Carlos Alex Sander Juvêncio Gulo

Técnicas de paralelização em GPGPU aplicadas em algoritmo para remoção de ruído multiplicativo

São José do Rio Preto
2012

Carlos Alex Sander Juvêncio Gulo

Técnicas de paralelização em GPGPU aplicadas em algoritmo para remoção de ruído multiplicativo

Dissertação apresentada para obtenção do título de Mestre em Ciência da Computação, área de Computação Aplicada junto ao Programa de Pós-Graduação em Ciência da Computação do Instituto de Biociências, Letras e Ciências Exatas da Universidade Estadual Paulista "Júlio de Mesquita Filho", Campus de São José do Rio Preto.

Orientador: Prof. Dr. Antonio Carlos Sementille

São José do Rio Preto
2012

Gulo, Carlos Alex Sander Juvêncio.
Técnicas de paralelização em GPGPU aplicadas em algoritmo para remoção de ruído multiplicativo / Carlos Alex Sander Juvêncio Gulo. - São José do Rio Preto : [s.n.], 2012.
80 f. : il. ; 30 cm.

Orientador: Antonio Carlos Sementille
Dissertação (Mestrado) - Universidade Estadual Paulista, Instituto de Biociências, Letras e Ciências Exatas

1. Processamento de Imagem. 2. Suavização. 3. Ruído Multiplicativo. I. Sementille, Antonio Carlos. II. Universidade Estadual Paulista, Instituto de Biociências, Letras e Ciências Exatas. III. Título.

CDU – 004.932

Ficha catalográfica elaborada pela Biblioteca do IBILCE
Campus de São José do Rio Preto - UNESP

Especialmente ao Mateus, meu filho *expert* e minha esposa Tatiana, pelo incentivo, apoio, paciência e dedicação ao aceitar nossa ideia de “mestrar” juntos. Preparem-se, papai está chegando para aquela disputa no tênis de quadra, porque no tênis de mesa ainda estou dominando!

Agradecimentos

Primeiramente à Deus, por ter sido agraciado com muita saúde, amizade, companheirismo e apoio da minha família e amigos. Graças a muita insistência, persistência, crença, e muita, mas muita fé mesmo, depois de percorridos mais de 110 mil KM conquistamos mais esta etapa em nossas vidas, Mateus e Tatiana! Obrigado Deus Pai todo poderoso por todas suas graças e por sempre iluminar nossos caminhos na vida e em cada viagem. SUA benção iluminou o caminho de cada motorista de ônibus na estrada, em travessias de balsa, no cumprimento de cada trajeto *destas estradas brasileiras* que *ligam* o centro-oeste à região sudeste do país. *Amém*!

À minha esposa Tatiana e meu filhote Mateus, pelo apoio, compreensão, companheirismo e pelo amor que me dedicaram durante todo esse tempo, me fazendo querer ser uma pessoa melhor e me ajudando nas horas mais difíceis com tanto carinho, atenção e muita paciência.

Aos meus familiares, pelo constante apoio, incentivo, amor, desejando força para sempre continuar em frente. Jurandir meu pai, Armelinda minha mãe, Vitor e Fábio meus queridos *brothers*, as respectivas cunhadas e sobrinhos lindos (*Juca, Cotchô e Dudona*). Sogrão Noé, cunhada "Fábia" e sogra Kitamura.

Aos amigos, pelas inúmeras vezes que estiveram ao lado da minha esposa e filho na minha ausência, fazendo companhia e oferecendo momentos de distração, alegria e amizade, para diminuir a saudade em cada período em que estive longe de casa. E não foram poucos!

Ao meu orientador, professor Sementille, meu agradecimento especial, não apenas pela oportunidade no desenvolvimento desta pesquisa, mas também por sua dedicação, boa vontade e nível de exigência adequados durante todas as etapas desta parceria. Muito obrigado pelos ensinamentos!

Aos meus colegas de Laboratório e aos professores do Programa, por compartilhar os vários momentos de reflexão, a troca de experiências, as caronas de ida ou volta para a rodoviária (André, Diego e Holdschip), e as colaborações para um aprendizado constante. Um agradecimento especial também ao Alex Araújo, sempre muito atencioso, prestativo e parceiro. O Henrique pela colaboração e disposição em aceitar os desafios produtivos para realizar experimentos "em cima da hora".

À CAPES, pelo financiamento da bolsa de Mestrado, à Universidade do Estado de Mato Grosso (UNEMAT), pelo apoio concedido e ao Laboratório Sistemas Adaptativos e Computação Inteligente (SACI), pelo uso de suas dependências.

"The reasonable man adapts himself to the world; the unreasonable one persists in trying to adapt the world to himself. Therefore all progress depends on the unreasonable man."

George B. Shaw.

Resumo

A evolução constante na velocidade de cálculos dos processadores tem sido uma grande aliada no desenvolvimento de áreas da Ciência que exigem processamento de alto desempenho. Associados aos recursos computacionais faz-se necessário o emprego de técnicas de computação paralela no intuito de explorar ao máximo a capacidade de processamento da arquitetura escolhida, bem como, reduzir o tempo de espera no processamento. No entanto, o custo financeiro para aquisição deste tipo de *hardware* não é muito baixo, implicando na busca de alternativas para sua utilização. As arquiteturas de processadores *multicore* e *General Purpose Computing on Graphics Processing Unit* (GPGPU), tornam-se opções de baixo custo, pois são projetadas para oferecer infraestrutura para o processamento de alto desempenho e atender aplicações de tempo real. Com o aperfeiçoamento das tecnologias multicomputador, multiprocessador e GPGPU, a paralelização de técnicas de processamento de imagem tem obtido destaque por viabilizar a redução do tempo de processamento de métodos complexos aplicados em imagem de alta resolução. Neste trabalho, é apresentado o estudo e uma abordagem de paralelização em GPGPU, utilizando a arquitetura CUDA, do método de suavização de imagem baseado num modelo variacional, proposto por Jin e Yang (2011), e sua aplicação em imagens com altas resoluções. Os resultados obtidos nos experimentos, permitiram obter um *speedup* de até quinze vezes no tempo de processamento de imagens, comparando o algoritmo sequencial e o algoritmo otimizado paralelizado em CUDA, o que pode viabilizar sua utilização em diversas aplicações de tempo real.

Palavras–chave

General Purpose Computing on Graphics Processing Units. Compute Unified Device Architecture. Processamento de Imagem. Suavização. Ruído Multiplicativo.

Abstract

Supported by processors evolution, high performance computing have contributed to development in several scientific research areas which require advanced computations, such as image processing, augmented reality, and others. To fully exploit high performance computing available in these resources and to decrease processing time, is necessary apply parallel computing. However, those resources are expensive, which implies the search for alternatives ways to use it. The multicore processors architecture and *General Purpose Computing on Graphics Processing Unit* (GPGPU) become a low cost options, as they were designed to provide infrastructure for high performance computing and attend real-time applications.With the improvements gained in technologies related to multicomputer, multiprocessor and, more recently, to GPGPUs, the parallelization of computational image processing techniques has gained extraordinary prominence. This parallelization is crucial for the use of such techniques in applications that have strong demands in terms of processing time, so that even more complex computational algorithms can be used, as well as their use on images of higher resolution. In this research, the parallelization in GPGPU of a recent image smoothing method based on a variation model is described and discussed. This method was proposed by Jin and Yang (2011) and is in-demand due to its computation time, and its use with high resolution images. The results obtained are very promising, revealing a speedup about fifteen times in terms of computational speed.

Keywords

General Purpose Computing on Graphics Processing Units. Compute Unified Device Architecture. Image Processing. Smoothing, Multiplicative Noise.

Lista de Figuras

Lista de Tabelas

Lista de Abreviaturas

ALU	*Arithmetic and Logic Unit*
API	*Application Programming Interface*
CPU	*Central Processing Unit*
CUDA	*Compute Unified Device Architecture*
GDRAM	*Graphics Dynamic Random-Access Memory*
GPGPU	*General Purpose Computing on Graphics Processing Unit*
GPU	*Graphics Processing Unit*
IBM	*International Business Machines Corporation*
IEEE	*Institute of Electrical and Electronics Engineers*
ITK	*Insight Segmentation and Registration Toolkit*
LD/ST	*load/store*
MFLOPS	*Millions of Floating Point Operations Per Second*
MIMD	*Multiple Instruction, Multiple Data Stream*
MIPS	*Million of Instructions per Second*
MISD	*Multiple Instruction, Single Data Stream*
MPI	*Message Passing Interface*
MSE	*Mean Squared Error*
MSSI	*Mean Structure Similitary Index*
NoC	*Network-on-Chip*
OpenCL	*Open Computing Language*
OpenGL	*Open Graphics Library*
PC	*Personal Computer*
PSNR	*Peak Signal-to-Noise Ratio*
RGB	*Red, Green and Blue*

SIMD	*Single Instruction, Multiple Data Stream*
SIMT	*Single Instruction, Multiple Thread*
SISD	*Single Instruction, Single Data Stream*
SM	*Stream Multiprocessor*
SMP	*Symmetric Multiprocessor*
SP	*Streaming Processors*
SPM	*Stream Programming Model*
SPMD	*Single Program Multiple Data Stream*
SSIM	*Structutal SIMilarity*
SFU	*Special Function Units*
UP	Unidade de Processamento
2D	bidimensional
3D	tridimensional

Sumário

CAPÍTULO 1

Introdução

Ao longo da história, o avanço da capacidade de processamento ficou marcado pelo aumento na velocidade de cálculos dos processadores. Em geral a indústria de processadores aumenta a velocidade de cálculos inserindo um maior número de transistores nos mesmos (KIRK; HWU, 2010). Esta abordagem tem enfrentado dificuldades devido às limitações físicas do silício, principalmente, pelo excesso de consumo de energia e aquecimento destes processadores.

Nos últimos anos, no entanto, os avanços nesta área seguiram outra direção, sendo populares atualmente as arquiteturas *multicore* de uso geral e as *General Purpose Computing on Graphics Processing Unit* (GPGPU). A indústria de processadores passou a oferecer *chips* com vários *cores* compondo o processador. Estes processadores baseados na arquitetura *multicore* de uso geral, oferecem recursos de paralelismo, proporcionando ganho de desempenho e processamento mais rápido. As GPGPUs possuem características para realizar processamento de alto desempenho, contendo dezenas, centenas e até milhares de unidades de processamento em um mesmo *chip*. Além do baixo custo financeiro, esta tecnologia está disponível para ser utilizada em computadores pessoais (VADJA, 2011; CHAPMAN et al., 2010).

A demanda por processamento de alto desempenho, em geral, tem sido atendida por equipamentos ou sistemas computacionais de custo elevado. Diante da popularização das GPGPUs e a aplicação de técnicas consolidadas de programação paralela, diversas áreas de pesquisa como a computação científica (KURZAK; BADER; DONGARRA, 2010), o processamento de imagens médicas (CHEN; ZHANG; XIONG, 2010), de satélite (KRISSIAN et al., 2005), imagens adquiridas e usadas na navegação de robôs (WURM et al., 2010), realidade aumentada (SCHONUNG, 2011), e muitas outras, podem conquistar avanços ainda mais significativos, sem a necessidade de grandes investimentos financeiros. Muitas destas aplicações podem utilizar imagens com alta resolução, ou necessitam ser processadas em tempo real para tomada de decisões como, por exemplo, em navegação robótica ou ainda porque estão envolvidos conjuntos com elevado número de imagens que devem ser processados de forma rápida, como em ultrassonografia (HWU, 2011). Assim, o uso de computação paralela para o processamento de imagem tem sido cada vez mais abordado, no sentido de obter métodos computacionais robustos, eficientes e de execução rápida.

O processamento de imagem tem se tornado cada vez mais uma área de pesquisa consolidada e com aplicação em diversos domínios do conhecimento. A obtenção de imagem, na maioria das vezes, inclui ruído e desta maneira, dificulta ou até mesmo inviabiliza a identificação de determinadas informações de interesse (GONZALEZ; WOODS, 2001). Desta maneira, torna-se necessária a realização de uma etapa de pré-processamento das imagens, conhecida como "suavização" ou "filtragem".

O problema torna-se mais complexo quando estas imagens possuem altas resoluções, ou ainda, quando exigem o processamento em tempo real. Satélites ou radares, por exemplo, capturam imagens com ruído em alta resolução, desta maneira, uma das alternativas é a suavização de imagem. Vários métodos de suavização de imagem podem ser encontrados na literatura especializada (GONZALEZ; WOODS, 2001; YANG et al., 2009; HUANG; NG; WEN, 2009; JIN; YANG, 2011; DASH; SA; MAJHI, 2011), e podem ser aplicados aos diferentes tipos de ruídos. O procedimento de suavizar imagem com alta resolução pode ser muito "caro" computacionalmente, por isso, os métodos desenvolvidos têm sido paralelizados a fim de obter seus resultados em menor tempo de processamento, inclusive atender demandas de processamento em tempo real (HWU, 2011).

1.1 Motivação

A área de processamento de imagem tem contribuído com vários setores da indústria e da ciência. Em aplicações médicas, por exemplo, permite a visualização de órgãos por meio de exames com imagens. A captura das imagens pode ser realizada por meio de exames de tomografia computadorizada, ressonância magnética e ultrassonografia. No entanto, para obter imagens com a menor perda de informação possível, uma das etapas iniciais requer o pré-processamento, no intuito de filtrar os ruídos incidentes.

Este ruído modifica os valores de intensidade de cada ponto da imagem de entrada, dificultando a utilização de técnicas computacionais automatizadas de extração de informações. Considerando este contexto, o método de suavização de imagem baseado em um modelo variacional, desenvolvido por Jin e Yang (2011), apresenta bons resultados quanto à remoção do ruído. No entanto, ao ser aplicado em imagem com alta resolução e, especialmente, aplicações que exigem processamento em tempo real, seu desempenho, com relação à implementações sequenciais, não é adequado.

Como a incidência deste tipo de ruído também é comum em radares de abertura sintética, a implementação deste e de outros modelos de suavização paralelizados em *Compute Unified Device Architecture* (CUDA), poderão contribuir consideravelmente no desenvolvimento da área de sensoriamento remoto, aplicações civis e militares, mapeamento e monitoração do

uso do solo, acompanhamento e discriminação de culturas, cartografia, planejamento urbano e rural, dentre outros (KRISSIAN et al., 2005; YI, 1999; HWU, 2011).

1.2 Objetivos

O objetivo desta pesquisa é reduzir o tempo de processamento da técnica de suavização de imagem baseada em modelo variacional, utilizando as principais técnicas de programação massivamente paralela em arquitetura CUDA. A técnica de suavização adotada é recente e promissora, foi desenvolvida por Jin e Yang (2011) a fim de remover ruído multiplicativo, comuns em imagens de satélites, radares, ultrassonografias, dentre outros. No entanto, requer um alto custo computacional para o processamento de imagem com altas resoluções. Outros objetivos deste trabalho são:

- identificar, de maneira geral, as principais técnicas de suavização de imagem baseadas em ruído multiplicativo e escolher a mais adequada para a paralelização;
- estruturar e implementar a paralelização do método de suavização adotado utilizando a arquitetura massivamente paralela, denominada CUDA; e
- avaliar a solução paralelizada quanto a capacidade de filtragem proposta no método original, bem como seu desempenho em comparação à execução paralelizada e sequencial.

