e Halide, sugerindo que, no contexto de processamento de imagens, dados não ficam na GPU tempo suficiente para que o uso desse tipo de memória mais rápida faça muita diferença no tempo total de execução.

Nosso experimento de validação com o Filtro Bilateral teve *speedups* menores que o artigo original da linguagem Halide (que teve *speedups* de 6x na CPU e 42x na GPU) simplesmente porque utilizamos uma implementação na CPU em C++ que faz uso de vetorização e *multi-threading* como base para comparação.

Como vimos, o desempenho na CPU é mais sujeita à variação devido a escolhas de implementação e diferentes otimizações feitas pelos compiladores C++, OpenCL e Halide. Por exemplo, no estudo CIELAB, Motion-Blur e o Filtro Bilateral, conseguimos melhor performance na CPU

com OpenCL e Halide porque houve muitos casos de expressões complexas que o compilador GCC não foi capaz de vetorizar, enquanto que o compilador OpenCL da Intel e Halide (com o *schedule* apropriado) conseguiram com sucesso.

No estudo CIELAB, temos resultados surpreendentes na CPU, pois Halide é significativamente mais lento que a implementação em C++. Através de inspeção do código *Assembly* gerado pelos 2 compiladores, vemos que o binário gerado pelo compilador Halide é cerca de 4 vezes maior que o gerado pelo compilador GCC e que inclui muitas chamadas desnecessárias a funções transcedentais que são computacionalmente caras. Isso acontece porque o compilador Halide no momento não possui um passo de otimização para eliminar subexpressões comuns (*Common Subexpression Elimination - CSE*), portanto muitas expressões reusadas são recomputadas sem necessidade. Infelizmente, é difícil fazer CSE em *Assembly* ou representações de baixo nível, pois muita informação é perdida nas consecutivas transformações e otimizações de código.

Por outro lado, CIELAB na GPU teve desempenho parecido com OpenCL e Halide, mesmo com os problemas mencionados anteriormente também presentes. A duplicação de funções transcedentais não é um problema tão grande na GPU pois a mesma possui hardware especial que acelera bastante exponenciações, logaritmos, etc. Podemos ver que ineficiências no código gerado são mascaradas pela grande capacidade de execução de operações em ponto-flutuante de GPUs.

No Detector de Harris, temos que a implementação em OpenCL na CPU é mais lenta que a implementação tradicional em C++. Isso se dá porque fizemos uso de texturas nesse filtro, assim simplificamos o código pois não temos que nos preocupar com casos especiais de acesso fora das bordas da imagem (como acontece em convoluções). Porém, enquanto que GPUs possuem hardware específico para tratar esses acessos, CPUs x86 não o possuem; o conjunto de instruções x86 não possui uma instrução *clamp* ou uma para interpolação linear (*lerp*).

Isso mostra que enquanto é possível confiar na portabilidade de OpenCL entre diferentes plataformas, o desempenho não é garantidamente ótimo em todas elas sem mudanças na implementação, algo que Halide oferece uma boa solução através de *schedules*.

No SIFT, temos na CPU uma combinação dos problemas anteriores. Em Halide, o cálculo da interpolação de extremos e da razão das curvaturas exige uma grande quantidade de operações aritméticas com matrizes que, assim como em CIELAB, cai no problema da recomputação desnecessária de expressões. Em OpenCL na CPU, a emulação de texturas causou a baixa performance do filtro em relação à implementação C++ com OpenMP. Na GPU, essas ineficiências foram mascaradas, fazendo com que tenhamos um bom *speedup* em relação à CPU.

5.2 Análise da dificuldade de implementação

A Tabela 5.2 mostra o número de linhas de código de cada implementação, como uma aproximação de sua dificuldade de implementação. Não incluímos na Tabela 5.2 em OpenCL a quantidade de código para inicializar a GPU e compilar os kernels, mas incluímos o gerenciamento de memória na GPU e chamadas a kernels. As valores de Halide incluem o algoritmo e *schedule* para CPU e GPU.

Tivemos o cuidado de que as implementações feitas nessa dissertação sejam escritas do modo mais eficiente em cada respectiva linguagem, mas com o cuidado de não se sacrificar a legibilidade. Por exemplo, em nossas implementações C++ poderíamos ter feito o uso de *intrisics* para gerar diretamente instruções vetoriais em linguagem de máquina nos casos que a auto-vetorização não funcionou, mas não o fizemos devido ao impacto negativo na legibilidade dessa abordagem.

