TONY ALEXANDER HILD

OTIMIZAÇÃO DO *INSIGHT SEGMENTATION AND REGISTRATION TOOLKIT* (ITK) UTILIZANDO *STREAMING SIMD EXTENSIONS* (SSE) e *OPENMP*

CURITIBA

2012

TONY ALEXANDER HILD

OTIMIZAÇÃO DO *INSIGHT SEGMENTATION AND REGISTRATION TOOLKIT* (ITK) UTILIZANDO *STREAMING SIMD EXTENSIONS* (SSE) e *OPENMP*

Dissertação de Mestrado apresentada ao Programa de Pós-Graduação em Informática, Setor de Ciências Exatas, Universidade Federal do Paraná.

Orientador: Prof. Dr. Daniel Weingaertner

CURITIBA

2012

Termo de Aprovação

TONY ALEXANDER HILD

OTIMIZAÇÃO DO *INSIGHT SEGMENTATION AND REGISTRATION TOOLKIT* (ITK) UTILIZANDO *STREAMING SIMD EXTENSIONS* (SSE) e *OPENMP*

Dissertação aprovada como requisito parcial para obtenção do grau de Mestre em Informática, pelo Programa de Pós-Graduação em Informática, Setor de Ciências Exatas, Universidade Federal do Paraná pela seguinte banca examinadora:

Prof. Dr. Daniel Weingaertner
Universidade Federal do Paraná

Prof. Dr. Roberto Hexel
Universidade Federal do Paraná

Prof. Dr. Carlos A. Maziero
Universidade Técnológica Federal do Paraná

Curitiba, 23 de fevereiro de 2012

Dedicatória

Dedico este trabalho ao meu amor, Graciane da Silva, que com paciência, carinho e compreensão ajudou-me a superar os desafios.

A meus queridos pais, Antonio e Therezinha, que me ensinaram os melhores valores e me oportunizaram uma vida digna.

Aos meus queridos irmãos, Giancarlo e Nádia, pela amizade e incentivo.

Agradecimentos

Agradeço em especial ao meu orientador, Prof. Daniel Weingaertner, por muito me ensiar, por acreditar no meu trabalho e entender os desafios que enfrentei.

Aos professores da UFPR e aos meus colegas do VRI (Grupo Visão, Robótica, Imagens).

A todos meus colegas da Unicentro, em especial ao Prof. Fábio Hernandes, pelo incentivo.

Aos meus amigos, pelo apoio e incentivo.

Ao Programa de Apoio a Planos de Reestruturação e Expansão das Universidades Federais (Reuni).

Enfim, agradeço a todos que de forma direta e/ou indireta contribuiram para a realização deste trabalho

Epígrafe

"Transforme as pedras que você tropeça nas pedras de sua escada"

Sócrates

Conteúdo

Lista de Siglas

AMD *Advanced Micro Devices*. 6

AoS *Array of Structures*. 18, 19

API *Application Programming Interface*. 9

ARM *Advanced RISC Machine*. 6

ASIC *Application Specific Integrated Circuit*. 1

AVX *Advanced Vector Extensions*. 6

CPU *Central Processing Unit*. 1, 5, 11, 29, 60

DRAM *Dynamic Random-Access Memory*. 11, 12, 15

FB-DIMM *Fully Buffered-Dual Inline Memory Model*. 15

FPGA *Field-Programmable Gate Array*. 2

FSB *Front-Side Bus*. 14, 15

GPGPU *General-Purpose computation on Graphics Processing Unit*. 1

GPU *Graphics Processing Unit*. 5

ILP *Instruction Level Paralelism*. 5

ITK *Insight Segmentation and Registration Toolkit*. xviii, xix, 2, 3, 26–28, 30, 31, 37

MIMD *Multiple Instruction, Multiple Data*. xviii, xix, 4, 5, 9

NUMA *Non-uniform memory access*. 14

OpenMP *Open Multi-Processing*. xviii, xix, 2, 9, 10, 29, 37

RAM *Random-Access Memory*. 11

SIMD *Single Instruction, Multiple Data*. xviii, xix, 2, 4–7, 18, 19, 22, 29

SoA *Structure of Arrays*. 18, 19

SRAM *Static Random-Access Memory*. 11, 12

SSE *Streaming SIMD Extensions*. xviii, xix, 2, 6, 7, 20, 21, 29, 37, 38, 40, 41, 43–45

TLB *Translation Lookaside Buffer*. 17

Lista de Figuras

Lista de Tabelas

Lista de Listagens

Lista de Algoritmos

Resumo

O processamento de imagens alcançou uma importância muito grande em um mundo que depende amplamente de estímulos visuais. Partindo desta premissa, implementações eficientes de algoritmos de processamento de imagens são necessárias para aumentar a produtividade em diversas áreas, como na medicina, indústria, e entretenimento. Para tal, este trabalho estudou técnicas de otimização *Multiple Instruction, Multiple Data* (MIMD) com a utilização de *Open Multi-Processing* (OpenMP), e *Single Instruction, Multiple Data* (SIMD), com a utilização de *Streaming SIMD Extensions* (SSE), e implementou tais técnicas em algoritmos de convolução bem como no algoritmo de detecção de bordas Canny, avaliando o desempenho de ambos. Além disso, foram propostas alterações à biblioteca *Insight Segmentation and Registration Toolkit* (ITK) para que outros filtros possam beneficiar-se das otimizações estudadas. Obteve-se ganho de desempenho de até 7 vezes nas implementações dos algoritmos de convolução, e de 16 vezes no algoritmo de detecção de bordas Canny. Conclui-se que ainda é possível aumentar o ganho de desempenho otimizando a implementação de algoritmos para serem executados em processadores existentes, com o custo de aumento da complexidade de codificação, e diminuição da portabilidade de código.

Palavras-chave: MIMD, SIMD; SSE; OpenMP; Processamento de Imagens; ITK; Algoritmos de Convolução; Detecção de Bordas; Canny.

Abstract

The image processing has achieved great importance in a world that depends largely on visual stimuli. On this assumption, efficient image processing algorithms implementations are needed to increase productivity in various fields such as medicine, industry and entertainment. For that, this work studied MIMD optimization techniques with the use of OpenMP and SIMD, with the use of SSE, and implemented these techniques in convolution algorithms, and Canny edge detection algorithm, evaluating their performance. In addition, was proposed changes to the ITK library so that others filters may benefit from the studied optimizations. Was obtained a performance gain up to 7 times in the convolution algorithms implementations and 16 times in the Canny edge detection algorithm. It was concluded that still can achieve performance gains by optimizing algorithms to run on existing processors, with the cost of increased complexity of coding, and reduction of code portability.

Key-words: MIMD, SIMD; SSE; OpenMP; Image Processing; ITK; Convolution Algorithms; Edge Detection; Canny.

1 Introdução

O processamento de imagens alcançou uma importância muito grande em um mundo que depende amplamente de estímulos visuais. Hoje, principalmente na indústria, entretenimento e medicina, o processamento de imagens é de suma importância, e cada vez mais busca-se eficiência para a diminuição do tempo de processamento.

Na indústria, o processamento de imagens pode ser utilizado para detecção de defeitos e controle de qualidade. Isto quer dizer que, dependendo do processo industrial, quanto mais rápido se der o controle de qualidade utilizando processamento de imagem, menor será o tempo total do processo industrial. Em outro segmento, no entretenimento, aplicações de realidade aumentada são capazes de criar objetos virtuais e fazê-los interagir com o mundo real, em tempo real. Na medicina, o processamento de imagens ajuda a detectar doenças e fazer diagnósticos mais precisos.

Entretanto, o processamento de imagens demanda uma grande capacidade computacional, sendo necessária a utilização de processadores que suportem computação paralela, além de técnicas de paralelização e otimização das implementações dos algoritmos utilizados no processamento de imagens. Mas isto está longe de ser problema fácil de se resolver.

Para obedecer a lei de Moore (MOORE, 2000) e manter aceitável o nível de consumo de energia, os fabricantes de processadores migraram de uma arquitetura *single-core* com altas frequências (*clock*) e alto consumo de energia, para uma arquitetura *multi-core*, que fornece desempenho por meio de processamento paralelo, com frequências menores e menor consumo de energia. As arquiteturas *multi-core* são bastante variadas, e incluem as conhecidas *Central Processing Unit*s (CPUs) de propósito geral, além de *General-Purpose computation*

*on Graphics Processing Unit*s (GPGPUs), *Application Specific Integrated Circuit*s (ASICs) e *Field-Programmable Gate Array*s (FPGAs) (BLAKE; DRESLINSKI; MUDGE, 2009; ASANOVIC et al., 2006).

O grande problema das arquiteturas paralelas ou *multi-core*, além da complexidade de projetá-las, está no modelo de programação que deve ser adotado para tirar proveito do poder computacional disponível. Até meados da década de 2000, a grande maioria das aplicações eram escritas para serem processadas de forma sequencial, em um único núcleo de processamento. Sendo assim, para aumentar seu desempenho bastava aumentar a frequência do processador, o famoso *"free lunch"* de Herb Sutter (SUTTER, 2005). Hoje, estas aplicações devem ser reescritas de forma paralelizada para haver ganho de desempenho (BLAKE; DRESLINSKI; MUDGE, 2009; ASANOVIC et al., 2006).

Para obter ganhos de desempenho satisfatórios, não basta apenas executar o programa em um processador com vários núcleos, outros problemas mais sutís devem ser observados, como a gerência de memória e a hierarquia de memória. A evolução das arquiteturas de memória não se deu na mesma velocidade da evolução dos processadores. Hoje, o acesso a memória chega a ter latências de centenas de ciclos de *clock*, tornando-se um dos principais gargalos na otimização (SUTTER, 2007; SUTTER, 2006; ALTED, 2010; DREPPER, 2007).

Em face a estas mudanças bruscas de paradigmas são necessárias novas ferramentas e abordagens que auxiliem o programador a extrair todo o potencial das diferentes arquiteturas. Algumas destas ferramentas estão disponíveis há alguns anos, mas somente agora estão tornando-se comuns e algumas abordagens ainda não são conhecidas da maioria dos programadores (BRAUNL et al., 2001; KIM; BOND, 2009).

1.1 Objetivos

O objetivo principal deste trabalho é propor alterações à biblioteca de processamento de imagens *Insight Segmentation and Registration Toolkit* (ITK) que permitam otimizar seu motor de processamento utilizando extensões *Single Instruction, Multiple Data* (SIMD), espe-

cialmente *Streaming SIMD Extensions* (SSE), e *Open Multi-Processing* (OpenMP). Para tal, será implementado o algoritmo para detecção de bordas Canny para ITK, testando-se várias abordagens.

Como objetivos secundários pode-se citar:

- Revisão bibliográfica sobre processamento de imagens, técnicas de paralelização e arquiteturas de hardware;
- Análise da utilização de memória pela implementação do algoritmo, principalmente da memória *cache*;
- Avaliação da latência das instruções *assembly* nas plataformas Core i7, Opteron e Core 2, e como isto afeta a performance geral da implementação;
- Implementação de um algoritmo de convolução otimizado; e
- Avaliação e comparação da performance entre as implementações dos algoritmos e entre os processadores testados.

1.2 Divisão do Trabalho

O presente trabalho está dividido da seguinte forma: o Capítulo 2 contém a revisão bibliográfica sobre processamento de imagens, arquitetura de hardware, técnicas de paralelismo e técnicas de otimização; o Capítulo 3 contém a análise dos algoritmos desenvolvidos, passo-a-passo; no Capítulo 4 encontram-se as plataformas e metodologias de testes; no Capítulo 5 são apresentados os resultados dos testes realizados e discutidos os pontos onde há problemas e possíveis melhorias; e, finalmente, no Capítulo 6 é feita a avaliação geral do trabalho e de sua contribuição.

2 Revisão Bibliográfica

Neste capítulo são descritos os conceitos teóricos utilizados para o desenvolvimento do trabalho.

2.1 Computação paralela

Computação paralela ou processamento paralelo, é o ato de realizar cálculos divididos em tarefas menores de forma simultânea ou concorrente para resolver algum problema. Existem diversos tipos de computação paralela como: de dados, tarefas e em nível de instrução (KIM; BOND, 2009; ALMASI; GOTTLIEB, 1989).

Segundo a taxonomia de Flynn (FLYNN, 1972), a organização do processamento paralelo pode ser dividido em vários modelos, dentre eles o SIMD e o *Multiple Instruction, Multiple Data* (MIMD).

No modelo SIMD cada elemento de processamento executa a mesma instrução em localidades diferentes de um conjunto de dados de forma síncrona. Este modelo tem a vantagem de ser menos complexo com relação à sincronização das tarefas, visto que existe um controlador central do fluxo. O modelo SIMD sofre com problemas de comunicação entre elementos de processamento, com acesso à memória, problemas de *gathering*/*scattering* (GRAMA et al., 2003), mas seu principal problema é a degradação do desempenho devido a desvios condicionais. Neste caso, alguns elementos de processamento estarão em um estado, enquanto outros permanecerão ociosos, pois o controlador central pode gerenciar apenas um estado por vez. Como a arquitetura dos processadores SIMD é mais simples que a dos processadores MIMD,

há maior densidade de transistores por CPU. Isto permite construir processadores massivamente paralelos com centenas de processadores de fluxo ou mais. O modelo SIMD está presente em processadores vetoriais, *Graphics Processing Unit*s (GPUs), e até mesmo em instruções especiais em CPUs (vistas adiante) (BRAUNL et al., 2001; KIM; BOND, 2009; FLYNN, 1972).

O modelo MIMD, por sua vez, é totalmente assíncrono e cada processador possui seu próprio controle de fluxo que executa seus próprios programas. Além disso, os processadores executam instruções diferentes em conjuntos de dados diferentes, e podem compartilhar recursos com outros processadores. Este modelo sofre com alguns problemas como custo adicional na comunicação entre os processadores e o compartilhamento de recursos, podendo impedir um ganho linear de desempenho com o aumento do número de processadores. Por exemplo, um processador pode bloquear certos recursos compartilhados (incluindo dados) enquanto executa algum processamento, fazendo com que outros processadores tenham que aguardar a liberação de tais recursos. Além do bloqueio de recursos, em algumas arquiteturas o uso compartilhado do barramento degrada sensivelmente o desempenho do sistema. Percebe-se que este modelo é bastante flexível, podendo ser utilizado para executar fluxos de processamento complexos, mas também necessitam de uma grande complexidade na gerência e sincronização destes fluxos, dificultando o trabalho do programador (BRAUNL et al., 2001; KIM; BOND, 2009; FLYNN, 1972).

A utilização de modelos de processamento paralelo pode aumentar muito o desempenho de sistemas baseados em computador. Por outro lado, devido à complexidade de tais modelos, sua utilização requer novos paradigmas de programação e diferentes algoritmos (BRAUNL et al., 2001).

2.1.1 *Streaming SIMD Extensions* (SSE)

Como já visto anteriormente, a computação SIMD processa múltiplos dados em paralelo com uma única instrução. Para aproveitar este modelo, a maioria das arquiteturas dos processadores atuais implementa extensões projetadas para fornecer instruções de computação

vetorial (*Instruction Level Paralelism* (ILP)) às unidades de processamento, como por exemplo a NEON da *Advanced RISC Machine* (ARM), Cell Broadband Engine da arquitetura Cell, AltiVec (DIEFENDORFF et al., 2000) da PowerPC, SSE (THAKKUR; HUFF, 1999), e *Advanced Vector Extensions* (AVX) (Intel, 2011c) ambas da Intel. Estas extensões são uma combinação de características de hardware e software, como registradores e instruções, que podem variar dependendo da arquitetura (SLINGERLAND; SMITH, 2005).

As extensões SIMD mais difundidas nos computadores pessoais são as SSE da Intel (também implementadas pela *Advanced Micro Devices* (AMD)), contendo de 8 a 16 registradores de 128 bits (dependendo da arquitetura), chamados XMM (Figura 1), que podem realizar operações com ponto-flutuante de precisão simples ou dupla, e inteiros (SLINGERLAND; SMITH, 2005; Intel, 2011b).

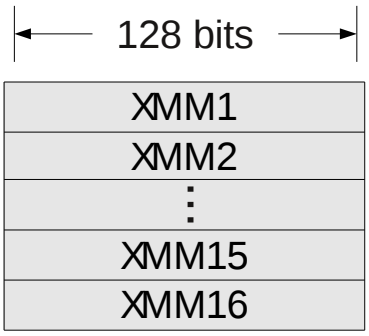


Figura 1 Registradores SSE

Fonte: O autor

SSE define dois tipos de operações: escalar e empacotada. A operação escalar opera somente no elemento de dados menos-significativo (bit 0-31), e as operações empacotadas computam todos os quatro elementos em paralelo (THAKKUR; HUFF, 1999), sendo que para obter máxima eficiência nas operações, os dados devem estar alinhados em 16 bytes na memória (THAKKUR; HUFF, 1999). Na Figura 2 podem ser vistos os dois tipos de operação com as instruções `mulss` e `mulps`, que realizam uma multiplicação.

Como o SSE é um conjunto de instruções *assembly* específicas, seria muito trabalhoso otimizar um algoritmo complexo escrevendo-o em *assembly*. Para simplificar este problema os

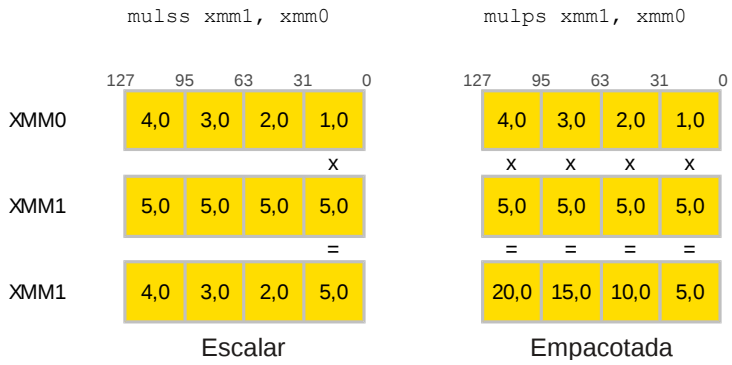


Figura 2 Tipos de operações SSE

Fonte: O autor

compiladores C/C++ mais avançados fornecem funções intrínsecas que emitem as instruções SSE necessárias. Por exemplo, para se fazer uma operação de produto-soma em SSE utiliza-se a função intrínseca `_mm_dp_ps` que emite a instrução `dpps` do SSE 4.1.

Na Listagem 1 há um exemplo que demonstra como estruturar um algoritmo para tirar vantagem do SSE. O algoritmo faz um laço que soma dois vetores com quatro elementos ponto-flutuante de precisão simples. Estas operações necessitam de quatro instruções de adição `fadd` com latência de 5 ciclos de *clock*, na arquitetura x87 (Intel, 2011c).

Listagem 1 Adição de vetores utilizando instruções escalares na arquitetura x87

```
1 void add(float *a, float *b, float *c)
2 {
3    int i;
4    for (i = 0; i < 4; i++) {
5       c[i] = a[i] + b[i];
6    }
7 }
```

Utilizando-se SSE é possível substituir as quatro instruções escalares do exemplo anterior por uma única instrução `addps` de 128 bits com latência de dois ciclos de *clock* (Intel, 2011c), como visto na Listagem 2, linha 5.

Apesar do significativo ganho de performance potencial com a utilização de tecnologias SIMD, identificar código que pode se beneficiar destas tecnologias é uma tarefa difícil e que consome tempo. Aplicações que necessitam de alto poder de computação podem se beneficiar,

Listagem 2 Exemplo de código SSE

```
1  void add(float *a, float *b, float *c)
2  {
3    __m128 va = _mm_load_ps (a);
4    __m128 vb = _mm_load_ps (b);
5    __m128 vc = _mm_add_pd (va, vb);
6    _mm_store_ps(c, vc);
7
8    /* Equivalente Assembly
9    ** mov eax, a
10   ** mov edx, b
11   ** mov ecx, c
12   ** movaps xmm0, XMMWORD PTR [eax]
13   ** addps xmm0, XMMWORD PTR [edx]
14   ** movaps XMMWORD PTR [ecx], xmm0
15   */
16 }
```

tais como (Intel, 2011c):

- Algoritmos e filtros de compressão de fala;
- Algoritmos de reconhecimento de fala;
- Rotinas de captura e exibição de vídeos;
- Rotinas de renderização;
- Gráficos 3D (geometria);
- Algoritmos de processamento de imagem e vídeo;
- Áudio espacial (3D);
- Modelagem física (gráficos, CAD);
- Aplicações para estações de trabalho;
- Algoritmos de encriptação;
- Aritmética complexa;

Neste mesmo sentido, códigos que contém pequenos laços e que operam em vetores sequenciais de inteiros de 8, 16 ou 32 bits, ponto-flutuantes de precisão simples de 32 bits ou de precisão dupla de 64 bits podem se beneficiar da utilização de extensões SIMD (Intel, 2011c).

2.1.2 *Open Multi-Processing* (OpenMP)

OpenMP[1] é um conjunto de diretivas de compilador e bibliotecas (*Application Programming Interface* (API)) que estendem as linguagens C/C++ e Fortran para expressar paralelismo de memória compartilhada (MIMD). A especificação do OpenMP (OpenMP Architecture Review Board, 2008) é livre, e existem implementações para vários compiladores e plataformas. A API facilita uma abordagem incremental de paralelização de programas sequencias com a adição das diretivas de paralelismo em laços ou em outras declarações do programa. Isto permite que o programa seja decomposto em alto nível com a criação de tarefas que podem ser executadas em diferentes *threads*. A API é direcionada à desenvolvedores que desejam paralelizar de forma rápida códigos científicos, mas é flexível o suficiente para suportar um conjunto variado de aplicações não-científicas. Sua primeira versão é datada de 1997 e atualmente encontra-se na versão 3.0 (DAGUM; MENON, 1998; SATO, 2002; CLARK, 1998).

OpenMP usa o modelo de execução paralela *fork-join*. O programa inicia como um processo único, com uma *thread* "inicial" ou "mestre". Assim que o programa entra em uma região paralela (fase *fork*), novas *threads* são criadas de acordo com as diretivas da região, e a execução passa a ser *multi-threaded*, como visualizado na Figura 3. Quando a execução sai da região paralela, as *theads* necessitam ser sincronizadas (fase *join*), e quando então a *thread* inicial segue com a execução (OpenMP Architecture Review Board, 2008; SATO, 2002).

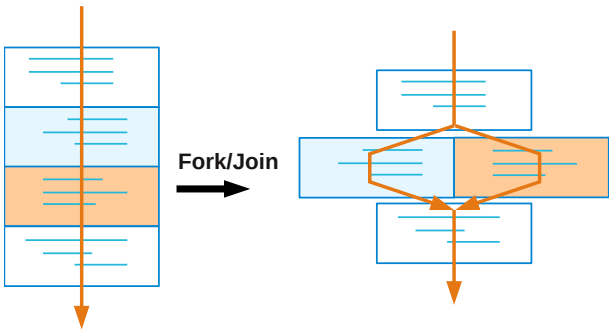


Figura 3 Fork/Join

Fonte: O autor

[1] http://openmp.org/

As diretivas de compilador são simples, permitindo paralelizar um programa existente com o mínimo de alterações no seu código. Na Listagem 3 encontra-se um exemplo de "Alô mundo!" em OpenMP:

Listagem 3 Exemplo Alô mundo em OpenMP

```
1  #include <omp.h>
2  #include <stdio.h>
3  #include <stdlib.h>
4
5  int main (int argc, char *argv[])
6  {
7    int nthreads, tid;
8    //Divide um conjunto de threads
9    //fornecendo suas proprias copias das
10   //variaveis
11   #pragma omp parallel \\
12           private(nthreads, tid)
13   {
14     //Obtendo o numero da thread
15     tid = omp_get_thread_num();
16     printf("Alo mundo da thread = %d\n",
17             tid);
18     //Apenas a thread principal faz isto
19     if (tid == 0) {
20       nthreads = omp_get_num_threads();
21       printf("Numero de threads = %d\n",
22           nthreads);
23     }
24   }  //Todas as threads unem-se a
25      //thread principal e finalizam
26 }
```

2.2 Aspectos relevantes sobre memória

Apesar de todos os esforços feitos nos últimos anos para a evolução das arquiteturas de memória, estas ainda continuam sendo o grande gargalo ou barreira a ser rompida para a melhoria significativa do desempenho dos programas. De pouco adianta paralelizar a transferência de dados e a computação dos dados se não existirem dados para serem computados em tempo devido a alta latência. A latência é o tempo gasto para um dado ser transportado desde a memória até o processador, outros *chipsets* ou dispositivos. Sendo assim, se o processador precisar de um dado para seguir seu processamento e o mesmo não estiver nas memórias *cache* mais rápidas, a quantidade de dados transportados em paralelo é irrelevante, pois eles levarão o mesmo tempo para chegar ao processador, ocasionando a parada no processamento até lá (ALTED, 2010; SUTTER, 2007; SUTTER, 2006; DREPPER, 2007).

Isto acontece devido a melhora no desempenho das memórias não ter acompanhado o

desempenho dos processadores. Na Figura 4 é mostrado o gráfico de desempenho relativo das CPUs com relação as *Dynamic Random-Access Memory*s (DRAMs), ou Memórias Dinâmicas de Acesso Aleatório. Enquanto os processadores dobram sua capacidade de processamento a cada dois anos, as memórias levam seis anos. Isto criou uma diferença enorme de desempenho entre os dois componentes, fazendo com que seja virtualmente impossível utilizar todo o poder de processamento de um processador moderno, pois o mesmo sofre de inanição de dados (ALTED, 2010; SUTTER, 2007; SUTTER, 2006; DREPPER, 2007; HENNESSY; PATTERSON, 2011).

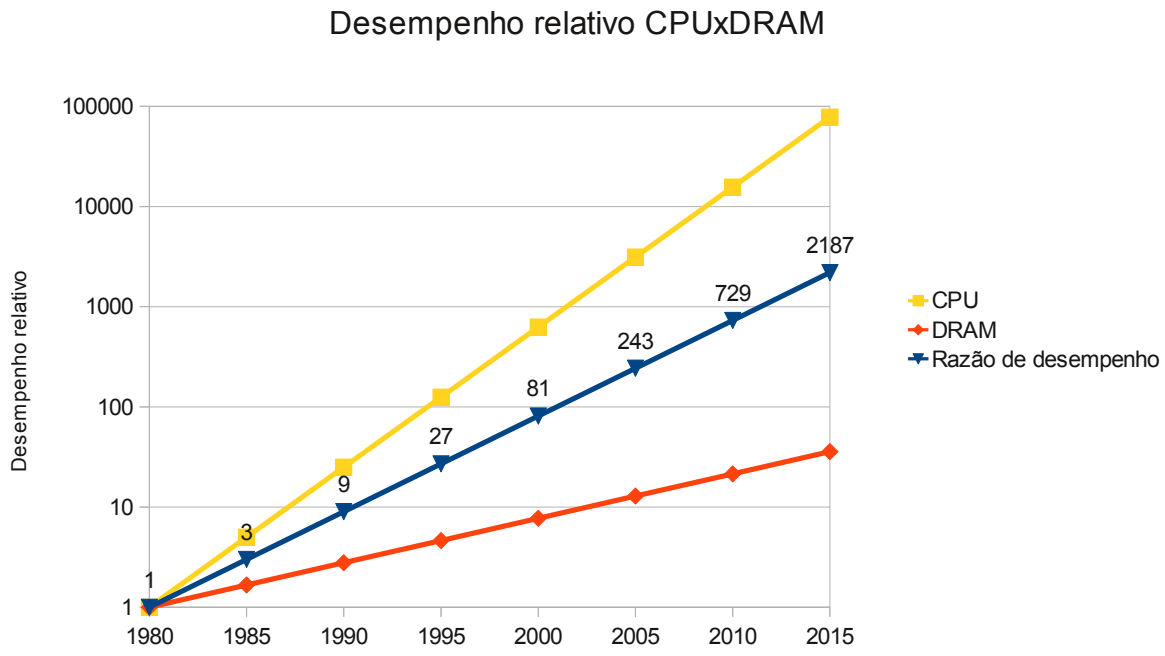


Figura 4 Desempenho relativo CPUxDRAM

Fonte: O autor

2.2.1 Arquitetura de memória

Quanto às arquiteturas de memória é pertinente mencionar as *Random-Access Memory*s (RAMs), ou Memórias de Acesso Aleatório, e duas de suas variantes que tem arquiteturas bastante distintas.

As *Static Random-Access Memory*s (SRAMs), ou Memórias Estáticas de Acesso Alea-

tório, são extremamente rápidas, mas complexas e caras de se produzir. As memórias *cache*, por exemplo, são deste tipo, explicando o porquê de seu tamanho ser relativamente pequeno. Além de memórias *cache*, as SRAMs são utilizadas em dispositivos que requerem alta velocidade no acesso a dados, como por exemplo *switches* de rede. A SRAM é construída utilizando-se seis transístores (Figura 5) e requer fluxo de energia constante para que mantenha o estado, o que também permite que seu estado seja lido e gravado rapidamente (DREPPER, 2007; BERNSTEIN; ROHRER, 2000; ITOH, 2008; HENNESSY; PATTERSON, 2011).

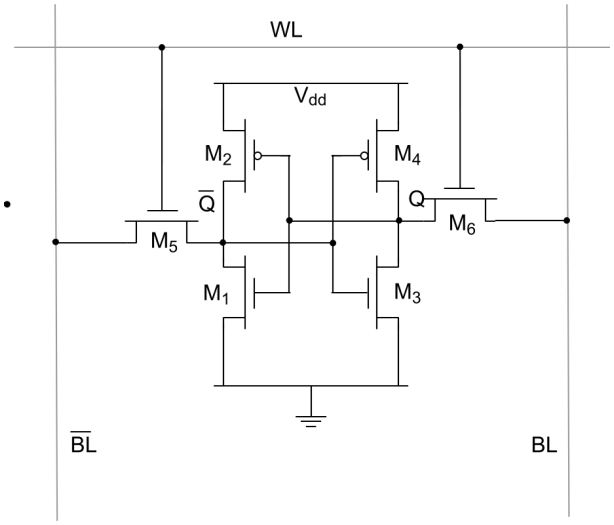


Figura 5 Célula SRAM

Fonte: Drepper (2007)

Por outro lado as DRAMs são mais simples e baratas de se produzir, mas seu desempenho é muito inferior às SRAMs. Esta discrepância de desempenho se deve por uma série de diferenças entre suas arquiteturas. As DRAMs são construídas utilizando-se apenas um transístor e um capacitor (Figura 6) para armazenar o estado. Esta diferença é crucial, pois o capacitor deve ser recarregado tipicamente a cada 64ms, impossibilitando neste tempo quaisquer leituras e escritas das células de memória. Além disso, a gravação deve ser feita carregando ou descarregando o capacitor, o que também não é instantâneo. As DRAMs têm mecanismos de acesso mais complexos, com a utilização de demultiplexadores de endereço, pois podem armazenar vários GBits de dados, inviabilizando o endereçamento direto. O desempenho de escrita e leitura das memórias DRAMs atuais é melhorado quando escreve-se ou lê-se em sequência,

permitindo, por exemplo, que uma linha inteira de memória *cache* seja carregada sem a demultiplexação de vários endereços (DREPPER, 2007; BERNSTEIN; ROHRER, 2000; ITOH, 2008; HENNESSY; PATTERSON, 2011).

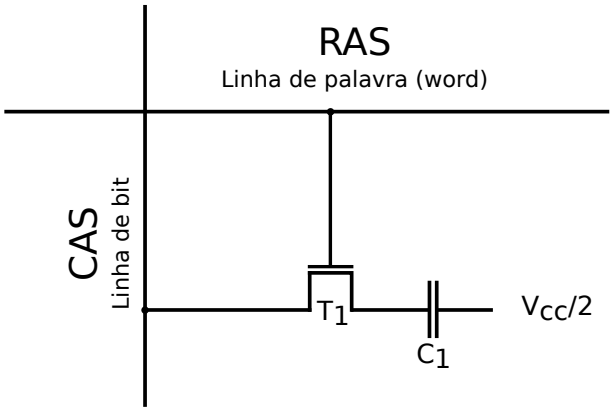


Figura 6 Célula DRAM

Fonte: O autor

2.2.2 Hierarquia de memória

Como não existem sistemas com recursos ilimitados de memória rápida, uma das formas de se contornar a latência de acesso aos dados é criar sistemas hierárquicos de memória (Figura 7), compostos de tipos e tamanhos de memórias diferentes. Como as memórias rápidas são caras, a hierarquia é organizada em vários níveis, cada um menor, mais rápido e com um custo maior por byte do que o nível inferior. O objetivo é criar uma hierarquia que custe tão pouco quanto o nível mais barato e seja tão rápida quanto o nível mais rápido (MAGGS; MATHESON; TARJAN, 1995; HENNESSY; PATTERSON, 2011).