1.3 Estrutura da Dissertação

Esta dissertação está organizada em seis capítulos, incluindo o presente capítulo de introdução:

- **Capítulo 2 - Arquiteturas Paralelas**, trata conceitos e terminologias da área de computação paralela e aspectos de evolução histórica, sobre arquiteturas GPGPUs e *multicores*. Descreve, de maneira introdutória, aspectos da computação de alto desempenho baseado na taxonomia de Flynn, métrica para avaliação de desempenho, bem como, apresenta o modelo de comunicação utilizado na arquitetura GPGPU;
- **Capítulo 3 - Técnicas de Computação Paralela**, apresenta os métodos de paralelização de maneira geral, descreve a necessidade de decompor problemas em tarefas menores e também, apresenta o funcionamento dos mecanismos de sincronização, utilizados na comunicação entre as unidades de processamento. O capítulo concentra-se em introduzir os Métodos de Paralelização em GPGPU, além de apresentar a arquitetura CUDA e uma breve comparação entre as linguagens de programação paralela: CUDA C/C++, OpenMP e MPI;

- **Capítulo 4 - Suavização de Imagem**, inicialmente apresenta conceitos relacionados a etapa de suavização de imagens, descreve as principais técnicas desta etapa do processamento de imagem. Em seguida, destaca os métodos mais comuns para suavização de imagem afetada por ruído multiplicativo, e também apresenta uma solução utilizando modelos variacionais;
- **Capítulo 5 - Trabalhos Relacionados**, neste capítulo são apresentados alguns métodos de suavização de imagens paralelizados, especialmente, em arquitetura GPGPU;
- **Capítulo 6 - Paralelização do Método**, descreve a implementação da paralelização em GPGPU. Também são fornecidos detalhes técnicos de implementação sobre a arquitetura CUDA e a linguagem CUDA C;
- **Capítulo 7 - Experimentos e Análise dos Resultados**, descreve as métricas de desempenho e avaliação de imagens, o cenário utilizado como infraestrutura para os experimentos, bem como realiza a análise dos resultados obtidos; e
- **Capítulo 8 - Conclusões e Trabalhos Futuros**, elenca as principais contribuições da presente pesquisa e aponta possíveis direcionamentos para o desenvolvimento de trabalhos futuros.

CAPÍTULO 2

Arquiteturas Paralelas

A maior demanda por desempenho computacional, em geral, é na realização de cálculos, a fim de atender aplicações científicas, médicas, militares, entretenimento, dentre outras. Na maioria das vezes, estas aplicações exigem o desenvolvimento de técnicas para processamento de alto desempenho. O presente capítulo apresenta conceitos e terminologias inerentes à computação paralela, além de um breve histórico sobre a evolução das arquiteturas paralelas.

2.1 Considerações Iniciais

A necessidade de processamento mais veloz é cada vez mais exigida em diversas áreas de pesquisa e da indústria em geral. A resolução de problemas em áreas como a computação científica, a computação gráfica, o processamento de imagem, a compressão de vídeo, manipulação de enormes quantidade de dados, previsão do tempo, dentre muitos outros, requer alto poder de processamento (PAGE, 2009).

A computação paralela tem sido utilizada para lidar com problemas desta magnitude. Por meio de técnicas de paralelização, utilizadas em processamento de alto desempenho, as aplicações são reescritas com objetivo de decompor o algoritmo ou os dados em partes menores. Estas partes menores, também chamadas de tarefas, são distribuídas para serem executadas em processadores ou núcleos, simultaneamente (GEBALI, 2011; DUNCAN, 1990; GURD, 1988; HWANG, 1987).

Durante estas etapas realiza-se o processo de comunicação e coordenação geral das tarefas entre os processadores envolvidos no processamento. Ao utilizar a programação paralela, dois fatores devem ser levados em consideração: o tipo de arquitetura paralela e o tipo de comunicação dos processadores. Dessa maneira, apresenta-se uma taxonomia baseada no fluxo de instruções e fluxo de dados de uma aplicação, conhecida como taxonomia de Flynn (STALLINGS, 2003; EL-REWINI; ABD-EL-BARR, 2005; PAGE, 2009). Esta classificação amplamente utilizada, é ilustrada na Figura 2.1:

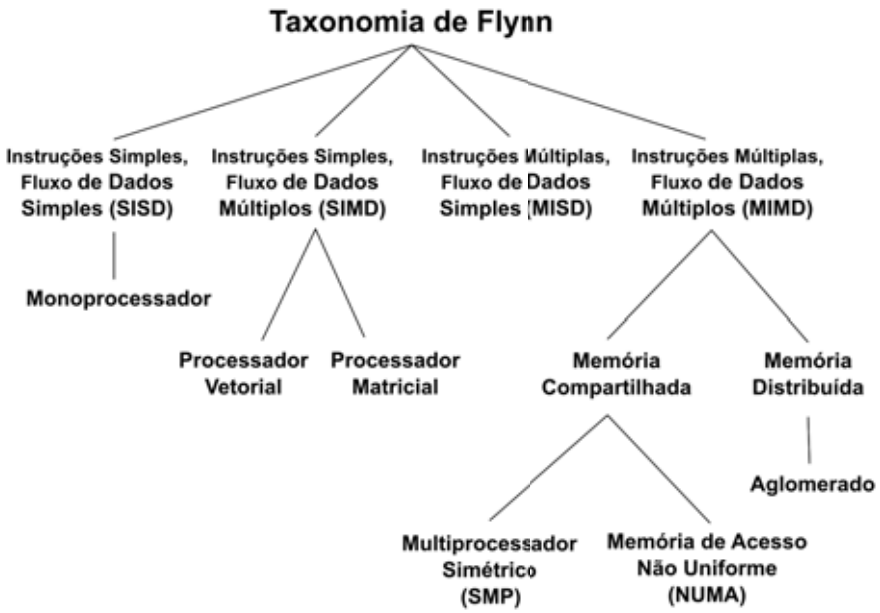
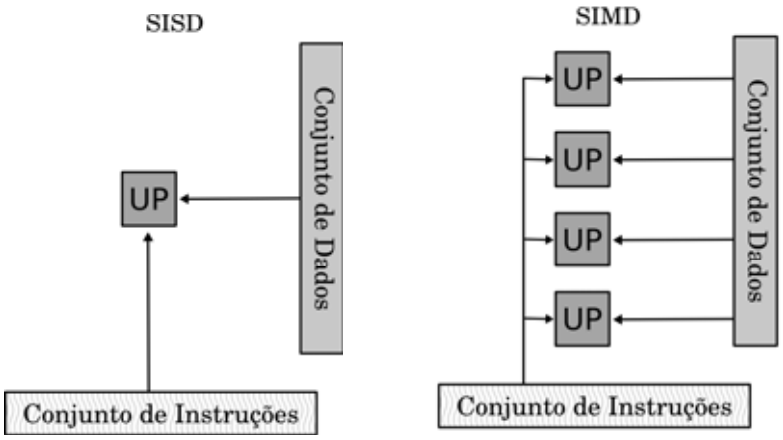


Figura 2.1: *Taxonomia de Flynn.*

Fonte: Traduzida e adaptada de Stallings (2003).

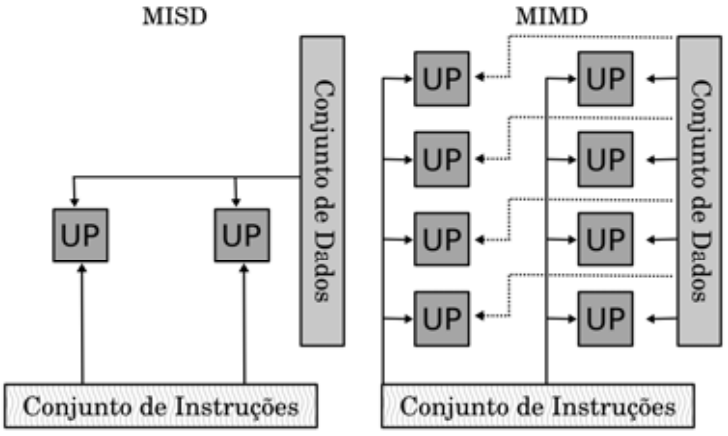
- *Single Instruction, Single Data Stream* (SISD): este modelo, representado na Figura 2.2(a), utiliza uma Unidade de Processamento (UP) para executar uma instrução por vez. Nos computadores e *mainframes* antigos, apesar de alguns possuírem múltiplas unidades e processamento, os dados e os fluxos de instruções não tinham qualquer relação ou dependência. Em outras palavras, é um computador que realiza processamento sequencial, baseado no modelo de von Neumann ou ainda, na máquina universal de Turing (KIRK; HWU, 2010). O desempenho dos computadores que utilizam esta arquitetura, geralmente, é medido em *Million of Instructions per Second* (MIPS), e o número de instruções enviadas para processamento é limitado no tempo (PAGE, 2009; GEBALI, 2011);
- *Single Instruction, Multiple Data Stream* (SIMD): supercomputadores vetoriais são baseados no modelo SIMD e permitem a paralelização da instrução, otimizando o tempo de processamento de um conjunto diferente de dados, conforme ilustrado na Figura 2.2(b). O desempenho é medido em *Millions of Floating Point Operations Per Second* (MFLOPS);



(a) *SISD* (b) *SIMD*

Figura 2.2: *Modelo de arquiteturas para fluxos de dados múltiplos.*
Fonte: Traduzida e adaptada de Schmeisser et al. (2009)

- *Multiple Instruction, Single Data Stream* (MISD): conforme apresentado na Figura 2.3(a), pode ser aplicado na resolução de problemas específicos, para garantir o processamento de cálculos matemáticos sólidos, sendo que os dados e as instruções são executados em unidades de processamento, necessariamente, diferentes (PAGE, 2009). Como exemplo conceitual pode-se citar, o processamento de vários algoritmos de criptografia tentando quebrar uma única mensagem codificada; e
- *Multiple Instruction, Multiple Data Stream* (MIMD): neste modelo apresentado na Figura 2.3(b) é possível realizar o paralelismo de instruções independentes para cada processador, com a vantagem de executar múltiplas tarefas simultaneamente. No entanto, há a necessidade de sincronizar em tempo real a execução das tarefas paralelas entre os processadores no final de uma aplicação simples (PAGE, 2009).



(a) *MISD* (b) *MIMD*

Figura 2.3: *Modelo de arquiteturas para fluxos de instruções múltiplas.*
Fonte: Traduzida e adaptada de Schmeisser et al. (2009)

O sincronismo entre os processadores é uma forma de comunicação, que pode ser obtida através do barramento de memória compartilhada ou em barramento de rede local. A maioria dos computadores modernos enquadram-se nesta categoria, os quais utilizam a programação paralela (PARHAMI, 2002).

A comunicação realizada entre os processadores, de acordo com o processamento por evento de comunicação, é classificada em "granularidade". Granularidade fina, para pequenas quantidades de tarefas executadas ou sincronizadas com menor frequência; e grossa, para grandes quantidades de tarefas executadas ou sincronizadas com maior frequência (KIRK; HWU, 2010; PARHAMI, 2002).

O modelo MIMD divide-se em multiprocessamento fortemente e fracamente acoplados. Na presente pesquisa foram levados em consideração apenas os sistemas fortemente acoplados, sendo esta a categoria para a arquitetura de GPGPU. Utilizar mais processadores, paralelizando a resolução dos problemas já citados, é uma prática que tem conquistado bons resultados. O desempenho da paralelização de um algoritmo pode ser medido através da eficiência e do *speedup* (JANG; PARK; JUNG, 2008; MERIGOT; PETROSINO, 2008).

O *speedup* é o ganho de desempenho na execução, em número de vezes, de um processo em n processadores em relação ao tempo de execução do processo, T_p, correspondente sequencialmente, em 1 processador, calculado pela Eq.2-1:

$$S(n) = \frac{T_p(1)}{T_p(n)} \tag{2-1}$$

A eficiência do algoritmo paralelizado pode ser calculado pela relação entre o *speedup* e o número de processadores, efetivamente utilizados. Esta medida indica quando um processador esté sendo subutilizado, quando o valor resultante for abaixo de 1, ou se o problema precisa ser decomposto de uma maneira mais eficiente para utilizar mais processadores, para um valor resultante acima de 1. Segundo a Lei de Amdahl e como ilustrado na Figura 2.4, aumentar arbitrariamente o número de processadores em um sistema paralelo não garante o aumento constante do *speedup* (EL-REWINI; ABD-EL-BARR, 2005; PARHAMI, 2002).

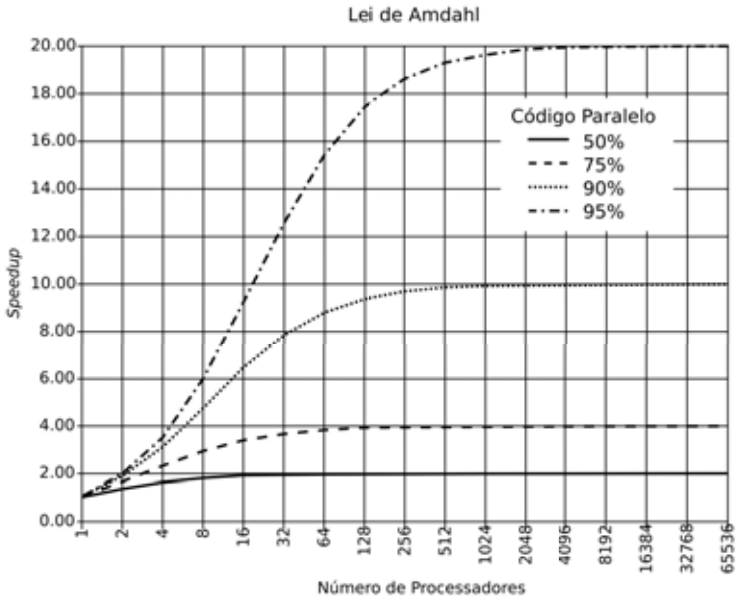


Figura 2.4: *Lei de Amdahl.*
Fonte: Traduzida e adaptada de Kirk e Hwu (2010)

Ao aumento arbitrário no número de processadores atribui-se o termo Escalabilidade, definido para representar o crescimento proporcional da capacidade de processamento paralelo, com a inclusão de novos processadores no sistema (EL-REWINI; ABD-EL-BARR, 2005).