Podemos ver que implementações em Halide são bem menores que as contrapartidas em C++ e OpenCL. Notamos especialmente a grande diferença em filtros mais complexos como o Filtro Bilateral em OpenCL comparado à implementação em Halide.

	C++ e OpenMP	OpenCL	Halide
CIELAB	61	77	33
U. MASK	59	134	37
M. BLUR	48	83	40
BIL. FILTER	138	365	45
HARRIS	196	209	104
SIFT	222	421	261

Tabela 5.2: Número de linhas de código para cada implementação.

Mesmo considerando que o número de linhas de código é uma aproximação simples para a real complexidade de uma implementação, a grande diferença apresentada torna claro o ponto de que é mais fácil implementar filtros de processamento de imagens que funcionam em CPU e GPU com Halide do que com as alternativas aqui apresentadas.

A metologia usada no trabalho para contagem de linhas de código foi:

- Excluímos linhas contendo apenas comentários.
- Não incluímos o cabeçalho da função de entrada de cada filtro.
- Incluímos linhas em branco, pois elas são necessárias para a leitura do código.

No Anexo A temos o código completo do filtro Motion-Blur em C++, OpenCL e Halide para comparação. Também incluímos o código para inicialização da GPU em OpenCL e compilação do kernel, que pode ser reusado em diferentes filtros. Todo o código utilizado nessa dissertação pode ser encontrado em https://github.com/victormatheus/halide-casestudies.

Nossa experiência é que portar código C++ para Halide é muito mais fácil do que fazer o mesmo para OpenCL. Não é necessária a preocupação com detalhes como gerenciamento de memória na GPU, compilação de kernels, entre outros detalhes de baixo nível, e podemos nos focar apenas em como expressar o algoritmo de modo que o paralelismo de dados fique claro.

Entretanto, não há no momento ferramentas confiáveis para depuração de código na GPU . O problema é que numa arquitetura massivamente paralela, em que não se sabe a ordem de execução das diversas *threads*, não há como reproduzir bugs de maneira determinística. Esse problema se manifesta tanto em Halide quanto OpenCL, pois embora seja possível executar as

duas linguagens também na CPU, muitos bugs só se apresentam em uma arquitetura e não em outra.

Halide possui a vantagem de limitar erros como acessos fora da imagem através de sua sintaxe, mas por ser uma linguagem ainda experimental, seu compilador possui bugs e permite schedules inválidos, como permitir a vetorização por um tamanho inválido, entre outros. Erros em compiladores introduzem bugs extremamente difíceis de serem encontrados.

Sobre *schedules*, temos que é essencial entender como eles alteram a geração de código por Halide. Podemos ver isso no estudo CIELAB, em que temos 3 canais. Usar vetorização entre canais é inefetivo, pois cada canal tem computações distintas, como podemos ver na Seção 4.1.2. Portanto, não há operações similares para colocar em instruções vetorizadas. A melhor forma de implementar o algoritmo é usar os canais não-intercalados e fazer a vetorização na direção x para cada um dos canais. Para isso, temos que entender como um schedule Halide ordena a imagem na memória, analisando a representação intermediária de Halide para o schedule (Halide IR).

Capítulo 6

Conclusão e Trabalhos Futuros

Programação de GPUs é radicalmente diferente dos modelos tradicionais de programação para CPU. É difícil para um programador ou pesquisador na área de imagens absorver os conceitos de programação paralela nestas arquiteturas. Isso pode levar a um código ineficiente e com *bugs*. Halide e linguagens similares fazem parte da ideia que é a de restringir o que pode ser feito pelo programador apenas ao que pode ser mapeado eficientemente no hardware. Temos então uma linguagem bem mais fácil de ser aprendida pelo programador sem experiência em otimização de código, mas com escopo de aplicação limitado.

Nesta dissertação, estudamos a viabilidade do uso de Linguagens de Domínio-Específico para processamento de imagens em arquiteturas CPU multi-core e GPU através de Halide. Mostramos a sintaxe de Halide e discutimos algumas limitações teóricas da linguagem. Fizemos comparações de desempenho e complexidade de implementação com 6 casos típicos que exploram diferentes padrões de implementação de algoritmos na área. Implementamos estes algoritmos em C++ com OpenMP, OpenCL e Halide e vimos as vantagens e desvantagens de Halide em relação a estas duas alternativas mais tradicionais.