2.2.3 Comunicação de dados entre memórias

Em arquiteturas massivamente paralelas, com muitos processadores e múltiplos núcleos por processador, a comunicação entre estes elementos passa a ser um problema. Como cada processador ou núcleo pode conter um ou mais níveis de memória *cache* não-compartilhada, a modificação de tal memória por outro núcleo (na memória principal ou em outra memória

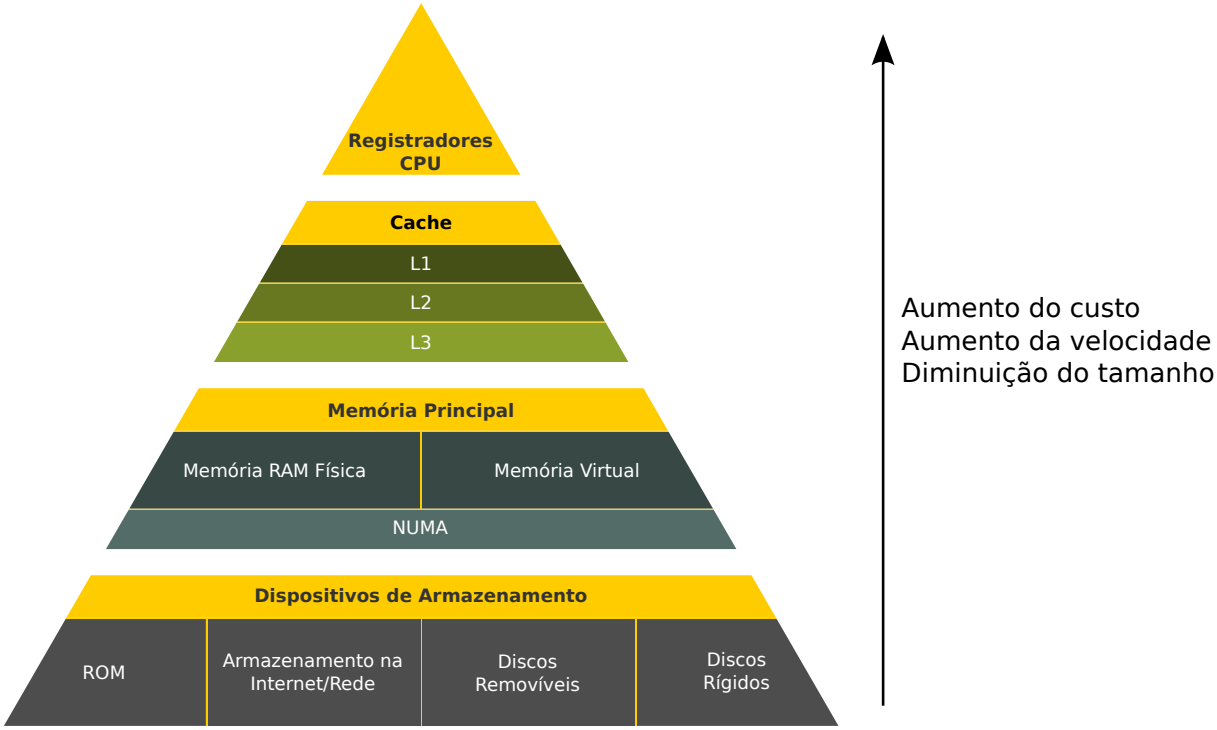


Figura 7 Hierarquia de memória

Fonte: O autor

compartilhada) implica na sua invalidação. Uma vez a memória invalidada, aquele processador ou núcleo terá que fazer outro acesso a memória principal para obter os novos dados, uma operação de alta latência. Caso a arquitetura de comunicação entre os processadores utilize um barramento comum (*Front-Side Bus* (FSB), por exemplo), possivelmente este barramento ficará sobrecarregado, aumentando ainda mais a latência de acesso à memória.

Para contornar este problema criaram-se arquiteturas chamadas *Non-uniform memory access* (NUMA), ou Acesso Não-Uniforme a Memória. Nestas arquiteturas cada processador tem sua própria memória e o acesso a memória de outros processadores não tem um tempo constante. Na arquitetura NUMA há uma interligação ponto-a-ponto de alta velocidade entre os núcleos. Apesar das vantagens, a arquitetura NUMA também sofre de problemas de acesso a dados, principalmente quando a quantidade de núcleos é grande o suficiente para inviabilizar uma ligação ponto-a-ponto entre todos os processadores (MAGGS; MATHESON; TARJAN, 1995; ALTED, 2010; DREPPER, 2007).

2.2.4 Técnicas de otimização de latência em referências a memória

Como já mencionado, a latência de acesso à memória é um grande problema nas arquiteturas atuais. Para diminuir a latência existem várias alternativas, como se segue.

Não obstante a taxa de transferência entre a memória principal e o processador (incluindo a largura do barramento e número de canais) ter importância no desempenho geral do sistema, a frequência do barramento e as frequências dos relógios do módulo de memória são as que desempenham o papel principal. Em um processador Intel Core 2 Duo de 2.933GHz a frequência de seu barramento (FSB) é de 266MHz. Como este barramento consegue enviar 4 palavras de 64 bits em paralelo, diz-se que sua frequência efetiva é de 1.066GHz, mas sua relação de frequência para o processador é de 11:1, então a cada bloqueio (*stall*) de um ciclo no barramento, haverá um bloqueio de 11 ciclos no processador. No entanto há um limite em aumentar a frequência do barramento, pois devido a transferência ser paralela, quando utilizadas memórias DDR (*Double Data Rate*), quanto maior a frequência, maior a interferência no sinal (DREPPER, 2007). A Intel chegou a desenvolver uma memória DRAM serial chamada *Fully Buffered-Dual Inline Memory Model* (FB-DIMM) para tentar resolver este problema, mas o padrão DDR3 teve melhor aceitação de mercado. As especificações do novo padrão DDR4 revelam que este usará um protocolo de transferência de memória serial ponto-a-ponto, trabalhando a 4.266GHz e com uma taxa de transferência de dados de 2.133GBps. A Samsung é a primeira empresa a fabricar um módulo de tal padrão, mas as estimativas indicam que apenas em 2015 este se tornará popular. (DREPPER, 2007; Samsung, 2011; Intel, 2006).

Caso não seja possível simplesmente aumentar a frequência de transferência da memória principal até o processador, ou a frequência do módulo de memória, com a adição de mais níveis de memória pode-se usar uma técnica chamada *latency hiding*[2] (Figura 8). Esta técnica consiste em previamente trazer dados a serem processados, da memória principal para a memória *cache*, em paralelo ao processamento, evitando os chamados *cache misses*. Como as memórias *cache* possuem uma latência de leitura e escrita muito menor do que a memó-

[2] Ocultar a latência

ria principal, isto permite que o processador sempre tenha dados para processar a baixo custo, não havendo necessidade de esperar (*stall*) o lento acesso à memória principal, além de permitir utilizar eficientemente a largura de banda de memória disponível (MAGGS; MATHESON; TARJAN, 1995; Intel, 2011b; DREPPER, 2007).

A ocultação da latência é atingida com a utilização de instruções de *prefetch*, que podem ser realizadas via hardware, compilador ou pelo programador. Tais instruções devem ser utilizadas de acordo com o tipo de referência de dados processados. Estes dados podem ter referência temporal, quando serão usados novamente em breve; espacial, quando dados adjacentes serão usados; ou não-temporal, quando os dados serão utilizados apenas uma vez, como por exemplo em um *stream*. Sendo assim, pode-se otimizar a utilização dos níveis de memória *cache* de acordo com o algoritmo (FOG, 2010; Intel, 2011c; Intel, 2011b; ALTED, 2010).

Apesar da técnica de *latency hiding* ser bastante eficiente, quando aplicada pelo programador, este deve ter total conhecimento do hardware responsável pela execução do software, pois o tamanho da linhas da memória *cache* podem variar de acordo com a arquitetura. Além disso, é necessário saber o tamanho do conjunto de dados, o padrão de acesso a estes, o tempo de acesso à memória em ciclos e o tempo de processamento em ciclos, para inserir as instruções de *prefetch* em locais estratégicos do código.

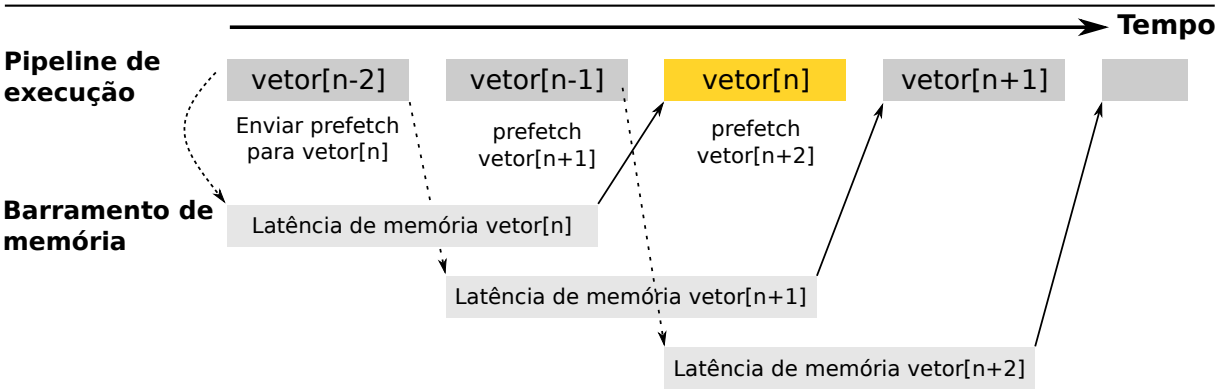


Figura 8 Escondendo a latência

Fonte: O autor baseado em Intel (2011b)

2.3 Otimização

Nesta seção são apresentadas algumas técnicas de otimização de código que podem ser aplicadas de forma simples, até mesmo em código pré-existente.

2.3.1 Desdobramento de laços (*loop unrolling*)

O desdobramento de laços (*loop unrolling*) (Intel, 2011b) permite que o mesmo código do corpo de um laço seja replicado várias vezes em uma mesma iteração (Figura 9).

As vantagens do *desdobramento de laços* são:

- Facilitar o trabalho do escalonador em rearranjar instruções, pois diminui a dependência de nomes entre os registradores;
- Aumentar o paralelismo de instruções, preenchendo mais eficientemente o *pipeline*;
- Diminuir a necessidade de desvios condicionais presentes no controle de fluxo do laço.

O número de desdobramentos deve ser balanceado, pois as vantagens acima podem ser anuladas devido ao aumento de *cache misses*[3] de instrução e à sobrecarga nos registradores.

2.3.2 *Loop blocking*

O *loop blocking* ou *loop tiling* (Intel, 2011b) (Figura 10) é outra técnica de otimização que permite melhorar a utilização da memória *cache*, com a criação de blocos de dados a serem processados compatíveis com a capacidade da memória *cache*.

Os blocos de memória devem ser pequenos o suficiente para preencher a memória *cache*, aumentando o reuso desta.

Outra vantagem da técnica é diminuir *misses* de *Translation Lookaside Buffer* (TLB) quando o domínio de memória é muito grande.

[3] *Cache miss*

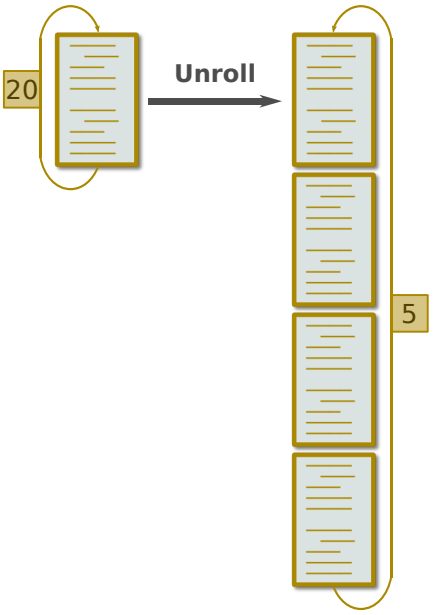


Figura 9 Desdobramento de laços (*Loop unrolling*)

Fonte: O autor

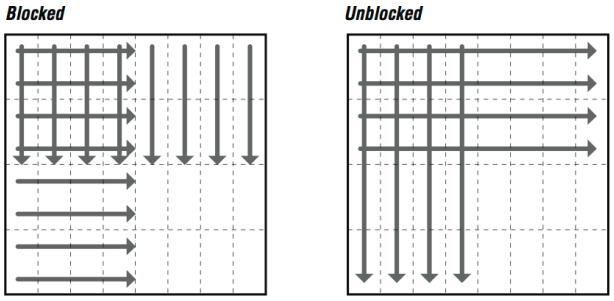


Figura 10 Loop blocking

Fonte: O autor

2.3.3 *Array of Structures* (AoS) e *Structure of Arrays* (SoA)

Array of Structures (AoS), ou Vetor de Estruturas, e *Structure of Arrays* (SoA), ou Estrutura de Vetores, são dois tipos de organização de estruturas de dados, como podem ser visualizados nas Listagens 4 e 5, respectivamente. O primeiro tipo, AoS, é menos eficiente para operações SIMD, pois seus elementos são independentes, ao contrário das estruturas do tipo SoA que têm um arranjo que permite que operações SIMD sejam executadas de forma otimizada.

O ideal é organizar as estruturas no formato SoA, mas nos casos em que isto não é

possível, pode-se realizar operações de rearranjo dos dados (*swizzle*) para tirarem maior proveito das operações SIMD.

Listagem 4 Exemplo organização de estruturas de dados AoS

```
1  typedef struct{
2    float x,y,z;
3    int a,b,c;
4    ...
5  } Vertex;
6
7  Vertex Vertices[NumOfVertices];
```

Listagem 5 Exemplo organização de estruturas de dados SoA

```
1  typedef struct{
2    float x[NumOfVertices];
3    float y[NumOfVertices];
4    float z[NumOfVertices];
5    int a[NumOfVertices];
6    int b[NumOfVertices];
7    int c[NumOfVertices];
8    ...
9  } VerticesList;
10 VerticesList Vertices;
```

2.3.4 *Data Swizzling/Deswizzling*

Data Swizzling/Deswizzling (mistura/separação de dados) é uma técnica utilizada para transformar um conjunto de dados SoA para AoS ou vice-versa, bastante utilizada na computação gráfica, em operações com vetores. A técnica consiste em rearranjar os dados de forma que seja possível fazer sua computação em paralelo de forma mais eficiente (Intel, 2011b). Na Figura 11 pode-se observar a aplicação da técnica em uma operação de produto-escalar. Repara-se que um vetor de três elementos é transformado em três vetores de quatro elementos, permitindo que sejam feitas quatro computações em paralelo.

2.3.5 Alinhamento de dados

Outro aspecto a se considerar é o alinhamento de dados, necessário para otimizar a leitura e escrita na hierarquia de memórias. Os dados são ditos como alinhados em memória quando se encontram em um endereço múltiplo do tamanho da palavra da arquitetura. Em

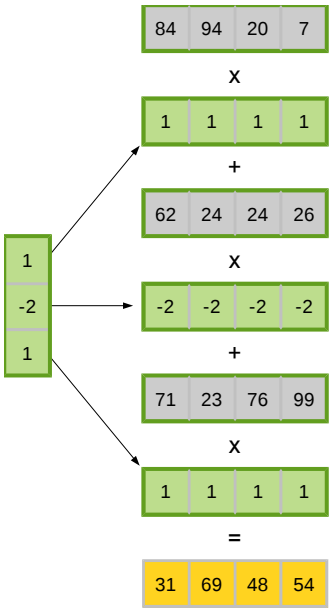


Figura 11 *Data Swizzling*

Fonte: O autor

arquiteturas de 32 bits ou 64 bits, variáveis, ponteiros, estruturas ou vetores são alinhados preferencialmente com 4 bytes ou 8 bytes respectivamente. Algumas arquiteturas impedem a leitura/escrita em endereços de memória não-alinhados, já outras, como a x86 permitem tal acesso, mas com degradação do desempenho.

Este alinhamento é importante, pois a transferência de dados pelo barramento de memória geralmente é realizada em fatias (*chunks*) com tamanhos múltiplos de uma palavra, e quando há um acesso à memória desalinhada é necessário que haja outro acesso para que todos os dados sejam lidos. Como isto é feito de forma atômica, aumenta-se a latência.

Manter a memória alinhada também aumenta a eficiência de utilização da memória *cache*. Por exemplo, se em uma determinada arquitetura o tamanho da linha de *cache* for de 64 bytes, manter os dados alinhados em múltiplos de 64 garante a plena utilização das linhas, o que pode ser bastante útil quando houver acesso a estruturas sequenciais como vetores (PAGE, 2009; HENNESSY; PATTERSON, 2011; FOG, 2010).

O acesso à dados utilizando-se algumas instruções SSE deve ser obrigatoriamente alinhado em 16 bytes (Figura 12). Para as instruções que permitem acessos desalinhados há ne-

cessidade de um carregamento adicional, mais uma instrução para trocar os elementos entre os registradores para que o resultado contenha o dado requisitado (Figura 13).

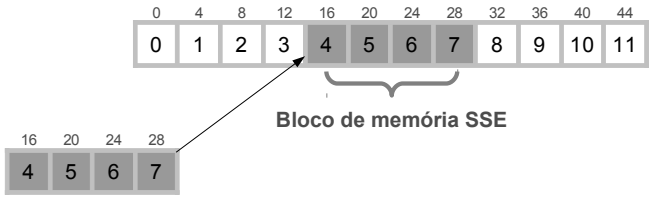


Figura 12 Acesso alinhado a memória

Fonte: O autor

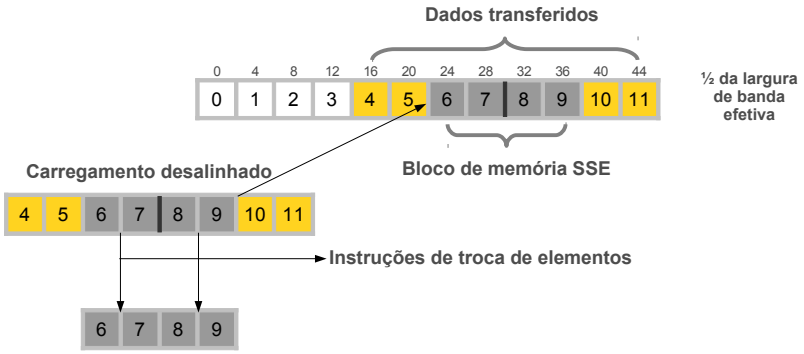
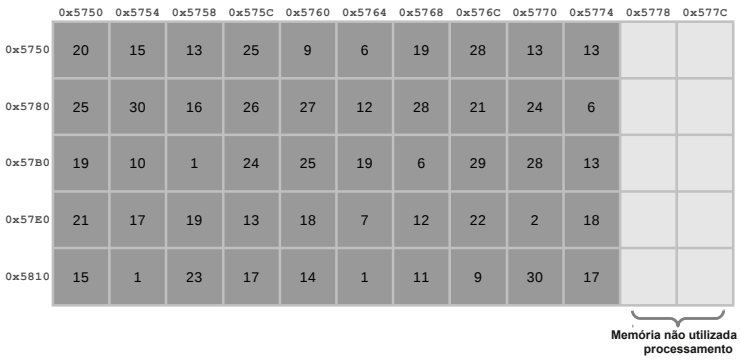


Figura 13 Acesso desalinhado a memória

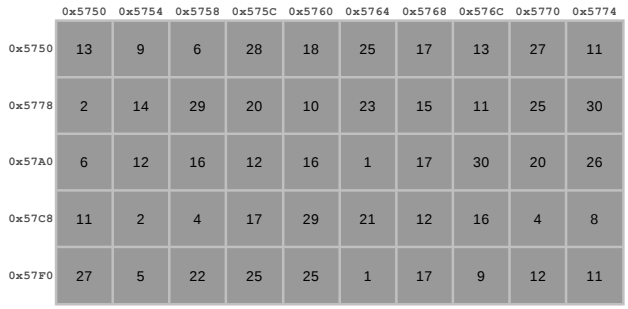
Como exemplo de otimização de acessos alinhados utilizando SSE pode-se citar o uso de imagens com *strides*[4] de tamanho variável. Na Figura 14 pode-se observar a técnica. Repara-se que na Figura 14a os endereços de memória de cada linha são múltiplos de 16 bytes, o que não ocorre na imagem da Figura 14b. Para fazer tal alinhamento é necessário aumentar o tamanho do *stride* da imagem. Para o cálculo do *stride* é utilizada a função `calculateAlignedStride` (Listagem 6). Esta função retorna a largura de *stride* em *pixels* múltipla do alinhamento em bytes maior e mais próxima da largura da imagem.

Com relação à organização da memória cache, a Figura 15 demonstra como geralmente se dá o acesso e a transferência de dados em uma arquitetura de 64-bits. Repara-se que, mesmo

[4] O *stride* de uma imagem é a largura da linha em bytes



(a) Stride alinhado em 16 bytes



(b) Stride desalinhado

Figura 14 Imagem com *stride* variável

Fonte: O autor

acessando somente uma palavra (*word*) de 64-bits, são carregadas 8 palavras para preencher a linha da memória cache.

2.3.6 Vetorização

Com já mencionado, muitas arquiteturas oferecem instruções SIMD, que podem ser utilizadas para a vetorização do processamento. A vetorização pode ser realizada de diversas maneiras, como por exemplo:

- Código *assembly* embarcado - maneira mais trabalhosa, menos portável, manutenível, e produtiva de vetorizar código, mas com a vantagem de poder criar um código otimizado;
- Funções intrínsecas - funções que emitem uma instrução específica de determinada arquitetura de processador, mas que ainda sofrem controle do compilador (Free Software

Listagem 6 Função para calcular um *stride* alinhado

```
1 int calculateAlignedStride (int width, int pixelSizeInBytes, int alignInBytes)
2 {
3     if(width < alignInBytes) return alignInBytes;
4     int widthInBytes = width * pixelSizeInBytes;
5     return widthInBytes % alignInBytes == 0 ? width :
6         (widthInBytes + alignInBytes – (widthInBytes % alignInBytes)) / pixelSizeInBytes;
7 }
```

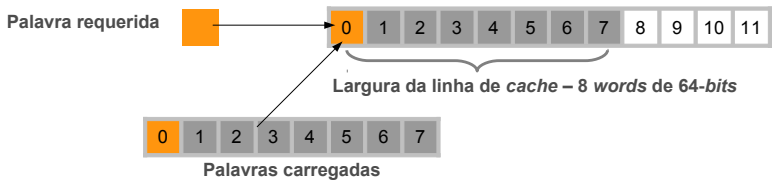


Figura 15 Acesso a memória

Fonte: O autor

Foundation, 2010; Microsoft, 2011).

- Pragmas - recurso oferecido pelo compilador que permite instruí-lo diretamente a vetorizar determinado laço (Intel, 2011d).

- Autovetorização - tipos de dados especiais que permitem ao compilador vetorizar automaticamente o código (Free Software Foundation, 2011; NAISHLOS, 2004; Intel, 2011a).

2.3.7 *Profiling*

Profiling é uma forma de análise dinâmica de um programa utilizada para medir a velocidade de processamento, carga do processador, uso de memória, consumo de energia, uso de instruções particulares, frequência/duração de chamadas a funções, etc., com o intuito de identificar gargalos, resolvê-los e melhorar o desempenho da implementação (WESCOTT; DOCEF, 2007; Valgrind™ Developers, 2011a).

O *profiling* é realizado por ferramentas chamadas *profilers*. Como exemplo pode-se citar o depurador e *profiler* de código aberto Valgrind (Valgrind™ Developers, 2011a).

2.4 Convolução

A convolução é uma operação onde duas funções, f e g, produzem uma terceira função que é tipicamente vista como uma versão modificada de uma das funções originais (WEISSTEIN, 2011). As convoluções são utilizadas para o tratamento de sinais digitais de n-dimensões. No caso de processamento de imagens utiliza-se uma matriz imagem e uma matriz chamada máscara ou *kernel*, operando-se o produto escalar destas duas matrizes, tendo como resultado desta operação o valor de um *pixel* da imagem de saída (GONZALEZ; WOODS, 2007). Na Figura 16 há um exemplo: para realizar a operação de convolução em uma imagem 2D basta percorrer a imagem calculando o produto escalar de todas as áreas de dimensões 3x3 com a máscara de mesma dimensão e armazenar o resultado no pixel que corresponde ao centro da área.

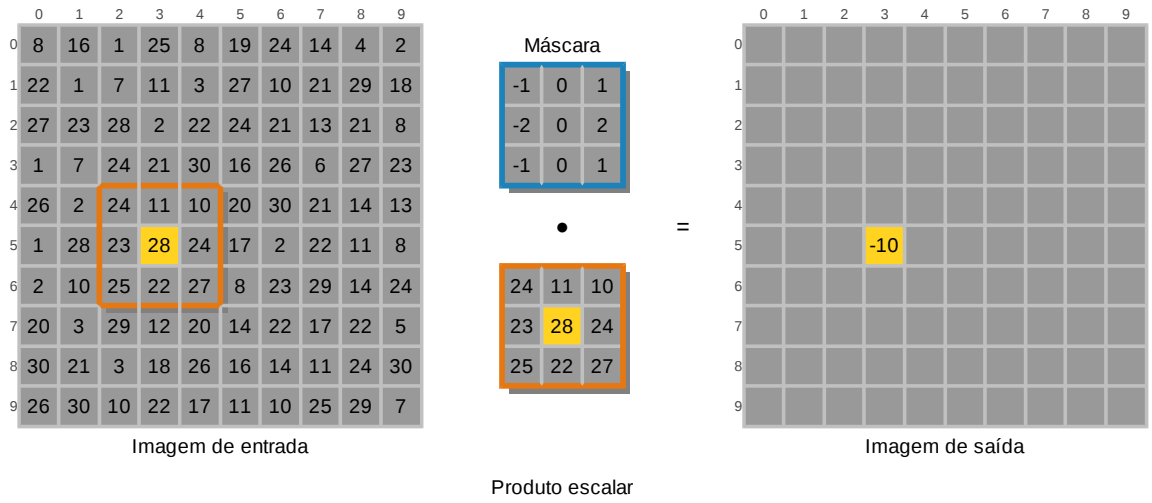


Figura 16 Convolução 2D

Fonte: O autor

A convolução 2D pode ser definda por:

$$C[x,y] = \sum_{y=-h}^{h} \sum_{x=-w}^{w} I[x,y] \times M[x,y]$$

Em alguns casos, a convolução pode ser separada em duas convoluções vetoriais (Figura 17), operando-se nos dois eixos x e y. Esta propriedade é válida quando a mesma máscara de duas dimensões pode ser obtida pela multiplicação de dois vetores de uma dimensão, como

por exemplo nas *máscaras* do operador Sobel (GONZALEZ; WOODS, 2007).

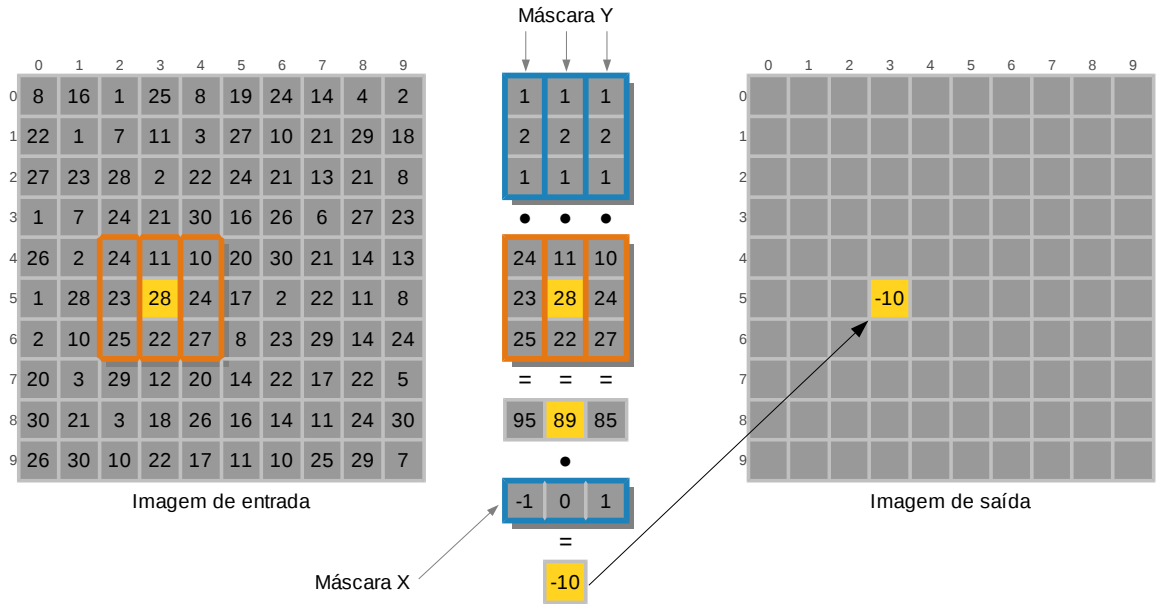


Figura 17 Convolução separável 2D

Fonte: O autor

As seguintes expressões definem as convoluções nos eixos x e y respectivamente:

$$C[x] = \sum_{y=-h}^{h} I[y+x] \times M[x]$$

$$C[y] = \sum_{y=-h}^{h} I[(y \times w)+x] \times M[x]$$

Onde C é a imagem convolucionanda, I é a imagem de entrada e M a máscara de convolução.

2.5 Processamento de imagens

Uma imagem pode ser definida como uma função bi-dimensional, $f(x,y)$, sendo x e y coordenadas no plano espacial, e a amplitude de f em quaisquer pares de coordenadas (x,y) a intensidade ou nível de cinza da imagem naquele ponto. Quando x, y, e os valores de amplitude de f são valores discretos, a imagem é digital, pois é formada por um número finito de elementos (*pixels*), os quais têm uma posição e um valor particular. Sendo assim, o

processamento de imagens é a aplicação de algoritmos que utilizam os dados da função $f(x,y)$ para alterar ou extrair informações destas imagens (GONZALEZ; WOODS, 2007).

2.5.1 Processamento de imagens médicas

O processamento de imagens tornou-se um componente essencial em muitos campos de pesquisas laboratoriais e médicas, e na prática clínica. Utilizando-se equipamentos de ressonância magnética ou de tomografia computadorizada é possível, por exemplo, detectar e quantificar tumores ou até mesmo reconstruir tecidos e outras estruturas humanas em 3D. Até pouco tempo atrás isto requeria estações de captura e visualização sofisticadas e caras, além de software customizado. Atualmente, quaisquer sistemas *desktop* com software e hardware gráficos apropriados podem realizar o mesmo tipo de análise (McAuliffe et al., 2001). Para viabilizar a utilização destes sistemas de baixo custo, nada mais natural do que a adoção de ferramentas e bibliotecas de processamento de imagens de código livre (CABAN; JOSHI; NAGY, 2007).

Software como ITK (Seção 2.5.2), VTK[5] e OpenCV[6] se tornaram referência no processamento de imagens médicas, com uma grande base de usuários e muitas funcionalidades. Com as funcionalidades destes software estabelecidas, o importante agora é a otimização das implementações de seus algoritmos para que se possa criar sistemas com respostas rápidas, agilizando, por exemplo, o resultado de diagnósticos, algo muito importante quando se trata da vida humana (CABAN; JOSHI; NAGY, 2007).

2.5.2 *Insight Segmentation and Registration Toolkit* (ITK)

O ITK[7] é um conjunto de ferramentas livres de código fonte aberto e orientadas a objeto para processamento, segmentação e registro de imagens, bastante utilizada na área médica. Tais ferramentas permitem ao usuário a possibilidade de criar fluxos de processamento de imagens complexos (IBÁÑEZ et al., 2005).

[5]Mais informações em: `http://www.vtk.org/`
[6]Mais informações em: `http://opencv.willowgarage.com/wiki/`
[7]Página oficial do projeto: `http://www.itk.org/`

O ITK é fomentado pelo consórcio *Insight Software Consortium* fundado por vários institutos de pesquisa, universidades e iniciativa privada dos Estados Unidos. Atualmente o projeto é capitaneado pela empresa Kitware[8], e seus principais objetivos são (YOO et al., 2002; ITK, 2011):

- Suporte ao projeto Homem Visível[9];
- Estabelecer uma base para pesquisas futuras;
- Criar um repositório para algoritmos fundamentais;
- Desenvolver uma plataforma para desenvolvimento avançado de produtos;
- Dar suporte comercial à aplicação de tecnologia;
- Criar convenções para trabalhos futuros;
- Fomentar uma comunidade autossustentável de desenvolvedores e usuários de software.

O ITK não foi projetado para ser utilizado diretamente por médicos. O objetivo do projeto é criar uma *API* que pode ser usada por desenvolvedores em programas médicos ou outros programas que resolvam problemas com processamento de imagens ou volumes (YOO et al., 2002).