Baseado na Lei de Amdahl (HILL; MARTY, 2008), de acordo com a porção de código otimizada f por um *speedup* S, o ganho de desempenho máximo é definido pela Eq. 2-2:

$$Speedup_{enhanced}(f,S) = \frac{1}{(1-f)+\frac{f}{S}} \tag{2-2}$$

Em busca do oferecimento de maior desempenho, sistemas com *multicores* e multiprocessadores, tem sido produzidos pela indústria de processadores. Na Figura 2.5, são apresentadas diferenças estruturais que influenciam, não só no desempenho, mas também no consumo de energia (NVIDIA, 2010).

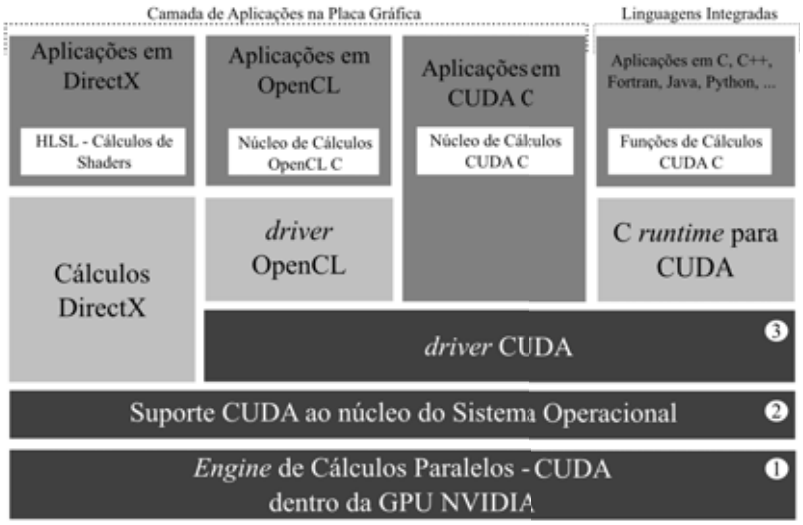


Figura 2.5: *Diferenças entre: CPU e GPGPU.*
Fonte: Traduzida e adaptada de Kirk e Hwu (2010)

A *Central Processing Unit* (CPU) utiliza grande parte de seus transistores para controle e cache, enquanto a GPGPU utiliza para cálculos. Os sistemas *multicores* possuem um bom desempenho e pouco consumo de energia, apesar dos *cores* serem projetados para baixo desempenho. Seus processadores são interconectados por um único barramento ou um *Network-on-Chip* (NoC) (KIRK; HWU, 2010; PAGE, 2009).

No caso dos multiprocessadores, os *chips* são separados e interconectados por barramento, permitindo que cada *chip* multiprocessado seja um *chip multicore*. Os processadores que compõem estes *chips* são de alto desempenho, com isso, influencia diretamente em seu poder computacional, entretanto, possuem o consumo de energia mais elevado em relação ao sistema *multicore* (GEBALI, 2011).

As GPGPUs são baseadas no modelo conhecido como *Single Program Multiple Data Stream* (SPMD) e permitem manipular um número muito grande de *threads* ao mesmo tempo (PAGE, 2009; GEBALI, 2011). *Threads* são fluxos de instruções que permitem compartilhar espaço de endereçamento, temporizadores, arquivos, e podem ser executados simultâneamente, ou ainda, concorrentemente (HENNESSY; PATTERSON, 2007; STALLINGS, 2003; TANENBAUM, 2010).

A GPGPU, escopo da presente pesquisa, pode ser classificada em um modelo do tipo SIMD e MIMD, em razão de ser capaz de executar conjuntos de instruções funcionais em um fluxo de dados por meio do *kernel*, conhecido como Modelo de Programação em Fluxo ou *Stream Programming Model* (SPM). *Kernel*, neste contexto, são funções aplicadas em cada elemento do conjunto de registros que exigem processamento similiar, definidos como fluxo ou *stream* (NVIDIA, 2010).

As GPGPUs, por meio do *Stream Multiprocessor* (SM), vêm sendo cada vez mais aplicadas à computação de propósito geral e intensivo, tornando-se desta maneira verdadeiras GPGPUs. As principais características das GPGPUs descritas por Gebali (2011), tais como intensidade de cálculos, paralelismo de dados e localidade temporal e espacial, são apresentadas com detalhes no Capítulo 4. As GPGPUs passaram por um processo de evolução até atingir a atual capacidade de processamento. Esta evolução é apresentada na próxima seção.

2.2 Histórico dos *Multicores* e GPUs

A partir do surgimento dos primeiros computadores pessoais na década de 80, denominados PC (*Personal Computer*), a indústria de computadores tem aumentado o desempenho destes equipamentos, principalmente, incrementando a velocidade da CPU.

A velocidade de processamento dos primeiros PCs era na frequência de 1MHz, já atualmente, é cerca de 1000 (mil) vezes mais rápido, alcançando *clocks* entre 1GHz e 4GHz. O crescimento na velocidade de processamento, conhecido desde 1965 como a Lei de Moore (PARHAMI, 2002), enfrenta dificuldades como limitações físicas no tamanho de transistores, consumo de energia e dissipação de calor.

Uma das alternativas encontradas pela indústria de processadores, a partir de 2005, foi oferecer soluções com o processamento de várias CPUs em um mesmo *chip*, conhecidos como *multicores* (KIRK; HWU, 2010). Juntamente com os *multicores*, o conceito de computação paralela tem sido disseminado ao consumidor doméstico como a solução para realizar várias tarefas simultaneamente, sem perder o desempenho (SANDERS; KANDROT, 2011). De qualquer maneira, é importante salientar a necessidade de reformulação do código da aplicação para obter aproveitamento da arquitetura *multicore*.

As primeiras GPUs (*Graphics Processing Unit*) foram lançadas, quase que juntamente, com os primeiros PCs. Entre as décadas de 80 e 90, diferentes fabricantes de placas gráficas surgiram no mercado. A primeira geração de processador gráfico, ainda nos anos 80, era capaz de mostrar textos e gráficos, tinha a capacidade de armazenamento em memória de apenas 16KBytes, além de representar cerca de 16 cores. Neste período todo o processamento era realizado diretamente na CPU.

Com o avanço dos sistemas operacionais no começo da década de 90, apresentando interfaces gráficas em 2D, surgiu uma demanda para um novo mercado de processadores gráficos. Neste contexto, a Silicon Graphics criou a biblioteca *Open Graphics Library* (OpenGL) para permitir interface com o *hardware* de aceleração gráfica. Durante este período, começaram a ser desenvolvidas aplicações gráficas em 3D, especialmente jogos como Doom, Duke Nukem 3D e Quake (BORGO; BRODLIE, 2009).

Outras empresas iniciaram suas atividades na produção de placas gráficas aceleradoras para aplicações 3D, como por exemplo a *International Business Machines Corporation* (IBM), primeira fabricante a oferecer aceleração gráfica bidimensional (2D) e tridimensional (3D), a S3, a NVIDIA a qual se tornou uma das principais fabricantes do setor e disputa a liderança de mercado com a ATI Technologies, 3dfx Interactive, dentre outras, contribuindo para a consolidação destas aplicações.

O termo GPU é empregado para o primeiro *chip* com funções integradas de realizar transformação, iluminação, recorte a ajuste de triângulos, texturização e renderização de imagens. A única maneira do programador interagir com a GPU era por meio das bibliotecas programáveis como OpenGL, desenvolvida pela Silicon Graphics, e DirectX, desenvolvida pela Microsoft.

Com o propósito de permitir maior controle de desenvolvedores sobre o processamento realizado na GPU, a NVIDIA incorporou o DirectX 8.0 em seus *chips* (SANDERS; KANDROT, 2011; BORGO; BRODLIE, 2009). A segunda geração de GPU oferecia funções de transformação geométrica, aplicação de iluminação, e principalmente o processamento paralelo, inovado com a empresa 3dfx.

Todas estas funções eram realizadas diretamente no *hardware* da GPU, reduzindo a carga de processamento da CPU (BORGO; BRODLIE, 2009). Continuando o processo de evolução das GPUs, surge o suporte para transformação de informações enviadas pela CPU, chamada de *vertex shaders*. Com isso, operações como transformações em vértices, iluminação, geração de coordenadas de textura, dentre outras, poderiam ser modificadas, marcando assim a terceira geração das GPUs.

A quarta geração é complementada com o recurso de *fragment shader*, o qual permite criar efeitos e transformações diretamente nos pixels antes de serem desenhados na tela do monitor. O *fragment shader*, também conhecido como *pixel shader*, atua em segmentos rasterizados permitindo o cálculo de cores, acesso e aplicação de texturas, cálculo de coordenadas de textura por pixel, além de outros efeitos. Pesquisadores passaram a realizar operações utilizando a unidade de processamento gráfico por meio da *Application Programming Interface* (API), conhecida como *shading language* (JANG; PARK; JUNG, 2008).

Conhecer bem a linguagem *shading*, dominar computação gráfica, além da dificuldade de manipulação de dados em ponto flutuante e monitorar se os cálculos eram finalizados sem falhas, foram algumas das principais limitações de uso destes processadores para uso geral (SANDERS; KANDROT, 2011).

Diante das limitações, a NVIDIA desenvolveu a *Compute Unified Device Architecture* (CUDA), seguindo padrões de requisitos do *Institute of Electrical and Electronics Engineers* (IEEE) na construção da *Arithmetic and Logic Unit* (ALU). Dessa maneira, a unidade

aritmética de ponto flutuante de precisão única foi projetada para utilizar um conjunto de instruções adequado para uso geral e não apenas processamento gráfico (SANDERS; KANDROT, 2011).

A Figura 2.6, ilustra a arquitetura modular CUDA, sendo que no módulo 1 a GPGPU com suporte CUDA, no módulo 2, o suporte de inicialização e configuração CUDA para o sistema operacional, e módulo 3 o *driver* CUDA.

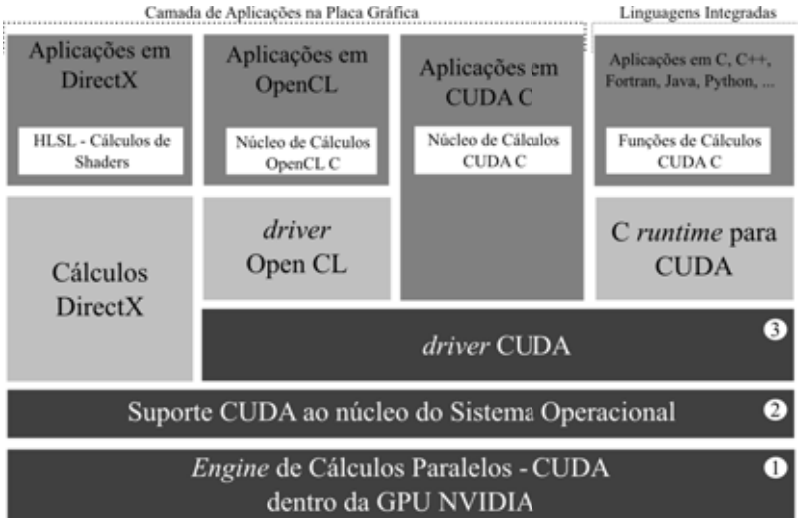


Figura 2.6: *Arquitetura CUDA.*

Fonte: Traduzida e adaptada de NVIDIA (2010)

O desenvolvimento de aplicações para executar nestes processadores gráficos, apesar de utilizar a arquitetura CUDA, exige conhecimentos avançados em programação OpenGL ou DirectX. No intuito de estender o recurso para um número maior de desenvolvedores, a NVIDIA incorporou uma pequena quantidade de *keywords* na linguagem C padrão adicionando funcionalidades específicas da arquitetura CUDA, criando desta maneira um compilador para esta linguagem, denominado CUDA C (SANDERS; KANDROT, 2011; KIRK; HWU, 2010; BORGO; BRODLIE, 2009).

2.3 Considerações Finais

Os computadores multiprocessados ou *multicores*, na maioria das vezes não executam aplicações completamente paralelas em todas as suas unidades de processamento, em razão de implementações sequenciais destas aplicações. Há décadas grandes empresas e instituições de pesquisas utilizam computadores de grande porte paralelizando aplicações que exigem alto desempenho computacional. No entanto, a aquisição deste tipo de equipamento exigia investimento de alto custo.

Desta maneira, com a popularização dos computadores *multicores* e a possibilidade de melhorar o desempenho das aplicações, o desenvolvimento de aplicações paralelas têm aumentado significativamente (KIRK; HWU, 2010). Assim, as GPGPUs vem sendo utilizadas em diversas pesquisas, e em alguns casos como infraestrutura para computação de alto desempenho. No próximo capítulo serão apresentas as técnicas de programação paralela mais significativas destinadas às GPGPUs.

CAPÍTULO 3

Técnicas de Computação Paralela

Diversas áreas da Ciência utilizam a computação de alto desempenho para resolver problemas cada vez mais complexos. Muitos destes problemas envolvem cálculos que exigem alta precisão e rapidez no processamento dos dados, o que pode ser alcançado por meio da paralelização dos algoritmos e execução em vários processadores. Neste capítulo são apresentadas algumas das técnicas e bibliotecas de programação paralela mais significativas para esta pesquisa, enfatizando a arquitetura CUDA.

3.1 Considerações Iniciais

A computação paralela, definida na Seção 2.1, tem como objetivo minimizar a carga de operações por processador e maximizar o *speedup* do processamento como um todo.

O processo de paralelização de um algoritmo, inicia-se com a decomposição de problemas maiores em tarefas. Em seguida as tarefas são distribuídas entre os processadores exigindo do programador, dentre outras ações, codificar a aplicação para garantir independência de dados e realizar o controle de sincronização ou de comunicação entre as UP, em busca de maior eficiência na paralelização das tarefas (PARHAMI, 2002).

3.1.1 Decomposição

De acordo com Vadja (2011), a técnica de decomposição pode ser classificada em funcional, baseada em dados e baseada em tarefas. Na decomposição funcional, é possível uma UP continuar realizando suas operações sem a necessidade de aguardar o resultado de processamento em outras UPs. Ao utilizar esta técnica, a aplicação pode ser implementada determinando, especificamente, as partes do código a serem paralelizadas.

Dessa maneira, a decomposição funcional pode ser subdivida em estática e dinâmica. Na paralelização de código, quando é definido um número específico de UP, a técnica é conhecida como decomposição funcional estática, sendo sua principal característica a falta de escalabilidade. No caso da decomposição funcional dinâmica, a aplicação é independente da

quantidade de processadores, permitindo escalabilidade e *speedup* ao sistema computacional (VADJA, 2011).

Para a decomposição baseada em dados, a paralelização em tempo de execução, inicia-se com a distribuição de processos, ou *threads*, entre as UPs disponíveis para o processamento de conjuntos de dados diferentes e não dependentes (GURD, 1988). No modelo *pipeline*, os dados são decompostos em etapas diferentes e cada UP realiza o processamento especializado destas etapas. No entanto, pode ser que algumas das UP executem o processamento da mesma etapa. Na Figura 3.1, as etapas são representadas como *Estágio* e as UPs, como *núcleos*.