Podemos ver que existem diferentes limitações nas formas tradicionais de programação de alto desempenho. Vimos que a auto-vetorização feita pelo compilador GCC não é capaz de explorar todas as oportunidades de otimização e que OpenCL, embora seja um padrão portável, exige que o código seja reescrito para tirar proveito de uma arquitetura, como vimos com os problemas de desempenho causados pelo uso de memória compartilhada e de textura na CPU.

Halide soluciona este problema de portabilidade de performance entre arquiteturas através de *schedules*, que de fato ajudaram bastante na exploração rápida de diferentes otimizações para nossos estudos de caso.

Vimos que Halide foi capaz de expressar algoritmos com diferentes estruturas de dados, como o Bilateral Grid, e pipelines bastante complexos, como o SIFT. Ao mesmo tempo, a limitação de uma saída por função e a impossibilidade de efeitos colaterais impede a implementação de alguns algoritmos, como vimos com o operador de Sobel.

Halide obteve um bom desempenho na CPU comparado com OpenCL e C++, mas teve problemas nos estudos CIELAB e SIFT, onde Halide muitas vezes recomputou desnecessariamente expressões similares, o que é problemático para estes filtros que possuem grande complexidade aritmética.

No entanto, Halide mostrou um bom desempenho na GPU comparado a OpenCL para todos os estudos de caso. Fatores como limitação do desempenho pela banda da porta PCIe e grande capacidade de processamento em ponto-flutuante do hardware fazem com que diferenças entre as linguagens sejam menos relevantes.

Além disso, todos os nossos estudos de caso mostraram que a complexidade do código Halide é bastante inferior ao código OpenCL equivalente.

Como esperado, escrever bons *schedules* é difícil. Fazê-lo requer não apenas conhecimento da arquitetura subjacente, mas também como Halide funciona internamente e como os *schedules* alteram o código gerado, um esforço similar ao que fizemos no Capítulo 3 com o operador de Sobel. Como estamos desenvolvendo aplicações onde desempenho é essencial, ter um modelo mental do funcionamento interno de Halide se torna necessário. Isso não é uma tarefa trivial no momento porque não há documentação suficiente a respeito, portanto tivemos que recorrer a ler o código-fonte de Halide e seus logs ao compilar nossos programas.

Além dos problemas de otimização mencionados, também tivemos alguns problemas durante o desenvolvimento de nossos estudos de caso porque o compilador Halide ainda é um esforço de pesquisa e como tal, possui *bugs*, especialmente em *schedules* complexos, alocação de memória e tempo de compilação, como vimos no SIFT. Entender como Halide funciona internamente foi necessário para compreender a causa destes problemas.

Temos em um trabalho recente do mesmo grupo que desenvolve Halide, um modo de encontrar automaticamente *schedules* através de algoritmos genéticos (Ragan-Kelley, Barnes, Adams, Paris, Durand & Amarasinghe 2013). Não tentamos neste trabalho esta abordagem, mas encontrar *schedules* automaticamente parece ser uma boa idéia especialmente para GPUs, pois muitas da transformações de Halide como vetorização não se aplicam à arquitetura. Em geral, *schedules* para GPU são relativamente simples de serem decididos, ficando apenas a escolha de quais funções devem ser avaliadas e guardadas na memória, quais devem ser recomputadas sob demanda (políticas *root* e *inline*) e o tamanho do bloco utilizado.

O autor deste trabalho tem trabalhado com a linguagem CUDA e OpenCL quase desde o começo da tecnologia e pode dizer que Halide de fato tem um impacto positivo na experiência de programação para processamento de imagens. A separação entre a especificação (que é similar a pseudo-código) e o *schedule* faz com que se pense primeiro em como mapear o problema em construções da linguagem que favorecem o paralelismo de dados. A habilidade de rapidamente verificar se o código implementado produz resultados corretos e a partir daí melhorar gradativamente os *schedules* é muito útil.

Portanto, temos que o uso de DSLs em processamento de imagens, Halide em particupar, é definitivamente promissor e pode ajudar bastante no uso eficiente de hardware paralelo, que está se tornando cada vez mais comum e necessário para um bom desempenho de algoritmos da área.