O *toolkit* é escrito em C++ e foi desenvolvido levando-se em conta conceitos como programação genérica com uso de *templates*, *smart pointers* para gerenciamento automático de memória, padrões de projeto e suporte a *multithreading*. Também possui *bindings*[10] à várias linguagens como Java, Python, Tcl, etc (IBÁÑEZ et al., 2005; YOO et al., 2002).

A representação de dados é realizada pelas classes `itk::Image` e `itk::Mesh`, e a manipulação destas é feita por vários tipos de "iteradores" e *containers*, que são utilizados para percorrer e armazenar os dados.

[8] Página oficial da empresa: `http://www.kitware.com`
[9] Página oficial do projeto: `http://www.nlm.nih.gov/research/visible`
[10] Bibliotecas de ligação

A classe `itk::Image` representa uma amostra de dados n-dimensional e regular. Esta classe é altamente flexível, podendo-se especificar, por exemplo, o tamanho dos pixels, o tipo de dados e o espaço entre eles. Outro conceito interessante do ITK é a representação de espaços retangulares contínuos de uma imagem como regiões. Os três tipos de regiões mais importantes são:

1. **LargestPossibleRegion** — a imagem em sua totalidade;
2. **BufferedRegion** — a porção da imagem mantida em memória;
3. **RequestedRegion** — a porção da região requerida por um filtro ou outra classe para realizar operações.

As instâncias das classes de representação de dados, ou objetos de dados, são operados por filtros (objetos de processamento) organizados em fluxos de processamento de dados (*pipelines*), para produzir novos objetos de dados. Tais objetos de processamento podem ser fontes de dados, objetos de filtro ou mapeadores. Os objetos de fonte de dados produzem ou fornecem dados, como os leitores de arquivos de imagem. Os objetos de filtro recebem dados, os processam, e produzem novos dados. Já os mapeadores atuam como objetos de saída de dados.

Outra característica interessante do ITK e sua arquitetura de *pipelines* de processamento de dados. Os *pipelines* permitem que vários filtros sejam compostos, vinculando objetos de dados, para resolver uma variada gama de problemas. Os *pipelines* suportam um mecanismo de atualização automática que permitem que o filtro seja executado somente quando sua entrada de dados ou estado interno sejam alterados. Além disso, estes suportam *streaming* (fluxo de dados) e *multithreading*, dividindo os dados em partes menores, processando as partes uma a uma e reagrupando os dados processados em um resultado final (IBÁÑEZ et al., 2005).

2.5.3 Processamento paralelo de imagens

Várias arquiteturas, bem como bibliotecas e otimizações, já foram propostas para processamento paralelo de imagens (STAMOPOULOS, 1975; BRAUNL et al., 2001; YANG; ZHU; PU, 2008; ASANO; MARUYAMA; YAMAGUCHI, 2009; KIM; LEE; CHEN, 2010; SLABAUGH; BOYES; YANG, 2010).

O processamento de imagens pode ser complexo, requerendo grande poder computacional e tempo de processamento. Uma simples imagem colorida de 1024x1024 *pixels* com 8 bits por canal de cor[11] ocupa 3MB de memória. No entanto, o processamento paralelo da imagem pode diminuir significantemente o tempo de processamento, particularmente se um grande número de processadores puder ser utilizado eficientemente, por exemplo, em processamentos realizados associando cada *pixel* a um diferente processador (BRAUNL et al., 2001).

Como visto anteriormente, existem várias arquiteturas *multi-core*, e cada uma delas contém suas peculiaridades. Kim, Lee e Chen (2010) utiliza a seguinte abordagem de otimização para processamento de imagens em CPUs: primeiramente define-se se um programa é limitado por memória ou computação, o que é crucial para determinar o desempenho potencial. Depois é otimizado o desempenho *single-core*, vetorizando fluxos com instruções SIMD. O terceiro passo é alcançar alta escalabilidade *multi-core*, sendo essencial para tal um bom particionamento de dados para minimizar a comunicação entre os núcleos e balancear a carga, o que pode ser feito utilizando-se OpenMP (SLABAUGH; BOYES; YANG, 2010). O último passo é a otimização dos acessos a memória, que deve se preocupar mais em economizar banda ao invés de ocultar a latência, devido a hierarquia de memória. Outra otimização pode ser obtida com o uso correto da memória virtual das arquiteturas modernas. Kamp (2010) propõe um algoritmo chamado *b-heap*, que é uma otimização do algoritmo clássico de *heap* binária. A otimização consiste em aumentar a altura da *heap* e diminuir a largura de seus subníveis, reduzindo assim o número de invalidações de páginas de memória virtual.

[11] Cada pixel colorido em uma imagem digital é criado através de alguma combinação das três cores primárias: vermelho, verde e azul. Cada cor primária é normalmente chamada de "canal de cor" (Cambridge in Colour, 2012).

Kim, Lee e Chen (2010) e Slabaugh, Boyes e Yang (2010) demonstram como o alinhamento de memória, a utilização de instruções SSE e OpenMP e a gerência eficiente da memória *cache* podem trazer ganhos significativos de desempenho a alguns algoritmos comumente utilizados em processamento de imagens como FFT (*Fast Fourier Transform*), convoluções, normalização e histogramas.

2.6 Detector de bordas Canny

O detector de bordas Canny é um algoritmo de detecção de bordas eficiente e muito utilizado no processamento de imagens. Canny definiu que para um algoritmo ser considerado bom, este deve ter baixa probabilidade de não detectar bordas reais, e baixa probabilidade de detectar não-bordas como bordas. Esta característica é diretamente ligada a ruídos na imagem. Em segundo lugar, um algoritmo detector de bordas deve ter boa localização. Os pontos marcados como bordas devem estar o mais próximo possível do centro da borda real. Terceiro, o algoritmo deve ter como resultado apenas uma única borda. Isto teve deve ser levado em consideração, pois as duas primeiras características do filtro não eliminam por completo a possibilidade de haver mais de uma resposta por borda (CANNY, 1986).

Baseado nestes critérios, o detector de bordas Canny primeiramente suaviza a imagem para eliminar ruídos utilizando o filtro Gaussiano. Posteriormente procura por gradientes na imagem. Gradientes são regiões de rápida transição de tons, causando grandes derivadas espaciais. Então o algoritmo suprime destas regiões os *pixels* com derivadas que não são máximas locais (*non-maximum suppression*). Por fim é feita a limiarização por histerese, usando-se dois limiares, um superior e um inferior, para eliminar o restante dos *pixels* que não são borda, como visto no Algoritmo 1 (CANNY, 1986).

Algoritmo 1: Algoritmo de detecção de bordas Canny

1. Suavizar a imagem de entrada;
2. Calcular o gradiente da imagem suavizada;
3. Suprimir os valores não máximos de gradiente;
4. Limiarizar os picos de gradiente para eliminar bordas falsas;

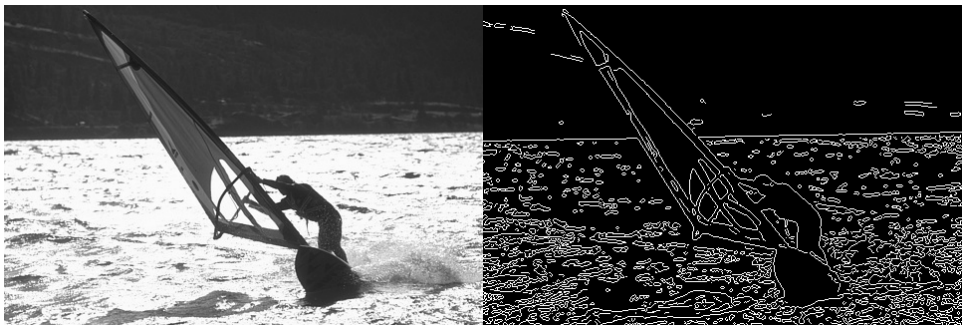


Figura 18 Exemplo de detecção de bordas usando o algoritmo de Canny

Fonte: O autor

2.6.1 Algoritmo de detecção de bordas Canny para ITK

Nesta seção é explicado como é implementado o algoritmo de detecção de bordas Canny para ITK.

A implementação do algoritmo Canny para ITK requer quatro parâmetros: variância e erro máximo, que são utilizados pela função Gaussiana, e os limiares máximos e mínimos para a limiarização por histerese. Primeiramente é feita a suavização da imagem utilizando um filtro Gaussiano. Em seguida são calculadas as derivadas direcionais de segunda ordem da imagem suavizada. Então é feita a detecção de bordas definitivas com a supressão dos não-máximos locais, detectando cruzamentos em zero (*zero crossing*) das derivadas segundas e analisando os sinais das derivadas de terceira ordem. Finalmente é aplicada a limiarização por histerese na magnitude do gradiente da imagem suavizada (multiplicada pelos cruzamentos em zero) da imagem suavizada para ligar as bordas.

Na Figura 19 pode-se visualizar como é o fluxo de dados do algoritmo.

2.6.2 Suavização gaussiana

A suavização gaussiana tem por objetivo diminuir ruídos da imagem de entrada (I) e é definida pela Equação 2.1:

$$L = I \cdot K \tag{2.1}$$

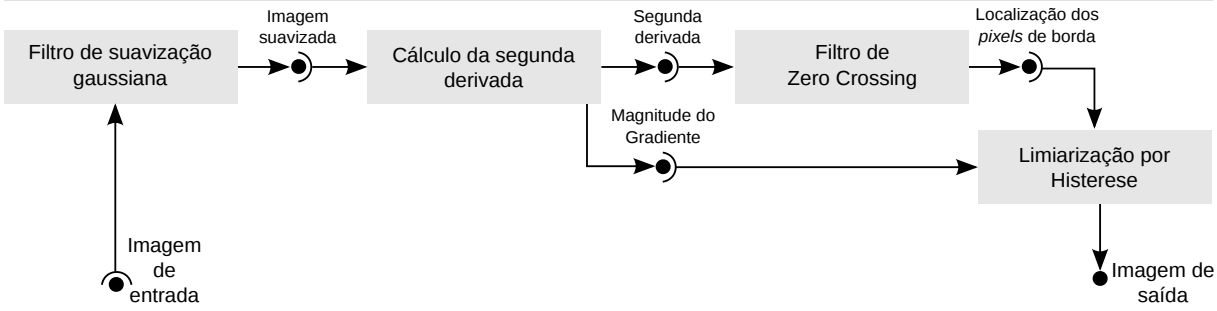


Figura 19 Fluxo de dados do algoritmo Canny para ITK

Fonte: O autor

onde K é uma máscara gaussiana separável (Seção 2.4) com tamanho definido pelo σ da função gaussiana, e L é a imagem de resultado.

2.6.3 Detecção de bordas com derivadas direcionais locais e geometria diferencial

A detecção de bordas com derivadas direcionais locais e geometria diferencial segue o algoritmo proposto por Lindeberg (1998).

O cálculo das derivadas direcionais locais de uma imagem bidimensional é realizado adicionando em cada ponto (x_0, y_0) da imagem um sistema de coordenadas (u, v), com o eixo-v paralela à direção do gradiente em (x_0, y_0), e a direção-u perpendicular, de acordo com a Equação 2.2. No Algoritmo 2 encontram-se os passos para o cálculo das derivadas de segunda ordem.

$$\left(\frac{cos\alpha}{sen\alpha}\right) = \frac{1}{\sqrt{L_x^2 + L_y^2}} \left(\frac{L_x}{L_y}\right)\Bigg|_{(x_0,y_0)} \tag{2.2}$$

As derivadas direcionais de primeira ordem no sistema-(u, v) são relacionadas à derivadas parciais no sistema de coordenadas cartesiano:

$$\begin{cases} \partial_u = sen\ \alpha\ \partial_x - cos\alpha\ \partial_y, \\ \partial_v = cos\ \alpha\ \partial_x + sin\alpha\ \partial_y, \end{cases}, \tag{2.3}$$

ou,

$$\begin{cases} L_x = L \cdot \partial_x \\ L_y = L \cdot \partial_y \\ L_{xx} = L \cdot \partial_{xx} \\ L_{xy} = L \cdot \partial_{xy} \\ L_{yy} = L \cdot \partial_{yy} \end{cases}, \tag{2.4}$$

onde:

$$\partial_x = \begin{bmatrix} \frac{-1}{2} & 0 & \frac{1}{2} \end{bmatrix}, \quad \partial_y = \begin{bmatrix} \frac{1}{2} \\ 0 \\ \frac{-1}{2} \end{bmatrix}, \tag{2.5}$$

$$\partial_{xx} = \begin{bmatrix} 1 & -2 & 1 \end{bmatrix}, \quad \partial_{yy} = \begin{bmatrix} 1 & -2 & 1 \end{bmatrix}^T, \tag{2.6}$$

$$\partial_{xy} = \begin{bmatrix} \frac{-1}{4} & 0 & \frac{1}{4} \\ 0 & 0 & 0 \\ \frac{1}{4} & 0 & \frac{-1}{4}, \end{bmatrix} \tag{2.7}$$

De acordo com a noção de *non-maximum suppression*, um ponto de borda é definido como um ponto com a magnitude do gradiente assumindo um máximo na direção do gradiente. Nos termos das derivadas direcionais, esta definição pode ser expressada como a derivada direcional de segunda ordem na direção-v L_{vv} sendo zero, e a derivada direcional de terceira ordem na mesma direção L_{vvv} sendo negativa (LINDEBERG, 1998 apud CANNY; KORN, 1986,

1988), definido pela Equação 2.8:

$$\begin{cases} L_{vv} = 0, \\ \\ L_{vvv} < 0. \end{cases} \tag{2.8}$$

Uma vez que somente a informação de sinal é importante, L_{vv} e L_{vvv} podem ser definidas aproximadamente pelas Equações 2.9 e 2.10, respectivamente:

$$\tilde{L}_{vv} = \frac{(L_x^2 L_{xx}) + 2L_x L_y L_{xy} + (L_y^2 L_{yy})}{L_x^2 + L_y^2} \tag{2.9}$$

$$\tilde{L}_{vvv} = L_{vvx}\left(\frac{L_x}{L_v}\right) + L_{vvy}\left(\frac{L_y}{L_v}\right), \tag{2.10}$$

onde:

$$\begin{cases} L_{vvx} = L_{vv} * \partial_x \\ \\ L_{vvy} = L_{vv} * \partial_y \end{cases} \tag{2.11}$$

O cálculo da magnitude do gradiente é demonstrado no Algoritmo 3.

Algoritmo 2: Cálculo das derivadas de segunda ordem

1. Para cada *pixel* da imagem L calcular as derivadas parciais locais L_x, L_y, L_{xx}, L_{yy}, L_{xy} com a Equação 2.4;
2. Computar as derivadas direcionais de segunda ordem na direção-v L_{vv} de acordo com a Equação 2.9;

2.6.4 *Non-maximum Suppression*

Non-maximum Suppression (*NMS*) é o terceiro passo do algoritmo, e é utilizado para encontrar os *pixels* com cruzamento em zero (*zero-crossing*) na derivada de segunda ordem (L_{vv}). Os cruzamentos em zero ocorrem nas posições onde o sinal muda ou onde encontram-

Algoritmo 3: Algoritmo de cálculo da magnitude do gradiente

para cada *pixel de L e* L_{vv} **faça**
  Calcular as derivadas parciais L_x, L_y de acordo com as Equações 2.4;
  Calcular as derivadas parciais L_{vvx} e L_{vvy} de acordo com as Equações 2.11;
  Calcular a derivada direcional de terceira ordem na direção-v L_{vvv} de acordo com a Equação 2.10;
  $L_v \leftarrow \begin{cases} \sqrt{L_x^2 + L_y^2}, \text{ se } L_{vvv} \leq 0, \\ 0, \text{caso contrário.} \end{cases}$
fim para cada

se valores nulos seguidos de valores não-nulos na vizinhança-4[12]. O resultado é uma imagem binária (Z_C) contendo *pixels* com valor 1 nas posições onde ocorre o cruzamento em zero.

2.6.5 Limiarização por histerese

A limiarização por histerese é o quarto e último passo do algoritmo, e seu objetivo é "ligar" os pontos de bordas. Ligar os pontos de borda significa seguir o caminho de maior L_v (saída do *NMS*), dependendo de dois limiares T_H (limiar superior) e T_L (limiar inferior). Sua saída é uma imagem binária contendo bordas com um *pixel* de espessura, como visto no Algoritmo 4.

Algoritmo 4: Algoritmo de limiarização por histerese

Dados: imagem *NMS*
Dados: limiar inferior T_L
Dados: limiar superior T_H
para *todos os pixels em* $NMS[i]$ **faça**
  se $NMS[i] > T_H$ **então**
    marcar como borda e enfileirar na lista l todos os *pixels* vizinhança-4 com $NMS[i] > T_L$;
    enquanto l *não estiver vazia* **faça**
      desenfileirar o primeiro *pixel* e marcá-lo como borda;
      verificar a vizinhança-4 e enfileirar em l os pixels com $NMS[i] > T_L$;
    fim enqto
  fim se
fim para

Finalmente, no Algoritmo 5 pode-se visualizar todos os passos seguidos na implemen-

[12] A vizinhança-4 tem as seguintes direções: Norte (N), Sul (S), Leste (E) ou Oeste (W)

tação.

Algoritmo 5: Algoritmo de detecção de bordas para ITK

1. Convolucionar máscaras Gaussianas separáveis na imagem de entrada (I) resultando em uma imagem suavizada L;
2. Calcular as derivadas de primeira (L_{vx}, L_{vy}), segunda e terceira (L_{vv}, L_{vvv}) ordens da imagem suavizada, e a magnitude do gradiente (L_v);
3. Detectar *Zero-Crossings* (ZC) em L_{vv} e multiplicar o resultado por L_v;
4. Fazer a limiarização por histerese (HT), ligando os pontos de borda;

3 Desenvolvimento

Neste capítulo são descritas as implementações dos algoritmos de convolução e do algoritmo de detecção de bordas Canny, otimizadas com SSE e OpenMP, bem como as alterações efetuadas na biblioteca ITK para que fosse possível utilizar *strides* de imagem com tamanho variável.

3.1 Implementações dos algoritmos de convolução

Para verificar as melhores abordagens de otimização implementou-se várias versões do algoritmo de convolução (Seção 2.4). A partir de um algoritmo simples (*naive*[1]), tanto para a convolução 2D quanto para a convolução 2D separável, realizaram-se incrementos na otimização para medir e comparar o desempenho.

Todos as imagens (*buffers*) e máscaras das implementações utilizam tipos de dados de ponto-flutuante de precisão simples de 32 bits (*floats*), sendo unidimensionais. A opção por dados ponto-flutuante é em razão de sua utilização na representação de imagens médicas. Além disso, todos os laços mais externos dos algoritmos são paralelizados utilizando OpenMP com escalonador dinâmico.

Nesta seção são descritos as seguintes implementações do algoritmo de convolução:

- `naiveConvolve` - Implementação do algoritmo de convolução 2D simples de referência;
- `alignedConvolve` - Implementação do algoritmo de convolução 2D com otimização de

[1] Ingênuo, óbvio, pouco sofisticado

alocação de imagens com *strides* alinhados com o tamanho da linha de *cache* da arquitetura;

- `loopUnrollConvolve` - Implementação do algoritmo de convolução 2D com otimização de desdobramento de laço;
- `sseUnalignedConvolve` - Implementação do algoritmo de convolução 2D com otimização de vetorização SSE explícita com instruções de carregamentos e armazenamentos desalinhados;
- `sseConvolve` - Implementação do algoritmo de convolução 2D com otimização de vetorização SSE explícita com instruções de carregamentos alinhados e armazenamentos desalinhados;
- `sseNConvolve` - Implementação do algoritmo de convolução 2D com otimizações específicas para cada tamanho de máscara, utilizando os registradores de maneira mais eficiente;
- `separableConvolve` - Implementação do algoritmo de convolução 2D separável simples de referência;
- `sseSConvolve` - Implementação do algoritmo de convolução 2D separável com otimização de vetorização SSE explícita com instruções de carregamentos alinhados e armazenamentos desalinhados;
- `sseNSConvolve` - Implementação do algoritmo de convolução 2D separável com otimizações específicas para cada tamanho de máscara, utilizando os registradores de maneira mais eficiente;
- `prefetchConvolve` - Implementação do algoritmo de convolução 2D com otimização de *prefetch* por software;

Todo o código-fonte das implementações dos algoritmos pode ser encontrado em:

`http://web.inf.ufpr.br/vri/alumni/2012-TonyHild`

Ressalta-se que o repositório citado ficará disponível por tempo indeterminado, sob responsabilidade do grupo de pesquisa Visão, Róbotica, Imagem (VRI) (`http://web.inf.ufpr.br/vri`).

3.1.1 Implementação `naiveConvolve`

A implementação `naiveConvolve` é utilizada como referência para a medida de desempenho dos outros convolucionadores 2D. Seu pseudocódigo é apresentado no Algoritmo 6.

Algoritmo 6: Implementação `naiveConvolve`

Dados: uma imagem de entrada inputImage de tamanho imageWidth $\times$ imageHeight
Dados: uma máscara kernel de tamanho kernelWidth $\times$ kernelWidth
Resultado: a imagem de saída outputImage de tamanho imageWidth $\times$ imageHeight convolucionada pelo kernel

```
1  halfKernel ← kernelWidth ÷ 2;
2  maxY ← imageHeight − 2 × halfKernel;
3  maxX ← imageWidth − 2 × halfKernel;
4  para y ← 0 até maxY em paralelo faça
5      para x ← 0 até maxX faça
6          dotproduct ← 0;
7          para r ← 0 até kernelWidth faça
8              idxM ← r × kernelWidth;
9              idxImg ← (y + r) × imageWidth + x;
10             para c ← 0 até kernelWidth faça
11                 dotproduct = dotproduct + kernel[idxM + c] ∗ inputImage[idxImg + c];
12             fim para
13         fim para
14         outputImage[(y + halfKernel) ∗ imageWidth + (x + halfKernel)] ←
           dotproduct;
15     fim para
16 fim para
17 processar pixels das bordas da imagem de largura da imagem;
```

3.1.2 Implementação `alignedConvolve`

A implementação `alignedConvolve` diferencia-se da implementação `naiveConvolve` (Seção 3.1.1) por utilizar imagens, tanto de entrada como de saída, com *strides* alinhados (verificado nas Linhas 9 e 14 do Algoritmo 7).

Algoritmo 7: Algoritmo `alignedConvolve`

Dados: uma imagem de entrada inputImage de tamanho imageWidth $\times$ imageHeight
Dados: uma máscara kernel de tamanho kernelWidth $\times$ kernelWidth
Resultado: a imagem de saída outputImage de tamanho imageWidth $\times$ imageHeight convolucionada pelo kernel

```
1  halfKernel ← kernelWidth ÷ 2;
2  maxY ← imageHeight − 2 × halfKernel;
3  maxX ← imageWidth − 2 × halfKernel;
4  para y ← 0 até maxY em paralelo faça
5      para x ← 0 até maxX faça
6          dotproduct ← 0;
7          para r ← 0 até kernelWidth faça
8              idxM ← r × kernelWidth;
9              idxImg ← (y + r) × imageStride + x;
10             para c ← 0 até kernelWidth faça
11                 dotproduct = dotproduct + kernel[idxM + c] * inputImage[idxImg + c];
12             fim para
13         fim para
14         outputImage[(y + halfKernel) * imageStride + (x + halfKernel)] ←
           dotproduct;
15     fim para
16 fim para
17 processar pixels das bordas da imagem de largura da imagem;
```

A variável *imageStride* (Linha 9) é calculada com o auxílio da função `calculateAlignedStride` (Listagem 6, Página 23);

3.1.3 Implementação `loopUnrollConvolve`

A implementação `loopUnrollConvolve` (Apêndice A, Listagem 21) foi otimizada com desdobramento de laço (Seção 2.3.1).

O laço mais interno (Algoritmo 7, Linha 5), que itera as colunas da imagem, foi desdobrado em 32 chamadas à função (`inline`) `convolution` (Listagem 7). Esta função realiza a operação de produto escalar da máscara com a respectiva região da imagem e retorna o resultado. Ressalta-se que o valor de 32 desdobramentos foi escolhido empiricamente.

Listagem 7 Função para calcular produto escalar

```
1  inline float convolution(const float* __restrict input, const int s,
2                           const float* __restrict kernel,
3                           const int ks, const int kw,
4                           const int x, const int y)   {
5      float sum = 0;
6      for (int r = 0; r < kw; ++r) {
7          const int idxFtmp = r * ks;
8          const int idxIntmp = (y + r) * s + x;
9          for (int c = 0; c < kw; ++c) {
10             sum += kernel[idxFtmp + c] * input[idxIntmp + c];
11         }
12     }
13     return sum;
14 }
```

3.1.4 Implementação `sseUnalignedConvolve`

A implementação `sseUnalignedConvolve` é a primeira versão que faz uso de instruções SSE diretamente no código, além das otimizações do compilador. São utilizadas instruções intrínsecas do compilador que emitem diretamente uma instrução SSE correspondente em *assembly* (Seção 2.3.6).

Como já mencionado na Seção 2.3.5, no conjunto de instruções SSE existem dois tipos de instruções de carregamento e armazenamento: alinhado ou desalinhado. No primeiro tipo, necessariamente, o endereço dos dados a serem carregados ou armazenados deve estar alinhado em 16 bytes. Já no segundo tipo, o alinhamento do endereço de memória não importa. Nesta implementação fez-se uso de instruções de carregamento e armazenamento desalinhados para a imagem, mas com carregamento alinhado para a máscara.

A utilização de SSE na função `sseUnalignedConvolve` (Listagem 8) inicia-se nas Linhas 11 e 12, com a declaração de duas variáveis do tipo `__m128`, que armazenam 4 *floats* de 32 bits. Na variável `kv` é carregada parte da máscara, e na variável `sum` é armazenado o produto escalar.

Pode-se observar na Linha 13 que as colunas da imagem são incrementadas em 4 elementos, pois este é o número de elementos que são carregados e processados, de uma só vez, por iteração. Na Linha 14, a variável `sum` é iniciada com zeros, utilizando a função `_mm_¬ setzero_ps`. O laço que itera as colunas da máscara, da mesma forma que o laço das colunas da imagem, é incrementado em 4. Na Linha 19, `kv` recebe 4 elementos da máscara, por meio

da função `_mm_load_ps`. Esta função emite a instrução `MOVAPS` e espera como parâmetro um ponteiro `p` *float* de 32 bits alinhado em 16 bytes, e caso o ponteiro não esteja alinhado uma exceção será disparada.

Das Linhas 23 até 26 é realizado o cálculo do produto escalar. A acumulação com o operador `+=`, implicitamente, emite a instrução `ADDPS`, que soma 4 *floats* de 32 bits de uma só vez. A função `_mm_loadu_ps`, que emite a instrução `MOVUPS`, diferentemente da função `_mm_load_ps`, carrega 4 *floats* de 32 bits a partir de um ponteiro `p`, independentemente de o mesmo estiver alinhado ou não. Esta função é responsável por carregar parte da imagem necessária para o cálculo de produto escalar. Finalmente, a função `_mm_dp_ps` (`dp` de *dot product*) calcula o produto escalar entre parte da máscara e parte da imagem carregadas. Esta função tem a seguinte assinatura:

```
__m128 _mm_dp_ps(__m128 a, __m128 b, const int mask);
```

onde `a` e `b` são os vetores operandos do produto escalar computado baseado na máscara `mask` (que pode ser inteira ou hexadecimal). A máscara indica quais elementos dos vetores serão utilizados para o cálculo e em qual posição o resultado será armazenado no vetor de retorno. Por exemplo, para `mask=0xf1`, os quatro bits mais significativos indicam que todos os elementos dos operandos serão utilizados para a operação de produto escalar, e os quatro bits menos significativos indicam que o resultado será copiado apenas na primeira posição do vetor de retorno. Na Linha 29 encontra-se a instrução de armazenamento do resultado final do produto escalar. A função `_mm_storeu_ps` recebe dois parâmetros: um ponteiro `p`, que pode ser desalinhado, indicando onde o vetor será armazenado, e o próprio vetor `a` do tipo `__m128`.

3.1.5 Implementação `sseConvolve`

A implementação `sseConvolve` realiza uma convolução com carregamentos alinhados e armazenamentos desalinhados, utilizando a técnica *rotação de vetores*, criada especialmente para este caso.

Listagem 8 Implementação `sseUnalignedConvolve`

```
void sseUnalignedConvolve (const int s, const int w, const int h,
                           const int ks, int kw,
                           const float* input, float* output,
                           const float* kernel) {

    int hk      = kw / 2;
    int startX  = 0;
    int stopX   = w - hk * 2;
    int startY  = 0;
    int stopY   = h - hk * 2;

    #pragma omp parallel for shared (input, output)
    for (int y = startY; y < stopY; ++y) {
        register __m128 sum;
        register __m128 kv;
        for (int x = startX; x < stopX; x += 4) {
            sum = _mm_setzero_ps();
            for (int r = 0; r < kw; ++r) {
                int idxFtmp = r * ks;
                int idxIntmp = (y + r) * s + x;
                for (int c = 0; c < kw; c += 4) {
                    kv = _mm_load_ps(&kernel[idxFtmp + c]);
                    sum += _mm_dpf1_ps(kv, _mm_loadu_ps(&input[idxIntmp + c]));
                    sum += _mm_dpf2_ps(kv, _mm_loadu_ps(&input[idxIntmp + c + 1]));
                    sum += _mm_dpf4_ps(kv, _mm_loadu_ps(&input[idxIntmp + c + 2]));
                    sum += _mm_dpf8_ps(kv, _mm_loadu_ps(&input[idxIntmp + c + 3]));
                }
            } //for (int r = 0...
            _mm_storeu_ps(&output[(y + hk) * s + (x + hk)], sum);
        } //for (int x = 0...
    } //for (int y = 0...
    processBoundaries2D (s, w, h,
                         ks, kw,
                         input, output, kernel);
}
```

A técnica *rotação de vetores* consiste em rotacionar, à esquerda ou à diretia, um ou mais vetores SSE sequenciais. A Figura 20 exemplifica a rotação de 4 vetores SSE à esquerda. A rotação do exemplo é realizada com o auxílio das funções macro `MOVE_ROTATE4_LEFT` (Listagem 11), `MOVE_ROTATE_LEFT` (Listagem 10) e `MOVE_ROTATE4_LEFT` (Listagem 9).

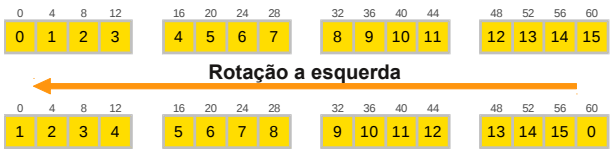


Figura 20 Técnica de *rotação de vetores* - Rotação de 4 vetores à esquerda

Fonte: O autor

Listagem 9 Macro MOVE_ROTATE4_LEFT

```
#define MOVE_ROTATE4_LEFT(vector0, vector1, vector2, vector3, vector4) \
    MOVE_ROTATE_LEFT(vector0, vector1) \
    MOVE_ROTATE_LEFT(vector1, vector2) \
    MOVE_ROTATE_LEFT(vector2, vector3) \
    MOVE_ROTATE_LEFT(vector3, vector4) \
    ROTATE_LEFT(vector4)
```

Listagem 10 Macro MOVE_ROTATE_LEFT

```
1 #define MOVE_ROTATE_LEFT(vector0, vector1) \
2     vector0 = _mm_move_ss(vector0, vector1); \
3     ROTATE_LEFT(vector0);
```

Listagem 11 Macro ROTATE_LEFT

```
1 #define ROTATE_LEFT(vector) \
2     vector = _mm_shuffle_ps(vector, vector, _MM_SHUFFLE(0,3,2,1));
```

As macros fazem uso de duas operações SSE: de *shuffle* (embaralhamento) e *move* (movimentação). Para a operação de *shuffle* é utilizada a função `_mm_shuffle_ps` que emite a instrução SSE `SHUFPS`. Esta função seleciona dois valores de cada um dos vetores de origem (`m1`, `m2`), de acordo com uma máscara `k`. A máscara pode ser obtida por meio da função macro `_MM_SHUFFLE(p3, p2, p1, p0)`, na qual a posição dos parâmetros indica o destino no vetor resultante, e o valor do parâmetro indica a origem nos vetores `m1 (p1, p2)` e `m2 (p3, p4)`.

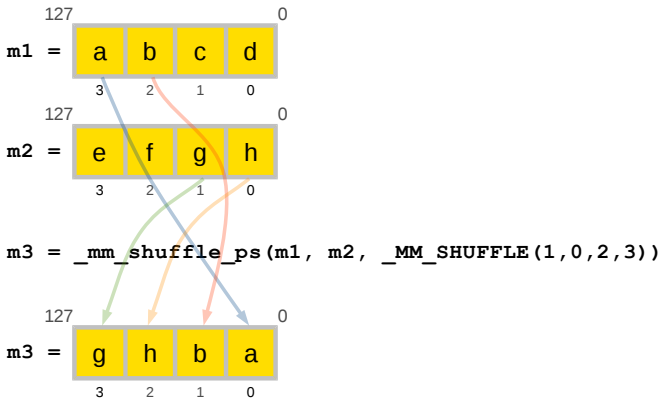


Figura 21 Macro _MM_SHUFFLE

Fonte: O autor

A operação de movimentação utiliza a função `_mm_move_ss (m1, m2)`. Esta função emite a instrução `MOVSS`, que move a palavra menos significativa de `m1` para `m2`.