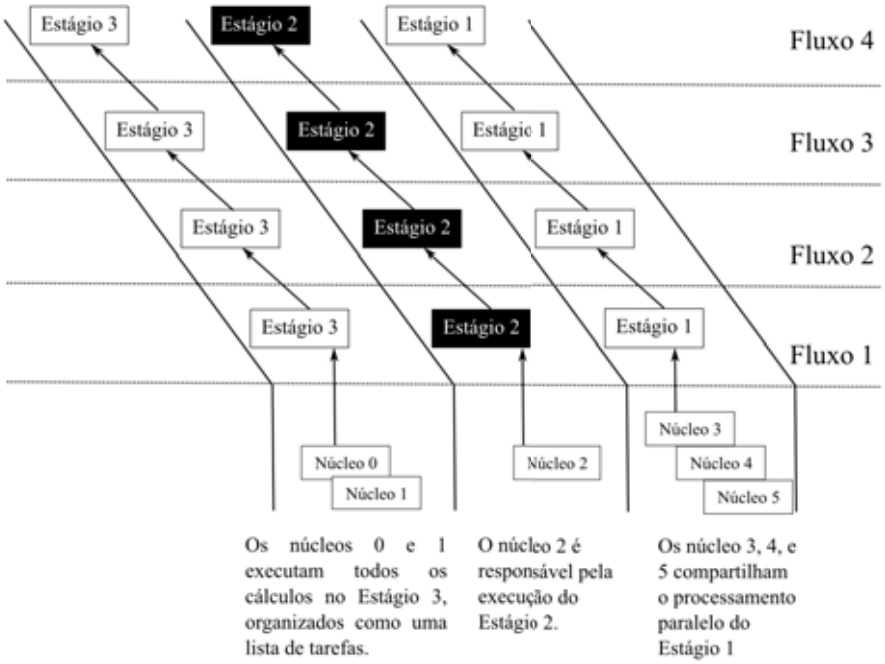


Figura 3.1: *Paralelização utilizando modelo pipeline.*
Fonte: Traduzida e adaptada de Vadja (2011)

As categorias do modelo apresentado anteriormente, podem ser inadequadas para problemas que exigem dependência de dados e processos. Para permitir o paralelismo eficiente nesta categoria de problemas, Vadja (2011) considera promissora a utilização de técnicas de previsão na busca e processamento de dados.

A implementação de técnicas de decomposição de tarefas, deve levar em consideração o funcionamento básico de processos e *threads*, em nível de sistema operacional. Conceitualmente, processos podem conter *threads*, desta maneira, caracteriza-se a dependência hierárquica entre eles. Os *threads* compartilham recursos como espaço de endereçamento, variáveis globais, arquivos, dentre outros. Por outro lado, os processos são completamente isolados, sendo considerados pelo sistema operacional como unidades de isolamento permitindo gerenciamento de falhas.

O sistema operacional gerencia o acesso aos recursos computacionais e utiliza o conceito

de prioridade, baseado na associação entre processos e *threads*, como pode ser observado na Figura 3.2.

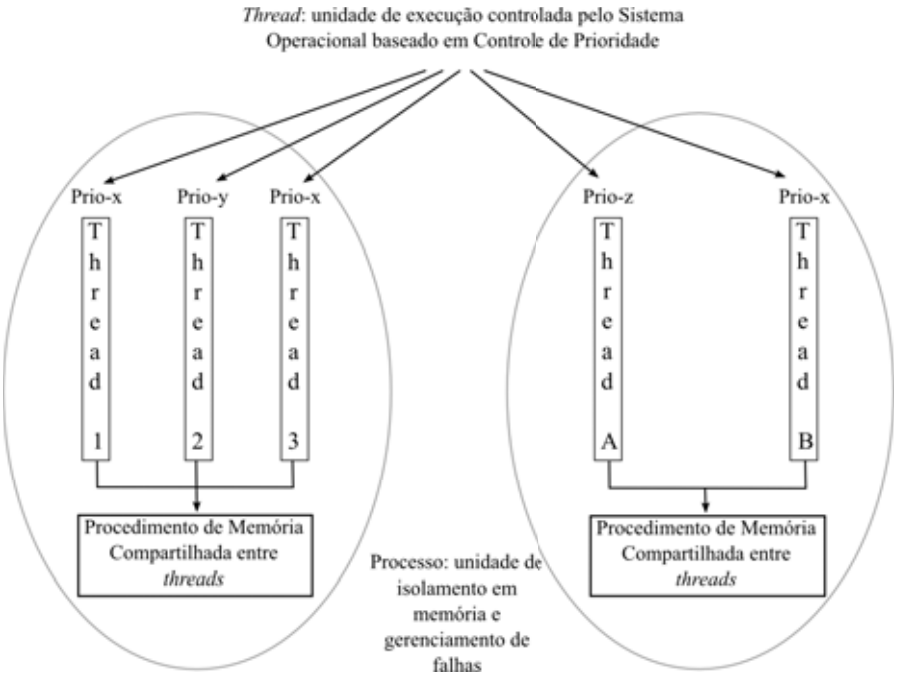


Figura 3.2: *Conceitos de processos, threads e prioridades.*
Fonte: Traduzida e adaptada de Vadja (2011)

3.1.2 Sincronização

Aplicações paralelas, em geral, exigem interações na troca de informações, acesso aos mesmos dados, espera no término de processamento, dentre outras formas. O uso destas interações é facilitado por meio de mecanismos de sincronização como *locks*, semáforos, regiões críticas, dentre outros.

Durante a execução de uma aplicação paralela, uma UP pode precisar de resultados em processamento noutra UP. Esta dependência de dados, ou dependência funcional, exige que a aplicação aguarde o processamento das UPs envolvidas e então, posteriormente, realize o sincronismo para concluir a tarefa. Em outra situação, partes da aplicação paralela podem realizar acessos frequentes a recursos do sistema, como *hardware*, acesso de memória compartilhada, dentre outros. Em ambos os casos, a decomposição exige a necessidade de sincronizar as diferentes tarefas paralelizadas.

3.2 Métodos de Paralelização em GPGPU

A maneira como pode ser realizada a comunicação entre as UPs pode ser dividida em sistemas fortemente e fracamente acoplados. A ênfase nesta pesquisa, como mencionado anteriormente, é em sistemas fortemente acoplados. Dessa maneira, o *Symmetric Multiprocessor* (SMP), consiste em um conjunto de processadores idênticos que podem executar tarefas em cooperação para resolver problemas científicos utilizando o compartilhamento de memória (HENNESSY; PATTERSON, 2007).

Sistemas computacionais compostos por SMP, também são conhecidos como computação distribuída, grade computacional ou computação massivamente paralela. As UPs envolvidas na execução das tarefas comunicam-se para trocar informações acerca do processamento, além de realizar operações, como por exemplo, atualização dos dados armazenados em memória.

As técnicas de paralelização mais comuns aos tipos de arquiteturas *multicores* e GPGPU (NVIDIA, 2010; KIRK; HWU, 2010), são: paralelismo de tarefas, o qual realiza a execução simultânea de múltiplas tarefas, que podem ser *threads* ou processos, em dados diferentes. O paralelismo de dados, onde múltiplos dados recebem a execução de uma tarefa e o paralelismo de instruções, quando há a execução de múltiplas instruções por processador.

A aplicação destas técnicas de paralelização exige o suporte específico do *hardware*. Kirk e Hwu (2010), levantaram as diferenças entre os projetos da CPU e GPGPU. O projeto da CPU é otimizado para desempenho de código sequencial, utilizando sofisticadas lógicas de controle para permitir que instruções de uma única *thread* executem em paralelo; a velocidade de acesso à memória não aumenta muito em razão da utilização do processador para propósito geral, satisfazendo dispositivos de entrada e saída, sistemas operacionais e aplicativos.

O projeto da GPGPU prioriza a realização de uma enorme quantidade de cálculos de pontos flutuantes, disponibilizando pequenas unidades de memória cache para facilitar o controle de requisitos de largura de banda. Permitem ainda, que múltiplos *threads* acessem os mesmos dados na memória, dispensando a necessidade de recuperação constante de dados na memória principal, sendo que a velocidade de acesso à memória é, em geral, 10 vezes mais rápida em processadores gráficos (KIRK; HWU, 2010).

Diante das diferenças de *hardware* apresentadas anteriormente, a Figura 3.3 ilustra a evolução na capacidade de cálculos de pontos flutuantes entre CPUs e GPGPUs, um dos principais requisitos na exigência por processamento de alto desempenho.

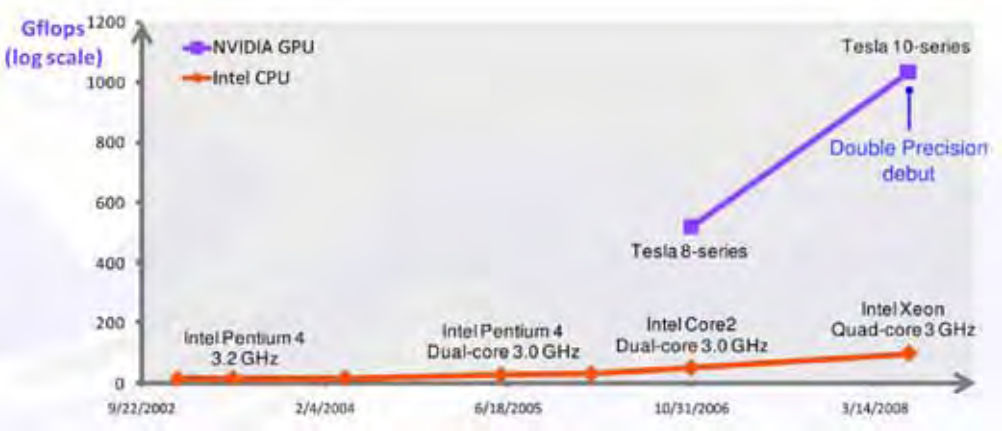


Figura 3.3: *Evolução de Operações por Ponto Flutuante em CPUs e GPUs.*
Fonte: NVIDIA (2010)

A partir destas considerações acerca das GPGPUs, ressaltam-se as principais características sobre o modelo SPM, denominado pela NVIDIA como *Single Instruction, Multiple Thread* (*Single Instruction, Multiple Thread* (SIMT)). Neste modelo, os dados são representados como um *stream*, ou seja, um fluxo de dados é classificado como um conjunto, e estruturado em um *array*. A execução de uma instrução é aplicada em um conjunto de dados.

Os *kernels* executam operações paralelamente em todo o *stream*, utilizando-o como dado de entrada e saída. As operações de gerenciamento podem ser divididas em cópia, indexação e decomposição de subconjuntos, ou podem ser calculados paralelamente utilizando *kernels* (KIRK; HWU, 2010; NVIDIA, 2010).

No modelo SIMT, a chamada de uma variedade de *kernels* é organizada como uma hierarquia de grupo de *threads*. O recurso de dividir *kernels* em blocos independentes, bem como o suporte eficiente aos *threads* em GPGPU garante escalabilidade transparente e portável, permitindo um programa em CUDA executar em qualquer número de *cores*. Os *threads* são utilizados para paralelismo de granularidade fina e são agrupados formando blocos. Os blocos são utilizados para paralelismo de granularidade grossa e são agrupados formando uma grade, a qual representa uma chamada de *kernel* na placa gráfica. Como ilustrado na Figura 3.4, esta hierarquia permite que cada *thread* dentro de um bloco, assim como, cada bloco dentro de uma grade, tenham um índice de identificação único (HOFMANN; BINNA, 2010).

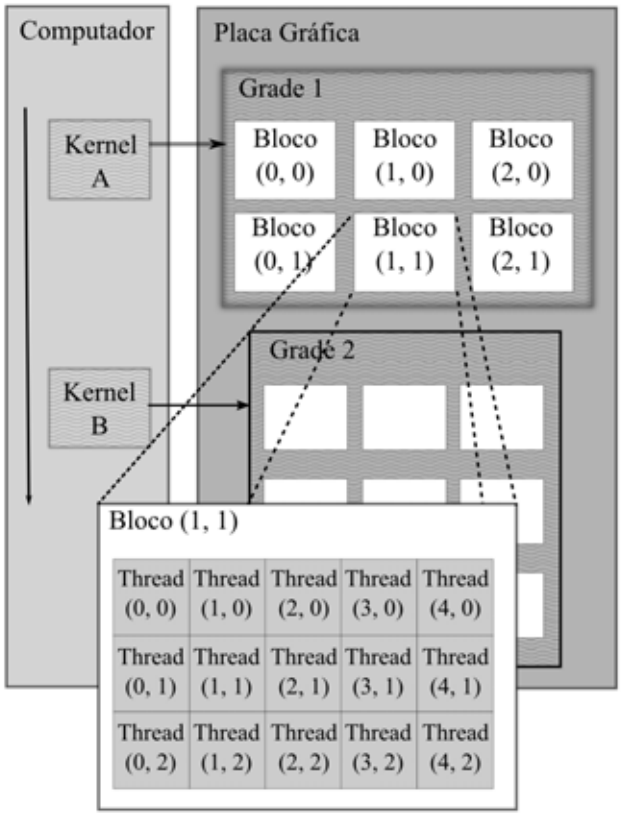


Figura 3.4: *Modelo Single Instruction, Multiple Thread.*
Fonte: Traduzida e adaptada de NVIDIA (2010)

Neste modelo, os *threads* de um bloco podem ser organizados em três dimensões para executar em um único multiprocessador na GPGPU. Também podem ser sincronizados e compartilhar dados com outros *threads* do mesmo bloco por meio de memória compartilhada, já este tipo de comunicação é mais rápida do que através da memória global da placa gráfica, utilizada na comunicação entre os blocos (KIRK; HWU, 2010).

Na Figura 3.5, é apresentado um diagrama sobre a composição da arquitetura GPGPU desenvolvida pela NVIDIA, conhecida como *Compute Unified Device Architecture* (CUDA). A arquitetura CUDA é adequada para realizar as operações descritas no modelo anterior, e conforme ilustrado nesta figura, a arquitetura consiste de *Stream Multiprocessor* (SM), podendo acoplar até dezenas destes multiprocessadores. Para cada SM, ilustrado na Figura 3.5 (b), é possível observar o agrupamento de *Streaming Processors* (SP) ou CUDA *cores*. As unidades de *load/store* (LD/ST) permitem que os endereços "origem" e "destino" sejam calculados para 16 *treads* por *clock*. Instruções do tipo *seno, cosseno e raiz quadrada* são executadas em unidades de funções especiais, representadas na ilustração por *Special Function Units* (SFU). Cada SM possui 4 unidades SFU e organiza as *threads* em grupos de 32 unidades paralelas, que são definidos como *warps*. Cada SM pode executar até 2 *warps*

concorrentemente, através de 2 controladores *warp* e 2 unidades de envio. A Figura 3.5 (c) ilustra o CUDA *core*, o qual é composto por uma unidade lógica aritmética (INT) e uma unidade de ponto flutuante (FP).