Outra contribuição deste trabalho foi que reportamos aos autores de Halide e corrigimos diversos bugs no compilador para GPUs. Além disso, a implementação do algoritmo *Bilateral Grid* mostrada aqui foi integrada no *software* GIMP, onde ela será usada para remoção de ruído e *tone-mapping*.

Recomendamos a implementação da otimização de *Common Subexpression Elimination* em

Halide, para evitar o problema da recomputação de expressões similares, e de um sistema de gerenciamento de memória em que resultados intermediários sejam desalocados conforme seja possível. Caso uma comunidade se forme em torno da linguagem é também esperado que haja maior documentação.

Um campo interessante para pesquisas futuras é o uso de Halide para *computação heterogênea*, em que temos a CPU e GPU trabalhando simultaneamente para resolver um problema. O surgimento de arquiteturas como APUs da AMD, que integram CPU e GPU no mesmo *die* com compartilhamento de memória, são uma ótima plataforma para este tipo de desenvolvimento. Além disso, a maior parte dos *systems on a chip* para *smartphones* e *tablets* também possui estas características.

Vimos também que foi possível expressar com sucesso em Halide muitos algoritmos da área de manipulação de imagens. Consideramos o resultado deste trabalho bastante positivo sobre o uso de Halide e outras DSLs no software GIMP e é algo que pode ser explorado posteriormente.

Bibliografia

Adobe (2013). *Pixel Bender Developer's Guide*.

Allen, R. & Kennedy, K. (2002). *Optimizing compilers for modern architectures*, Morgan Kaufmann.

Apple (2013). *Core Image Programming Guide*.

Bailey, D., Masters, I. & Warner, M. (2011). Gpu fluids in production: a compiler approach to parallelism, *ACM Transactions on Graphics Talks*, pp. 4:1–4:1.

Bunks, C. (2000). *Grokking the Gimp*, New Riders Publishing, Thousand Oaks, CA, USA.

Dower, S. (2012). Automatic implementation of evolutionary algorithms on gpus using esdl, *IEEE Congress on Evolutionary Computation*, pp. 1–8.

Flynn, M. (1972). Some computer organizations and their effectiveness, *IEEE Transactions on Computers* **C-21**(9): 948–960.

Guenter, B. & Nehab, D. (2010). Neon: A domain-specific programming language for image processing, *Microsoft TechReport MSR-TR-2010-175, Microsoft Research* .

Harris, C. & Stephens, M. (1988). A combined corner and edge detector, *Alvey Vision Conference*, pp. 1–6.

Khronos Group (2010). *The OpenCL Specification*.

Kirk, D. B. & Hwu, W. W. (2010). *Programming Massively Parallel Processors: A Hands-on Approach*, 1st edn, Morgan Kaufmann Publishers Inc.

Lattner, C. & V., A. (2004). LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation, *International Symposium on Code Generation and Optimization*.

Lowe, D. (1999). Object recognition from local scale-invariant features, *International Conference on Computer Vision*, Vol. 2, pp. 1150–1157 vol.2.

Mainland, G. & Morrisett, G. (2010). Nikola: embedding compiled gpu functions in haskell, *ACM Symposium on Haskell*, pp. 67–78.

Munshi, A., Gaster, B., Mattson, T. G., Fung, J. & Ginsburg, D. (2011). *OpenCL Programming Guide*, 1 edn, Addison-Wesley Professional.

NVIDIA (2008). Parallel thread execution, *ISA Version* **1**.

NVIDIA (2011). *NVIDIA CUDA programming guide*.

Owens, J. D., Luebke, D., Govindaraju, N., Harris, M., Krüger, J., Lefohn, A. & Purcell, T. J. (2007). A survey of general-purpose computation on graphics hardware, *Computer Graphics Forum* **26**(1): 80–113.

Paris, S. & Durand, F. (2006). A fast approximation of the bilateral filter using a signal processing approach, *ACM Transactions on Graphics Talks*, pp. 568–580.

Paris, S., Kornprobst, P. & Tumblin, J. (2009). *Bilateral filtering: Theory and applications*, Vol. 1, Now Publishers Inc.

Ragan-Kelley, J., Adams, A., Paris, S., Levoy, M., Amarasinghe, S. & Durand, F. (2012). Decoupling algorithms from schedules for easy optimization of image processing pipelines, *ACM Transactions on Graphics* pp. 32:1–32:12.