O Algoritmo 8 e a Figura 22 demonstram a implementação do algoritmo `sseReuse¬4Convolve` (Apêndice A, Listagem 22). Nesta implementação em específico[2], para todo *ix*

[2] Versão da implementação do algoritmo sseConvolve que rotaciona 5 vetores de imagem e armazena 4 vetores

(coluna da imagem, Figura 22a), são criados 4 vetores acumuladores, que armazenam o resultado final da convolução.

Algoritmo 8: Implementação `sseConvolve`

Dados: uma imagem de entrada inputImage de tamanho imageWidth $\times$ imageHeight
Dados: uma máscara kernel de tamanho kernelWidth $\times$ kernelWidth
Resultado: a imagem de saída outputImage de tamanho imageWidth $\times$ imageHeight convolucionada pelo kernel

```
1  halfKernel ← kernelWidth ÷ 2;
2  maxY ← imageHeight − 2 × halfKernel;
3  maxX ← imageWidth − 2 × halfKernel;
4  para y ← 0 até maxY em paralelo faça
5  |   sum0, sum1, sum2, sum3 ← setzero ;
6  |   para x ← 0 até maxX incrementado por 16 faça
7  |   |   Ver Figura 22b
8  |   fim para
9  |   processar pixels das bordas da imagem de largura da imagem;
10 fim para
```

Para todo *ky* (linha da máscara, Figura 22b), são carregados 4 blocos[3] de imagem em vetores SSE. Para todo *kx*, (coluna da máscara, Figura 22b) é carregado um bloco de máscara, o quinto bloco de imagem, e operado o produto escalar utilizando a técnica de *rotação de vetores*. Ao final de uma iteração tem-se quatro vetores SSE contendo o resultado de 16 *pixels* convolucionados da imagem, como visualizado na Figura 22c

de resultado por iteração de *x*

[3] Quatro vetores contendo quatro elementos

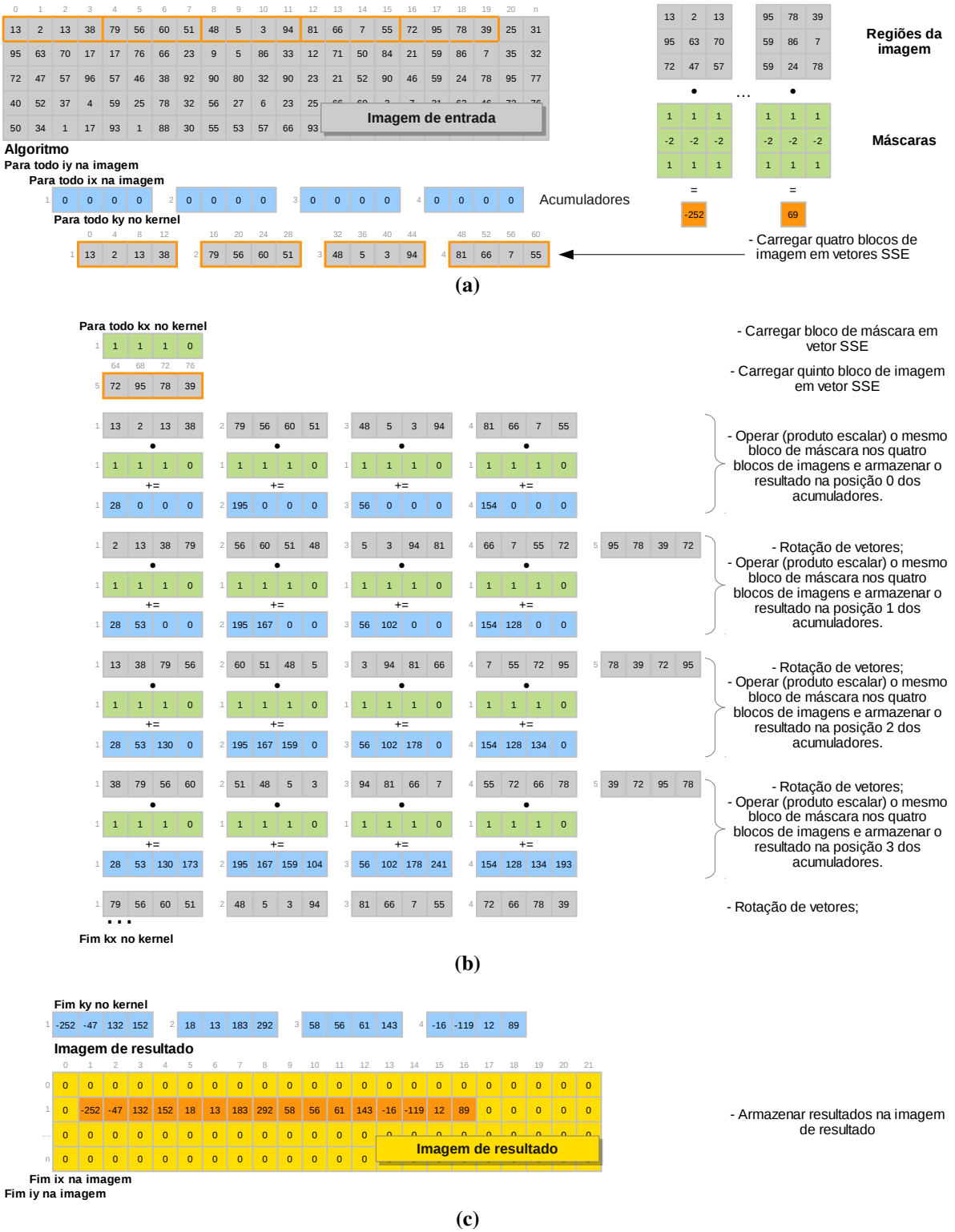


Figura 22 Implementação de convolução alinhada com SSE

Fonte: O autor

3.1.6 Implementação `separableConvolve`

A implementação `separableConvolve` (Apêndice A, Listagem 23) é utilizada de referência para os testes de convolução 2D separável (Seção 2.4). Seu pseudocódigo é apresentado no Algoritmo 9.

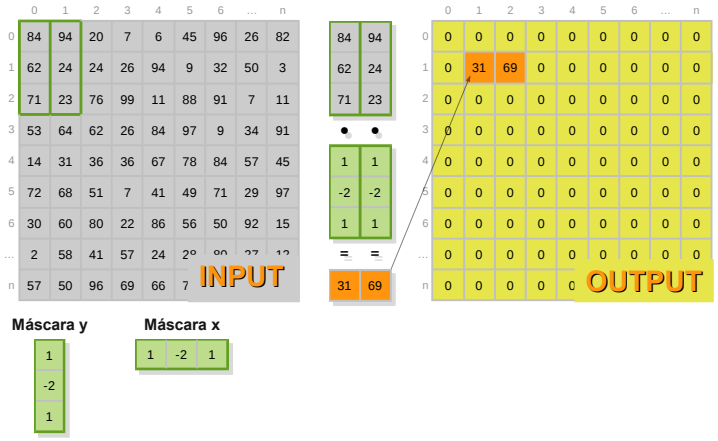
Primeiramente, é realizada a convolução com a máscara *y* (Figura 23a) de tantos elementos quanto a *largura da máscara - 1* e o resultado é armazenado na imagem de saída, na mesma posição do resultado final. Então, é processado o elemento correspondente à *largura da máscara* (Figura 23b). Depois é operada a máscara *x* com o resultado das operações da máscara *y*, e o resultado final é armazenado (Figura 23c).

Há que se destacar que mesmo sendo realizadas duas convoluções, das máscaras *x* e *y*, não é necessária uma imagem temporária para armazenar o resultado intermediário, pois a própria imagem de saída serve a este propósito. Esta é uma otimização, que além de economizar memória, mantém a localidade espacial de memória *cache*.

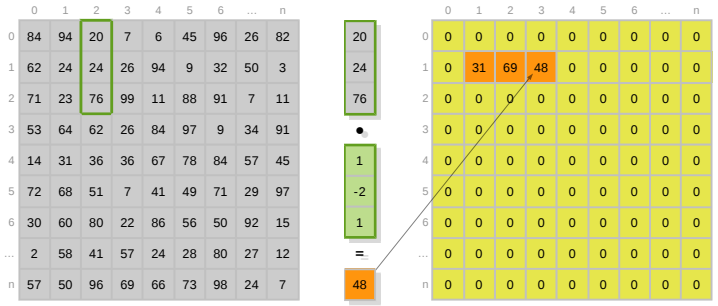
Algoritmo 9: Algoritmo de convolução 2D separável simples

Dados: uma imagem de entrada inputImage de tamanho imageWidth $\times$ imageHeight
Dados: duas máscaras kernelY e kernelX de tamanho kernelWidth $\times$ kernelWidth
Resultado: a imagem de saída outputImage de tamanho imageWidth $\times$ imageHeight convolucionada pelas máscaras

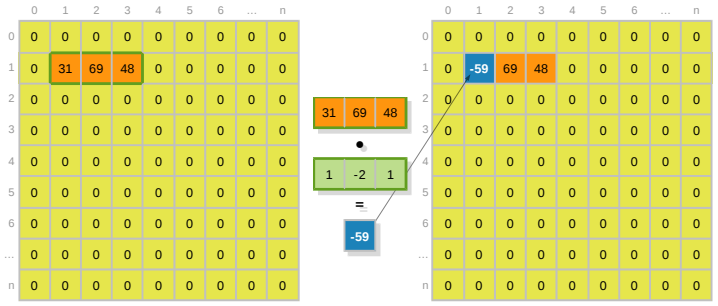
```
1  halfKernel ← kernelWidth ÷ 2;
2  maxY ← imageHeight − 2 × halfKernel;
3  maxX ← imageWidth − 2 × halfKernel;
4  para y ← 0 até maxY em paralelo faça
5      para x ← 0 até kernelWidth − 1 faça
6          Convolucionar x colunas com o kernelY (Figura 23a);
7      fim para
8      para x ← 0 até max faça
9          Convolucionar a coluna x + kernelWidth − 1 com o kernelY (Ver Figura 23b);
10         Convolucionar a linha y + halfKernel com o kernelX (Ver Figura 23c);
11     fim para
12 fim para
13 processar pixels das bordas da imagem de largura da imagem;
```



(a) Passo 1



(b) Passo 2



(c) Passo 3

Figura 23 Algoritmo de convolução 2D separável simples

Fonte: O autor

3.1.7 Implementação `sseSConvolve`

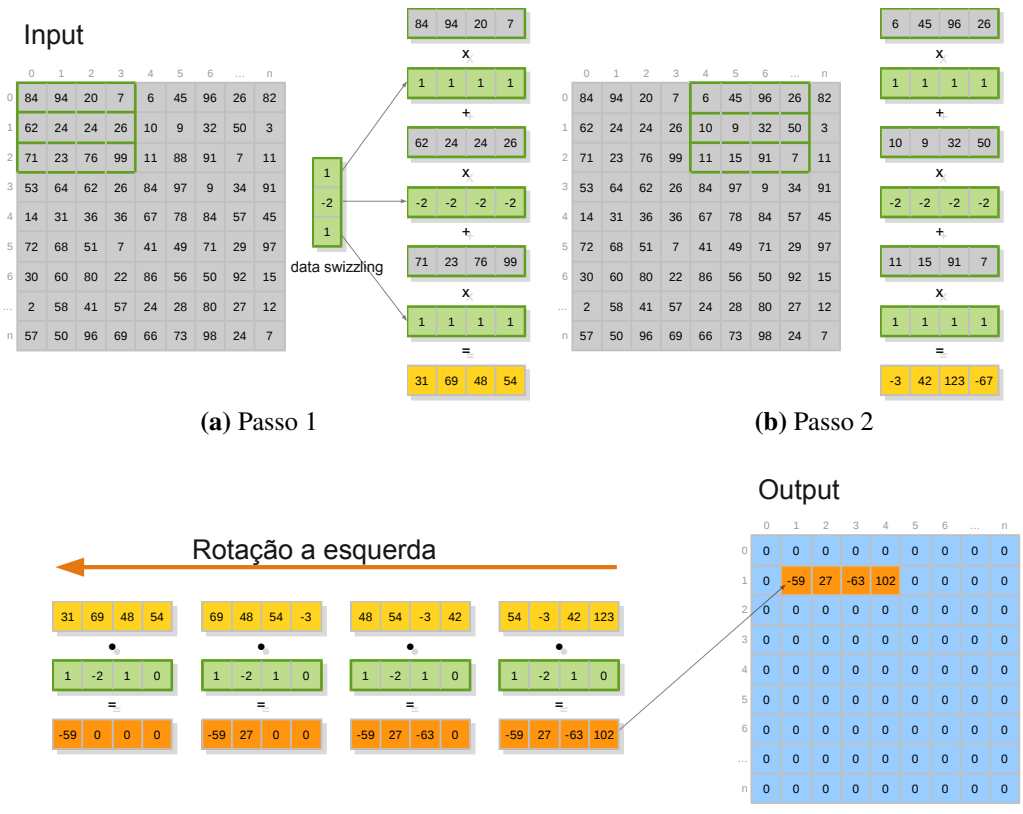
A implementação `sseSConvolve` (Apêndice A, Listagem 24) segue o mesmo princípio da implementação `separableConvolve` (Seção 3.1.6), mas ao invés de utilizar a própria imagem de saída como imagem temporária, são utilizados vetores SSE armazenados em registradores, permitindo operações de transferência de memória com latência menor.

Para a convolução da máscara *y* é utilizada a técnica de *data swizzling* (Seção 2.3.4),

permitindo que sejam calculados os produtos escalares de quatro colunas de uma só vez, como pode ser visualizado na Figura 24a e Figura 24b.

Para a convolução da máscara *x* é utilizada a mesma técnica de *rotação de vetores* da implementação `sseConvolve` (Seção 3.1.5), com a diferença de ser unidimensional. A Figura 24c mostra apenas o resultado de um vetor como exemplo, mas na implementação são calculados quatro vetores por iteração.

A operação das máscaras *y* e *x* é feita de forma intercalada, parcial e incremental. Mesmo sendo operada apenas parte da máscara *y*, seu resultado é acumulado com o resultado parcial da máscara *x*. Isto permite que sejam realizadas operações com máscaras grandes sem haver problemas de falta acumuladores temporários (registradores SSE) das operações da máscara *y*.



(a) Passo 1

(b) Passo 2

(c) Passo 3

Figura 24 Algoritmo de convolução 2D separável com SSE

Fonte: O autor

3.1.8 Implementações `sseNConvolve` e `sseNSConvolve`

As implementações `sseNConvolve` e `sseNSConvolve` são implementações otimizadas dos algoritmos `sseConvolve` e `sseSConvolve` (Seção 3.1.5, Seção 3.1.7) para máscaras com larguras de três, cinco, sete e nove. Para os algoritmos de convolução 2D as implementações são: `sse3Convolve`, `sse5Convolve`, `sse7Convolve`, `sse9Convolve`. Para os algoritmos de convolução 2D separável as implementações são: `sse3SConvolve`, `sse5SConvolve`, `sse7¬SConvolve`, `sse9SConvolve`.

A principal diferença destas implementações com relação às anteriores, reside no fato destas utilizarem de forma mais racional os registradores SSE, permitindo, por exemplo, manter toda ou parte da máscara nos registradores, evitando acessos à memória, além de utilizar a técnica de *rotação de vetores* otimizada para cada caso.

3.1.9 Implementação `prefetchConvolve`

A implementação de convolução 2D com utilização de *prefetch* visa testar a eficiência da utilização de *prefetch* manual feita pelo programador. Como base desta implementação foi utilizada a implementação com desdobramento de laço `loopUnrollConvolve` (Seção 3.1.3). Antes de cada iteração do laço x (das colunas da imagem) é realizado o *prefetch* de N linhas, onde N é a largura da máscara, a partir do endereço de memória da iteração seguinte de x.

3.2 Alterações realizadas na biblioteca ITK

Para que fosse possível utilizar instruções SSE com acessos a memória alinhados na biblioteca ITK, foi necessária a realização de alterações, adicionando à biblioteca a funcionalidade de trabalhar com imagens com *strides* variáveis e alocar *buffers* de imagem alinhados.

A partir destas alterações criou-se um *patch* que pode encontrado em

`http://web.inf.ufpr.br/vri/alumni/2012-TonyHild`

A seguir mostra-se um resumo das classes adicionadas:

- `itk::AlignedBuffer` - Métodos de ajuda para cálculo de *strides* alinhados e alocação alinhada;
- `itk::StopWatch` - Classe utilizada para medir desempenho em alta resolução (explicada com detalhes na Seção 4.1).

A seguir mostra-se um resumo das classes alteradas:

- `itk::ImageBase` - Para trabalhar com *strides* variáveis alterou-se o escopo do método `CalculateOffsetTable`, de protegido para virtual.
- `itk::Image` - Sobrescrita do método `CalculateOffsetTable` permitindo que todas as imagens tenham o *stride* calculado de forma alinhada com auxílio da classe `itk::AlignedBuffer`;
- `itk::ImportImageContainer` - Na classe `itk::ImportImageContainer`, responsável pela alocação do *buffer*, tal alocação é marcada com o atributo de alinhamento do compilador (Listagem 12), garantindo que todas as linhas da imagem iniciem-se com os mesmos múltiplos de endereço (Seção 2.3.5);
- `itk::ImageFileReader` - Conversão do *buffer* desalinhado para alinhado;
- `itk::ImageFileWriter` - Conversão do *buffer* alinhado para desalinhado;
- `itk::ConstNeighborhoodIterator` - Alteração na forma como são calculadas as distâncias entre elementos da imagem, e.g., *pixels*, linhas, fatias (*slices*).

Listagem 12 Atributo de alinhamento do compilador g++

```
1 TElement* data __attribute__ ((aligned(ALIGMENT_BYTES)));
```

Após realizadas estas alterações surgiram alguns problemas. O primeiro problema foi relacionado às operações de I/O. Estas operações são tratadas, em alto nível, pelas classes `itk::ImageFileReader` (responsável pela leitura) e `itk::ImageFileWriter` (responsável pela escrita), que servem como uma interface para as classes especializadas para cada tipo de imagem (i.e, PNG, JPEG, DICOM). As classes especializadas são responsáveis por ler e gravar os dados efetivamente, interagindo com bibliotecas especializadas como libpng, libgdcm, etc, e as classes `itk::ImageFileReader` e `itk::ImageFileWriter` responsáveis pela conversão dos *buffers*, caso necessário, para serem consumidos pela classe `itk::ImportImageContainer` e pelas bibliotecas de imagem. Para resolver este problema optou-se por uma conversão dos *strides* dos *buffers* de entrada e de saída, de alinhado para desalinhado e vice-versa. Apesar de não ser a solução ideal, preveniu-se a alteração de todas as classes especializadas para que retornassem *buffers* alinhados da leitura, ou aceitassem *buffers* alinhados para escrita.

O segundo problema está relacionado à forma da biblioteca iterar a imagem. De maneira geral, as operações nas imagens utilizam regiões de interesse, que são iteradas utilizando "iteradores". Existem vários tipos de "iteradores", e estes são utilizados dependendo do tipo de operação que se pretende realizar. Um dos mais comuns é o "iterador" de vizinhança, representado pela classe `itk::NeighborhoodIterator`, que pode ser utilizado, por exemplo, para operar uma convolução. Nestes "iteradores" são calculadas as distâncias (*offsets*) entre componentes da imagem, como `pixels`, linhas e fatias. Na versão original, quando é necessário acessar elementos da próxima linha da imagem, a biblioteca utiliza a largura da imagem em *pixels* para calcular esta distância, mas no caso de *buffers* com *strides* variáveis, isto resulta em um acesso errôneo. Então alterou-se o método `SetBound` da classe `itk::ConstNeighborhoodIterator` (especialização apenas-escrita da classe verblitk::NeighborhoodIteratorl) para que esta utilize o valor de *stride* para calcular a distância da próxima linha.

Dada a complexidade da biblioteca, há que se destacar que as alterações realizadas são específicas para imagens bidimensionais com *pixels* ponto flutuante de precisão simples (*floats* de 32 bits), e que foram somente para a execução do algoritmo de detecção de bordas Canny. Outro ponto a se destacar é o fato do suporte a *strides* alinhados fazer parte da "lista

de desejos" (KitwarePublic, 2012) da quarta versão da biblioteca, mas que não foi incluído na última versão lançada em dezembro de 2011.

3.3 Implementação do algoritmo de detecção de bordas Canny

Nesta seção é explicado como se deu a implementação do algoritmo de detecção de bordas Canny para a biblioteca ITK, otimizado com SSE e OpenMP (`OpCanny`). A descrição geral deste algoritmo e como foi originalmente implementado para a biblioteca ITK pode ser encontrada na Seção 2.6.

As principais razões para a escolha de implementação do algoritmo `OpCanny`:

- O algoritmo Canny naturalmente contém passos sequenciais e paralelizáveis;
- A computação é complexa o suficiente para permitir análise de desempenho;
- A implementação nativa para ITK (`ItkCanny`) é ineficiente.

A implementação `OpCanny` segue exatamente os mesmos passos da implementação `ItkCanny`, com o objetivo de obter os mesmos resultados. Para a maioria dos passos do algoritmo foram utilizadas as técnicas testadas no desenvolvimento dos algoritmos de convolução vistos na Seção 3.1.

3.3.1 Suavização gaussiana

Para K com $kernelSize = \{3,5,7,9\}$ é utilizado o algoritmo `sseNSConvolve` (Seção 3.1.8). Para outros tamanhos de máscaras é utilizado o algoritmo `sseSConvolve` (Seção 3.1.7).

3.3.2 Detecção de bordas com derivadas direcionais locais e geometria diferencial

O cálculo das derivadas direcionais-v de segunda ordem (L_{vv}) é feito de acordo com o Algoritmo 2. No Algoritmo 3 são descritos os passos para o cálculo da magnitude do gradiente (derivada primeira na direção-v) L_v. Estes algoritmos são implementados nos métodos: `Compute2ndDerivative` (Apêndice B, Listagem 25) e `Compute2ndDerivativePos` (Apêndice B, Listagem 26), respectivamente.

O método `Compute2ndDerivative` calcula as derivadas parciais utilizando a mesma técnica do algoritmo `sse3SConvolve` (Seção 3.1.8). As máscaras das derivadas L_y e L_{yy} são carregadas em registradores usando *data swizzling* (Seção 2.3.4). Para as derivadas L_x e L_{xx} é utilizada a técnica de *rotação de vetores* (Seção 3.1.5). O cálculo de L_{vv} é feito utilizando operações aritméticas SSE empacotadas, como pode ser visto na Listagem 13.

Listagem 13 Método SSE otimizado para cálculo da derivada L_{vv}

```
1 extern __inline __m128 __attribute__((__gnu_inline__, __always_inline__, __artificial__))
2 _mm_lvv_ps (__m128 lx, __m128 ly, __m128 lxx, __m128 lyy, __m128 lxy) {
3         lxy = _mm_set1_ps(2.0) * lx * ly * lxy;
4         lx *= lx;
5         ly *= ly;
6         return (lx * lxx + lxy + ly * lyy) / (lx + ly);
7 }
```

L_{vvv} é calculada pelo método `Compute2ndDerivativePos`. As técnicas utilizadas na implementação deste método são semelhantes às utilizadas no método `Compute2ndDerivative`.

Vale a pena destacar a utilização da função intrínseca `_mm_sqrt_ps`, que emite a instrução `SQRTPS`, e calcula a raiz quadrada de quatro números ponto-flutuante de precisão simples de forma empacotada.

Também foram utilizadas as seguintes funções SSE:

```
__m128 _mm_cmple_ps(__m128 a , __m128 b );
```

e

```
__m128 _mm_and_ps(__m128 a , __m128 b );
```

A primeira função emite a instrução CMPLEPS e que faz a seguinte operação (Listagem 14):

Listagem 14 Instrução CMPLEPS

```
1 r0 := (a0 <= b0) ? 0xffffffff : 0x0
2 r1 := (a1 <= b1) ? 0xffffffff : 0x0
3 r2 := (a2 <= b2) ? 0xffffffff : 0x0
4 r3 := (a3 <= b3) ? 0xffffffff : 0x0
```

A segunda função emite a instrução ANDPS (Listagem 15) e computa o E lógico em nível de bit (operação *bitwise*) entre os dois vetores. Estas funções foram utilizadas para calcular L_{vvv}, como visto na Listagem 16.

Listagem 15 Instrução ANDPS

```
1 r0 := a0 & b0
2 r1 := a1 & b1
3 r2 := a2 & b2
4 r3 := a3 & b3
```

Listagem 16 Cálculo de L_{vvv} realizada pelo OpCanny

```
1 lv = lx * lx + ly * ly;
2 lv = _mm_sqrt_ps (lv);
3 lvvv = lvvx * lx / lv + lvvy * ly / lv;
4 lvvv = _mm_cmple_ps(lvvv,  _mm_setzero_ps());
5 lvvv = _mm_and_ps(lv, _mm_set1_ps(0x00000001));
6 lvvv *= lv;
```

3.3.3 *Non-maximum Suppression*

A maior inovação na implementação deste passo do OpCanny é a verificação simultânea de quatro *pixels* de cada uma das direções, conforme a Figura 25 e o Algoritmo 10. A comparação vertical entre os elementos é simples, pois estes ficam alinhados. Para a comparação horizontal é utilizada a técnica de *rotação de vetores*, movendo elementos do vetor central para os vizinhos esquerdos e direitos, tornando possível compará-los da mesma forma como é realizada com os vizinhos superiores e inferiores (Figura 25).

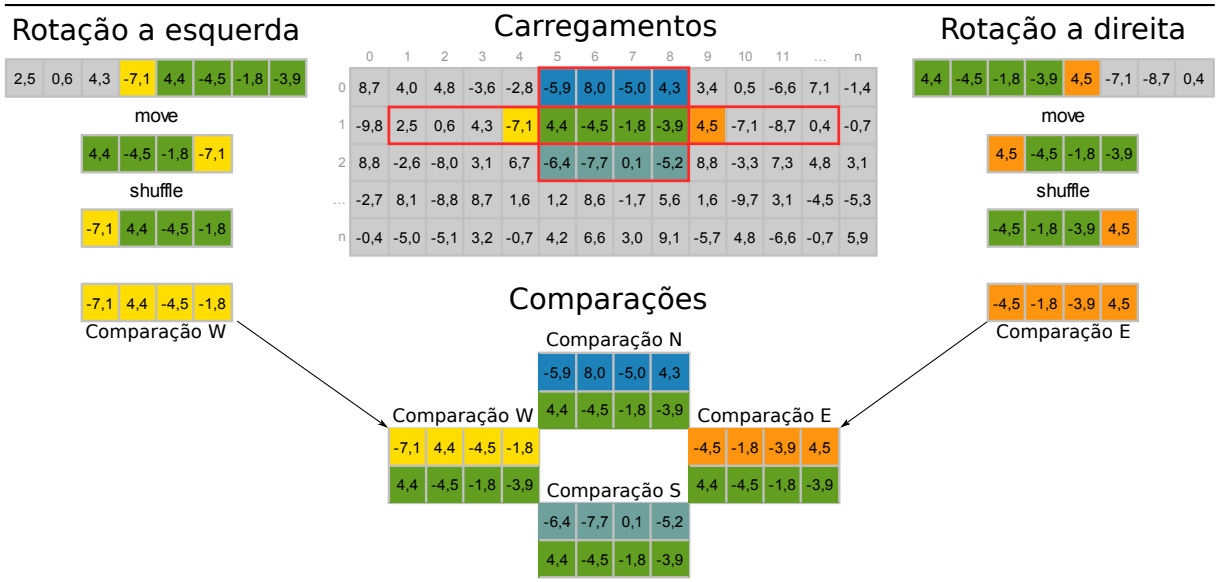


Figura 25 Comparações do método `ZeroCrossing`

Fonte: O autor

A implementação deste algoritmo utilizando SSE encontra-se no método `ZeroCrossing` (Apêndice B, Listagem 27). O fluxo de dados da vizinhança-N pode ser visto na Figura 26, e na Listagem 17 encontra-se o trecho de código deste fluxo de dados. Neste trecho há que se destacar as operações de obtenção da máscara de sinais. Inicialmente são carregados dois vetores de $L_{vv}[i]$: `thisOne` (pixels centrais da vizinhança-4) e `that` (pixels da vizinhança-N). A seguir são obtidos seus valores absolutos e armazenados em `absThis` e `absThat`, aplicando a função `_mm_and_ps` (`&`) entre estes vetores e o vetor constante, `abs`, contendo quatro elementos `0x7FFFFFFF`. Esta operação apaga o bit mais significativo que contém o sinal. Para obter a máscara de sinais também é utilizada a função `_mm_and_ps`, mas agora com a máscara `0x80000000`, preservando apenas o bit de sinal de cada elemento e armazenado-os nos vetores `maskThis` e `maskThat`. Este passo retorna um vetor contendo zeros com sinais, que é um valor especial da linguagem *C/C++* conhecido como `NaN` (*Not a Number*). Então é necessário utilizar a função `_mm_or_ps` (`|`) com a máscara `0x00000001` para transformar os ± 0 em ± 1, pois a comparação de valores `NaN` sempre é falsa, mesmo sendo os dois valores `NaN`.

Finalmente, a multiplicação de $Z_C \times L_v$ é realizada pelo método `Multiply`, utilizando SSE. O armazenamento do resultado é feito utilizando a função `_mm_stream_ps`, que faz a

Algoritmo 10: Algoritmo de NMS

1 Carregar em vetores SSE (v) os *pixels* centrais com dados da derivada de segunda ordem $L_{vv}[i]$, mais sua vizinhança-4 (Figura 25);
2 Realizar operação de *rotação de vetores* nos vetores esquerdo e direito (Figura 25);
3 Calcular o cruzamento em zero:

$$\begin{cases} v(west) \leftarrow L_{vv}[i-4]; \\ v(north) \leftarrow L_{vv}[i-stride]; \\ v(east) \leftarrow L_{vv}[i+4]; \\ v(south) \leftarrow L_{vv}[i+stride]; \end{cases}$$

$v(aux1) \leftarrow (v(L_{vv}[i]) * v(west) \leq 0) * (|v(L_{vv}[i])| < |v(west)|);$
$v(aux2) \leftarrow (v(L_{vv}[i]) * v(north) \leq 0) * (|v(L_{vv}[i])| < |v(north)|);$
$v(aux3) \leftarrow (v(L_{vv}[i]) * v(east) \leq 0) * (|v(L_{vv}[i])| < |v(east)|);$
$v(aux4) \leftarrow (v(L_{vv}[i]) * v(south) \leq 0) * (|v(L_{vv}[i])| < |v(south)|);$
$Z_C[i...i+4] \leftarrow aux1 \vee aux2 \vee aux3 \vee aux4;$

4 Retornar uma imagem NMS contendo as magnitude de gradientes (L_v) máximas locais:

$$NMS = Z_C \times L_v$$

Listagem 17 Comparações da vizinhança-4 do método `ZeroCrossing` com SSE

```
1  static const __m128 sign = _mm_castsi128_ps(_mm_set1_epi32(0x80000000));
2  static const __m128 one = _mm_set1_ps(0x00000001);
3  static const __m128 abs = _mm_castsi128_ps(_mm_set1_epi32(0x7FFFFFFF));
4
5  __m128 thisOne = _mm_load_ps(&inputImage[(y + 1) * imageStride + x]);
6  __m128 that = _mm_load_ps(&inputImage[y * imageStride + x]);
7  __m128 absThat = _mm_and_ps(that, abs);
8  __m128 absThis = _mm_and_ps(thisOne, abs);
9  __m128 maskThis = _mm_and_ps(thisOne, sign);
10 __m128 maskThat = _mm_and_ps(that, sign);
11
12 maskThis = _mm_or_ps(maskThis, one);
13 maskThat = _mm_or_ps(maskThat, one);
14
15 maskThat = _mm_cmpneq_ps(maskThis, maskThat);
16
17 __m128 lessThan = _mm_cmplt_ps(absThis, absThat);
18
19 __m128 equal = _mm_cmpeq_ps(absThis, absThat);
20
21 lessThan = _mm_or_ps(lessThan, equal);
22
23 __m128 out = _mm_and_ps(lessThan, maskThat);
```

transferência de dados sem poluir a memória *cache*. Isto é ideal neste caso pois o resultado da multiplicação não será reutilizado.