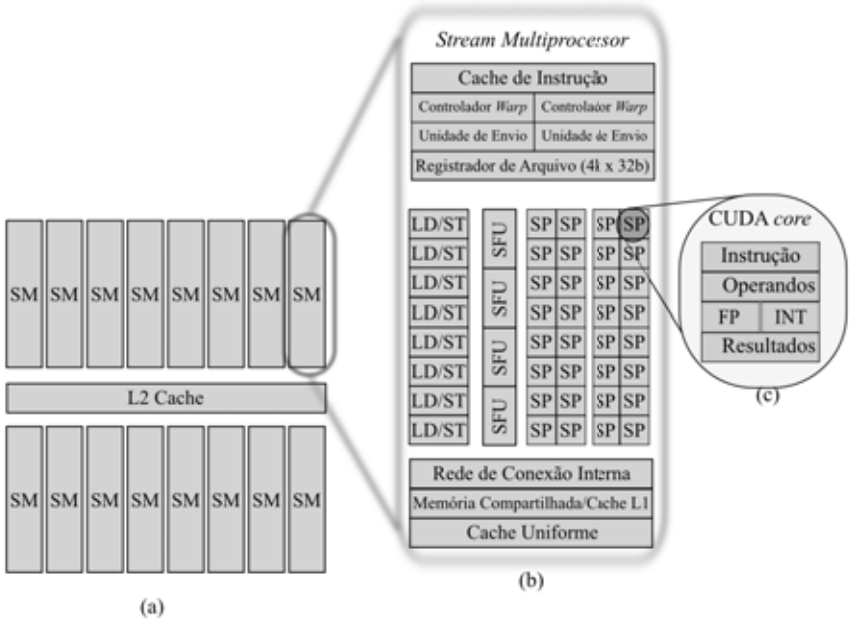


Figura 3.5: *Visão simplificada da GPGPU Fermi (Nvidia).*
(a) Diagrama de bloco do *stream multiprocessor*. b) Diagrama de bloco do Processador de *thread/stream*. c) Diagrama de bloco do Processador CUDA *core*. Fonte: Traduzida e adaptada de Hwu (2011)

O suporte em linguagens de alto nível para desenvolvimento em CUDA utilizam o estilo tradicional da linguagem C/C++, dessa maneira utilizam fluxos de controle como *if-then-else*, *for* e *while*. A implementação destes controles são diferenciadas de acordo com a arquitetura do processador, GPGPU ou CPU.

Como visto anteriormente, uma parte do processador executa operações baseadas no modelo SIMD durante várias partes do tempo, dessa maneira caso porções de código em execução utilizem *branch* diferentes, ambos serão processados. Esta é uma das situações de redução de desempenho para programas com implementações de *branchs*. *Branch* são ramificações no código de implementação fazendo com que dois caminhos de execução sejam sempre verificados, como por exemplo em controles do tipo *if-then-else*.

Os mecanismos de controle utilizados em CUDA, de maneira geral, implementam MIMD *branching*, SIMD *branching* e previsão de desvios. MIMD *branching* permite que diferentes processadores executem diferentes ramificações dependentes de dados sem penalidades, similar ao que acontece em CPU. SIMD *branching* exige que todos processadores executem

instruções idênticas, por outro lado, ramificações divergentes podem ocasionar redução de desempenho (NVIDIA, 2010).

O mecanismo de previsão de desvios é empregado na avaliação de ramificações, e de acordo com a condição identificada, é atribuído um marcador de condição. Este marcador será verificado antes do processador percorrer a próxima ramificação. O custo de processamento de todas as ramificações será o mesmo para cada parte da ramificação, somado ao custo de avaliação do ramo (PARHAMI, 2002).

3.3 Aspectos de Programação Paralela

Baseando-se em arquiteturas paralelas e técnicas de computação paralela, linguagens de programação paralela foram desenvolvidas a fim de permitir interação na utilização destes recursos e aplicação em áreas de interesse comercial ou científico.

Kirk e Hwu (2010) classificam os modelos de programação paralela em interação de processos e decomposição de problemas. De acordo com os mecanismos de comunicação entre os processos paralelos, o modelo chamado interação de processos pode ser dividido em: memória compartilhada, troca de mensagens e implícita.

O modelo de decomposição de problemas caracteriza a maneira como os processos são formulados para o processamento paralelo, sendo este dividido em paralelismo de tarefas, de dados, e implícito. Modelos de computação paralela que exigem troca de mensagens entre processadores e sistemas multiprocessados de memória distribuída, podem ser melhores explorados por meio da linguagem de programação *Message Passing Interface* (MPI), e OpenMP para sistemas multiprocessados de memória compartilhada. Seguem algumas das principais características destas linguagens (KIRK; HWU, 2010):

- Aplicações implementadas em MPI, podem executar em mais de 100.000 processadores de um sistema em *cluster*, no entanto, pode ser extremamente custoso implementar modificações para executar a aplicação em sistema de memória compartilhada.
- A linguagem OpenMP é considerada ideal para sistemas de memória compartilhada, porém, dificuldades no gerenciamento de *overhead* limitam sua escalabilidade para algumas centenas de processadores.

A linguagem CUDA C, desenvolvida especialmente para GPGPUs, não possui os problemas e limitações apresentadas pelas linguagens descritas anteriormente. Como vantagem em relação ao MPI, permite a comunicação entre GPGPU e CPU utilizando troca de mensagens unilateral. Em comparação com o OpenMP, possui alta escalabilidade e baixo *overhead* no

gerenciamento de *threads*. No entanto, não oferece suporte à variedade de aplicações as quais o OpenMP suporta com sucesso.

Com o objetivo de padronizar o modelo de programação para GPGPU, alguns fabricantes como Apple, Intel, AMD, e a própria NVIDIA, desenvolvedora da arquitetura CUDA, uniram-se no desenvolvimento do modelo de programação denominado *Open Computing Language* (OpenCL). Com a padronização, pretende-se permitir que as aplicações desenvolvidas utilizando este modelo, possam executar em quaisquer processadores com suporte às extensões e API da linguagem, além de garantir ao programador o gerenciamento do paralelismo em processadores massivamente paralelos.

As linguagens CUDA e OpenCL possuem várias similaridades, diante disso, Kirk e Hwu (2010) afirmam haver uma certa vantagem para a linguagem CUDA, em velocidade de execução quando o *hardware* oferece suporte para ambas as linguagens, e garantem também que a linguagem OpenCL exige desenvolvimento de aplicações em mais baixo nível em relação ao CUDA.

3.4 Considerações Finais

Diante da demanda por processamento de alto desempenho, no processo de paralelização dos algoritmos deve-se levar em consideração a arquitetura de processadores para, posteriormente, definir-se a melhor estratégia de paralelização e linguagem de programação.

O objetivo é explorar ao máximo a capacidade de processamento da arquitetura selecionada, obtendo resultados satisfatórios. O emprego de técnicas de paralelismo, na maioria das vezes, visa alcançar resultados tais como a redução no tempo de processamento.

CAPÍTULO 4

Métodos de Suavização de Imagem e Métricas de Avaliação

O objetivo do presente capítulo é descrever alguns métodos de suavização de imagem mais representativos, principalmente quando a mesma é obtida por meio de sensores do tipo: infravermelho, radar, ultrassom, dentre outros. Neste capítulo, também são apresentadas algumas métricas para a avaliação de suavizações. Ruídos, em processamento de imagem, são erros que dificultam ou diminuem a precisão na identificação de características de uma imagem, e são comuns durante a aquisição de imagem.

4.1 Considerações Iniciais

A captura de imagem pode ser realizada por meio de diversos dispositivos como câmera fotográfica, sensor *laser*, aparelho de ultrassonografia, radares, satélites, dentre outros (BANDEIRA, 2009; KRISSIAN et al., 2005; YI, 1999). Uma imagem, em escala de cinza, pode ser definida como uma função $f(x,y)$ sendo cada ponto desta função, conhecido como pixel, associado à uma intensidade luminosa. No caso de imagem colorida, cada ponto da função é associado a um vetor com três valores, comumente representados por *Red, Green and Blue* (RGB).

Durante a etapa de aquisição de imagem é comum o registro de erros ou distorções, conhecidos como ruídos. Os ruídos, normalmente, são distribuídos na imagem de forma aleatória e com uma distribuição de probabilidade que possuem desvio padrão e média específicos.

Uma imagem pode ter suas características corrigidas, suavizadas ou realçadas por meio de técnicas de transformação. Uma das técnicas, conhecida como abordagem local é empregada "pixel a pixel", sendo a mudança de um ponto P qualquer, dependente dos pontos considerados na "vizinhança" de P. A contribuição de pontos mais próximos é mais acentuada do que aqueles pontos mais distantes.

De maneira geral, a partir da identificação de cada tipo de ruído, vários métodos foram propostos (GONZALEZ; WOODS, 2001). No entanto, o interesse do presente estudo

concentra-se em um ruído multiplicativo (KRISSIAN et al., 2005). A captura de imagem a partir ultrassom, sonar, radares de abertura sintética, infravermelho, resulta em imagem com este tipo de ruído por causa da radiação incidente sobre a superfície observada (KRISSIAN et al., 2005; YI, 1999).

A tarefa de realizar a redução deste tipo de ruído pode ser classificada em duas categorias. Na primeira, são utilizados algoritmos de filtragem para reduzir o ruído multiplicativo após a captura da imagem, bem como podem ser aplicados no domínio espacial e da frequência. Já na segunda categoria, durante a formação da imagem são aplicadas técnicas para reduzir a intensidade do ruído calculando uma média de cada posição da imagem (BANDEIRA, 2009). Na próxima seção são introduzidos alguns exemplos de filtros, inicialmente os filtros da média e da mediana, os quais não adotam um modelo para um ruído específico e posteriormente, serão introduzidos filtros para ruído multiplicativo.

Os filtros podem ser divididos em filtros lineares, capazes de suavizar, manter detalhes e minimizar efeitos de ruídos sem modificar o nível médio de cinza da imagem; ou filtros não lineares, caracterizados por realizar as mesmas operações dos filtros lineares, no entanto, sem a preocupação de manter o nível médio de cinza da imagem original.

4.1.1 Filtro da Média

A interpretação da maioria das imagens descreve uma superfície contínua e uma resolução de imagem a partir do mesmo nível de intensidade. O filtro da média é linear e geralmente, não adota modelos específicos de ruídos, bem como é considerado um dos filtros mais utilizados na remoção de ruídos.

Nesta abordagem de suavização, o elemento central de uma pequena região de um pixel recebe o valor da média entre os elementos desta pequena região, também conhecida como "janela" ou "máscara". Esta mesma operação é realizada para cada pixel da imagem, como por exemplo, o valor do pixel central (12) da janela será substituído pelo resultado do cálculo da média dos valores 12+238+244+244+245+245+247+250+252, que resulta em 219. Observe o exemplo utilizando uma janela de 3x3, ilustrado pela Figura 4.1.

244	247	245
252	**12**	238
244	245	250

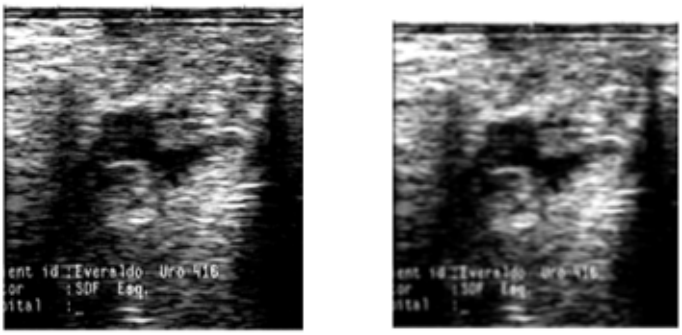
(a)

244	247	245
252	**219**	238
244	245	250

(b)

Figura 4.1: *Janela de elementos com dimensões de 3x3.*

A abordagem consiste em eliminar os ruídos, tendo como principal vantagem a preservação do contorno, porém causa pequenas distorções em linhas finas e curvas agudas, o que pode ser contornado utilizando o filtro da mediana, descrito na seção a seguir. Como exemplo, a Figura 4.2 ilustra o filtro da média aplicado em uma imagem com ruído multiplicativo.



(a) *Imagem Original.* (b) *Imagem suavizada.*

Figura 4.2: *Demonstração do método de suavização de imagem: Média.*

4.1.2 Filtro da Mediana

O filtro da mediana é não linear, não realiza cálculos, apenas ordena os valores dos pixels de uma dada janela e substitui o valor do pixel central com o valor mediano da janela do pixel. Para uma janela de n x n pixels, a mediana dos valores dos pixels ordenados encontra-se na posição $\frac{(n^2+1)}{2}$. Para o exemplo anterior, 12+238+244+244+**245**+245+247+250+252, o valor da mediana é o 245, como pode ser observado pela Figura 4.3.

244	247	245
252	12	238
244	245	250

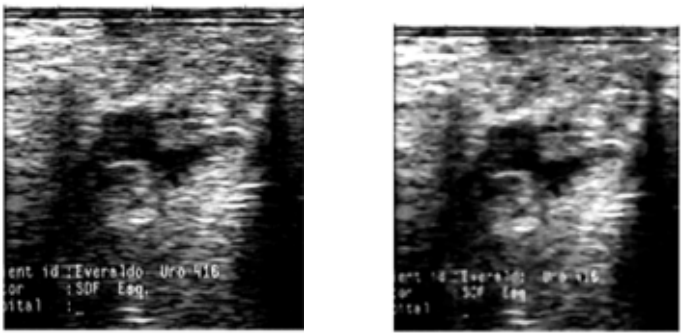
(a)

244	247	245
252	**245**	238
244	245	250

(b)

Figura 4.3: *Janela de elementos com dimensões de 3x3.*

Este filtro é considerado adequado para ruídos que apresentam componentes impulsivos, como o ruído sal e pimenta. Desta maneira, com a variação significativa nos níveis de cinza dos pixels vizinhos, o filtro realiza o descarte destes elementos em uma dada janela, reduzindo a nitidez das bordas. Na Figura 4.4, é ilustrado um exemplo de aplicação do filtro da mediana em uma imagem com ruído multiplicativo.



(a) *Imagem Original.* (b) *Imagem suavizada.*

Figura 4.4: *Demonstração do método de suavização de imagem: Mediana.*

4.2 Métodos de Redução de Ruído Multiplicativo

4.2.1 Filtros Localmente Adaptativos

Esta categoria de filtros é conhecida como "adaptativos" em função de utilizar informações de uma vizinhança próxima ao pixel a ser processado, e assim permitir a redução de ruído em imagem contaminada por ruído aditivo (ex: gaussiano, impulsivo, dentre outros), multiplicativo ou ambos combinados. Dentre os filtros que compõem esta categoria, destacam-se os filtros de Lee e de Frost (BANDEIRA, 2009; YI, 1999). Basicamente, estes filtros buscam realizar a minimização de erro médio quadrático sobre o modelo multiplicativo do ruído, resultando em um modelo de ruído aditivo.