Ragan-Kelley, J., Barnes, C., Adams, A., Paris, S., Durand, F. & Amarasinghe, S. (2013). Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines, *ACM SIGPLAN conference on Programming Language Design and Implementation*, ACM.

Shantzis, M. A. (1994). A model for efficient and flexible image computing, *ACM Transactions on Graphics*, pp. 147–154.

Stone, J., Gohara, D. & Guochun, S. (2010). Opencl: A parallel programming standard for heterogeneous computing systems, *Computing in Science Engineering* **12**(3): 66–73.

Apêndice A

Código completo do filtro Motion-Blur em C++, OpenCL e Halide

A.1 C++ com OpenMP

```
1  void
2  motion_blur (float const * __restrict__ _input,
3               float       * __restrict__ _output,
4               int    width,
5               int    height,
6               float  length,
7               float  angle)
8  {
9    const int channels = 4;
10
11   float theta = angle * M_PI / 180.0;
12   float offset_x = fabs(length * cos(theta));
13   float offset_y = fabs(length * sin(theta));
14   int num_steps = ceil(length) + 1;
15
16   float * __restrict__ input  =
17     (float * __restrict__) __builtin_assume_aligned(_input,  32);
18
19   float * __restrict__ output =
20     (float * __restrict__) __builtin_assume_aligned(_output, 32);
21
22   #define INPUT(x,y,c) input[c+channels*(x + width * y)]
23
24   #pragma omp parallel for
25   for (int y=0; y < height; y++)
26     for (int x=0; x < width; x++)
27       for(int c = 0; c < channels; c++)
28         {
29           float sum = 0.0;
30
31           for(int step = 0; step < num_steps; step++)
32             {
33               float t = num_steps == 1 ? 0.0f :
34                   step / (float)(num_steps - 1) - 0.5f;
```

```
35
36              float xx = x + t * offset_x;
37              float yy = y + t * offset_y;
38
39              int ix = (int)(xx);
40              int iy = (int)(yy);
41
42              float dx = xx - ix;
43              float dy = yy - iy;
44
45              float mixy0, mixy1, pix0, pix1, pix2, pix3;
46
47              pix0 = INPUT(clamp(ix,   0, width-1), clamp(iy,   0, height-1), c);
48              pix1 = INPUT(clamp(ix+1, 0, width-1), clamp(iy,   0, height-1), c);
49              pix2 = INPUT(clamp(ix,   0, width-1), clamp(iy+1, 0, height-1), c);
50              pix3 = INPUT(clamp(ix+1, 0, width-1), clamp(iy+1, 0, height-1), c);
51
52              mixy0 = dy * (pix2 - pix0) + pix0;
53              mixy1 = dy * (pix3 - pix1) + pix1;
54
55              sum  += dx * (mixy1 - mixy0) + mixy0;
56            }
57
58          output[c+channels*(x + y * width)] = sum / num_steps;
59        }
60  }
```

A.2 OpenCL

A.2.1 Código OpenCL (GPU)

```
1   float4 get_pixel_color(const __global float4 *in_buf,
2                          int     width,
3                          int     height,
4                          int     x,
5                          int     y)
6   {
7       int ix = clamp(x, 0, width-1);
8       int iy = clamp(y, 0, height-1);
9
10      return in_buf[iy * width + ix];
11  }
12
13  __kernel void motion_blur(__global const float4 *src_buf,
14                            __global       float4 *dst_buf,
15                            int     width,
16                            int     height,
17                            int     num_steps,
18                            float   offset_x,
19                            float   offset_y)
20  {
21      int gidx = get_global_id(0);
22      int gidy = get_global_id(1);
23
```

```
    float4 sum = 0.0f;

    for(int step = 0; step < num_steps; step++)
    {
        float t = num_steps == 1 ? 0.0f :
            step / (float)(num_steps - 1) - 0.5f;

        float xx = gidx + t * offset_x;
        float yy = gidy + t * offset_y;

        int   ix = (int)floor(xx);
        int   iy = (int)floor(yy);

        float dx = xx - floor(xx);
        float dy = yy - floor(yy);

        float4 mixy0,mixy1,pix0,pix1,pix2,pix3;

        pix0 = get_pixel_color(src_buf, width, height, ix,   iy);
        pix1 = get_pixel_color(src_buf, width, height, ix+1, iy);
        pix2 = get_pixel_color(src_buf, width, height, ix,   iy+1);
        pix3 = get_pixel_color(src_buf, width, height, ix+1, iy+1);

        mixy0 = dy * (pix2 - pix0) + pix0;
        mixy1 = dy * (pix3 - pix1) + pix1;

        sum  += dx * (mixy1 - mixy0) + mixy0;
    }

    dst_buf[gidy * width + gidx] = sum / num_steps;
}
```