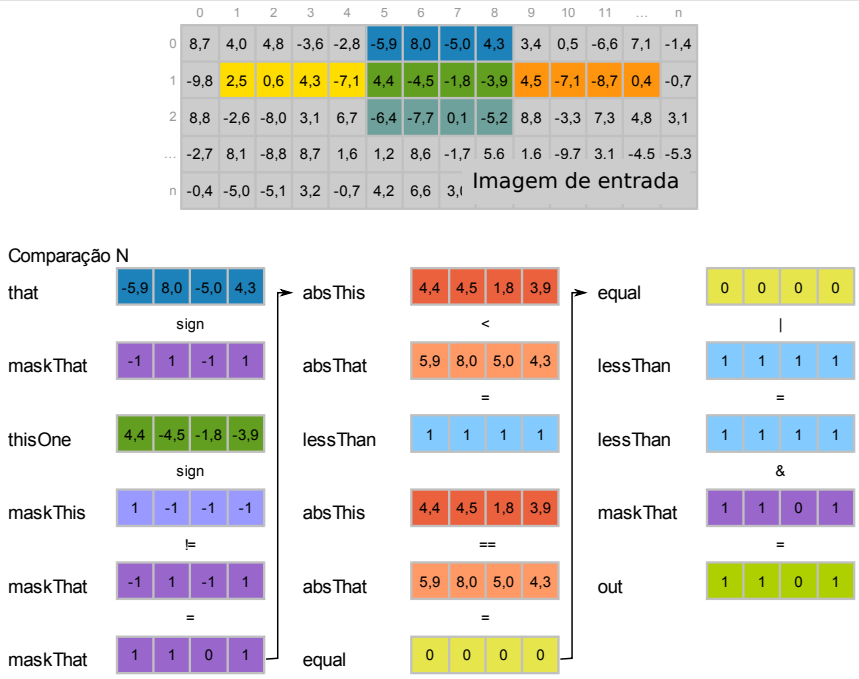


Figura 26 Fluxo de dados da "Comparação N" do método `ZeroCrossing`

Fonte: O autor

3.3.4 Limiarização por histerese

Na limiarização por histerese, o enfileiramento de *pixels* é realizado utilizando uma fila circular. Para utilizar paralelização MIMD (Seção 2.1) nesta etapa, são criadas várias filas circulares em um mesmo *buffer* temporário (utilizado pelas etapas anteriores). Tais filas são particionadas no *buffer* dividindo a altura (y) do *buffer* pelo número máximo de *threads* alocadas para a execução. Há que se destacar esta otimização por este passo ser de difícil paralelização. Apesar da possibilidade de acessos concorrentes quando uma *thread* seguir a mesma borda que outra *thread*, como a operação de histerese é idempotente, haverá somente uma gravação duplicada. Na Figura 27 encotra-se um exemplo de um possível acesso concorrente entre duas *threads*.

A implementação do algoritmo encontra-se no método `HysteresisThresholding` (Apêndice B, Listagem 28). As únicas utilizações de instruções SSE são feitas na busca por *pixels* com $L_v > T_H$. Neste caso compara-se os valores de entrada, carregados em um vetor SSE, com outro vetor contendo T_H, utilizando a função `_mm_cmpgt_ps` (>), e extraindo a más-

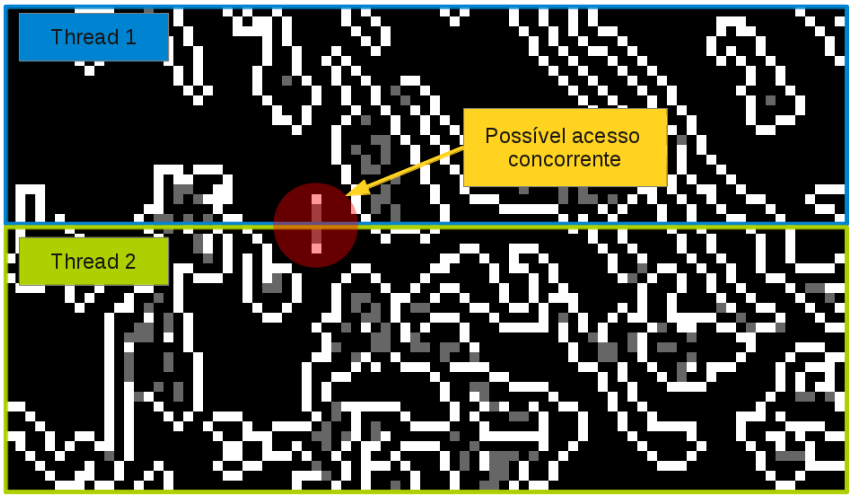


Figura 27 Possibilidade de acesso concorrente - Histerese

Fonte: O autor

cara de bits deste resultado por meio da função `_mm_movemask_ps`. Esta função, que emite a instrução `MOVMSKPS`, retorna uma máscara de 4 bits com os bits mais significativos dos quatro elementos do vetor. Caso o resultado da máscara seja 0, o laço continua, senão a máscara é verificada bit a bit indicando qual *pixel* seguir.

4 Materiais e Métodos

Neste capítulo são descritas as plataformas e os métodos utilizados para realizar os testes dos algoritmos de convolução e de detecção de bordas Canny, e das alterações realizadas na biblioteca ITK.

Para as implementações dos algoritmos de convolução foram realizados testes de desempenho, que comparam o desempenho das várias versões com relação ao algoritmo `naiveConvolve`, e testes de *profiler* para investigar como se comportou a memória *cache* e o número de instruções realizadas. Para o `OpCanny` foram realizados testes de desempenho, comparando a implementação `OpCanny` com relação ao `ItkCanny`, e de conformidade, verificando se o resultado do `OpCanny` é o mesmo do `ItkCanny`, tanto para a biblioteca otimizada, quanto para a biblioteca original.

4.1 Hardware e software

Os testes dos algoritmos foram realizados em três plataformas, conforme a Tabela 1.

Os dados das plataformas foram coletados da seguinte maneira:

- As informações da CPU foram obtidas com os comandos `$ cat /proc/cpuinfo` e `$ cpuid`;
- As informações sobre a memória RAM foram obtidas com o comando `$ sudo dmidecode --type 17`;
- GFLOPS de pico foram calculados multiplicando a frequência das CPUs pela quantidade

Tabela 1 Configurações dos ambientes de testes

Característica/Plataforma	Intel(R) Core(TM)2 Duo P7450	Intel(R) Core(TM) i7 975	AMD Opteron(tm) 6136
Apelido	**c2d**	**ci7**	**opteron**
Microarquitetura	Core 2	Nehalem	K10
Frequência CPU (Ghz)	2,13	3,33	2,4
Sockets	1	1	4
CPUs	1	1	4
Núcleos	2	8	32
Máx. FLOPS[a]/Ciclo	8	8	4
GFLOPS[b] por núcleo	17,04	26,64	9,6
GFLOPS totais	34,08	213,12	307,2
Tipo do barramento	FSB[c]	QuickPath Interconect (NUMA)	HyperTransport (NUMA)
Freq. Barramento (MHz)	266	3200	3200
Transferências por ciclo	4	2	2
Total GT[d]/s	1064	6400	6400
Bandwidth[e] CPU (GB/s)	8,512	51,2	51,2
Memória RAM (GB)	4	6	120
Freq. De memória (MHz)	800	1333	1333
Bandwidth/Canal (GB/s)	6,4	10,664	10,664
Número de canais	2	3	2
Bandwidth CPU (GB/s)	12,8	31,992	21,328
Cache L1	64KB 8-way	32KB 8-way	32KB 8-way
Cache L2	3MB 12-way	256KB 8-way	2MB 8-way
Cache L3	-	8MB 16-way	10MB
Linha de cache	64B	64B	64B
Compilador	gcc 4.6.2	gcc 4.6.1	gcc 4.6.1
S.O.	SUSE Linux 11.4 64bit	Debian 4.6.1-15 64bit	Debian 4.6.1-15 64bit

[a]Operações de ponto flutuante/s
[b]Milhões de FLOPS/s
[c]Front Side Bus - Tipo de barramento de transferência de memória
[d]Milhões de transferências de memória/s
[e]Largura de banda de memória

de operações ponto-flutuante por ciclo;

As arquiteturas dos processadores podem ser visualizados nos Diagramas 28, 29 e 30, criados pela ferramenta *likwid-topology*[1].

A medição do tempo foi realizada pela classe `StopWatch`, que encapsula as funcionalidades do relógio monotônico do sistema (Listagem 18), com resolução de milissegundos.

Listagem 18 Relógio utilizado para medir o tempo de execução

```
1 struct timespec ts;
2 clock_gettime(CLOCK_MONOTONIC, &ts);
3 n = (i64)ts.tv_sec * 1000000LL + (i64)ts.tv_nsec / 1000LL;
```

Todos os testes foram compilados com otimizações específicas para cada arquitetura e

[1]likwid-topology - ferramenta para consulta de topologias de nós de computadores

diagram 28 Diagrama físico da arquitetura **ci7**

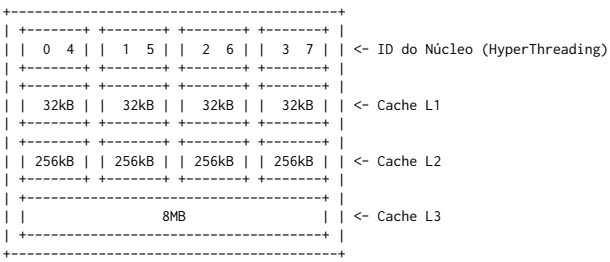
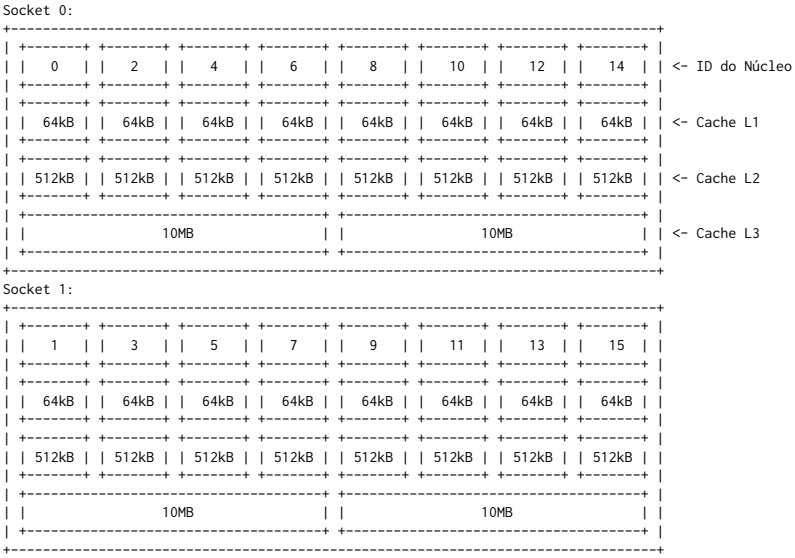


diagram 29 Diagrama físico da arquitetura **opteron**



com a vetorização automática (Seção 2.3.6) habilitada.

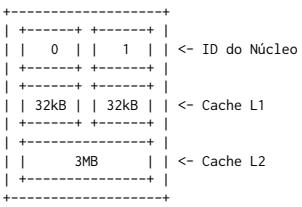
As *flags* de compilador e *linker* comuns para todas as plataformas são:

- **Flags de compilador**: `-O3 -fopenmp -msse -msse2 -msse3 -fabi-version=5 -¬ mfpmath=sse -march=native -mtune=native -ffast-math -ftree-vectorize`
- **Flags de linker**: `-lgomp -lboost_system -lboost_filesystem`

Para as plataformas **ci7** e **c2d** foram adicionadas as *flags* de compilador `-msse4 -¬ msse4.1 -msse4.2`.

Desta forma o desempenho das implementações foi avaliado com a melhor compilação possível, dentro das limitações da arquitetura e compilador. Além disso, todos os algoritmos fo-

diagram 30 Diagrama físico da arquitetura **c2d**



ram paralelizados com a utilização de OpenMP com um número fixo de *threads* por plataforma: **ci7** = 8 *threads*; **opteron** = 16 *threads*; e **c2d** = 2 *threads*.

4.2 Características específicas de cada arquitetura

Para uma análise mais ampla do desempenho das implementações, um dos requisitos de testes foi a possibilidade de execução em três plataformas distintas: **ci7**, **opteron**, **c2d**. Em todas as plataformas utilizou-se técnicas de programação paralela, tanto MIMD como SIMD, sendo que como técnica MIMD utilizou-se a biblioteca OpenMP para execução *multithread*. Como técnica SIMD, para as plataformas Intel© utilizou-se o conjunto de instruções SSE 4.1 e para a plataforma AMD o conjunto de instruções SSE 3.0.

Na primeira versão das implementações, tanto dos algoritmos de convolução como do `OpCanny`, utilizou-se intensivamente duas instruções SSE 4.1: `dpps` (produto-soma) e `blendps` (mistura). Como estas instruções não estão presentes no SSE 3.0, tiveram de ser emuladas para que fosse possível executar os algoritmos na plataforma **opteron**. Para tal criou-se as versões das funções intrínsecas `_mm_dp_ps` e `_mm_blend_ps` para SSE 3.0, com a desvantagem do aumento do número de instruções e da latência. Tais funções foram implementas refatorando-se o código da biblioteca Open Source SSE Plus[1], disponibilizada pela AMD. O código destas instruções pode ser visualizado no Apêndice C, Listagem 29.

Após testes verificou-se que o desempenho das funções emuladas estava longe do ideal, então todas as chamadas para estas funções foram substituídas. As funções `_mm_dp_ps` foram substituídas por funções específicas para cada parâmetro de máscara. Estas funções foram implementadas de forma otimizada, utilizando apenas instruções de `&` lógico, multiplicação (`_mm_mul_ps`) e soma horizontal (`_mm_hadd_ps`). Um exemplo de tal implementação pode ser observado na Listagem 19.

As chamadas para as funções `_mm_blend_ps`, onde possível, foram substituídas por funções `_mm_mov_ps`, vistas na Seção 3.1.5.

[1] http://sseplus.sourceforge.net/

Listagem 19 Exemplo função de produto-soma com máscara estática

```
1 extern __inline __m128 __attribute__((__gnu_inline__, __always_inline__, __artificial__))
2 _mm_dpf8_ps (__m128 a, __m128 b) {
3     static const __m128 omask = _mm_castsi128_ps(_mm_set_epi32(0xFFFFFFFF, 0x0, 0x0, 0x0));
4     a = _mm_mul_ps( a, b );
5     a = _mm_hadd_ps( a, a ); // Horizontally add the 4 values
6     a = _mm_hadd_ps( a, a ); // Horizontally add the 4 values
7     return _mm_and_ps( a, omask );// Clear output using low bits of the mask
8 }
```

4.3 Implementações dos algoritmos de convolução

Nesta seção são descritos os testes de desempenho e de *profiler* realizados para as implementações dos algoritmos de convolução.

4.3.1 Testes de desempenho

Foram realizados testes de desempenho para um conjunto de implementações de algoritmos de convolução (A), comparando o tempo de execução das versões `naiveConvolve` e `separableConvolve` (Seção 3.1.1 e Seção 3.1.6, respectivamente), com as outras implementações dos algoritmos de convolução.

A seguir mostra-se o conjunto dos algoritmos testados:

- `naiveConvolve` (Seção 3.1.1)
- `alignedConvolve` (Seção 3.1.2)
- `sseUnalignedConvolve` (Seção 3.1.4)
- `loopUnrollConvolve` (Seção 3.1.3)
- `prefetch128Convolve` (Seção 3.1.9)
- `sse3Convolve` (Seção 3.1.8)
- `sse5Convolve` (Seção 3.1.8)
- `sse7Convolve` (Seção 3.1.8)

- `sse9Convolve` (Seção 3.1.8)
- `sseReuse1Convolve` (Seção 3.1.5)
- `sseReuse4Convolve` (Seção 3.1.5)
- `separableConvolve` (Seção 3.1.6)
- `sseSConvolve` (Seção 3.1.7)
- `sse3SConvolve` (Seção 3.1.8)
- `sse5SConvolve` (Seção 3.1.8)
- `sse7SConvolve` (Seção 3.1.8)
- `sse9SConvolve` (Seção 3.1.8)

Foram realizados baterias de testes computando a média aritmética e o desvio padrão do tempo de execução de cada algoritmo utilizando um conjunto de imagens $I = \{I_1, ..., I_6\}$ (Tabela 2), e um conjunto de máscaras $K = \{3, 5, 7, 9, 11, 23, 33\}$, todas de tipo ponto flutuante de precisão simples de 32 bits. Os valores de I_n são aleatórios, variando entre zero e um, e gerados pela função `rand` da biblioteca `libc`. A máscaras têm valores calculados por funções gaussianas uni e bidimensionais.

Tabela 2 Resolução das imagens de testes dos algoritmos de convolução

Imagem	Resolução em pixels
I_1	321x481
I_2	642x962
I_3	1284x1924
I_4	2568x3848
I_5	5136x7696
I_6	10272x15392

Os testes foram realizados de acordo com o Algoritmo 11:

O tempo começa a ser medido imediatamente antes da chamada do método da implementação, e é finalizado imediatamente após o retorno da chamada, como visto no exemplo

Algoritmo 11: Algoritmo dos testes de convolução

```
1  para cada plataforma ∈ {ci7, opteron, c2d} faça
2      para cada A_i ∈ A faça
3          para cada I_n ∈ {I_1, ..., I_6} (Tabela 2) faça
4              para cada K_s ∈ {3, 5, 7, 9, 11, 23, 33} faça
5                  para iter = 1 até 100 faça
6                      limpar cache;
7                      iniciar contagem de tempo;
8                      chamar A_i(I_n, K_s);
9                      parar contagem de tempo;
10                 fim para
11                 calcular média aritmética e desvio padrão de A_i;
12             fim para cada
13         fim para cada
14     fim para cada
15 fim para cada
```

da Listagem 20. Apesar da média de 100 iterações ter sido escolhida de forma empírica, testes preliminares demonostraram que este número de iterações é suficiente para a validação estatística do desvio padrão.

Listagem 20 Método de medição de tempo de execução da implementação do algoritmo

```
1 m_StopWatch.StartNew();
2 implementacaoAlgoritmoTeste(img, kernel);
3 m_StopWatch.Stop();
```

Todos os testes utilizam os mesmos *buffers* de entrada e saída. A cada iteração a memória *cache* é completamente saturada por dados aleatórios, prevenindo assim a localidade temporal. O *buffer* de saída também é apagado a cada iteração, sendo preenchido por zeros.

Os testes foram executados com a carga do sistema o mais baixa possível para evitar possíveis preempções e falhas de *cache*. Na plataforma **c2d** a execução se deu com *Run Level 1*, sem habilitação de rede e multitarefas. Para as outras plataformas os testes foram executados com exclusividade de hardware, sendo que apenas o autor teve acesso ao hardware durante a execução dos testes.

4.3.2 Testes de *profiling*

Os testes de *profiling* (Seção 2.3.7) tem por objetivo inspecionar o número de instruções executadas, o número de acessos à memória, e o número `cache misses` dos algoritmos `naiveConvolve` e `sse3Convolve` utilizando a imagem I_4 e a máscara K_3 para todas as plataformas. O *profiling* foi realizado com auxílio do *profiler* Valgrind (Valgrind™ Developers, 2011a), utilizando a ferramenta `cachegrind` (Valgrind™ Developers, 2011b)

Também foi realizado um *profiling* de contagem de número de instruções SSE chamadas pelas implementações `sseUnalignedConvolve` e `sseReuse1Convolve`, com auxílio dos comandos `$ objdump`, `$ gdb` e de um *script* que pode ser encontrado em:

`http://web.inf.ufpr.br/vri/alumni/2012-TonyHild`

4.4 Implementação do algoritmo de detecção de bordas Canny

Para a implementação do algoritmo de detecção de bordas Canny foram realizados dois tipos de testes: de conformidade e de desempenho. Para ambos os testes os parâmetros do filtro foram definidos de forma empírica, sendo utilizados os mesmos valores para todas as bases de imagens testadas: $\sigma = 1,4$; $T_L = 4,0$; e, $T_H = 7,0$;

4.4.1 Bases de imagens

Para os testes foi utilizada a base de imagens *Berkeley Segmentation Dataset*[2]. Esta base contém 100 imagens *JPG* de 321x481 e 481x321 *pixels* que foram convertidas para *PNG* em escala de cinza de 8 bits. Ao resultado da conversão deu-se o nome de B_1.

As imagens das bases B_2 a B_6 (Tabela 3) foram criadas duplicando-se a resolução horizontal e vertical das imagens com a replicação da mesma imagem da base anterior. Por exemplo, para a criação de uma imagem da base B_2, replicou-se horizontalmente a mesma

[2]Base de dados livre para fins de pesquisa não comerciais e de educação (MARTIN et al., 2001). Disponível em: http://www.eecs.berkeley.edu/Research/Projects/CS/vision/bsds/

imagem da base B_1 e o resultado foi novamente replicado verticalmente. Com isto, a base B_2 contém imagens com resolução quatro vezes maior que a base B_1, e assim por diante.

Tabela 3 Configuração das bases de imagens utilizadas nos testes do algoritmo de detecção de bordas Canny

Base	Resolução das imagens em pixels
B_1	321x481 e 481x321
B_2	642x962 e 962x642
B_3	1284x1924 e 1924x1284
B_4	2568x3848 e 3848x2568
B_5	5136x7696 e 7696x5136
B_6	10272x15392 e 15392x10272

4.4.2 Testes de conformidade

Primeiramente o resultado do `ItkCanny` foi validado comparando o resultado de sua execução, quando compilado com a biblioteca alterada (Seção 3.2), com o resultado do mesmo algoritmo compilado com biblioteca ITK original. Isto foi necessário para avaliar se as alterações na biblioteca afetaram de alguma forma o resultado do algoritmo. Para tal foram calculados os *hashs* das imagens resultadas das execuções das duas bibliotecas utilizando comando `md5sum` (algoritmo MD5) e comparadas com o comando `diff`.

O segundo teste de conformidade valida o resultado obtido pelo algoritmo `OpCanny`, confrontando-o com o `ItkCanny`, que serve como resultado de referência. Neste caso não foi utilizada a mesma metodologia do primeiro teste de conformidade, pois os algoritmos têm implementações diferentes.

Este teste calcula a porcentagem média, por base, de pixels detectados corretamente (TP), a quantidade de pixels não detectados pelo algoritmo otimizado (FN), e a quantidade de *pixels* detectados erroneamente pelo algoritmo otimizado (FA), utilizando o método de (BOAVENTURA I.A.G. ; GONZAGA, 2009). Para tal são utilizadas as seguintes métricas:

Porcentagem de *pixels* de bordas corretamente detectados:

$$P_{co} = \frac{TP}{max(N_I, N_B)},$$

onde N_I é o número de bordas detectadas pelo algoritmo de referência e N_B é o número de bordas detectadas pelo algoritmo testado.

Porcentagem de *pixels* de bordas não detectados:

$$P_{nd} = \frac{FN}{max(N_I, N_B)}.$$

E porcentagem de *pixels* de bordas detectados erroneamente (falso alarme):

$$P_{fa} = \frac{FP}{max(N_I, N_B)}.$$

Se os valores de *pixels* detectados corretamente no teste de conformidade forem muito próximos a 100%, pode-se dizer que os algoritmos são equivalentes e intercambiáveis.

4.4.3 Testes de desempenho

Os testes de desempenho tem por objetivo comparar o tempo de execução do algoritmo `ItkCanny` e `OpCanny`.

São coletados dados totais e parciais da execução, sendo:

- Tempo total (T_B) de execução por base - Soma dos tempos de execução do método `Update` do filtro Canny;
- Tempo total de cada passo (T_P) do filtro - Soma dos tempos de execução dos passos `GaussianBlur`, `Compute2ndDerivative`, `Compute2ndDerivativePos`, `ZeroCrossing`, `Multiply`, `HysteresisThresholding`.

Após a coleta dos tempos totais é realizado cálculo da média aritmética e o desvio padrão por base, de acordo com o Algoritmo 12:

A coleta dos dados de todos os métodos sempre é iniciada imediatamente antes da chamada do método e terminada imediatamente após o retorno do mesmo. Para todos os testes foram usadas as mesmas configurações de números de *threads* OpenMP para a ITK.

Algoritmo 12: Algoritmo dos testes do detector de bordas Canny

```
1  para cada plataforma ∈ {ci7, opteron, c2d} faça
2      para cada B_i ∈ {B_1, ..., B_6} (Tabela 2) faça
3          para iter = 1 até 100 para B_1..B_4 ou iter = 1 até 10 para B_5..B_6 faça
4              para cada I_n ∈ B_i faça
5                  iniciar contagem de tempo;
6                  chamar método canny->Update();
7                  para cada passo ∈ P faça
8                      iniciar contagem de tempo;
9                      chamar método passo();
10                     parar contagem de tempo;
11                     acumular resultados parciais;
12                 fim para cada
13                 parar contagem de tempo;
14                 acumular resultados totais;
15             fim para cada
16             calcular média aritmética e desvio padrão dos tempos totais e parciais de B_i;
17         fim para
18     fim para cada
19 fim para cada
```

4.4.4 Testes de *profiling*

Os testes de *profiling* (Seção 2.3.7) tem por objetivo inspecionar o número de instruções executadas, o número de acessos à memória, e o número *cache misses* da execução do método `HysteresisThresholding` em um subconjunto de seis imagens da base B_4, dos algoritmos `ItkCanny` e `OpCanny`, em todas as plataformas. O subconjunto é formado pelas imagens que têm o nome de arquivo iniciados pelo número 8, e foi escolhido de forma empírica.

O *profiling* foi realizado com auxílio do *profiler* Valgrind (Valgrind™ Developers, 2011a), utilizando a ferramenta `callgrind` (Valgrind™ Developers, 2011b).

5 Resultados e Discussões

Neste capítulo são apresentados e discutidos os resultados dos testes das implementações dos algoritmos de convolução e da implementação algoritmo de detecção de bordas Canny.

5.1 Implementações dos algoritmos de convolução

Nesta seção são apresentados e discutidos os resultados dos testes dos algoritmos de convolução otimizados, realizados nas plataformas descritas anteriormente.

5.1.1 Testes de desempenho

Nas Figuras 31, 32, 33 pode-se visualizar os gráficos de comparação de desempenho (*speedup*) dos algoritmos de convolução 2D otimizados com relação ao algoritmo de referência simples (`naiveConvolve`) em todas as plataformas. Valores acima de $1,0$ indicam ganho de desempenho dos algoritmos otimizados. O eixo x representa o tamanho das máscaras agrupadas pelas imagens, o eixo y representa o ganho de desempenho, e o eixo z representa as implementações dos algoritmos.

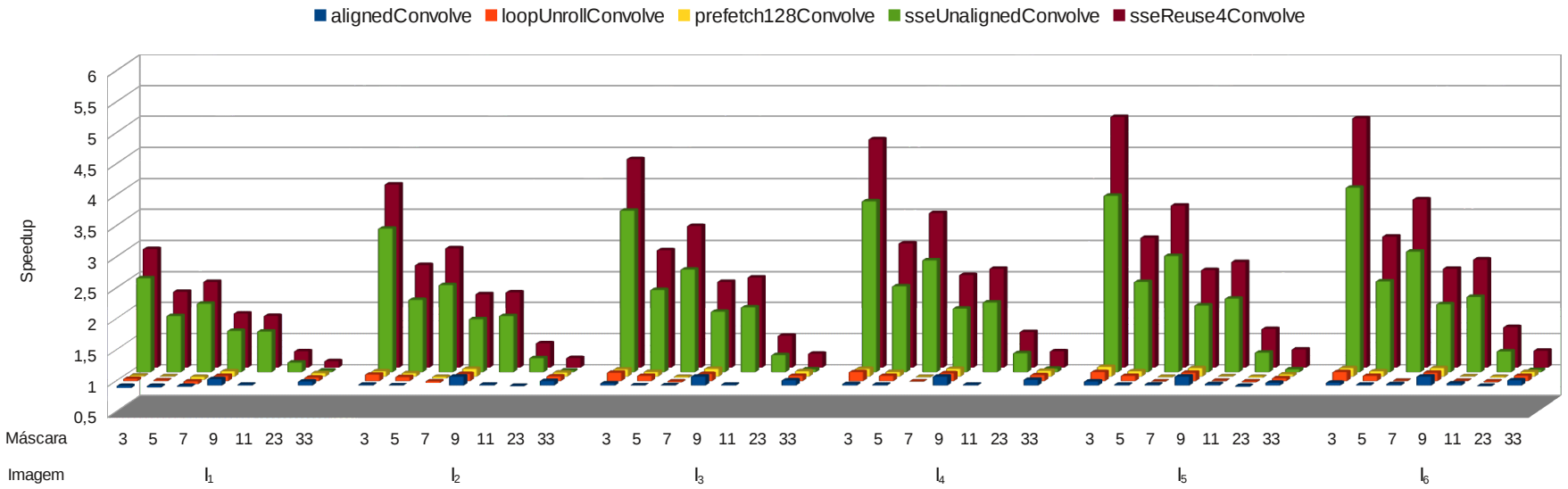


Figura 31 Desempenho `naiveConvolve` x algoritmos otimizados 2D - **ci7**

Fonte: O autor

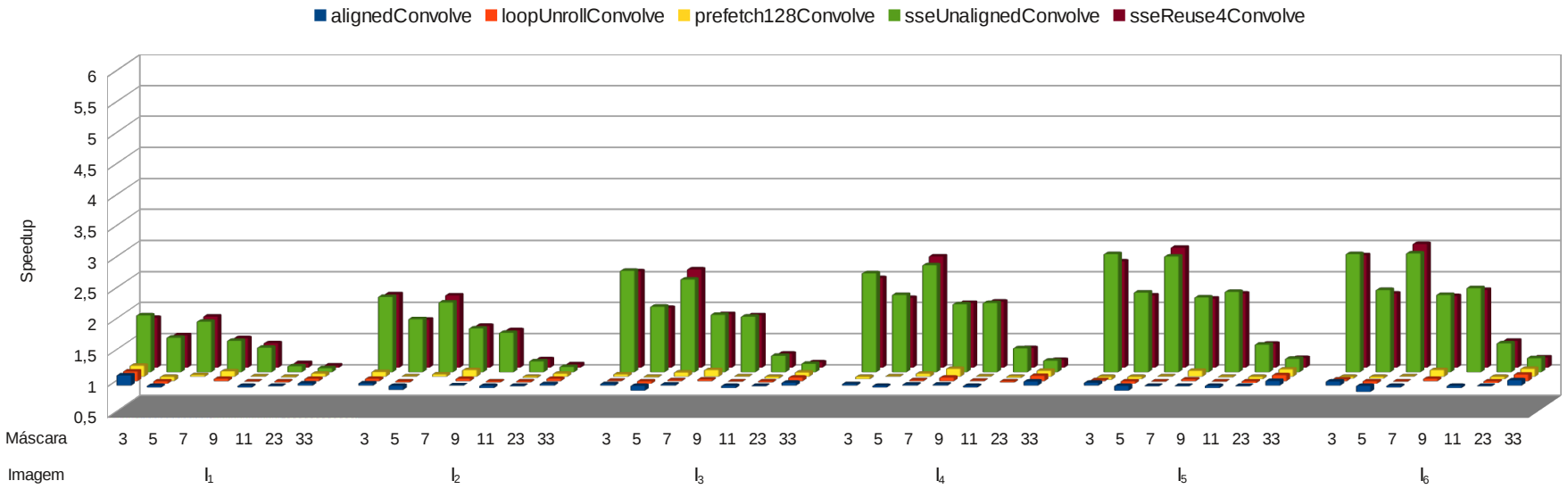


Figura 32 Desempenho `naiveConvolve` x algoritmos otimizados 2D - **opteron**

Fonte: O autor

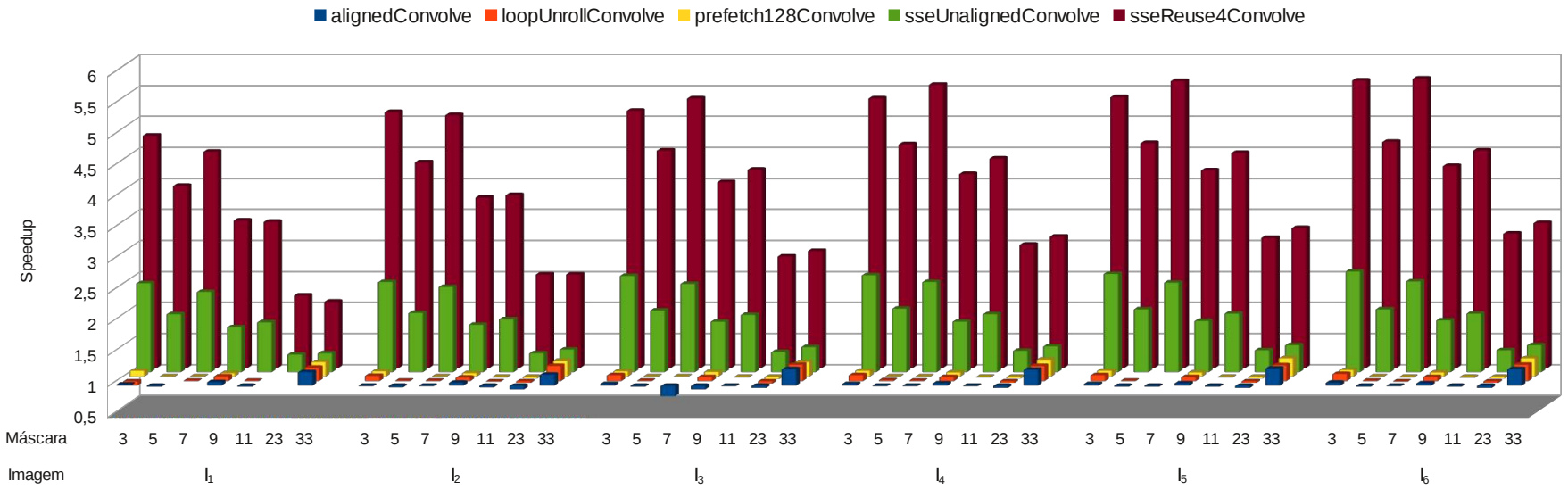


Figura 33 Desempenho `naiveConvolve` x algoritmos otimizados 2D - **c2d**

Fonte: O autor

Os gráficos das Figuras 31, 32, 33 revelam ganhos de desempenho de até 5,5 vezes com o algoritmo `sseReuse4Convolve` na plataforma **c2d**, 5 vezes na plataforma **ci7**, e de até 3 vezes na plataforma **opteron**.