O filtro de Lee é capaz de preservar a estrutura original de bordas nas áreas com muito contraste, no entanto, praticamente não reduz o ruído em áreas homogêneas, como pode ser observado graficamente na Figura 4.5(b). Por outro lado, o filtro de Frost é capaz de reduzir o ruído presente nestas mesmas áreas, e mantém as bordas com um grau de nitidez adequado, representado na Figura 4.5(c).

(a) *Imagem Original.* (b) *Imagem suavizada com o método de Lee.* (c) *Imagem suavizada com o método de Frost.*

Figura 4.5: *Demonstração dos métodos de suavização de imagem, Lee e Frost.*

4.2.2 Método de Suavização baseado num Modelo Variacional

Um dos tipos de ruído muito explorado nas pesquisas atuais de processamento de imagem é o ruído multiplicativo. Uma imagem com ruído multiplicativo é considerada como sendo formada pela multiplicação de uma imagem original I, por uma imagem ruído I_n. Este tipo de ruído é considerado como sendo mais complexo de ser removido de imagem do que os ruídos aditivos (AUBERT; AUJOL, 2008).

O uso de modelos variacionais para remoção de ruído multiplicativo tem sido um dos caminhos mais adotados no desenvolvimento de métodos de suavização de imagem afetada por este tipo de artefato (AUBERT; AUJOL, 2008; HUANG; NG; WEN, 2009). Jin e Yang (2011) propuseram um método promissor para remoção deste tipo de ruído em imagem usando o modelo variacional proposto por Rudin, Osher e Fatemi (1992) para a remoção de ruído aditivo, e descrito pela Eq. 4-1:

$$min_u \left\{ J(u) + \lambda \int_\Omega (f-u)^2 \right\}, \tag{4-1}$$

onde Ω é um domínio fechado pertencendo a R^2, f é a imagem afetada pelo ruído, u é a imagem original na iteração atual, $J(u)$ é um termo regulador da equação, e λ é um parâmetro de peso. O método proposto em Jin e Yang (2011) foi projetado para remover ruído multiplicativo em imagem de ultrassonografia. Para isso, Jin e Yang (2011) investigaram como o ruído afeta este tipo de imagem, e concluíram que poderiam adotar a função proposta por Krissian et al. (2005) para resolver o modelo variacional da Eq. 4-1 considerando:

$$E(u) = \int_\Omega \frac{(f-u)^2}{u}, \tag{4-2}$$

onde u é a imagem original, $f = u + \sqrt{ug}$ é a imagem afetada pelo ruído sendo g uma variável Gaussiana com média não nula, ou seja diferente de 0 (zero). Assim, o modelo variacional adotado por Jin e Yang (2011) pode ser descrito pela Eq. 4-3:

$$min_u \left\{ J(u) + \lambda \int_\Omega \left[\frac{(f-u)^2}{u} \right] \right\}, \tag{4-3}$$

onde $\lambda > 0$ é um parâmetro de peso. Desta forma, o problema de remoção de ruído multiplicativo foi resolvido a partir do modelo dado pela Eq. 4-3, que foi resolvida numericamente adotando a Eq. 6-1:

$$\partial_t u = div\left(\frac{\nabla u}{|\nabla u|}\right) + \lambda\left(\frac{f^2}{u^2} - 1\right), \tag{4-4}$$

onde ∇ é o operador gradiente e div o operador divergente. Para discretizar a parte contínua da Eq. 6-1, os autores adotaram o esquema de diferenças finitas (RUDIN; OSHER; FATEMI, 1992), descrito pela Eq. 6-2, considerando o tamanho do passo $h = 1$ e o intervalo de tempo como sendo Δt:

$$\begin{aligned}
D_{\overset{+}{x}}(u_{i,j}) &= u_{i+1,j} - u_{i,j}, \\
D_{\overset{-}{x}}(u_{i,j}) &= u_{i,j} - u_{i-1,j}, \\
D_{\overset{+}{y}}(u_{i,j}) &= u_{i,j+1} - u_{i,j}, \\
D_{\overset{-}{y}}(u_{i,j}) &= u_{i,j} - u_{i,j-1}, \\
|D_x(u_{i,j})| &= \sqrt{\left[D_{\overset{+}{x}}(u_{i,j})\right]^2 + \left(m\left[D_{\overset{+}{y}}(u_{i,j}), D_{\overset{-}{y}}(u_{i,j})\right]\right)^2 + \delta}, \\
|D_y(u_{i,j})| &= \sqrt{\left[D_{\overset{+}{y}}(u_{i,j})\right]^2 + \left(m\left[D_{\overset{+}{x}}(u_{i,j}), D_{\overset{-}{x}}(u_{i,j})\right]\right)^2 + \delta}.
\end{aligned} \tag{4-5}$$

O parâmetro $\delta > 0$ é uma constante, sendo adotado próximo de 0 (zero) e o termo m é definido pela Eq. 4-6:

$$m[a,b] = \frac{sign(a) + sign(b)}{2} min(|a|, |b|), \tag{4-6}$$

onde $min(|a|, |b|)$ é uma função que retorna o menor dos valores absolutos de a e b, e $sign(a)$ é uma função que determina o sinal de a, retornando 1 (um) se a for positivo, -1 (menos um) se for negativo e 0 (zero) se a for igual a 0 (zero). Considerando $n = 1,2,...$ como

sendo as iterações do modelo, a Eq. 6-1 pode ser reescrita na forma:

$$u_{i,j}^{n+1} = \delta t \left[\frac{-D_{\overset{+}{x}}(u_{i-1,j}^n) + D_{\overset{+}{x}}(u_{i,j}^n)}{-|D_x(u_{i-1,j}^n)| + |D_x(u_{i,j}^n)|} + \frac{-D_{\overset{+}{y}}(u_{i,j-1}^n) + D_{\overset{+}{y}}(u_{i,j}^n)}{-|D_y(u_{i,j-1}^n)| - |D_y(u_{i,j-1}^n)|} \right] + \lambda^n \left(\frac{f^2}{(u_{i,j}^n)^2} - 1 \right) + u_{i,j}^n, \tag{4-7}$$

onde f é a imagem de entrada afetada pelo ruído. O parâmetro λ foi calculado automaticamente a cada nova iteração n do método por intermédio de:

$$\lambda^n = \frac{1}{\sigma^2} \left(\Sigma_{i,j} \left[\left(D_{\overset{-}{x}} \left(\frac{D_{\overset{+}{x}}(u_{i,j}^n)}{|D_x(u_{i,j}^n)|} \right) + D_{\overset{-}{y}} \left(\frac{D_{\overset{+}{y}}(u_{i,j}^n)}{|D_y(u_{i,j}^n)|} \right) \right) \frac{(u_{i,j}^n - f)u_{i,j}^n}{u_{i,j}^n + f} \right] \right) \tag{4-8}$$

onde σ^2 é a variância da imagem de entrada. O método de suavização de imagem é demonstrado nas Figuras 4.6 e 4.7.

(a) *128 x 128 pixels.* (b) *Imagem com ruído.* (c) *Imagem suavizada.*

Figura 4.6: *Demonstração do método de suavização de imagem.*

(a) *128 x 96 pixels.* (b) *Imagem com ruído.* (c) *Imagem suavizada.*

Figura 4.7: *Demonstração do método de suavização de imagem.*

4.3 Métricas de Avaliação da Suavização

A comparação entre imagens é uma tarefa natural para o sistema visual humano, mas a realização desta mesma tarefa por um sistema computacional é mais complexa e não tão natural. Existem vários estudos que tentam apresentar técnicas capazes de comparar estatisticamente o desempenho de métodos de processamento de imagem (WANG et al., 2004;

ECKERT; BRADLEY, 1998; RAMPONI et al., 1996), em particular, baseadas no erro e em informação estrutural.

4.3.1 Análise baseada em erro

Os índices de comparação baseados em erro fazem uma estimativa do erro entre a imagem restaurada, isto é, a imagem após suavização, e a imagem original antes de ser degradada pela adição de ruído.

As técnicas desta categoria têm como principal desvantagem a possibilidade de falhar em casos de existir deslocamentos/translação entre as imagens a comparar. Por exemplo, imagens semelhantes, mas que um dos objetos apresentados tenha sido deslocado numa delas podem ser consideradas distintas por estas técnicas. Além disso, estes índices realizam a comparação baseando-se na variação das intensidades dos pixels das imagens, o que pode originar que imagens com diferentes tipos de distorções têm índices semelhantes (WANG et al., 2004). Apesar disso, os índices baseados em erro são muito usados para comparar o desempenho de métodos de realce (GHITA; WHELAN, 2010) e suavização de imagem (JIN; YANG, 2011; CHEN; SUN; XIA, 2010), devido à sua simplicidade.

O índice *Peak Signal-to-Noise Ratio* (PSNR) é muito utilizado para análise do desempenho dos métodos de restauração e suavização de imagem. Este índice calcula a relação entre a maior força possível de um sinal, no caso de imagens, a maior intensidade, e sua força afetada pelo ruído (DASH; SA; MAJHI, 2011). Por conveniência, o PSNR é representado em função da escala logarítmica na base 10 (decibel), devido ao fato de alguns sinais possuírem um valor muito elevado. O PSNR pode ser calculado a partir do *Mean Squared Error* (MSE):

$$MSE = \frac{1}{m \quad n} \sum_{i=0}^{m-1} \sum_{j=1}^{n-1} [I(i,j) - I_r(i,j)]^2 \tag{4-9}$$

onde m e n são as dimensões das imagens a comparar, I é a imagem original e I_r é a imagem afetada pelo processamento, da seguinte forma:

$$PSNR = 10log_{10}\left(\frac{MAX_I^2}{MSE}\right) \tag{4-10}$$

onde MAX_I é o valor máximo que um pixel pode assumir; no caso de imagens em níveis de cinza representadas por 8 bits, $MAX_I = 255$. Assim, quanto maior for o seu valor do índice PSNR, mais semelhantes são as duas imagens em comparação. É importante destacar que para imagens idênticas, o valor do índice MSE será nulo, e consequentemente o PSNR será indefinido.

4.3.2 Análise baseada na informação estrutural

Nesta abordagem, tenta-se perceber as mudanças na informação estrutural da imagem para quantificar as degradações ocorridas. A análise da informação estrutural parte do princípio de que o sistema de visão humano é adaptado para extrair informação estrutural daquilo que é visualizado, buscando alterações nestas estruturas para detectar mudanças e, consequentemente, possíveis degradações geradas por algum processo (WANG; BOVIK; LU, 2002). O índice *Structutal SIMilarity* (SSIM) é um dos índices desta categoria mais usado para analisar a qualidade de métodos de processamento computacional de imagem (CHEN; SUN; XIA, 2010).

O índice SSIM foi proposto por Wang et al. (2004), na tentativa de evitar que imagens com qualidade visual muito distinta tenham o mesmo índice, como pode acontecer nos índices baseados em erro. Este índice é calculado com base em três componentes de comparação: luminância, contraste e estrutura da imagem. O primeiro componente é calculado pela intensidade média do sinal da imagem em análise. O contraste é calculado a partir do desvio padrão, e o parâmetro estrutura é computado a partir da imagem normalizada pelo desvio padrão da mesma. Assim, o índice SSIM pode ser obtido pela equação:

$$SSIM(x,y) = l(x,y)^{\alpha} \cdot c(x,y)^{\beta} \cdot s(x,y)^{\gamma}, \tag{4-11}$$

onde a componente l refere-se à luminância, c ao contraste e s à estrutura, e $\alpha 0$, $\beta 0$ e $\gamma 0$ são parâmetros constantes usados para definir o peso de cada uma das componentes no índice final. As três componentes l, c e s são relativamente independentes, e a alteração em uma delas não afeta as demais. Em (WANG et al., 2004) tem-se uma análise de cada uma destas componentes, indicando-se detalhadamente como as mesmas são calculadas.

O SSIM apresenta um índice por cada pixel da imagem, e para tornar o seu uso mais prático, costuma-se utilizar um índice SSIM médio, também chamado de *Mean Structure Similitary Index* (MSSI), que é calculado a partir da média dos elementos SSIM obtidos. Para imagens iguais, este índice é igual a 1 (um positivo), sendo reduzido a medida que as imagens se diferem, até atingir valor -1 (um negativo) para duas imagens exatamente opostas, isto é, uma é a negação da outra.

4.4 Considerações Finais

A captura de imagem, de maneira geral, registra interferências que dificulta a identificação de informações de interesse. Cada categoria de sensores utilizados na etapa de captura de imagem está suscetível a estes tipos de interferências ou ruídos.

Este capítulo descreveu, de maneira geral, como a área de processamento de imagem contribui na solução deste tipo de problema. Inicialmente foram apresentados os métodos mais comuns para suavização de imagem, e posteriormente, uma ênfase maior para o método de suavização de imagem baseado em um modelo variacional aplicado ao ruído multiplicativo. Por fim, apresentou-se a descrição dos métodos SSIM e PSNR, que serão utilizados como ferramentas de análise para aferir a eficiência do método de suavização adotado.

CAPÍTULO 5

Trabalhos Relacionados

Este capítulo descreve as principais abordagens encontradas na literatura que exploram métodos de paralelização aplicados em processamento de imagem, buscando obter o máximo de desempenho da arquitetura computacional disponível, e consequentemente, reduzir o tempo de processamento da aplicação. Uma vez que não foram encontradas pesquisas envolvendo a utilização de métodos de paralelização em GPGPU e CUDA na etapa de suavização de imagens afetadas por ruídos multiplicativos, foram consideradas pesquisas, de maneira geral, envolvendo aplicações paralelizadas em processamento de imagens.

5.1 Considerações Iniciais

A demanda por processamento de alto desempenho, comum em diversas áreas da indústria ou da Ciência, é objeto de interesse em muitas instituições de pesquisas em todo o mundo. Com isso, torna-se cada vez mais frequente a adaptação de métodos computacionais, tradicionalmente sequenciais, para o modelo de programação paralelizado. Isto significa a possibilidade de resolver, computacionalmente, diversos problemas de áreas como processamento de imagem e sinais (HOFMANN; BINNA, 2010), dinâmica de fluídos (KURZAK; BADER; DONGARRA, 2010), realidade virtual e aumentada (ARAS; SHEN, 2010), biologia computacional (KURZAK; BADER; DONGARRA, 2010), dentre outras, desenvolvendo métodos e algoritmos paralelizados.