A.2.2 Código na máquina hospedeira (CPU)

```
static cl_context       context;
static cl_platform_id   platform;
static cl_device_id     device;
static cl_command_queue command_queue;
static cl_kernel        kernel;

static const char* program_source =
  /* codigo-fonte opencl do filtro motion-blur */ ;

void
motion_blur_cl_prepare()
{
  clGetPlatformIDs (1, &platform, NULL);
  clGetDeviceIDs (platform, CL_DEVICE_TYPE_GPU, 1, &device, NULL);
  context = clCreateContext(0, 1, &device, NULL, NULL, NULL);
  command_queue = clCreateCommandQueue(context, device, 0, NULL);

  size_t length = strlen(program_source);

  cl_program program = clCreateProgramWithSource(context, 1,
```

```
                                                &program_source,
                                                &length,
                                                NULL);

  clBuildProgram(program, 0, NULL, NULL, NULL, NULL);

  kernel = clCreateKernel(program, "motion_blur", NULL);
}

void
motion_blur_cl (cl_mem input,
                cl_mem output,
                int    width,
                int    height,
                float  length,
                float  angle)
{
  float theta = angle * M_PI / 180.0;
  float offset_x = length * cos(theta);
  float offset_y = length * sin(theta);
  int num_steps = ceil(length) + 1;

  size_t global_ws[2];

  {
  global_ws[0] = width;
  global_ws[1] = height;

  clSetKernelArg(kernel, 0, sizeof(cl_mem),   & intput);
  clSetKernelArg(kernel, 1, sizeof(cl_mem),   & output);
  clSetKernelArg(kernel, 2, sizeof(cl_int),   & width);
  clSetKernelArg(kernel, 3, sizeof(cl_int),   & height);
  clSetKernelArg(kernel, 4, sizeof(cl_int),   & num_steps);
  clSetKernelArg(kernel, 5, sizeof(cl_float), & offset_x);
  clSetKernelArg(kernel, 6, sizeof(cl_float), & offset_y);

  clEnqueueNDRangeKernel(command_queue,
                         kernel, 2,
                         NULL, global_ws, NULL,
                         0, NULL, NULL);
  }
}
```

A.3 Halide

```
Func acc_mb("acc_mb"), output("output");

float theta = angle * (float)M_PI / 180.0f;
float offset_x = length * std::cos(theta);
float offset_y = length * std::sin(theta);
int   num_steps = (int)(length+0.5f) + 1;

Var x("x"), y("y"), c("c");

```

```
RDom step(0, num_steps);

Expr t = (num_steps == 1)? 0.0f : step / (float)(num_steps - 1) - 0.5f;

Expr xx = x + t * offset_x;
Expr yy = y + t * offset_y;

Expr dx = xx - floor(xx);
Expr dy = yy - floor(yy);

Expr ix = cast<int>(xx);
Expr iy = cast<int>(yy);

Expr mixy0 = dy * (clamped(ix,  iy+1,c) - clamped(ix,  iy,c)) + clamped(ix,  iy,c);
Expr mixy1 = dy * (clamped(ix+1,iy+1,c) - clamped(ix+1,iy,c)) + clamped(ix+1,iy,c);

acc_mb(x,y,c) += dx * (mixy1 - mixy0) + mixy0;

motion_blur(x,y,c) = acc_mb(x,y,c) / float(num_steps);

if (use_gpu())
  {
    acc_mb.reorder(c,x,y).root().cudaTile(x,y,16,16);
    acc_mb.update().reorder(step,c,x,y).root().cudaTile(x,y,16,16);
    motion_blur.reorder(c,x,y).root().cudaTile(x,y,16,16);
  }
else
  {
    motion_blur.root().reorder(c,x,y).parallel(y).unroll(c,4).vectorize(x, 4);
    acc_mb.update().reorder(c,x,y).parallel(y).unroll(c,4).vectorize(x, 4);
  }

Image<float> out (input.width(), input.height(), input.channels());

motion_blur.realize(out);
```