A plataforma **opteron** obteve ganhos mais modestos por não possuir o conjunto de instruções SSE 4.1. O uso intensivo da função intrínseca `_mm_dp_ps` nas plataformas **ci7** e **c2d**, e a falta da mesma para a plataforma **opteron**, degradou o desempenho da última, pois a função necessitou ser substituída por funções compostas com maior latência.

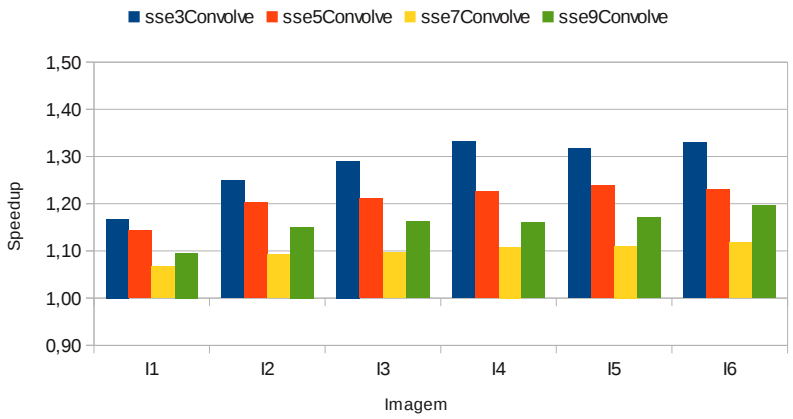
Apesar dos resultados com a otimização SIMD terem superado as expectativas, ainda é possível melhorar a eficiência do algoritmo aumentando a ocupação dos vetores. Em todas as plataformas fica clara a perda de eficiência ao se utilizar registradores SSE de forma parcial. A utilização dos vetores para máscaras com tamanho cinco e nove é de 60% e 77%, respectivamente, enquanto que para máscaras de tamanho sete e onze a utilização dos vetores chega à 86% e 90%, respectivamente. Uma das maneiras de melhorar a ocupação dos vetores é otimizando as implementações para máscaras com tamanho fixo. Apesar do aumento do tamanho da máscara diminuir a razão de perda de eficiência, máscaras com tamanho entre três e nove são mais utilizadas para processamento de imagens, fazendo valer a pena a especialização.

Em todas as plataformas houve diminuição brusca do ganho de desempenho nas máscaras de tamanho 23 e 30, evidenciado eficiência da autovetorização do compilador.

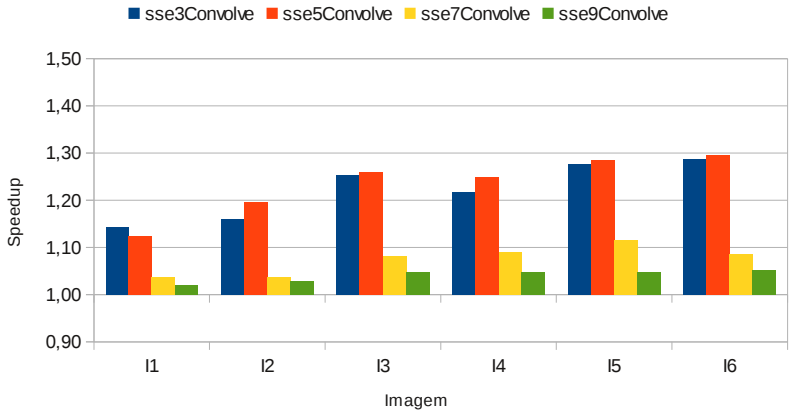
As otimizações de desdobramento de laços, sem utilização explícita de SSE, permitiram ganho de desempenho de até 15% nas plataformas **ci7** e **c2d** para as máscaras de tamanho três. Para máscaras de maior tamanho, o aumento no número de instruções aumentou e eficiência do *pipeline*, diminuindo a eficiência relativa do desdobramento de laços. A implementação que faz uso de *prefech* explícito (`prefechConvolve`) não obteve ganhos conclusivos.

Na Figura 34 pode-se visualizar o gráfico de comparação de desempenho da implementação `sseConvolve` em relação à implementação `sseNConvolve`, em cada plataforma. Valores acima de $1,0$ indicam ganho de desempenho dos algoritmos `sseNConvolve`. O eixo x representa as imagens, o eixo y representa o ganho de desempenho, e as colunas representam os

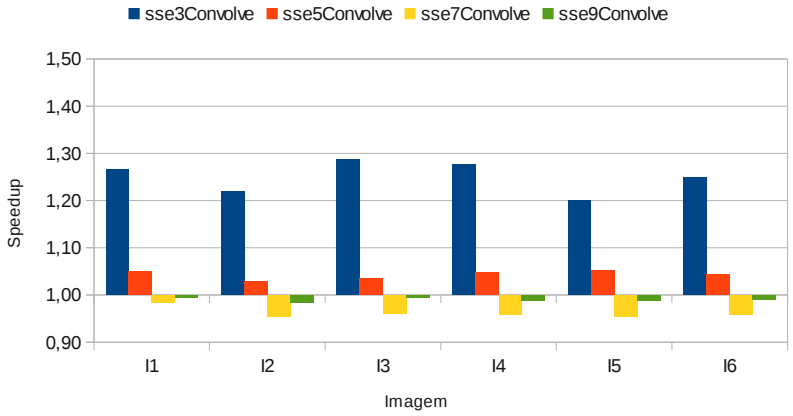
algoritmos.



(a) ci7



(b) opteron



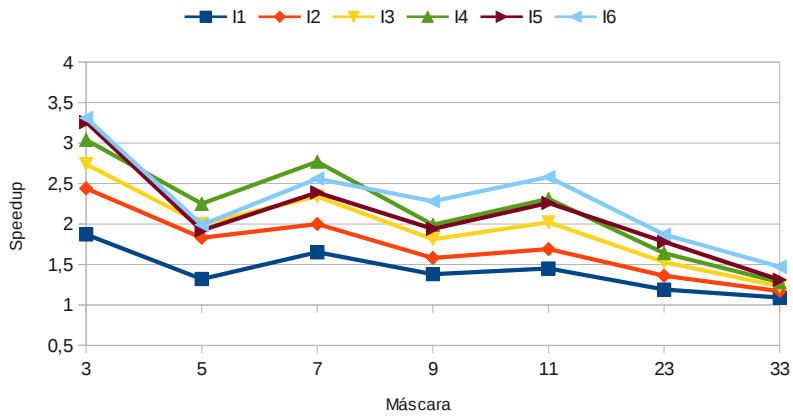
(c) c2d

Figura 34 Desempenho `sseConvolve` x `sseNConvolve` em todas as plataformas

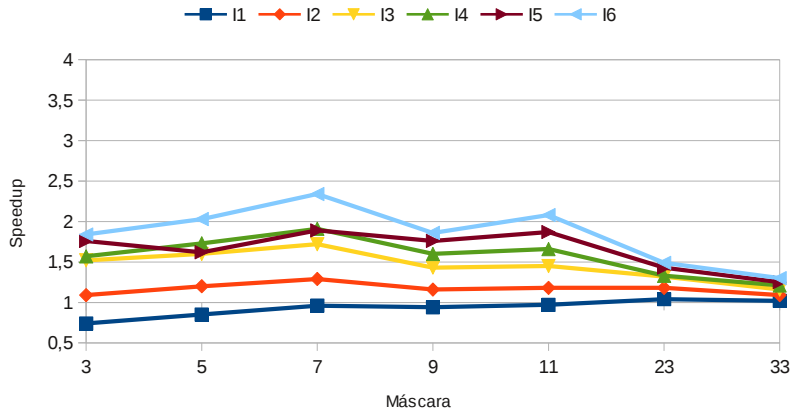
Fonte: O autor

As implementações `sseNConvolve` (Figura 34) demonstraram ganho de desempenho em todos os tamanhos de máscaras nas plataformas **ci7** e **opteron**. Na plataforma **c2d**, os ganhos se restringiram às implementações para máscaras de tamanho três e cinco. O maior ganho para máscaras de tamanho três e cinco, chegando a 30%, se deve ao fato da diminuição de carregamentos das mesmas. A máscara de tamanho três é inteiramente carregada ao início de cada iteração, sendo desdobrados 16 laços de x (colunas da imagem) para cada carregamento. Para a máscara de tamanho cinco, 75% da máscara é carregada ao início de cada iteração, e da mesma forma da implementação para máscaras de tamanho três, são desdobrados 16 laços de x, para cada carregamento.

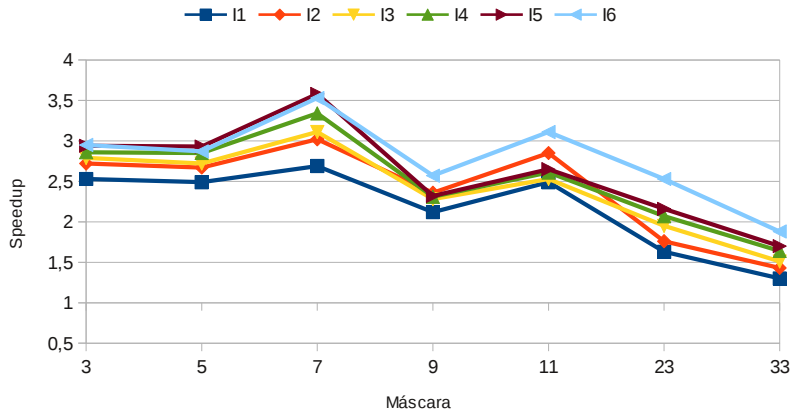
Na Figura 35 encontra-se o gráfico de comparação de desempenho da implementação `separableConvolve` em relação à implementação `sseSConvolve`, em cada plataforma. Valores acima de $1,0$ indicam ganho de desempenho da implementação `sseSConvolve`. O eixo x representa o tamanho das máscaras, o eixo y representa o ganho de desempenho, e as linhas representam as imagens.



(a) ci7



(b) opteron



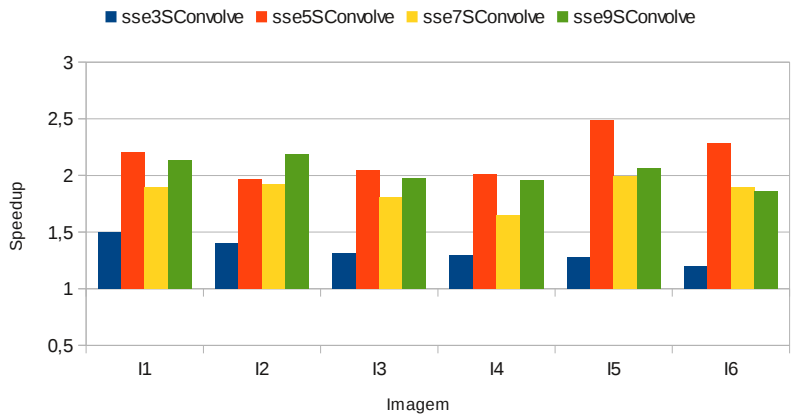
(c) c2d

Figura 35 Desempenho `separableConvolve` x `sseSConvolve` em todas as plataformas

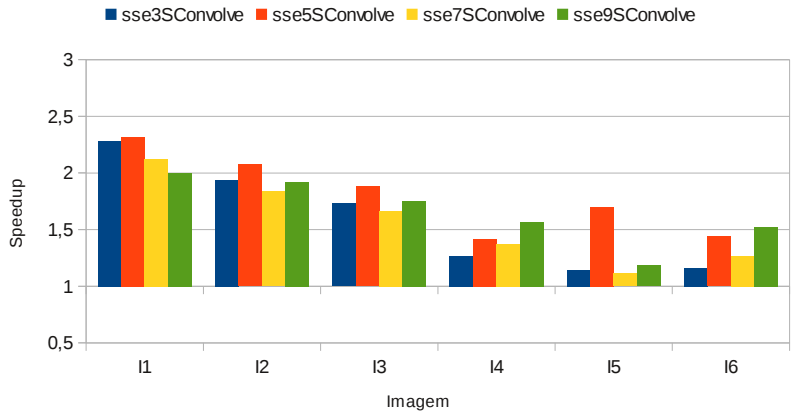
Fonte: O autor

Analisando o gráfico da Figura 35, percebe-se que o comportamento do desempenho dos algoritmos de convolução 2D separáveis é semelhante ao comportamento dos algoritmos convolução 2D. No geral, há ganho de desempenho maior para as máscaras três, sete e onze, e o desempenho cai de forma abrupta para as máscaras maiores que onze.

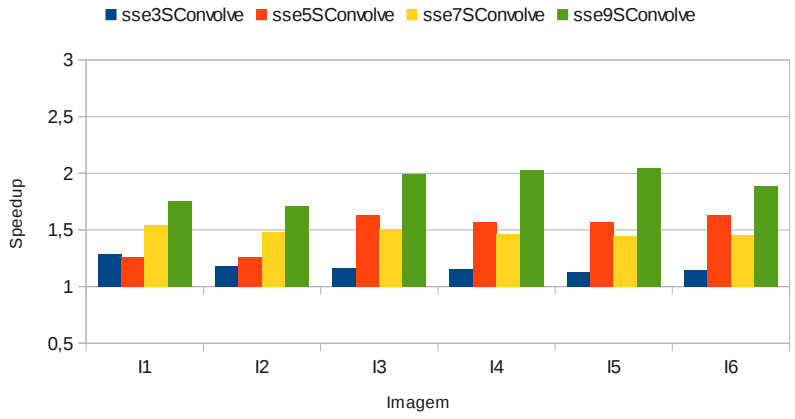
Na Figura 36 encontra-se o gráfico de comparação de desempenho da implementação `sseSConvolve` em relação à implementação `sseNSConvolve`, em cada plataforma. Valores acima de $1,0$ indicam ganho de desempenho da implementação `sseNSConvolve`. O eixo x representa o tamanho das máscaras, o eixo y representa o ganho de desempenho, e as colunas representam as imagens.



(a) ci7



(b) opteron



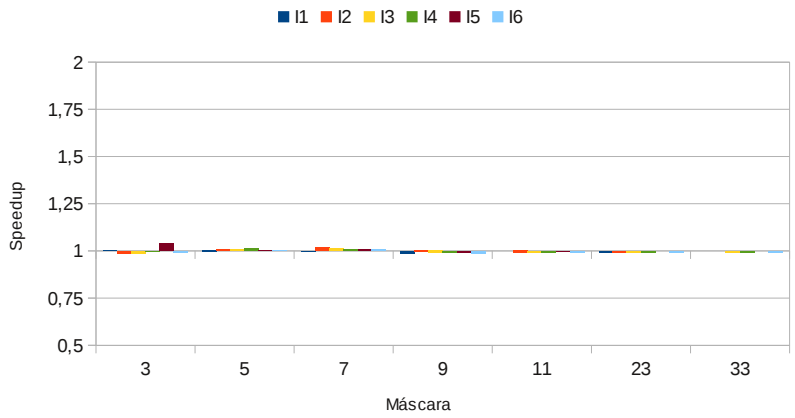
(c) c2d

Figura 36 Desempenho `sseSConvolve` x `sseNSConvolve` em todas as plataformas

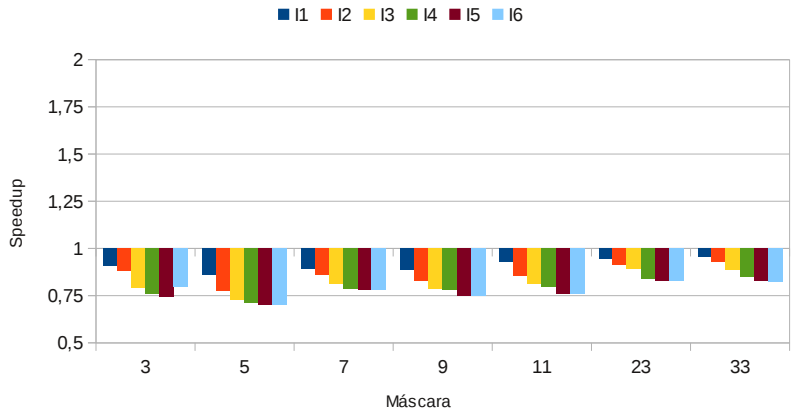
Fonte: O autor

Com a análise do gráfico da Figura 36 percebe-se que houve ganho de desempenho de até duas vezes dos algoritmos `sseNSConvolve` com relação ao algoritmo `sseSConvolve`, devido a utilização otimizada dos registradores SSE e o menor número de desvios condicionais na implementação.

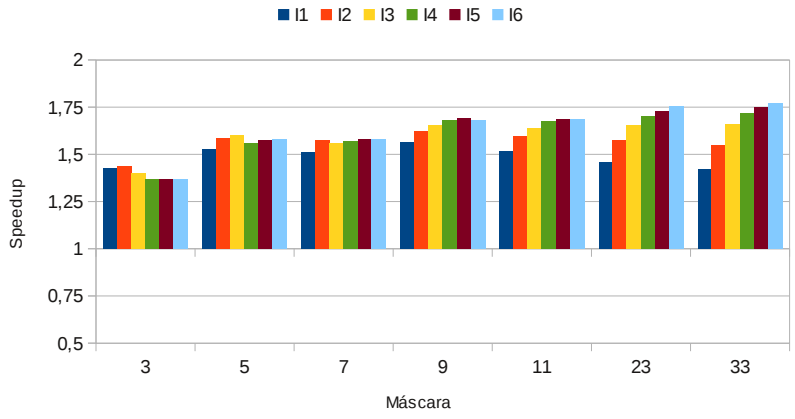
Finalmente na Figura 37 pode-se visualizar o gráfico de comparação de desempenho da implementação com carregamentos e armazenamentos dasalinhados (`sseUnalignedConvolve`) em relação à implementação com carregamentos alinhados e armazenamentos desalinhados (`sseReuse1Convolve`), em cada plataforma. Valores acima de 1,0 indicam ganho de desempenho do algoritmo alinhado. O eixo *x* representa o tamanho das máscaras, o eixo *y* representa o ganho de desempenho, e as colunas representam as imagens.



(a) ci7



(b) opteron



(c) c2d

Figura 37 Desempenho `sseUnalignedConvolve` x `sseReuse1Convolve` em todas as plataformas

Fonte: O autor

Com relação ao alinhamento, as plataformas **ci7** e **opteron**, assim como atestam os fabricantes (Intel, 2011b; Advanced Micro Devices, 2011), foram otimizadas para diminuir a latência de carregamentos desalinhados. Isto fica evidenciado analisando a Figura 37. Na plataforma **ci7**, o desempenho das implementações `sseUnalignedConvolve` e `sseReuse1¬ Convolve` (equivalentes em número de desdobramento de laços e armazenamentos por laço) foram praticamente idênticos. Na plataforma **opteron**, a implementação `sseReuse1Convolve` teve um desempenho médio 20% inferior à implementação `sseUnalignedConvolve`, mas neste caso, devido novamente à emulação da instrução `_mm_dp_ps` não se pode medir com precisão se a eficiência do código desalinhado está na otimização da arquitetura ou na ineficiência do código alinhado.

5.1.2 Testes de *profiling*

Na Tabela 4 encontram-se os dados de `profiling` da execução dos algoritmo `naiveConvolve` e `sse3Convolve`.

Tabela 4 Instruções `naiveConvolve` x `sse3Convolve`

ci7			
	`naiveConvolve`	`sse3Convolve`	**Razão**[a]
Instruções[b]	1.985.789.307	154.998.314	12,8
Ref. Dados[c]	484.126.912	16.568.648	29,2
Ref. *Cache* LL[d]	2.493.670	2.509.060	1,0
LL *misses*[e]	1.242.932	1.253.707	1,0
opteron			
	`naiveConvolve`	`sse3Convolve`	**Razão**
Instruções	2.015.462.774	244.250.112	8,3
Ref. Dados	464.394.699	24.027.630	19,3
Ref. *Cache* LL	1.252.763	1.271.357	1,0
LL *misses*	1.246.534	1.255.582	1,0
c2d			
	`naiveConvolve`	`sse3Convolve`	**Razão**
Instruções	1.985.791.751	155.000.757	12,8
Ref. Dados	484.126.843	16.568.579	29,2
Ref. *Cache* LL	2.485.973	2.509.062	1,0
LL *misses*	1.243.429	1.254.753	1,0

[a] `naiveConvolve` / `sse3Convolve`
[b] Número de instruções executadas
[c] Número de referências à dados em memória
[d] Número de referências à memória *cache* de mais alto nível
[e] Número de *cache misses* na memória *cache* de mais alto nível

Na Tabela 4 pode-se reparar diminuição de número de instruções em todas as plataformas, além da diminuição do número de acesso à dados e de *cache misses*. A diminuição do número de instruções se deu pela melhor vetorização do código (lembrando que a versão `naiveConvolve` é vetorizada pelo compilador), e pela otimização de desdobramento de laços, que permitiu que o compilador pudesse otimizar o *pipeline* de execução, escondendo a latência. A diminuição das referências a dados foi possível graças à técnica de *rotação de vetores*, que utiliza somente referências a registradores, e que é uma operação de baixa latência.

Na Tabela 5 pode-se observar a contagem de instruções SSE dos métodos `sseReuse`¬

1Convolve e sseUnalignedConvolve.

Tabela 5 Instruções SSE - sseReuse1Convolve e sseUnalignedConvolve

Plataforma	ci7		opteron		c2d	
Instrução/Algoritmo	**a**[a]	**b**[b]	**a**	**b**	**a**	**b**
addps	4	4	20	4	4	4
andps	-	-	20	4	-	-
dpps	4	4	-	-	4	4
haddps	-	-	40	8	-	-
movaps	8	1	36	4	8	1
movss	4	-	20	1	4	-
movups	1	5	1	5	1	5
mulps	-	-	20	4	-	-
prefetcht0	-	-	2	-	-	-
shufps	7	-	35	-	7	-
xorps	2	2	2	2	2	2
Total	**30**	**16**	**196**	**32**	**30**	**16**

[a] sseReuse1Convolve
[b] sseUnalignedConvolve

A Tabela 5 demonstra que a implementação alinhada para a plataforma **opteron** executa seis vezes mais instruções SSE do que a versão desalinhada, mesmo assim é apenas 25% mais lenta, em média. Ainda analisando a Tabela 5, percebe-se que as implementações das plataformas **ci7** e **c2d** executam exatamente a mesma quantidade de instruções SSE para ambos algoritmos. A plataforma **c2d** foi a que obteve o maior ganho de desempenho, chegando a 75%, justamente por não ter acessos desalinhados otimizados.

5.2 Implementação do algoritmo de detecção de bordas Canny

Nesta seção são apresentados os resultados dos testes dos algoritmos ItkCanny e OpCanny, realizados nas plataformas descritas anteriormente.

5.2.1 Testes de conformidade

O resultado do cálculo dos *hashs* das imagens geradas pelo algoritmo ItkCanny, tanto na biblioteca alterada quanto na biblioteca original são idênticos.

O resultado dos testes de conformidade foi de 99,999% de coincidência de *pixels* detectados corretamente em todas as bases e plataformas.

Os testes de conformidade demonstraram que a diferença entre o `OpCanny` e `ItkCanny` é desprezível, portanto pode-se considerar as implementações equivalentes.

5.2.2 Testes de desempenho

Na Tabela 6 encontram-se os valores do tempo de execução médio por base e desvio padrão dos algoritmo `OpCanny` e `ItkCanny`.

Tabela 6 Tempo de execução médio e desvio padrão dos algoritmo `OpCanny` e `ItkCanny`

		OpCanny		ItkCanny	
		Tempo (s)	DP[a]	Tempo (s)	DP
ci7	B_1	0,149	3,0	2,007	1,4
	B_2	0,544	0,5	7,671	1,5
	B_3	2,140	0,1	29,439	0,8
	B_4	8,001	0,1	117,738	1,0
	B_5	31,166	0,0	471,604	1,5
	B_6	123,947	0,6	1917,983	2,3
opteron	B_1	0,810	20,3	5,696	1,9
	B_2	2,174	3.8	14,796	2,2
	B_3	4,076	0,8	46,655	4,9
	B_4	12,246	1,5	169,108	4,4
	B_5	43,248	1,9	710,801	4,1
	B_6	164,339	1,9	2697,714	0,4
c2d	B_1	0,534	1,1	5,328	0,3
	B_2	1,928	2,5	20,865	0,2
	B_3	7,501	0,0	82,447	0,1
	B_4	32,784	0,1	328,103	0,1
	B_5	130,417	0,1	1309,151	0,2
	B_6	520,966	0,1	5235,557	0,1

[a] Desvio padrão em %

Como demonstrado na Tabela 6, desprezando as bases B_1 e B_2 pelo seu tempo de execução ser muito baixo e pela sobrecarga de criação de *threads*, o desvio padrão das médias indica coerência nos testes. Não foram evidenciadas grandes discrepâncias entre o algoritmo nativo e otimizado.

No gráfico da Figura 38 pode-se visualizar a comparação de desempenho do algoritmo `OpCanny` em relação ao `ItkCanny`, medido no método `Update`. O eixo *x* reprenta as bases, e o eixo *y* representa o ganho de desempenho, e as colunas representam as plataformas. Valores acima de $1,0$ indicam ganho de desempenho.

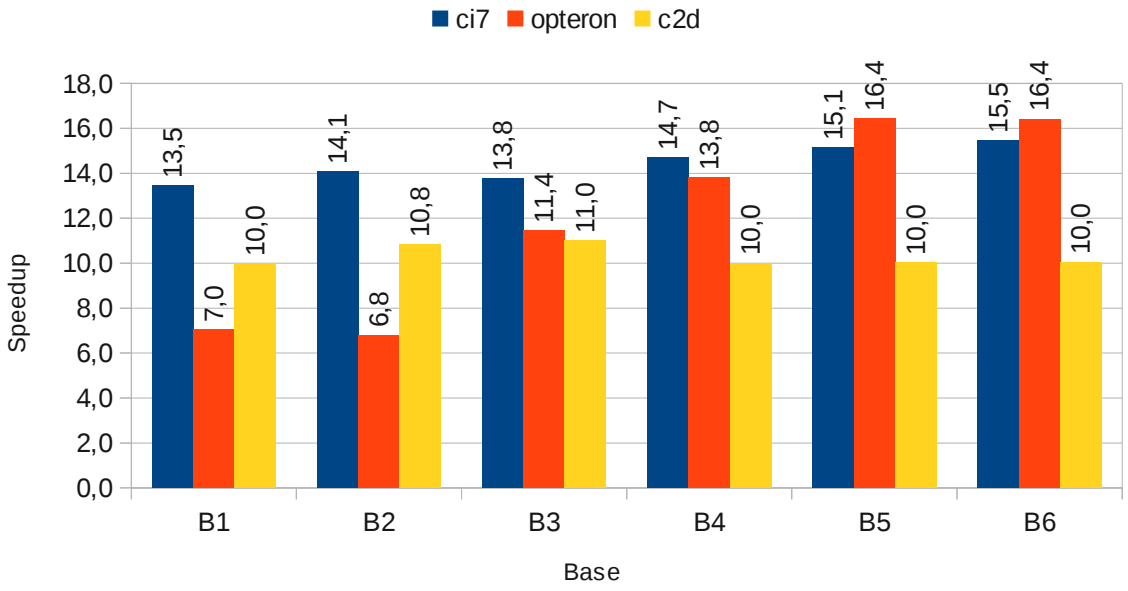


Figura 38 *Speedup* da execução do filtro Canny (método `Update`)

Fonte: O autor

De maneira geral, os ganhos de desempenho variaram de 600% a 1500%, com uma evidente vantagem para a plataforma **ci7**, como pode ser visto na Figura 38. As plataformas **ci7** e **opteron** escalaram adequadamente com o aumento da base, mas devido à maior largura de banda e número de *threads*, a plataforma **opteron** foi a que obteve o melhor desempenho nas bases B_5 e B_6. A plataforma **c2d** manteve-se com um ganho de desempenho constante ao redor de 10 vezes em relação ao `ItkCanny`. Isto se deve ao fato desta plataforma possuir apenas dois núcleos.

No gráfico da Figura 39 pode-se visualizar a comparação de desempenho (*speedup*) da execução de todas as etapas do filtro em separado. Os valores indicam quantas vezes cada etapa do algoritmo `OpCanny` é mais rápida que a etapa correspondente do algoritmo `ItkCanny`. O eixo *x* reprenta as bases agrupadas por plataforma, o eixo *y* representa o ganho de desempenho, e as colunas representam as etapas. Valores acima de $1,0$ indicam ganho de desempenho.

O gráfico da Figura 40 representa a porcentagem de tempo que cada etapa do filtro utiliza em relação ao tempo total. O eixo *x* reprenta as bases agrupadas por algoritmo e por plataforma, o eixo *y* representa a porcentagem de tempo, e as colunas representam as etapas.