A capacidade de cálculo massivamente paralelo das GPGPUs, permite realizar o processamento de dezenas de milhares de *threads* concorrentes. O processamento de imagens pode exigir o processamento de grandes volumes de dados ou imagens com altas resoluções, e ainda, processamento em tempo real.

5.2 Paralelização do filtro da mediana

A paralelização do filtro da mediana, paralelizada em um aglomerado de computadores, foi modelada para cada processador ficar responsável pelos cálculos dos novos pixels e suas

respectivas vizinhanças, definida como uma máscara de 3x3. A exploração do paralelismo ocorreu por meio de uma implementação orientada por troca de mensagens e memória compartilhada, aproveitando-se a composição dos nós com processadores *dual core* (SCHNORR, 2012).

O emprego de processadores *dual core* aliados ao uso das funcionalidades da biblioteca OpenMP, permitiu a utilização do paralelismo em dois níveis, sendo o paralelismo entre os nós e no próprio nó. No primeiro nível, o paralelismo no próprio nó, realiza a divisão dos dados de entrada para o processamento entre N instâncias. Cada instância, deve ser executada em um processador *dual core*, sendo os dados, divididos novamente, a nível de memória compartilhada. O paralelismo entre os nós, é realizado por meio da biblioteca MPI, responsável por coordenar a primeira divisão dos dados através do modelo de troca de mensagens.

O cenário utilizado nos testes foi composto por um aglomerado de computadores *dual core*, conectados por uma rede de alta velocidade. Os experimentos foram realizados com um número gradativo de nós, chegando a 20 computadores, já as imagens utilizadas, possuíam dimensões com 1024Kbytes, 6144Kbytes e 20504Kbytes. A abordagem proposta foi testada com o objetivo de encontrar o cenário de processamento mais adequado para a utilização da combinação das bibliotecas OpenMP e MPI.

Os resultados demonstraram ganho constante de *speedup*, no entanto, a escalabilidade ficou limitada a um número de 16 nós. A partir deste limite, a inclusão de nós passou a reduzir o ganho de *speedup* em razão do alto custo de comunicação provido pela abordagem de troca de mensagens. O ganho de desempenho em relação ao cenário com 2 e 16 nós, foi cerca de 4 vezes maior, como pode ser observado na Figura 5.1.

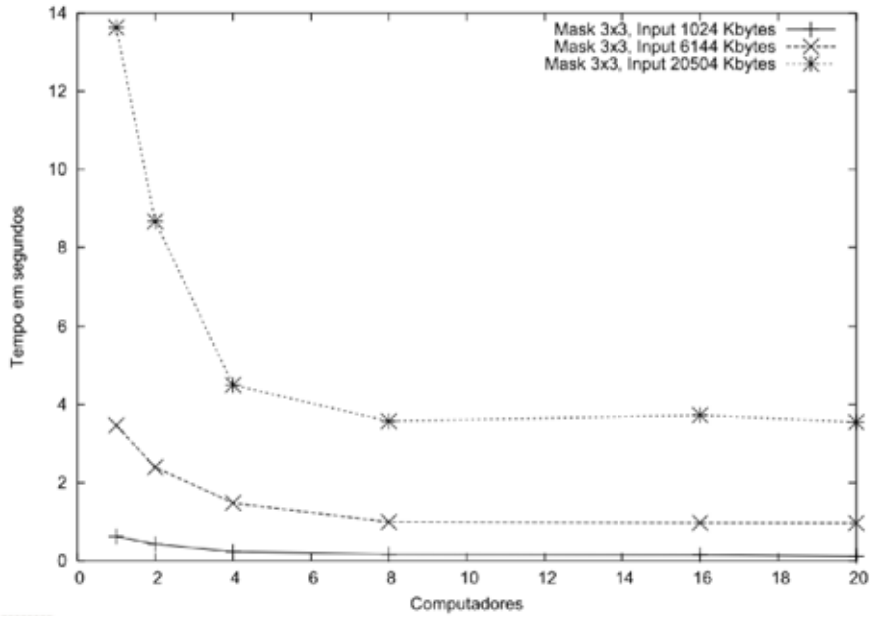


Figura 5.1: *Comparação de desempenho.*

Fonte: (SCHNORR, 2012)

5.3 Método de Canny paralelizado em CUDA

A programação massivamente paralela em GPGPU utilizando a arquitetura CUDA torna-se uma alternativa viável para acelerar a execução do método de detecção de contorno de *Canny*, integrado em bibliotecas de processamento de imagens médicas, como por exemplo, a biblioteca *Insight Segmentation and Registration Toolkit* (ITK). A arquitetura CUDA oferece infraestrutura de alto desempenho no processamento deste tipo de aplicação (LOURENCO, 2011).

O ITK é um conjunto de ferramentas com código fonte aberto, multiplataforma, utilizado no processamento, segmentação e registro de imagens médicas. A filosofia utilizada na implementação do ITK, permite combinar classes implementadas em diversas linguagens de programação como o TCL, Python e Java, graças à sua arquitetura baseada em fluxo de dados. Outro projeto, conhecido como CUDAITK, tem como objetivo explorar a capacidade de paralelismo de GPGPUs em busca de maior desempenho para implementações de filtros do ITK. Para maiores detalhes sobre o ITK, consultar Lourenco (2011).

É importante destacar a implementação de filtros da média, mediana, gaussiano em CUDA para o projeto CUDAITK. O desempenho obtido por estes filtros foi cerca de 140, 25 e 60 vezes mais rápidos que suas versões originais em ITK, respectivamente. Por outro lado, existe outro projeto com o propósito de explorar as arquiteturas GPGPUs, conhecido como *Cuda Insight Toolkit* (CITK). No entanto, esta iniciativa propõe realizar pequenas mudanças na

arquitetura ITK para obter suporte à arquitetura CUDA. Lourenco (2011) descreve a iniciativa como, compatível com os componentes ITK existentes, permitindo gerenciamento de dados em memória da GPU e CPU, transparente e eficiente, pois reduz a quantidade de cópias de dados entre memórias para o mínimo necessário.

Em relação á paralelização do método Canny, a pesquisa descreve a necessidade de otimização do algoritmo CUDA, destacando maior importância para o uso eficiente de estratégias de acesso à memória da placa gráfica. O ganho de desempenho, obtido nos experimentos, foi proporcionado pela exploração do modelo de programação paralela SIMT, e aos ajustes na definição de tamanhos para os blocos e grades manterem um número adequado de *threads* em processamento.

A implementação foi definida para cada pixel da imagem ser processada por uma *thread*, e cada bloco foi configurado para processar 256 *threads*. Os ajustes para configurar o tamanho da grade, e assim, definir a taxa de ocupação da GPU foi determinado pela Equação 5-1:

$$N_{Bl} = \frac{N_{Px} + N_{Th} - 1}{N_{Th}} \tag{5-1}$$

onde N_{Bl} expressa a quantidade de blocos ideal para processar N_{Px} pixels, utilizando a configuração de N_{Th}*threads* por bloco (LOURENCO, 2011). Os experimentos foram realizados utilizando 100 imagens com dimensões 321x481, 642x962, 1284x1924, 2568x3848 e 2568x3848 pixels, denominados na pesquisa como Bases B1, B2, B3, B4 e B5, respectivamente. Para avaliar a escalabilidade do código implementado, foram utilizadas 4 placas gráficas com diferentes configurações de CUDA *cores*, capacidade de memória e capacidade de cálculos.

A paralelização do método Canny obteve um *speedup* entre 1,2 e 3,0 vezes em placas gráficas, como pode ser observado na Figura 5.2. A comparação de desempenho envolvendo CPUs e GPU variou de 25,5 a 278,9 vezes. O melhor desempenho obtido pela placa Fermi, segundo o autor, foi em razão da quantidade 4 vezes maior de processadores, além de caches de memória global e o modelo de execução com 2 escalonadores, capazes de reduzir a latência de acesso.

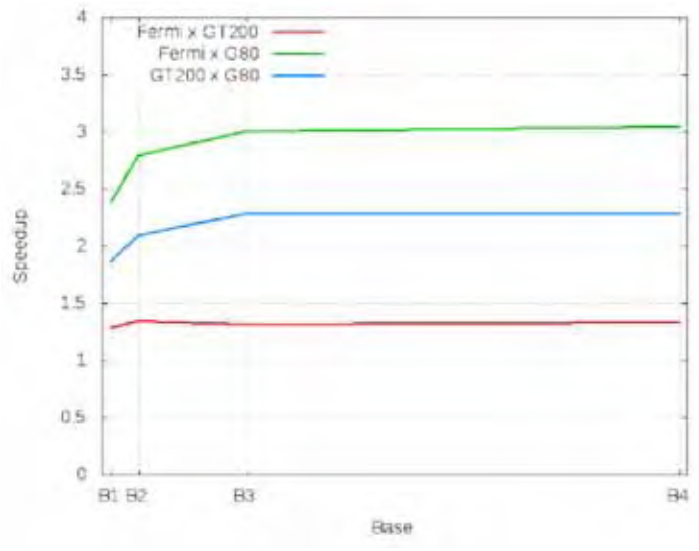


Figura 5.2: *Comparação de desempenho.*
Fonte: (LOURENCO, 2011)

Os experimentos também envolveram CPUs *dual core* e *quad core*, no entanto, não foram levados em consideração para a composição deste capítulo, pois considerou-se mais proveitoso demonstrar a variação de desempenho para diferentes placas gráficas. Neste caso, vale ressaltar a necessidade de configuração adequada de *threads* por *warps*, *warps* por multiprocessador, registradores por multiprocessador e memória compartilhada por multiprocessador, lembrando que o ideal é evitar o acesso serial à memória, definindo *warps* com *threads* múltiplos de 32. A partir de ajustes adequados, a otimização no tempo de execução alcança ganhos de desempenho de 82% na arquitetura G80, 84% na GT200 e 89% na Fermi.

O objetivo da pesquisa foi obter o maior ganho de desempenho no processamento de imagens utilizando arquitetura GPGPU, o qual foi alcançado explorando o modelo de paralelização SIMT e CUDA.

5.4 Modelo de filtros por vizinhança acelerados pelo uso da arquitetura CUDA

O processo de remoção de ruídos em imagens médicas de tomografia computadorizada pode ser realizado com a redução da radiação projetada, o que também reduz a qualidade das imagens obtidas, ou através de técnicas de processamento de imagens, como filtros para a suavização de imagens.

Zheng, Xu e Mueller (2011) desenvolveram um método paralelizado para suavização de imagens, aplicando técnicas avançadas de otimização de acesso à memória da placa gráfica.

Os filtros adotados foram o bilateral e o filtro não local. A abordagem proposta foi a subdivisão da imagem em partes iguais, e distribuir cada parte desta imagem para o processamento em um SM.

Os diferentes tipos de memórias disponíveis na arquitetura CUDA possuem uma enorme diferença na largura de banda, sendo a memória global a memória com maior custo para transferências de dados, centenas de ciclos por *clock* (PARK et al., 2011). Para reduzir a latência, a imagem de entrada foi armazenada na memória de textura, e cada *warp* foi responsável por ler/escrever apenas um segmento na memória, com 128 bytes. Aproveitando-se da localização dos pixels por vizinhanças cada *thread* deverá processar 4 bytes, os quais podem representar 4 caracteres, 2 números inteiros do tipo *short* ou 1 número de ponto flutuante com precisão simples (ZHENG; XU; MUELLER, 2011).

O custo de transferência de dados para a memória foi reduzido ainda mais, adotando a estratégia de busca antecipada (*prefetching*), a qual consiste em armazenar uma parte da imagem na memória para o processamento de um bloco de *threads*, e uma janela com dimensões superiores à máscara do filtro determina os dados que serão processados.

Os pixels da janela processada são sobrepostos a cada novo carregamento de parte da imagem. A utilização da memória compartilhada também reduz os acessos à memória global, pois este tipo de filtro reutiliza os dados de entrada várias vezes. Desta maneira, a estratégia de busca antecipada segue ilustrada na Figura 5.3, demonstrando o cenário de otimização para alcançar a redução de latência. No caso de apenas um bloco CUDA com 16x16 *threads* (representado na Figura 5.3(a) em verde) seriam necessários 32x32 pixels lidos em sua vizinhança (Figura 5.3(a)). Adotando o carregamento por vizinhança de pixels na memória compartilhada (representado em cinza), o método antecipa 4 partes de 16x16 pixels, carregados sequencialmente (Figura 5.3(b-e)). Finalmente, com os dados carregados completamente, as *threads* podem realizar o acesso mais rapidamente na memória compartilhada.

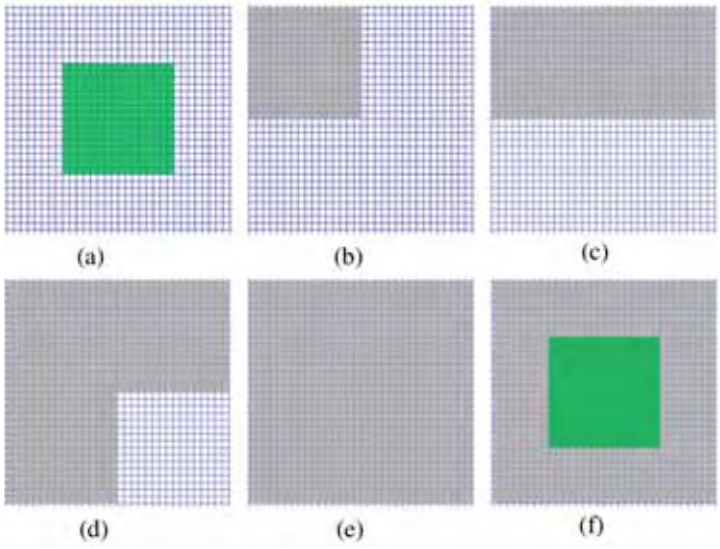


Figura 5.3: *Carregamento de partes da imagem para execução do filtro.*
Fonte: (ZHENG; XU; MUELLER, 2011)

Os experimentos foram realizados utilizando os blocos CUDA com 32x32 *threads*, no entanto, para otimizar a implementação poderia ajustar os blocos para 16x16 *threads*, pois a limitação da placa gráfica utilizada (Fermi) é de 1536 *threads* por bloco. Foram processadas imagens com 256x256, 512x512, 1024x1024 pixels de dimensão, e o *speedup* obtido foi de 4x para o filtro não local, e 1,2x para o filtro bilateral. Os resultados demonstram ganho de desempenho ao explorar a arquitetura CUDA, tornando uma alternativa eficiente, quando comparado à implementações sequenciais.

5.5 Considerações Finais

A pesquisa bibliográfica apresentou, de maneira geral, a utilização de infraestrutura computacional baseada em *multicore* e GPGPU aplicados em processamento de imagens. Como pode ser observado, a área de processamento de imagem requer alta capacidade de desempenho na execução de suas tarefas, no entanto, para obter maior ganho de desempenho é necessário combinar a infraestrutura de processamento com técnicas de paralelismo.