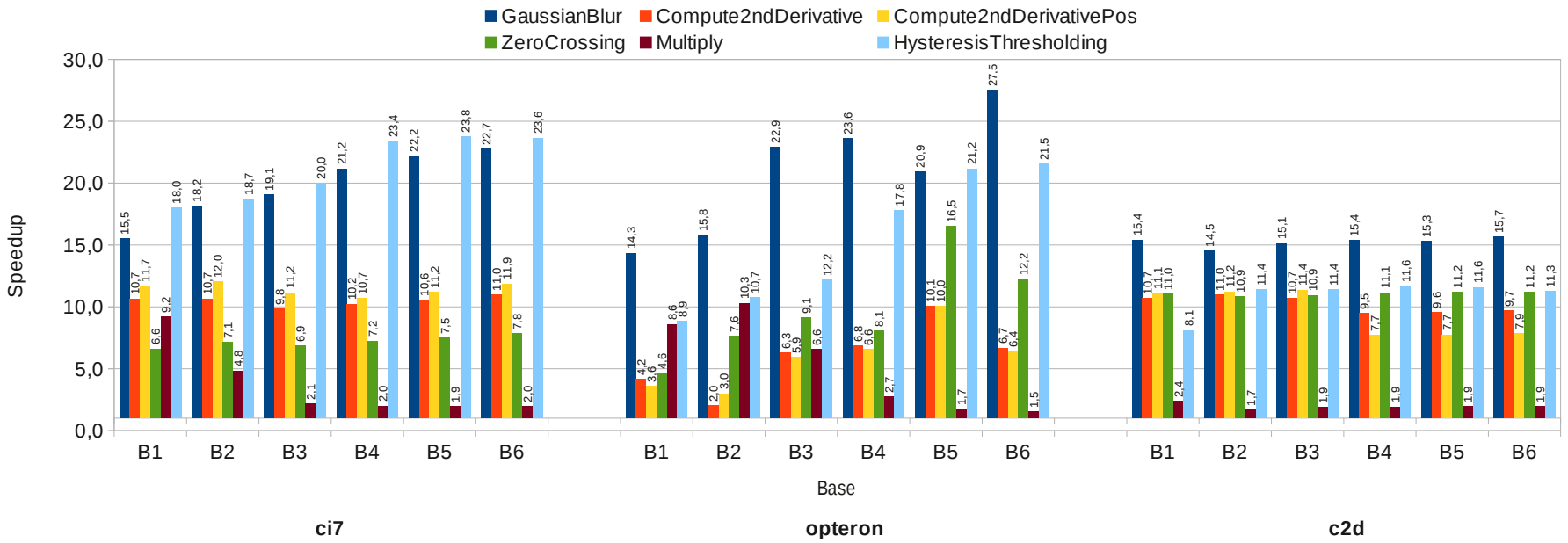


Figura 39 *Speedup* por etapa do filtro Canny

Fonte: O autor

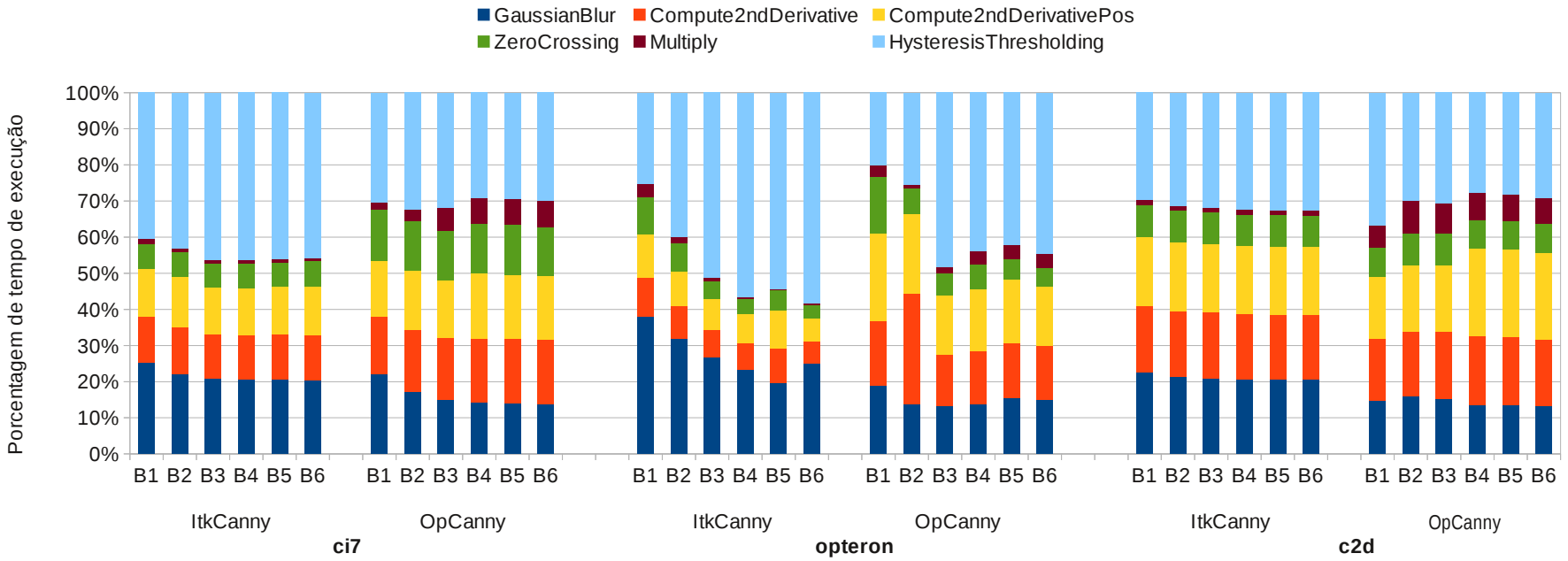


Figura 40 Tempo acumulado de execução das etapas do filtro Canny

Fonte: O autor

O gráfico da Figura 39 evidencia claramente duas coisas: a implementação tanto da suavização gaussiana quanto da histerese do `ItkCanny` é bastante ineficiente. Pode-se chegar a esta conclusão confrontando o desempenho do ganho de desempenho da suavização gaussiana no `OpCanny` contra o ganho de desempenho do algoritmo `sse7SConvolve`. Levando-se em conta que o ganho de desempenho do algoritmo `sse7SConvolve` variou de 2,5 a 5 vezes, a implementação da suavização gaussiana `ItkCanny` é no mínimo 3 vezes mais lenta que o algoritmo `separableConvolve` na plataforma **c2d**, e 4 vezes mais lenta na plataforma **ci7**.

Na etapa da histerese também houve um expressivo ganho de desempenho devido à paralelização e implementação mais eficiente das filas circulares e da verificação de *pixels* não-borda. As filas circulares foram implementadas reutilizando um *buffer* pré alocado de forma sequencial, já a implementação do algoritmo nativo utiliza uma implementação encadeada, e é penalizado pelo acesso aleatório. Não obstante estas otimizações, a paralelização também teve grande peso no ganho de desempenho, já que o código nativo não é paralelizado.

O cálculo das derivadas é realizado com a mesma técnica de *rotação de vetores* utilizado no algoritmo `sse3Convolve`. A plataforma **c2d** teve um maior ganho de desempenho relativo por não ter otimizações de acesso a dados desalinhados, lembrando que nesta plataforma os testes foram realizados com apenas duas *threads*. A plataforma **ci7** teve ganho de desempenho constante e a plataforma **opteron** sofreu mais uma vez com as instruções emuladas de maior latência, e teve um ganho de desempenho não linear devido ao maior número de *threads*.

A etapa de *zero crossing*, que é uma boa candidata à autovetorização, teve melhoria de performance equivalente nas plataformas **ci7** e **opteron**, e ganho adicional na plataforma **c2d**, pelos mesmos motivos da etapa do cálculo das derivadas. Um ganho adicional foi possível pela remoção das estruturas condicionais, já que todas as operações são realizadas utilizando operações lógicas (Figura 26), e também pela comparação de quatro *pixels* vizinhos de uma só vez.

No gráfico de tempo acumulado (Figura 40) percebe-se que em todas as plataformas

o foco foi otimizar justamente as duas etapas que consumiam mais tempo: `GaussianBlur` e `HysteresisThresholding`. A suavização gaussiana teve o desempenho melhorado a ponto de rivalizar com cálculo de *zero crossings*, que é uma operação mais simples.

O gráfico da Figura 41 representa o *throughput* em *pixels*/s (eixo *y*). As linhas representam os algoritmos/plataformas, o eixo *x* representa o número de pixels da imagem, e os pontos das linhas representam o tamanho da imagem (em *pixels*) da base.

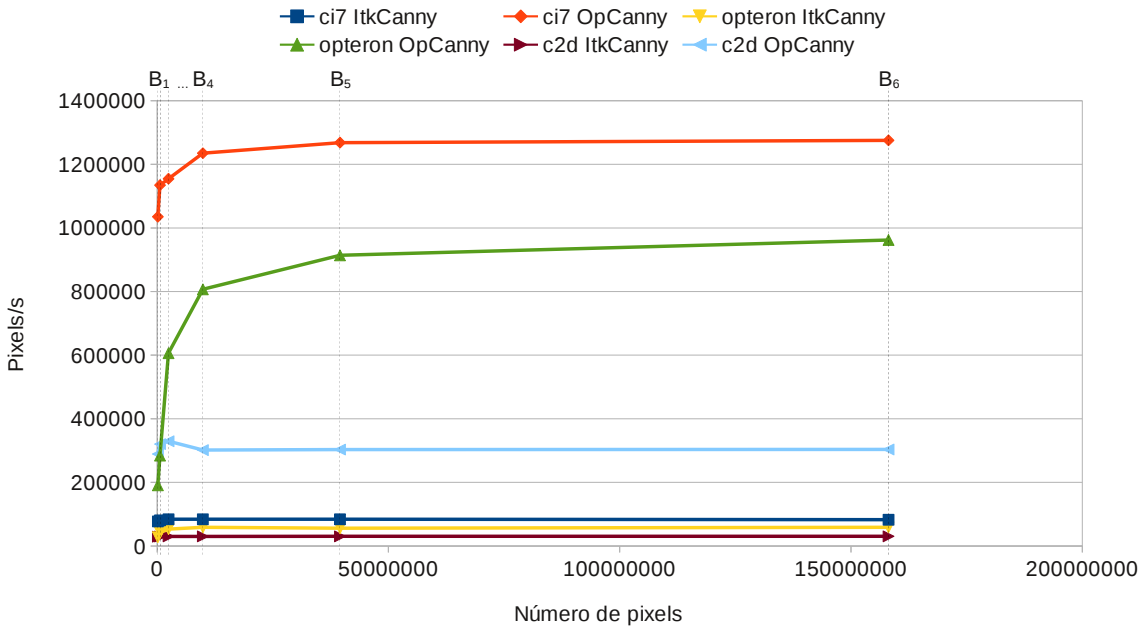


Figura 41 *Throughput* em *pixels*/s do filtro Canny

Fonte: O autor

Analisando o gráfico Figura 41 percebe-se que apesar do *throughput* final da plataforma **ci7** ter sido melhor, a plataforma **opteron** escalou com uma tendência mais definida devido ao seu maior *throughput* teórico e número de núcleos.

Finalmente, no gráfico da Figura 42 pode-se visualizar a comparação de desempenho das outras plataformas em relação à plataforma **ci7**.

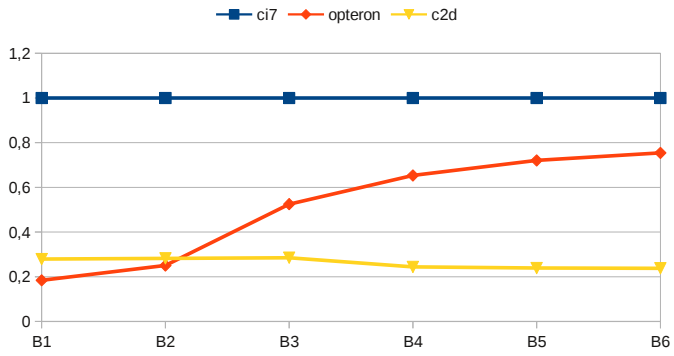


Figura 42 Ganho de desempenho por plataforma do filtro Canny

Fonte: O autor

Apesar da plataforma na plataforma **opteron** o `OpCanny` ter conseguido o maior ganho de desempenho em relação ao `ItkCanny`, seu tempo absoluto de execução foi em média 25% inferior à plataforma **ci7**, como evidenciado pelo gráfico da Figura 42.

No geral pode-se concluir que as otimizações aplicadas à implementação `OpCanny` foram bem sucedidas, e que podem ser aplicadas em outros algoritmos com a desvantagem de serem complexas e pouco portáveis. Apesar disto, o trabalho demonstrou testes em três plataformas amplamente utilizadas, o que pode valer o esforço de otimizações especificas por plataforma.

5.2.3 Testes de *profiling*

Na Tabela 7 encontram-se os dados de *profiling* da execução do método `HysteresisThresholding`.

A Tabela 7 demonstra o número de instruções e *caches misses* do `ItkCanny` comparada ao `OpCanny`. A execução do `OpCanny` com somente uma *thread*, executou 10 vezes menos instruções e acessou 14 vezes menos memória que o `ItkCanny`. Uma das razões da diminuição no número de instruções se deu na comparação com o limiar superior. Como o filtro detecta bordas finas, com apenas um *pixel* de largura, grande parte da imagem contém apenas *pixels* com valor 0, sendo assim, a comparação com quatro *pixels* de uma só vez aumenta muito a

Tabela 7 Número de instruções do método `HysteresisThresholding`

ci7					
	# Threads	**Instruções**[a]	**Ref. Dados**[b]	**Ref. Cache LL**[c]	**LL misses**[d]
`ItkCanny`	1	250.279	138.826	223	118
`OpCanny`	1	22.244	9.412	263	163
Razão[e]		11,3	14,8	0,8	0,7
`ItkCanny`	8	249.676	139.156	224	118
`OpCanny`	8	3.174	1.452	108	97
Razão		78,7	95,9	2,1	1,2
opteron					
	# Threads	**Instruções**	**Ref. Dados**	**Ref. Cache LL**	**LL misses**
`ItkCanny`	1	248.166	138.826	255	160
`OpCanny`	1	22.180	9.413	253	202
Razão		11,2	14,7	1,0	0,8
`ItkCanny`	2	243.964	137.907	211	160
`OpCanny`	2	1.608	816	96	93
Razão		151,7	169,1	2,2	1,7
c2d					
	# Threads	**Instruções**	**Ref. Dados**	**Ref. Cache LL**	**LL misses**
`ItkCanny`	1	248.058	138.818	216	126
`OpCanny`	1	22.251	9.415	263	170
Razão		11,1	14,7	0,8	0,7
`ItkCanny`	16	250.770	139.811	216	126
`OpCanny`	16	11.074	4.743	172	131
Razão		22,6	29,5	1,3	1,0

[a] Número de instruções executadas / 100000
[b] Número de referências à dados em memória / 100000
[c] Número de referências à memória *cache* de mais alto nível / 100000
[d] Número de *cache misses* na memória *cache* de mais alto nível / 100000
[e] `ItkCanny` / `OpCanny`

eficiência da etapa. Nos gráficos da Figura 39 percebe-se claramente que, a partir da base B_4, a sobrecarga relativa de criação das *threads* é desprezível, e o algoritmo alcança sua eficiência máxima com um ganho de desempenho de aproximadamente 11 vezes na plataforma **c2d**, 20 vezes na plataforma **opteron** e de 23 vezes na plataforma **ci7**.

6 Conclusão

Este trabalho apresentou técnicas de otimização e paralelização SIMD e MIMD testadas em vários algoritmos, principalmente no algoritmo de detecção de bordas Canny para ITK, cumprindo todos os objetivos propostos.

Com o estudo percebeu-se que é possível otimizar a biblioteca ITK melhorando substancialmente seu desempenho. A utilização de técnicas de paralelização e instruções de baixo nível torna o desenvolvimento árduo, mas é ideal para a otimização de bibliotecas como a ITK, permitindo-a tirar total proveito do potencial de paralelismo dos processadores modernos. Os compiladores modernos são sofisticados o suficiente para otimizar automaticamente boa parte do código, mas ainda podem ser superados pela otimização manual.

O uso de OpenMP para a otimização MIMD se mostrou bastante eficaz, tanto na codificação quanto no desempenho. A diminuição do tempo de codificação de código paralelo com a utilização da biblioteca permite utilizar este tempo para otimizações SIMD em nível de instrução.

De forma geral, pode-se afirmar que diminuir o número de acessos à memória traz grandes vantagens com relação ao desempenho, pois as memórias atuais ainda são o grande gargalo dos sistemas computadorizados.

Espera-se que refatorando e evoluindo as implementações presentes neste trabalho consiga-se um ganho ainda maior de desempenho, trazendo vantagens para todos os usuários do ITK, bem como de outras bibliotecas de processamento de imagens que façam uso do algoritmo de convolução. Espera-se também que as implementações aqui apresentadas sirvam como base

para trabalhos futuros, pois as mesmas estão disponibilizadas como software livre.

Bibliografia

Advanced Micro Devices, A. Software optimization guide for AMD family 10h and 12h processors. *AMD Publication*, v. 40546, 2011.

ALMASI, G. S.; GOTTLIEB, A. *Highly parallel computing*. Redwood City, CA, USA: Benjamin-Cummings Publishing Co., Inc., 1989. ISBN 0-8053-0177-1.

ALTED, F. Why modern CPUs are starving and what can be done about it. *Computing in Science and Engineering*, v. 12, n. 2, p. 68–71, 2010. ISSN 0740-7475.

ASANO, S.; MARUYAMA, T.; YAMAGUCHI, Y. Performance comparison of FPGA, GPU and CPU in image processing. In: *2009 International Conference on Field Programmable Logic and Applications*. Prague, Czech Republic: [s.n.], 2009. p. 126–131.

ASANOVIC, K. et al. *The Landscape of Parallel Computing Research: A View from Berkeley*. [S.l.], Dec 2006. Disponível em: <http://www.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-183.html>.

BERNSTEIN, K.; ROHRER, N. J. *SOI Circuit Design Concepts*. 1. ed. [S.l.]: Springer, 2000. ISBN 0792377621.

BLAKE, G.; DRESLINSKI, R.; MUDGE, T. A survey of multicore processors. *IEEE Signal Processing Magazine*, v. 26, n. 6, p. 26–37, 2009. ISSN 1053-5888.

BOAVENTURA I.A.G. ; GONZAGA, A. Método de avaliação de detector de bordas em imagens digitais. *V Workshop de Visão Computacional*, Anais do V Worskhop de Visão Computacional, 2009.

BRAUNL, T. et al. *Parallel Image Processing*. 1. ed. [S.l.]: Springer, 2001. ISBN 3540674004.

CABAN, J. J.; JOSHI, A.; NAGY, P. Rapid development of medical imaging tools with Open-Source libraries. *Journal of Digital Imaging*, v. 20, n. Suppl 1, p. 83–93, nov. 2007. ISSN 0897-1889. PMID: 17680307 PMCID: 2039808.

Cambridge in Colour. *Profundidade de Bits*. fev. 2012. Disponível em: <http://www-.cambridgeincolour.com/pt-br/tutorials/bit-depth.htm>.

CANNY, J. A computational approach to edge detection. *IEEE Trans. Pattern Anal. Mach. Intell.*, v. 8, n. 6, p. 679–698, 1986.

CLARK, D. OpenMP: a parallel standard for the masses. *Concurrency, IEEE*, v. 6, n. 1, p. 10–12, 1998. ISSN 1092-3063.

DAGUM, L.; MENON, R. OpenMP: an industry standard API for shared-memory programming. *Computational Science & Engineering, IEEE*, v. 5, n. 1, p. 46–55, 1998. ISSN 1070-9924.

DIEFENDORFF, K. et al. AltiVec extension to PowerPC accelerates media processing. *IEEE Micro*, v. 20, n. 2, p. 85–95, 2000.

DREPPER, U. *What Every Programmer Should Know About Memory*. 114 p. Tese (Doutorado), 2007. Disponível em: <http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.91.957>.

FLYNN, M. J. Some computer organizations and their effectiveness. *IEEE Transactions on Computers*, C-21, n. 9, p. 948–960, 1972. ISSN 0018-9340.

FOG, A. *The microarchitecture of Intel, AMD and VIA CPUs: An optimization guide for assembly programmers and compiler makers*. [S.l.]: Copenhagen University College of Engineering, fev. 2010.

Free Software Foundation, I. *Target Builtins - Using the GNU Compiler Collection (GCC)*. 2010. Disponível em: <http://gcc.gnu.org/onlinedocs/gcc-4.6.2/gcc/Target-Builtins.htm>.

Free Software Foundation, I. *Auto-vectorization in GCC*. 2011. Disponível em: <http://gcc-.gnu.org/projects/tree-ssa/vectorization.html>.

GONZALEZ, R. C.; WOODS, R. E. *Digital Image Processing*. 3. ed. [S.l.]: Prentice Hall, 2007. ISBN 013168728X.

GRAMA, A. et al. *Introduction to Parallel Computing*. 2. ed. [S.l.]: Addison Wesley, 2003. ISBN 0201648652.

HENNESSY, J. L.; PATTERSON, D. A. *Computer Architecture, Fifth Edition: A Quantitative Approach*. 5. ed. [S.l.]: Morgan Kaufmann, 2011. ISBN 012383872X.

IBÁÑEZ, L. et al. The ITK software guide. 2005.

Intel. *Intel FBD Spec Addendum rev 0.9*. mar. 2006. Disponível em: <http://www.intel.com-/technology/memory/FBDIMM/spec/Intel_FBD_Spec_Addendum_rev_p9.pdf>.

Intel. topic, *C++ Classes and SIMD Operations*. 2011. Disponível em: <http://software-.intel.com/sites/products/documentation/studio/composer/en-us/2011/compiler_c/cref_cls-/common/cppref_class_cpp_simd.htm>.

Intel. *Intel® 64 and IA-32 Architectures Optimization Reference Manual*. Intel, nov. 2011. Disponível em: <http://www.intel.com/Assets/en_US/PDF/manual/248966-.pdf?wapkw=Architectures%20Optimization%20Reference%20Manual>.

Intel. *Intel® 64 and IA-32 Architectures Software Developer's Manual: Basic Architecture*. Intel, set. 2011. Disponível em: <http://www.intel.com/content/dam/doc/manual/64-ia-32-architectures-software-developer-vol-1-manual.pdf>.

Intel. OptionRef, *#Pragma vector*. 2011. Disponível em: <http://software.intel.com-/sites/products/documentation/hpc/compilerpro/en-us/cpp/lin/compiler_c/cref_cls/common-/cppref_pragma_vector.htm>.

ITK. *ITK - Segmentation & Registration Toolkit*. maio 2011. Disponível em: <http://www.itk-.org/>.

ITOH, K. The history of DRAM circuit designs – at the forefront of DRAM development –. *Solid-State Circuits Newsletter, IEEE*, v. 13, n. 1, p. 27–31, 2008. ISSN 1098-4232.

KAMP, P. You're doing it wrong. *Queue*, v. 8, n. 6, p. 20:20–20:27, jun. 2010. ISSN 1542-7730.

KIM, D.; LEE, V.; CHEN, Y. Image processing on multicore x86 architectures. *IEEE Signal Processing Magazine*, v. 27, n. 2, p. 97–107, 2010. ISSN 1053-5888.

KIM, H.; BOND, R. Multicore software technologies. *IEEE Signal Processing Magazine*, v. 26, n. 6, p. 80–89, 2009. ISSN 1053-5888.

KitwarePublic. *ITK Release 4/Wish List - KitwarePublic*. jan. 2012. Disponível em: <http://www.cmake.org/Wiki/ITK_Release_4/Wish_List>.

KORN, A. F. Toward a symbolic representation of intensity changes in images. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, v. 10, n. 5, p. 610–625, set. 1988. ISSN 0162-8828.

LINDEBERG, T. Edge detection and ridge detection with automatic scale selection. *International Journal of Computer Vision*, Springer Netherlands, v. 30, p. 117–156, 1998. ISSN 0920-5691. 10.1023/A:1008097225773.

MAGGS, B.; MATHESON, L.; TARJAN, R. Models of parallel computation: a survey and synthesis. In: *Proceedings of the Twenty-Eighth Hawaii International Conference on System Sciences*. Wailea, HI, USA: [s.n.], 1995. p. 61–70.

MARTIN, D. et al. A database of human segmented natural images and its application to evaluating segmentation algorithms and measuring ecological statistics. In: *Proc. 8th Int'l Conf. Computer Vision*. [S.l.: s.n.], 2001. v. 2, p. 416–423.

McAuliffe, M. et al. Medical image processing, analysis and visualization in clinical research. In: *Proceedings 14th IEEE Symposium on Computer-Based Medical Systems. CBMS 2001*. Bethesda, MD, USA: [s.n.], 2001. p. 381–386.

Microsoft. *MMX, SSE, and SSE2 Intrinsics (C++)*. 2011. Disponível em: <http://msdn-.microsoft.com/en-us/library/y0dh78ez(v=vs.80).aspx>.

MOORE, G. E. Cramming more components onto integrated circuits. In: *Readings in computer architecture*. [S.l.]: Morgan Kaufmann Publishers Inc., 2000. p. 56–59. ISBN 1-55860-539-8.

NAISHLOS, D. Autovectorization in GCC. *GCC Developers Summit*, 2004.

OpenMP Architecture Review Board. *OpenMP Application Program Interface*. maio 2008. Disponível em: <http://www.openmp.org/mp-documents/spec30.pdf>.

PAGE, D. A practical introduction to computer architecture. In: ____. 1. ed. [S.l.]: Springer, 2009. p. 73–74, 430. ISBN 1848822553.

Samsung. *SAMSUNG Develops Industry's First DDR4DRAM, Using 30nm Class Technology*. jan 2011. Disponível em: <http://www.samsung.com/global/business/semiconductor-/newsView.do?news_id=1228>.

SATO, M. OpenMP: parallel programming API for shared memory multiprocessors and on-chip multiprocessors. In: *System Synthesis, 2002. 15th International Symposium on*. [S.l.: s.n.], 2002. p. 109–111.

SLABAUGH, G.; BOYES, R.; YANG, X. Multicore image processing with OpenMP [Applications corner. *IEEE Signal Processing Magazine*, v. 27, n. 2, p. 134–138, 2010. ISSN 1053-5888. Disponível em: <http://ieeexplore.ieee.org/xpl/downloadCitations>.

SLINGERLAND, N. T.; SMITH, A. J. Multimedia extensions for general purpose microprocessors: a survey. *Microprocessors and Microsystems*, v. 29, n. 5, p. 225–246, jun. 2005. ISSN 0141-9331.

STAMOPOULOS, C. D. Parallel image processing. *Computers, IEEE Transactions on*, C-24, n. 4, p. 424 – 433, abr. 1975. ISSN 0018-9340.

SUTTER, H. The free lunch is over: A fundamental turn toward concurrency in software. *Dr. Dobb's Journal*, v. 30, n. 3, 2005.

SUTTER, H. *Concur and C++ Futures*. set. 2006. Disponível em: <http://video.google.com-/videoplay?docid=7625918717318948700>.

SUTTER, H. *Machine Architecture: Things Your Programming Language Never Told You*. set. 2007. Disponível em: <http://video.google.com/videoplay?docid=-4714369049736584770>.

THAKKUR, S.; HUFF, T. Internet streaming SIMD extensions. *Computer*, v. 32, n. 12, p. 26–34, 1999. ISSN 0018-9162.

Valgrind™ Developers. *Valgrind Home*. 2011. Disponível em: <http://valgrind.org/>.

Valgrind™ Developers. *Valgrind User Manual - CallGrind*. 2011. Disponível em: <http://valgrind.org/docs/manual/cl-manual.html>.

WEISSTEIN, E. W. Text, *Convolution – from Wolfram MathWorld*. 2011. Disponível em: <http://mathworld.wolfram.com/Convolution.html>.

WESCOTT, T.; DOCEF, A. Optimi-Ware: software profiling and optimization [Best of the web]. *IEEE Signal Processing Magazine*, v. 24, n. 3, p. 131–133, maio 2007. ISSN 1053-5888.

YANG, Z.; ZHU, Y.; PU, Y. Parallel image processing based on CUDA. In: *2008 International Conference on Computer Science and Software Engineering*. Wuhan, China: [s.n.], 2008. p. 198–201.

YOO, T. S. et al. Engineering and algorithm design for an image processing API: a technical report on ITK - the insight toolkit. 2002.

APÊNDICE A – Códigos das implementações dos algoritmos de convolução

Listagem 21 Código-fonte do algoritmo `loopUnrollConvolve`

```
void loopUnroll32Convolve (const int s, const int w, const int h,
                          const int ks, const int kw,
                          const float* __restrict input,
                          float* __restrict output,
                          const float* kernel) {
    int hk          = kw / 2;
    int startX      = 0;
    int stopX       = w - 2 * hk;
    int startY      = 0;
    int stopY       = h - 2 * hk;
    #pragma omp parallel for shared (input, output)
    for (int y = startY; y < stopY; y++) {
        int x = 0;
        int yXstride = (y + hk) * s;
        for (x = startX; x < stopX - 31;) {
            output[yXstride + x + hk] = convolution(input, s, kernel, ks, kw, x, y); ++x;
            output[yXstride + x + hk] = convolution(input, s, kernel, ks, kw, x, y); ++x;
            output[yXstride + x + hk] = convolution(input, s, kernel, ks, kw, x, y); ++x;
            output[yXstride + x + hk] = convolution(input, s, kernel, ks, kw, x, y); ++x;
            output[yXstride + x + hk] = convolution(input, s, kernel, ks, kw, x, y); ++x;
            output[yXstride + x + hk] = convolution(input, s, kernel, ks, kw, x, y); ++x;
            output[yXstride + x + hk] = convolution(input, s, kernel, ks, kw, x, y); ++x;
            output[yXstride + x + hk] = convolution(input, s, kernel, ks, kw, x, y); ++x;
            output[yXstride + x + hk] = convolution(input, s, kernel, ks, kw, x, y); ++x;
            output[yXstride + x + hk] = convolution(input, s, kernel, ks, kw, x, y); ++x;
            output[yXstride + x + hk] = convolution(input, s, kernel, ks, kw, x, y); ++x;
            output[yXstride + x + hk] = convolution(input, s, kernel, ks, kw, x, y); ++x;
            output[yXstride + x + hk] = convolution(input, s, kernel, ks, kw, x, y); ++x;
            output[yXstride + x + hk] = convolution(input, s, kernel, ks, kw, x, y); ++x;
            output[yXstride + x + hk] = convolution(input, s, kernel, ks, kw, x, y); ++x;
            output[yXstride + x + hk] = convolution(input, s, kernel, ks, kw, x, y); ++x;
            output[yXstride + x + hk] = convolution(input, s, kernel, ks, kw, x, y); ++x;
            output[yXstride + x + hk] = convolution(input, s, kernel, ks, kw, x, y); ++x;
            output[yXstride + x + hk] = convolution(input, s, kernel, ks, kw, x, y); ++x;
            output[yXstride + x + hk] = convolution(input, s, kernel, ks, kw, x, y); ++x;
            output[yXstride + x + hk] = convolution(input, s, kernel, ks, kw, x, y); ++x;
            output[yXstride + x + hk] = convolution(input, s, kernel, ks, kw, x, y); ++x;
            output[yXstride + x + hk] = convolution(input, s, kernel, ks, kw, x, y); ++x;
            output[yXstride + x + hk] = convolution(input, s, kernel, ks, kw, x, y); ++x;
            output[yXstride + x + hk] = convolution(input, s, kernel, ks, kw, x, y); ++x;
            output[yXstride + x + hk] = convolution(input, s, kernel, ks, kw, x, y); ++x;
            output[yXstride + x + hk] = convolution(input, s, kernel, ks, kw, x, y); ++x;
            output[yXstride + x + hk] = convolution(input, s, kernel, ks, kw, x, y); ++x;
            output[yXstride + x + hk] = convolution(input, s, kernel, ks, kw, x, y); ++x;
            output[yXstride + x + hk] = convolution(input, s, kernel, ks, kw, x, y); ++x;
            output[yXstride + x + hk] = convolution(input, s, kernel, ks, kw, x, y); ++x;
            output[yXstride + x + hk] = convolution(input, s, kernel, ks, kw, x, y); ++x;
        } //for (int x = 0...
        for ( ; x < stopX; ++x) {
            output[yXstride + x + hk] = convolution(input, s, kernel, ks, kw, x, y);
        }
    } //for (int y = 0...
    processBoundaries2D (s, w, h,
                        ks, kw,
                        input, output, kernel);
}
```

Listagem 22 Algoritmo de convolução 2D com SSE

```
void sseReuse4Convolve (const int s, const int w, const int h,
                           const int ks, int kw,
                           const float* input, float* output, const float* kernel) {

    int hk = kw / 2;
    int startX  = 0;
    int stopX   = w - 2 * hk;
    int startY  = 0;
    int stopY   = h - 2 * hk;

    #pragma omp parallel for shared (input, output)
    for (int y = startY; y < stopY; ++y) {
```

```
13          for (int x = startX; x < stopX; x += 16) {
14              register __m128 sum0, sum1, sum2, sum3;
15              sum0 = sum1 = sum2 = sum3 = _mm_setzero_ps();
16              for (int r = 0; r < kw; ++r) {
17                  const int idxFtmp = r * ks;
18                  const int idxIntmp = (y + r) * s + x;
19                  __m128 iv0, iv1, iv2, iv3, iv4;
20                  iv0 = _mm_load_ps(&input[idxIntmp]);
21                  iv1 = _mm_load_ps(&input[idxIntmp + 4]);
22                  iv2 = _mm_load_ps(&input[idxIntmp + 8]);
23                  iv3 = _mm_load_ps(&input[idxIntmp + 12]);
24
25                  for (int c = 0; c < kw; c += 4) {
26
27                      register const __m128 kv = _mm_load_ps(&kernel[idxFtmp + c]);
28                      iv4 = _mm_load_ps(&input[idxIntmp + c + 16]);
29
30                      sum0 += _mm_dp241_ps(kv, iv0);
31                      sum1 += _mm_dp241_ps(kv, iv1);
32                      sum2 += _mm_dp241_ps(kv, iv2);
33                      sum3 += _mm_dp241_ps(kv, iv3);
34
35                      BLEND_ROTATE4_LEFT(iv0, iv1, iv2, iv3, iv4);
36                      sum0 += _mm_dp242_ps(kv, iv0);
37                      sum1 += _mm_dp242_ps(kv, iv1);
38                      sum2 += _mm_dp242_ps(kv, iv2);
39                      sum3 += _mm_dp242_ps(kv, iv3);
40
41                      BLEND_ROTATE4_LEFT(iv0, iv1, iv2, iv3, iv4);
42                      sum0 += _mm_dp244_ps(kv, iv0);
43                      sum1 += _mm_dp244_ps(kv, iv1);
44                      sum2 += _mm_dp244_ps(kv, iv2);
45                      sum3 += _mm_dp244_ps(kv, iv3);
46
47                      BLEND_ROTATE4_LEFT(iv0, iv1, iv2, iv3, iv4);
48                      sum0 += _mm_dp248_ps(kv, iv0);
49                      sum1 += _mm_dp248_ps(kv, iv1);
50                      sum2 += _mm_dp248_ps(kv, iv2);
51                      sum3 += _mm_dp248_ps(kv, iv3);
52
53                      BLEND_ROTATE4_LEFT(iv0, iv1, iv2, iv3, iv4);
54                  }
55              } //for (int r = 0...
56              _mm_storeu_ps(&output[(y + hk) * s + (x + hk)], sum0);
57              _mm_storeu_ps(&output[(y + hk) * s + (x + hk) + 4], sum1);
58              _mm_storeu_ps(&output[(y + hk) * s + (x + hk) + 8], sum2);
59              _mm_storeu_ps(&output[(y + hk) * s + (x + hk) + 12], sum3);
60          } //for (int x = 0...
61      } //for (int y = 0...
62      processBoundaries2D (s, w, h,
63                           ks, kw,
64                           input, output, kernel);
65  }
```

Listagem 23 Algoritmo de convolução 2D separável simples

```
1   void separableConvolve (const int s, const int w, const int h, const int kw,
2                           const float* __restrict input, float* __restrict output,
3                           const float* __restrict kernelX,
4                           const float* __restrict kernelY) {
5
6       const int hk = kw / 2;
7       const int startX  = 0;
8       const int stopX   = w - 2 * hk;
9       const int startY  = 0;
10      const int stopY   = h - 2 * hk;
11      const int kernelOffset = 2 * hk;
12
13      #pragma omp parallel for shared (input, output)
14      for (int y = startY; y < stopY; ++y) {
15
16          for (int x = 0; x < kw - 1; ++x) {
17              float sum = 0;
18              int idxIntmp = y * s + x;
19              for (int r = 0; r < kw; ++r) {
20                  sum += kernelY[r] * input[idxIntmp + (r * s)];
21              }
22              output[(y + hk) * s + x + hk] = sum;
23          }
24
25          for (int x = startX; x < stopX; ++x) {
26              float sum = 0;
27              int idxIntmp = y * s + x + (kw - 1);
28              for (int r = 0; r < kw; ++r) {
29                  sum += kernelY[r] * input[idxIntmp + (r * s)];
30              }
31              output[(y + hk) * s + x + (kw - 1) + hk] = sum;
32              sum = 0;
33              for (int c = 0; c < kw; ++c) {
34                  sum += kernelX[c] * output[(y + hk) * s + x + c + hk];
35              }
36              output[(y + hk) * s + x + hk] = sum;
37          }
38
39      }
40      processBoundariesS2D (s, w, h,
41                            kw,
42                            input, output,
43                            kernelX, kernelY);
44
45  }
```

Listagem 24 Algoritmo de convolução 2D separável com SSE

```
void scSSE (const int s, const int w, const int h, int kw,
            const float* input, float* output,
            const float* kernelX, const float* kernelY) {

    int hk = kw / 2;
    int startX  = 0;
    int stopX   = w - hk * 2;
    int startY  = 0;
    int stopY   = h - (kw + (4 - (kw % 4)));

    #pragma omp parallel for shared (input, output)
    for (int y = startY; y < stopY; ++y) {
        for (int x = startX; x < stopX; x += 16) {
            register __m128 sum0, sum1, sum2, sum3,
                            sumy0, sumy1, sumy2, sumy3, sumy4,
                            kvx, kvy;
            sum0 = sum1 = sum2 = sum3 =
            sumy0 = sumy1 = sumy2 = sumy3 = sumy4 = _mm_setzero_ps();
            __m128 iv0, iv1, iv2, iv3, iv4;
            for (int r = 0; r < kw; r += 4) {
                const int idxIntmp = (y + r) * s + x;
                kvx = _mm_load_ps(kernelY + r);

                kvy =  _mm_shuffle_ps(kvx, kvx, 0);
                iv0 = _mm_load_ps(&input[idxIntmp]);
                sumy0 += kvy * iv0;

                iv0 = _mm_load_ps(&input[idxIntmp + 4]);
                sumy1 += kvy * iv0;

                iv0 = _mm_load_ps(&input[idxIntmp + 8]);
                sumy2 += kvy * iv0;

                iv0 = _mm_load_ps(&input[idxIntmp + 12]);
                sumy3 += kvy * iv0;


                kvy = _mm_shuffle_ps(kvx, kvx, 85);
                iv0 = _mm_load_ps(&input[idxIntmp + s]);
                sumy0 += kvy * iv0;

                iv0 = _mm_load_ps(&input[idxIntmp + 4 + s]);
                sumy1 += kvy * iv0;

                iv0 = _mm_load_ps(&input[idxIntmp + 8 + s]);
                sumy2 += kvy * iv0;

                iv0 = _mm_load_ps(&input[idxIntmp + 12 + s]);
                sumy3 += kvy * iv0;


                kvy = _mm_shuffle_ps(kvx, kvx, 170);
                iv0 = _mm_load_ps(&input[idxIntmp + (s * 2)]);
                sumy0 += kvy * iv0;

                iv0 = _mm_load_ps(&input[idxIntmp + 4 + (s * 2)]);
                sumy1 += kvy * iv0;

                iv0 = _mm_load_ps(&input[idxIntmp + 8 + (s * 2)]);
                sumy2 += kvy * iv0;

                iv0 = _mm_load_ps(&input[idxIntmp + 12 + (s * 2)]);
                sumy3 += kvy * iv0;


                kvy = _mm_shuffle_ps(kvx, kvx, 255);
                iv0 = _mm_load_ps(&input[idxIntmp + (s * 3)]);
                sumy0 += kvy * iv0;

                iv0 = _mm_load_ps(&input[idxIntmp + 4 + (s * 3)]);
                sumy1 += kvy * iv0;

                iv0 = _mm_load_ps(&input[idxIntmp + 8 + (s * 3)]);
                sumy2 += kvy * iv0;

                iv0 = _mm_load_ps(&input[idxIntmp + 12 + (s * 3)]);
                sumy3 += kvy * iv0;

            } //for (int r = 0...

            iv0 = sumy0;
            iv1 = sumy1;
            iv2 = sumy2;
            iv3 = sumy3;

            for (int c = 0; c < kw; c += 4) {
                sumy4 = _mm_setzero_ps();

                for (int r = 0; r < kw; r += 4) {

                    const int idxIntmp = (y + r) * s + x + c;

                    kvx = _mm_load_ps(kernelY + r);

                    kvy =  _mm_shuffle_ps(kvx, kvx, 0);
                    iv4 = _mm_load_ps(&input[idxIntmp + 16]);
                    sumy4 += kvy * iv4;

                    kvy = _mm_shuffle_ps(kvx, kvx, 85);
                    iv4 = _mm_load_ps(&input[idxIntmp + 16 + s]);
                    sumy4 += kvy * iv4;

                    kvy = _mm_shuffle_ps(kvx, kvx, 170);
                    iv4 = _mm_load_ps(&input[idxIntmp + 16 + (s * 2)]);
                    sumy4 += kvy * iv4;

                    kvy = _mm_shuffle_ps(kvx, kvx, 255);
                    iv4 = _mm_load_ps(&input[idxIntmp + 16 + (s * 3)]);
                    sumy4 += kvy * iv4;

                } //for (int r = 0...

                kvx = _mm_load_ps(&kernelX[c]);

```

```
115                 iv4 = sumy4;
116
117                 sum0 += _mm_dp_ps(kvx, iv0, 241);
118                 sum1 += _mm_dp_ps(kvx, iv1, 241);
119                 sum2 += _mm_dp_ps(kvx, iv2, 241);
120                 sum3 += _mm_dp_ps(kvx, iv3, 241);
121
122                 BLEND_ROTATE4_LEFT(iv0, iv1, iv2, iv3, iv4);
123
124                 sum0 += _mm_dp_ps(kvx, iv0, 242);
125                 sum1 += _mm_dp_ps(kvx, iv1, 242);
126                 sum2 += _mm_dp_ps(kvx, iv2, 242);
127                 sum3 += _mm_dp_ps(kvx, iv3, 242);
128
129                 BLEND_ROTATE4_LEFT(iv0, iv1, iv2, iv3, iv4);
130
131                 sum0 += _mm_dp_ps(kvx, iv0, 244);
132                 sum1 += _mm_dp_ps(kvx, iv1, 244);
133                 sum2 += _mm_dp_ps(kvx, iv2, 244);
134                 sum3 += _mm_dp_ps(kvx, iv3, 244);
135
136                 BLEND_ROTATE4_LEFT(iv0, iv1, iv2, iv3, iv4);
137
138                 sum0 += _mm_dp_ps(kvx, iv0, 248);
139                 sum1 += _mm_dp_ps(kvx, iv1, 248);
140                 sum2 += _mm_dp_ps(kvx, iv2, 248);
141                 sum3 += _mm_dp_ps(kvx, iv3, 248);
142
143                 BLEND_ROTATE4_LEFT(iv0, iv1, iv2, iv3, iv4);
144
145             }
146
147             _mm_storeu_ps(&output[(y + hk) * s + (x + hk)], sum0);
148             _mm_storeu_ps(&output[(y + hk) * s + (x + hk) + 4], sum1);
149             _mm_storeu_ps(&output[(y + hk) * s + (x + hk) + 8], sum2);
150             _mm_storeu_ps(&output[(y + hk) * s + (x + hk) + 12], sum3);
151         } //for (int x = 0...
152     } //for (int y = 0...
153
154     int hh = h - stopY;
155     const float *inputT = &input[stopY * s];
156     float *outputT = &output[stopY * s];
157
158     separableConvolve (s, w, hh, kw, inputT, outputT, kernelX, kernelY, false);
159
160     processBoundariesS2D (s, w, h,
161                           kw,
162                           input, output, kernelX, kernelY);
163 }
```

APÊNDICE B – Códigos da implementação do algoritmo de detecção de bordas Canny otimizado com SSE e OpenMP para ITK

Listagem 25 Método `Compute2ndDerivative`

```
template< class TInputImage, class TOutputImage >
void
OpCannyEdgeDetectionImageFilter< TInputImage, TOutputImage >
::Compute2ndDerivative(const int imageStride, const int imageWidth,
                       const int imageHeight,
                       const float* inputImage, float* outputImage, const int outStride)
{
    const int kernelWidth = 3;
    const int halfKernel = kernelWidth / 2;

    int startX  = 0;
    int stopX   = imageWidth - 2 * halfKernel;
    int startY  = 0;
    int stopY   = (imageHeight - 2 * halfKernel);

    #pragma omp parallel for shared (inputImage, outputImage)
    for (int y = startY; y < stopY; ++y) {
      for (int x = startX; x < stopX; x += 4) {
        register __m128 dx, dy, dxx, dyy, dxy;
        register __m128 kv0, kv1, kv2;
        const register __m128 kdxy0 = _mm_set_ps(0.25, 0.25, -0.25, -0.25);
        const register __m128 kdxy2 = _mm_set_ps(-0.25, -0.25, 0.25, 0.25);
        register __m128 iv00, iv01, iv02, iv10, iv11, iv12;

        const int y0 = (y + 0) * imageStride + x;
        const int y1 = (y + 1) * imageStride + x;
        const int y2 = (y + 2) * imageStride + x;

        iv00 = _mm_load_ps(&inputImage[y0]);
        iv01 = _mm_load_ps(&inputImage[y1]);
        iv02 = _mm_load_ps(&inputImage[y2]);

        iv10 = _mm_load_ps(&inputImage[y0 + 4]);
        iv11 = _mm_load_ps(&inputImage[y1 + 4]);
        iv12 = _mm_load_ps(&inputImage[y2 + 4]);

        //dx
        kv0 = _mm_set_ps(0.5, 0.5, -0.5, -0.5);
        dx = _mm_dpil_ps(iv01, kv0);
        //dxx
        kv0 = _mm_set_ps(0.0, 1.0, -2.0, 1.0);
        dxx = _mm_dp113_ps(iv01, kv0);
        //dxy
        dxy = _mm_vdpil_ps(iv00, kdxy0, iv02, kdxy2);

        BLEND_ROTATE31_LEFT(iv00, iv10, iv01, iv11, iv02, iv12)

        //dy
        kv0 = _mm_set_ps(0.5, 0.5, 0.5, 0.5);
        kv2 = _mm_set_ps(-0.5, -0.5, -0.5, -0.5);
        dy = _mm_v2dp_ps(iv00, kv0, iv02, kv2);
        //dyy
        kv1 = _mm_set_ps(-2.0, -2.0, -2.0, -2.0);
        dyy = _mm_v3mdp_ps(iv00, iv01, iv02, kv1);
        //dxx
        kv0 = _mm_set_ps(0.0, 1.0, -2.0, 1.0);
        dxx += _mm_dp114_ps(iv01, kv0);

        BLEND_ROTATE31_LEFT(iv00, iv10, iv01, iv11, iv02, iv12)
        //dx
        kv0 = _mm_set_ps(0.5, 0.5, -0.5, -0.5);
        dx += _mm_dpih_ps(iv01, kv0);
        //dxx
        kv0 = _mm_set_ps(0.0, 1.0, -2.0, 1.0);
        dxx += _mm_dp116_ps(iv01, kv0);
        //dxy
        dxy += _mm_vdpih_ps(iv00, kdxy0, iv02, kdxy2);

        BLEND_ROTATE1_LEFT(iv01, iv11)
        //dxx
        kv0 = _mm_set_ps(0.0, 1.0, -2.0, 1.0);
        dxx += _mm_dp120_ps(iv01, kv0);
        //calc
```

```
74          _mm_storeu_ps(&outputImage[(y + halfKernel) * outStride + (x + halfKernel)],
75                        _mm_lvv_ps(dx, dy, dxx, dyy, dxy));
76      }
77    }
78  }
```

Listagem 26 Método `Compute2ndDerivativePos`

```
1   template< class TInputImage, class TOutputImage >
2   void
3   OpCannyEdgeDetectionImageFilter< TInputImage, TOutputImage >
4   ::Compute2ndDerivativePos(const int imageStride, const int imageWidth,
5                         const int imageHeight,
6                         const float* lvInput, const float* lvvInput,
7                         float* outputImage, const int outStride)
8   {
9
10      const int kernelWidth = 3;
11      const int halfKernel = kernelWidth / 2;
12
13      int startX  = 0;
14      int stopX   = imageWidth - 2 * halfKernel;
15      int startY  = 0;
16      int stopY   = imageHeight - 2 * halfKernel;
17
18      #pragma omp parallel for shared (lvInput, lvvInput, outputImage)
19      for (int y = startY; y < stopY; ++y) {
20        for (int x = startX; x < stopX; x += 4) {
21          register __m128 lx, ly, lvvx, lvvy, lv;
22          const register __m128 kv0 = _mm_set_ps(0.5, 0.5, -0.5, -0.5);
23          const register __m128 kv1 = _mm_set_ps(0.5, 0.5, 0.5, 0.5);
24          const register __m128 kv2 = _mm_set_ps(-0.5, -0.5, -0.5, -0.5);
25          __m128 iv00, iv01, iv02, iv10, iv11, iv12;
26
27          const int y0 = (y + 0) * imageStride + x;
28          const int y1 = (y + 1) * imageStride + x;
29          const int y2 = (y + 2) * imageStride + x;
30
31          iv00 = _mm_load_ps(&lvInput[y0]);
32          iv01 = _mm_load_ps(&lvInput[y1]);
33          iv02 = _mm_load_ps(&lvInput[y2]);
34
35          iv10 = _mm_load_ps(&lvInput[y0 + 4]);
36          iv11 = _mm_load_ps(&lvInput[y1 + 4]);
37          iv12 = _mm_load_ps(&lvInput[y2 + 4]);
38
39          //lx
40          lx = _mm_dpil_ps(iv01, kv0);
41          BLEND_ROTATE31_LEFT(iv00, iv10, iv01, iv11, iv02, iv12)
42          //ly
43          ly = _mm_v2dp_ps(iv00, kv1, iv02, kv2);
44
45          BLEND_ROTATE1_LEFT(iv01, iv11)
46          lx += _mm_dpih_ps(iv01, kv0);
47
48          iv00 = _mm_load_ps(&lvvInput[y0]);
49          iv01 = _mm_load_ps(&lvvInput[y1]);
50          iv02 = _mm_load_ps(&lvvInput[y2]);
51
52          iv10 = _mm_load_ps(&lvvInput[y0 + 4]);
53          iv11 = _mm_load_ps(&lvvInput[y1 + 4]);
54          iv12 = _mm_load_ps(&lvvInput[y2 + 4]);
55          //Lvvx
56          lvvx = _mm_dpil_ps(iv01, kv0);
57          BLEND_ROTATE31_LEFT(iv00, iv10, iv01, iv11, iv02, iv12)
58          //Lvvy
59          lvvy = _mm_v2dp_ps(iv00, kv1, iv02, kv2);
60
61          BLEND_ROTATE1_LEFT(iv01, iv11)
62          //lvvx
63          lvvx += _mm_dpih_ps(iv01, kv0);
64
65          iv00 = lx * lx;
66          iv01 = ly * ly;
67          lv = iv00 + iv01;
68          lv = _mm_sqrt_ps (lv);
69          iv00 = lvvx * (lx / lv);
70          iv00 += lvvy * (ly / lv);
71          iv00 = _mm_cmple_ps(iv00,  _mm_setzero_ps());
72          iv00 = _mm_and_ps(iv00, _mm_set1_ps(0x00000001));
73          iv00 *= lv;
74
75          _mm_storeu_ps(&outputImage[(y + halfKernel) *
76       outStride + (x + halfKernel)], iv00);
77      }
78    }
79  }
```

Listagem 27 Médodo `ZeroCrossing`

```
1   template< class TInputImage, class TOutputImage >
2   void
3   OpCannyEdgeDetectionImageFilter< TInputImage, TOutputImage >
4   ::ZeroCrossing()
5   {
6
7     typename TInputImage::SizeType regionSize =
8           this->GetInput()->GetRequestedRegion().GetSize();
9
10      int imageWidth = regionSize[0];
11      int imageHeight = regionSize[1];
```

```
12
13      float* inputImage = this->GetOutput()->GetBufferPointer();
14      float* outputImage  = m_GaussianBuffer->GetBufferPointer();
15
16      const int kernelWidth = 3;
17      const int halfKernel = kernelWidth / 2;
18      const int imageStride = this->GetInput()->GetOffsetTable()[1];
19
20      int startX  = 4;
21      int stopX   = (imageWidth - 2 * halfKernel) - 4;
22      int startY  = 0;
23      int stopY   = imageHeight - 2 * halfKernel;
24
25
26      #pragma omp parallel for shared (inputImage, outputImage)
27      for (int y = startY; y < stopY; ++y) {
28        for (int x = startX; x < stopX; x += 4) {
29          static const __m128 sign = _mm_castsi128_ps(_mm_set1_epi32(0x80000000));
30          static const __m128 one = _mm_set1_ps(0x00000001);
31          static const __m128 abs = _mm_castsi128_ps(_mm_set1_epi32(0x7FFFFFFF));
32
33          __m128 thisOne = _mm_load_ps(&inputImage[(y + 1) * imageStride + x]);
34          __m128 that = _mm_load_ps(&inputImage[y * imageStride + x]);
35          __m128 absThat = _mm_and_ps(that, abs);
36          __m128 absThis = _mm_and_ps(thisOne, abs);
37          __m128 maskThis = _mm_and_ps(thisOne, sign);
38          __m128 maskThat = _mm_and_ps(that, sign);
39
40          maskThis = _mm_or_ps(maskThis, one);
41          maskThat = _mm_or_ps(maskThat, one);
42
43          maskThat = _mm_cmpneq_ps(maskThis, maskThat);
44
45          __m128 lessThan = _mm_cmplt_ps(absThis, absThat);
46
47          __m128 equal = _mm_cmpeq_ps(absThis, absThat);
48
49          lessThan = _mm_or_ps(lessThan, equal);
50
51          __m128 out = _mm_and_ps(lessThan, maskThat);
52
53          if (_mm_movemask_ps(out) == 0xf) {
54            _mm_storeu_ps(&outputImage[(y + 1) * imageStride + x], _mm_and_ps(out, one));
55            continue;
56          }
57
58          that = _mm_load_ps(&inputImage[(y + 2) * imageStride + x]);
59
60          maskThat = _mm_and_ps(that, sign);
61          maskThat = _mm_or_ps(maskThat, one);
62
63          maskThat = _mm_cmpneq_ps(maskThis, maskThat);
64          maskThat = _mm_or_ps(maskThat, out);
65
66          absThat = _mm_and_ps(that, abs);
67  //        absThis = _mm_and_ps(thisOne, abs);
68
69          lessThan = _mm_cmplt_ps(absThis, absThat);
70
71          equal = _mm_cmpeq_ps(absThis, absThat);
72
73          lessThan = _mm_or_ps(lessThan, equal);
74
75          maskThat = _mm_and_ps(lessThan, maskThat);
76
77          out = _mm_or_ps(maskThat, out);
78
79          if (_mm_movemask_ps(out) == 0xf) {
80            _mm_storeu_ps(&outputImage[(y + 1) * imageStride + x], _mm_and_ps(out, one));
81            continue;
82          }
83
84          that = _mm_load_ps(&inputImage[(y + 1) * imageStride + x - 4]);
85          that = _mm_blend_ps(thisOne, that, 8);
86
87          ROTATE_RIGHT(that);
88
89          maskThat = _mm_and_ps(that, sign);
90          maskThat = _mm_or_ps(maskThat, one);
91
92          maskThat = _mm_cmpneq_ps(maskThis, maskThat);
93          maskThat = _mm_or_ps(maskThat, out);
94
95          absThat = _mm_and_ps(that, abs);
96  //        absThis = _mm_and_ps(thisOne, abs);
97
98          lessThan = _mm_cmplt_ps(absThis, absThat);
99
100         maskThat = _mm_and_ps(lessThan, maskThat);
101
102         out = _mm_or_ps(maskThat, out);
103
104
105         if (_mm_movemask_ps(out) == 0xf) {
106           _mm_storeu_ps(&outputImage[(y + 1) * imageStride + x], _mm_and_ps(out, one));
107           continue;
108         }
109
110         that = _mm_load_ps(&inputImage[(y + 1) * imageStride + x + 4]);
111         that = _mm_blend_ps(thisOne, that, 1);
112         ROTATE_LEFT(that);
113         maskThat = _mm_and_ps(that, sign);
114         maskThat = _mm_or_ps(maskThat, one);
115
116         maskThat = _mm_cmpneq_ps(maskThis, maskThat);
117         maskThat = _mm_or_ps(maskThat, out);
118
119         absThat = _mm_and_ps(that, abs);
120 //        absThis = _mm_and_ps(thisOne, abs);
121
122         lessThan = _mm_cmplt_ps(absThis, absThat);
123
124         maskThat = _mm_and_ps(lessThan, maskThat);
125
126         out = _mm_or_ps(maskThat, out);
```

```
127
128           _mm_storeu_ps(&outputImage[(y + 1) * imageStride + x], _mm_and_ps(out, one));
129
130
131       }
132     }
133   //  Compute2ndDerivativeBondaries(imageStride, imageWidth,
134   //                                imageHeight, inputImage, inputImage,
135   //                                outputImage);
136   }
```

Listagem 28 Médodo `HysteresisThresholding`

```
1   template< class TInputImage, class TOutputImage >
2   void
3   OpCannyEdgeDetectionImageFilter< TInputImage, TOutputImage >
4   ::HysteresisThresholding()
5   {
6
7     typename TInputImage::SizeType regionSize =
8           this->GetInput()->GetRequestedRegion().GetSize();
9
10      int imageWidth = regionSize[0];
11      int imageHeight = regionSize[1];
12
13      float* inputImage  = m_GaussianBuffer->GetBufferPointer();
14      float* outputImage = this->GetOutput()->GetBufferPointer();
15
16      PRINT_LABEL ("HysteresisThresholding");
17
18
19      const int imageStride = this->GetInput()->GetOffsetTable()[1];
20
21      ClearBuffer( outputImage,
22                   imageStride,
23                   imageHeight );
24
25      ClearBuffer( m_UpdateBuffer->GetBufferPointer(),
26                   imageStride,
27                   imageHeight );
28
29      PRINT(imageWidth);
30
31      int startX  = 0;
32      int stopX   = imageWidth;
33      int startY  = 0;
34      int stopY   = imageHeight;
35
36      HysteresisEdgeIndex* buffer = (HysteresisEdgeIndex*)(m_UpdateBuffer->GetBufferPointer());
37      int offset = imageHeight / omp_get_num_threads();
38
39      HysteresisQueue* queues = new HysteresisQueue[omp_get_max_threads()];
40
41      for(int i = 0; i < omp_get_max_threads(); ++i) {
42        queues[i] = HysteresisQueue(buffer);
43        buffer += offset;
44      }
45
46      #pragma omp parallel for shared (queues, inputImage, outputImage)
47      for (int y = startY; y < stopY; ++y) {
48        HysteresisQueue queue = queues[omp_get_thread_num()];
49          const register __m128 upperThreshold = _mm_set1_ps(m_UpperThreshold);
50
51        for (int x = startX; x < stopX; x += 4) {
52
53          __m128 value = _mm_load_ps(&inputImage[y * imageStride + x]);      PRINT_VECTOR(thisOne);
54
55          unsigned int mask = _mm_movemask_ps(_mm_cmpgt_ps(value, upperThreshold));
56          if (mask == 0x0) continue;
57
58          if(mask & 1) //pixel 0
59          {
60            queue.Enqueue(x, y);
61            FollowEdge(imageStride, imageWidth, imageHeight, queue, inputImage, outputImage);
62          }
63
64          if((mask & 2) >> 1) //pixel 1
65          {
66            queue.Enqueue(x + 1, y);
67            FollowEdge(imageStride, imageWidth, imageHeight, queue, inputImage, outputImage);
68          }
69
70          if((mask & 4) >> 2) //pixel 2
71          {
72            queue.Enqueue(x + 2, y);
73            FollowEdge(imageStride, imageWidth, imageHeight, queue, inputImage, outputImage);
74          }
75
76          if((mask & 8) >> 3) //pixel 3
77          {
78            queue.Enqueue(x + 3, y);
79            FollowEdge(imageStride, imageWidth, imageHeight, queue, inputImage, outputImage);
80          }
81        }
82      }
83  }
84
85  template< class TInputImage, class TOutputImage >
86  inline void
87  OpCannyEdgeDetectionImageFilter< TInputImage, TOutputImage >
88  ::FollowEdge(int imageStride, int imageWidth, int imageHeight, HysteresisQueue& queue,
89               float* input, float* output)
90  {
91      HysteresisEdgeIndex* idx = queue.Pick();
92
93      if(output[idx->Y * imageStride + idx->X] == NumericTraits<OutputImagePixelType>::One )
94      {
95        // we must remove the node if we are not going to follow it!
```

```
96          // Pop the front node from the list and read its index value.
97          queue.Dequeue();
98          return;
99        }
100
101       int x;
102       int y;
103       while(!queue.IsEmpty())
104       {
105
106           idx = queue.Dequeue();
107           x = idx->X;
108           y = idx->Y;
109           VerifyEdge(x, y, imageStride, imageWidth, imageHeight, queue,
110                      input, output);
111       }
112   }
113
114   template< class TInputImage, class TOutputImage >
115   inline void
116   OpCannyEdgeDetectionImageFilter< TInputImage, TOutputImage >
117   ::VerifyEdge(int x, int y, int imageStride, int imageWidth, int imageHeight, HysteresisQueue& queue,
118                float* input, float* output)
119   {
120         int i;
121         output[y * imageStride + x] = 1;
122
123         if(x > 0) {
124           i = y * imageStride + x - 1; //west
125           if(input[i] > m_LowerThreshold &&
126              output[i] != 1) {
127               queue.Enqueue(x - 1, y);
128               output[i] = 1;
129           }
130         }
131
132         if(x > 0 && y > 0) {
133           i = (y - 1) * imageStride + x - 1; //north west
134           if(input[i] > m_LowerThreshold &&
135              output[i] != 1) {
136               queue.Enqueue(x - 1, y - 1);
137               output[i] = 1;
138           }
139         }
140
141         if(y > 0) {
142           i = (y - 1) * imageStride + x; //north
143           if(input[i] > m_LowerThreshold &&
144              output[i] != 1) {
145               queue.Enqueue(x, y - 1);
146               output[i] = 1;
147           }
148         }
149
150         if(x < imageWidth && y > 0) {
151           i = (y - 1) * imageStride + x + 1; //north east
152           if(input[i] > m_LowerThreshold &&
153              output[i] != 1) {
154               queue.Enqueue(x + 1, y - 1);
155               output[i] = 1;
156           }
157         }
158
159         if(x < imageWidth) {
160           i = y * imageStride + x + 1; //east
161           if(input[i] > m_LowerThreshold &&
162              output[i] != 1) {
163               queue.Enqueue(x + 1, y);
164               output[i] = 1;
165           }
166         }
167
168         if(x < imageWidth && y < imageHeight) {
169           i = (y + 1) * imageStride + x + 1; //south east
170           if(input[i] > m_LowerThreshold &&
171              output[i] != 1) {
172               queue.Enqueue(x + 1, y + 1);
173               output[i] = 1;
174           }
175         }
176
177         if(y < imageHeight) {
178           i = (y + 1) * imageStride + x; //south
179           if(input[i] > m_LowerThreshold &&
180              output[i] != 1) {
181               queue.Enqueue(x, y + 1);
182               output[i] = 1;
183           }
184         }
185
186         if(x > 0 && y < imageHeight) {
187           i = (y + 1) * imageStride + x - 1; //south west
188           if(input[i] > m_LowerThreshold &&
189              output[i] != 1) {
190               queue.Enqueue(x - 1, y + 1);
191               output[i] = 1;
192           }
193         }
194   }
```

APÊNDICE C – Funções SSE 4.1 emuladas

Listagem 29 Funções SSE 4.1 emuladas

```
#ifndef __SSE4_1__

    typedef union
    {
        __m128  f;
        __m128i i;
    } ssp_m128;

    extern __inline __m128 __attribute__((__gnu_inline__, __always_inline__, __artificial__))
    _mm_dp_ps (__m128 a, __m128 b, const int mask) {
        //http://sseplus.sourceforge.net/group__emulated___s_s_e3.html
        // Shift mask multiply moves 0,1,2,3 bits to left, becomes MSB
        const static __m128i mulShiftImm_0123 = _mm_set_epi32( 0x010000, 0x020000,
                                                               0x040000, 0x080000 );
        // Shift mask multiply moves 4,5,6,7 bits to left, becomes MSB
        const static __m128i mulShiftImm_4567 = _mm_set_epi32( 0x100000, 0x200000,
                                                               0x400000, 0x800000 );

        // Begin mask preparation
        ssp_m128 mHi, mLo;
        mLo.i = _mm_set1_epi32( mask ); // Load the mask into register
        mLo.i = _mm_slli_si128( mLo.i, 3 );// Shift into reach of the 16 bit multiply

        mHi.i = _mm_mullo_epi16( mLo.i, mulShiftImm_0123 );  // Shift the bits
        mLo.i = _mm_mullo_epi16( mLo.i, mulShiftImm_4567 );  // Shift the bits

        // FFFFFFFF if bit set, 00000000 if not set
        mHi.i = _mm_cmplt_epi32( mHi.i, _mm_setzero_si128() );
        // FFFFFFFF if bit set, 00000000 if not set
        mLo.i = _mm_cmplt_epi32( mLo.i, _mm_setzero_si128() );
        // End mask preparation - Mask bits 0-3 in mLo, 4-7 in mHi
        a = _mm_and_ps( a, mHi.f ); // Clear input using the high bits of the mask
        a = _mm_mul_ps( a, b );

        a = _mm_hadd_ps( a, a ); // Horizontally add the 4 values
        a = _mm_hadd_ps( a, a ); // Horizontally add the 4 values
        a = _mm_and_ps( a, mLo.f );// Clear output using low bits of the mask
        return a;
    }

    extern __inline __m128i
    __attribute__((__gnu_inline__, __always_inline__, __artificial__))
    ssp_movmask_imm8_to_epi32_SSE2( int mask ) {
        __m128i screen;
        // Shift mask multiply moves all bits to left, becomes MSB
        const static __m128i mulShiftImm =
          _mm_set_epi16( 0x1000, 0x0000, 0x2000, 0x0000,
                         0x4000, 0x0000, 0x8000, 0x0000 );
        // Load the mask into register
        screen = _mm_set1_epi16 ( mask );
        // Shift bits to MSB
        screen = _mm_mullo_epi16( screen, mulShiftImm );
        // Shift bits to obtain all F's or all 0's
        screen = _mm_srai_epi32 ( screen, 31  );
        return screen;
    }

    extern __inline __m128
    __attribute__((__gnu_inline__, __always_inline__, __artificial__))
    _mm_blend_ps( __m128  a, __m128  b, const int mask )// _mm_blend_ps [SSE4.1]
    {

        ssp_m128 screen, A, B;
        A.f = a;
        B.f = b;
        screen.i = ssp_movmask_imm8_to_epi32_SSE2 ( mask );
        B.i = _mm_and_si128   ( B.i, screen.i);  // clear a where mask = 0
        A.i = _mm_andnot_si128( screen.i, A.i ); // clear b where mask = 1
        screen.i = _mm_or_si128  ( A.i, B.i );   // a = a OR b
        return screen.f;
    }

#endif
```