Independente da arquitetura paralela adotada, ao comparar implementações sequenciais e paralelizadas, o principal elemento para garantir o ganho de desempenho é a etapa de modelagem da paralelização do problema. No caso de utilização de GPGPUs, esta etapa equivale à configuração e ajustes do *kernel*, bem como o emprego de técnicas avançadas de otimização de memória em CUDA.

CAPÍTULO 6

Implementação do Método Adotado

O estudo sobre métodos e técnicas de paralelização em GPGPU, concentrando em programação massivamente paralela, bem como a pesquisa sobre os métodos de suavização mais comuns demonstraram a necessidade de adotar uma estratégia de paralelização baseada na decomposição de dados, conforme descrito na seção 3.1.1, e para obter o máximo da capacidade de processamento na arquitetura CUDA, buscou-se incorporar o modelo de execução SIMT. Este capítulo descreve a paralelização do método de suavização baseado em um modelo variacional.

6.1 Considerações Iniciais

O método de suavização adotado, descrito na seção 4.2.2, é formado basicamente por três equações fundamentais para reduzir o ruído em imagens. A Equação 6-1 é a solução para a remoção de ruído multiplicativo.

$$\partial_t u = div\left(\frac{\nabla u}{|\nabla u|}\right) + \lambda\left(\frac{f^2}{u^2} - 1\right), \tag{6-1}$$

As equações abaixo representam o esquema de diferenças finitas, adotadas para discretizar as partes contínuas da equação anterior.

$$\begin{aligned}
&D_{\overset{+}{x}}(u_{i,j}) = u_{i+1,j} - u_{i,j},\\
&D_{\overset{-}{x}}(u_{i,j}) = u_{i,j} - u_{i-1,j},\\
&D_{\overset{+}{y}}(u_{i,j}) = u_{i,j+1} - u_{i,j},\\
&D_{\overset{-}{y}}(u_{i,j}) = u_{i,j} - u_{i,j-1},\\
&|D_x(u_{i,j})| = \sqrt{\left[D_{\overset{+}{x}}(u_{i,j})\right]^2 + \left(m\left[D_{\overset{+}{y}}(u_{i,j}), D_{\overset{-}{y}}(u_{i,j})\right]\right)^2 + \delta},\\
&|D_y(u_{i,j})| = \sqrt{\left[D_{\overset{+}{y}}(u_{i,j})\right]^2 + \left(m\left[D_{\overset{+}{x}}(u_{i,j}), D_{\overset{-}{x}}(u_{i,j})\right]\right)^2 + \delta}.
\end{aligned} \tag{6-2}$$

Após a discretização da Equação 6-1, o método calcula o valor da função *lambda* em cada elemento da matriz, por meio da Equação 6-3, determinando o parâmetro de peso automaticamente para cada iteração, sendo f utilizado para representar a imagem afeada pelo ruído.

$$\lambda^n = \frac{1}{\sigma^2}\left(\Sigma_{i,j}\left[\left(D_{\bar{x}}\left(\frac{D_{\dot{x}}(u^n_{i,j})}{|D_x(u^n_{i,j})|}\right)+D_{\bar{y}}\left(\frac{D_{\dot{y}}(u^n_{i,j})}{|D_y(u^n_{i,j})|}\right)\right)\frac{(u^n_{i,j}-f)u^n_{i,j}}{u^n_{i,j}+f}\right]\right) \tag{6-3}$$

A última etapa, é a aplicação da Equação 6-4 para obter o valor final de cada pixel de acordo com a iteração em andamento.

$$u^{n+1}_{i,j} = \delta t\left[\frac{-D_{\dot{x}}(u^n_{i-1,j})+D_{\dot{x}}(u^n_{i,j})}{-|D_x(u^n_{i-1,j})|+|D_x(u^n_{i,j})|}+\frac{-D_{\dot{y}}(u^n_{i,j-1})+D_{\dot{y}}(u^n_{i,j})}{-|D_y(u^n_{i,j-1})|-|D_y(u^n_{i,j-1})|}\right]+\lambda^n\left(\frac{f^2}{(u^n_{i,j})^2}-1\right)+u^n_{i,j}, \tag{6-4}$$

6.2 Metodologia Adotada

A partir das equações descritas na subseção 6.1, foi definida uma sequência de passos para a paralelização do método de suavização. A seguir, destaca-se um roteiro o qual foi desenvolvido a partir de conceitos e recomendações estabelecidas no guia de boas práticas de programação da NVidia (NVIDIA, 2010), além de basear-se no modelo SIMT.

6.2.1 Passo 1 - Identificação da Capacidade Computacional

O procedimento de implementação da paralelização do método de suavização iniciou-se com a identificação da capacidade computacional da placa gráfica disponível, ilustrada na Figura 6.1.

6.2.2 Passo 2 - Configuração do nível de ocupação

A configuração do *kernel* deve ser ajustada para utilizar a quantidade de blocos e *threads* suficientes para realizar a leitura/escrita de todos os elementos das matrizes em execução. Todos os *kernels* implementados foram invocados com a configuração de 256 *threads* por bloco. O número de blocos B, foi definido através do calculo da seguinte equação:

$$B = \frac{T_{px}+T_{Tb}-1}{T_{Tb}} \tag{6-5}$$

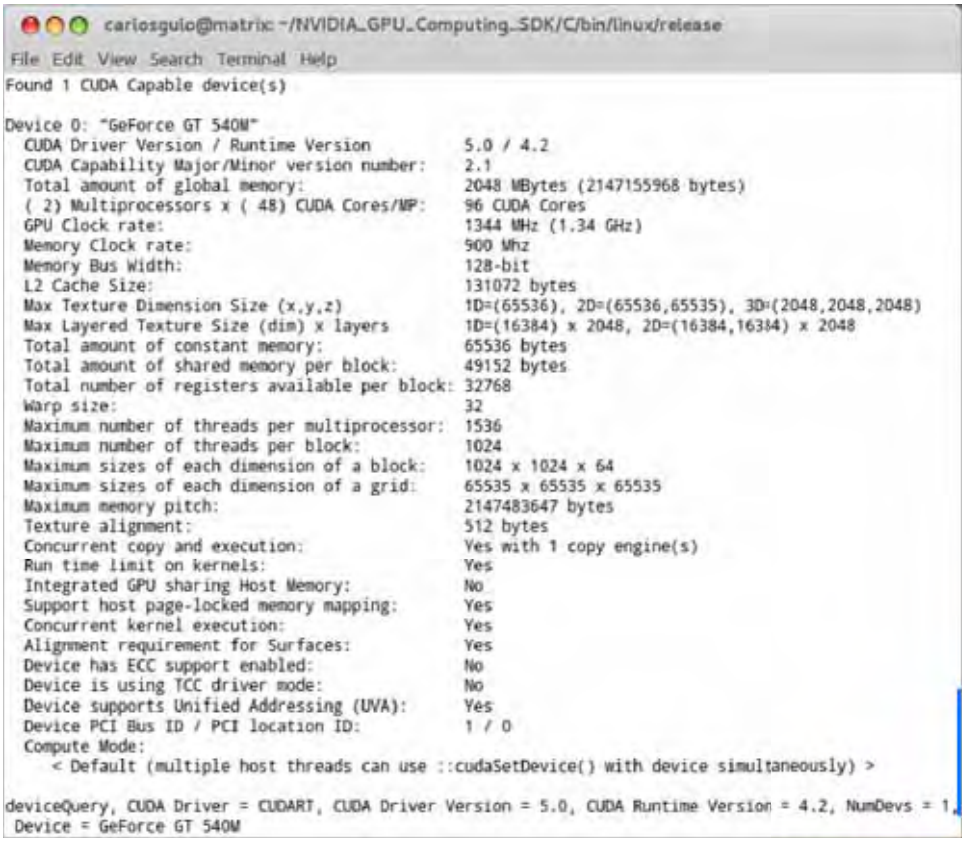


Figura 6.1: *Configuração detalhada da placa gráfica utilizada nos experimentos - NVidia GT 540M (Fermi).*

sendo T_{px} o total de pixels e T_{Tb} o número de *threads* por bloco. Realizando estes cálculos, obtém-se a configuração para realizar o processamento de um *thread* por elemento da matriz, ilustrado na Figura 6.2.

```
1       dim3 threadsPerBlock(16, 16)
2       dim3 numBlocks((alturaImagem + 15) / threadsPerBlock.y, (larguraImagem + 15) /
            threadsPerBlock.x)
```

Figura 6.2: *Configuração dos kernels utilizados na implementação.*

O desenvolvimento de aplicações para arquiteturas massivamente paralelas alcança maior desempenho quando os recursos disponíveis na placa gráfica são utilizados de maneira eficiente. Com isso, o *nível de ocupação* da GPU mede a proporção de processadores ativos na placa gráfica durante a execução de um *kernel*. A ocupação deve levar em consideração os limites de *threads* por bloco, blocos por multiprocessador, registradores por multiprocessador e memória compartilhada por multiprocessador. O ideal é ocupar a GPU com o número de máximo de *threads* concorrentes, definido para cada arquitetura e modelo de placa. Realizar um alto nível de ocupação, na prática, não garante ganho de desempenho adicional (NVIDIA,

2010), pois, existe o problema de latência de memória e um alto nível de ocupação pode reduzir o desempenho da aplicação, de maneira geral (PARK et al., 2011).

6.2.3 Passo 3 - Otimização no uso da hierarquia de memória em CUDA

Como pode ser observado na Figura 6.3, cada multiprocessador pode utilizar quatro tipos de memória: um conjunto de registradores para cada SM, memória compartilhada entre os SM, uma cache constante compartilhada entre os SM e a cache de textura para otimizar o acesso à memória de texturas. O acesso mais rápido são os registradores, e assim como os demais tipos de memória, podem ser acessados pelos *threads*. É importante destacar a possibilidade dos *threads* acessar dados em diferentes espaços de memória. Cada SM possui 64kB em memória de registradores (LOURENCO, 2011).

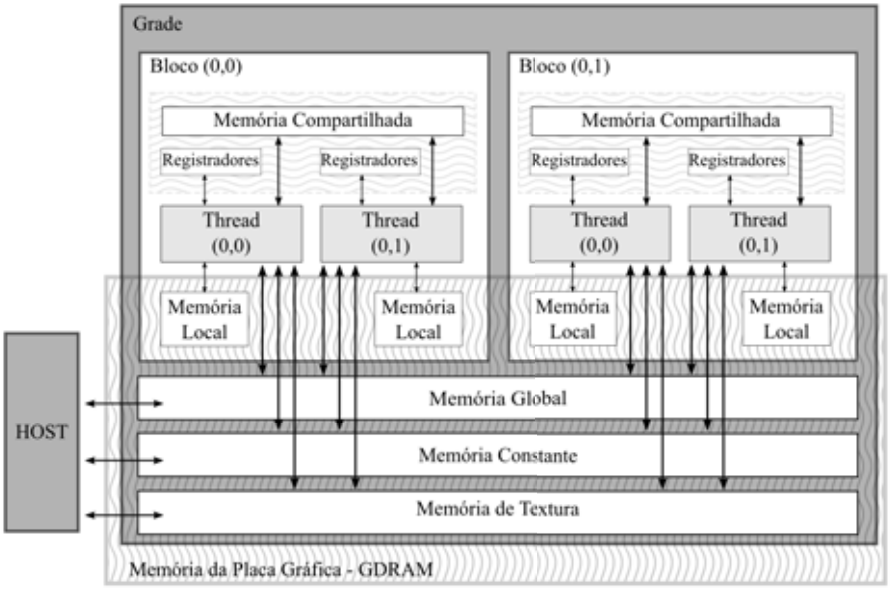


Figura 6.3: *Espaços de memória acessados por cada thread.*
Fonte: Traduzida e adaptada de Hwu (2011)

No caso da memória compartilhada, a velocidade de acesso é parecida aos registradores, qualquer *threads* de um bloco podem compartilhar acesso a mesma memória compartilhada entre os SM, e cada SM possui 64KB de memória, na arquitetura Fermi (NVIDIA, 2010). Cada módulo de memória possui 32 bits de largura, desta maneira, torna-se mais eficiente acessos de *threads* consecutivas às posições de um vetor de dados. Um módulo pode receber múltiplas requisições para um mesmo dado, gerando conflitos, no entanto, as operações de acesso à memória tornam-se serializadas automaticamente, até que todas as requisições de acesso sejam atendidas. Como a serialização pode reduzir o desempenho de acesso, um

dispositivo de *broadcast* é responsável por evita-la, impedindo a leitura de todas as *threads* ao mesmo endereço de memória (LOURENCO, 2011; PODLOZHNYUK, 2007).

O acesso à memória global pode ser realizado simultaneamente por todas as *threads*, entretanto, possui algumas restrições para melhorar a velocidade de acesso, considerada a memória mais lenta na GPU é também a memória com maior capacidade de armazenamento e pode ser acessada pela CPU. Este tipo de memória possui a maior largura de banda disponível na placa gráfica, no entanto, para obter esta vantagem, é necessário realizar acessos com todas as *threads* de um mesmo *half-warp* à posições contíguas de memória (PODLOZHNYUK, 2007). Este procedimento é conhecido como acesso coalescido de memória, que permitirá a realização de operações paralelas em um menor número de transações (NVIDIA, 2010; KIRK; HWU, 2010).

Todos os *threads* possuem acesso apenas de leitura à memória cache constante, que disponibiliza 8KB para cada SM. No entanto, apenas para um endereço de memória pode ser lido por *threads* de um *half-warp*. Este tipo de memória, permite ser escrita apenas a partir de instruções vindas da CPU, e é persistente através de chamadas múltiplas de *kernel* na mesma aplicação (NVIDIA, 2010).

A memória de textura também pode ser acessada por todas as *threads*, mas de uma mesma grade, e torna-se somente leitura a partir de *kernels*. Este tipo de memória utiliza uma cache separada das demais memórias, com capacidade de 8KB para cada SM, disponibiliza alto desempenho de acesso quando todas as *threads* realizam operações em endereços de memória próximos. A redução no número de acessos, disponível pelo tipo de dados *float4*, torna a leitura de texturas 4 vezes mais rápida do que utilizar o tipo de dados *float* (PARK et al., 2011).

A relação entre o tipo de acesso (“l” para leitura e “e” para escrita) e o espaço de memória, localização (interior “INT” ou exterior “EXT”) da memória no *chip* e a latência em ciclos de máquina, podem ser observados na Tabela 6.1.

Tabela 6.1: *Tipos de acesso em memórias em CUDA.*

Memória	Localização	*Thread*	Bloco	Grade	Latência
Local	EXT	l/e	–	–	200-300
Compartilhada	INT	l/e	l/e	–	= registradores
Global	EXT	l/e	l/e	l/e	200-300
Texturas	INT cache	l/e	l/e	apenas leitura	= > 100
Constantes	INT cache	l/e	l/e	apenas leitura	= registradores

6.2.4 Passo 4 - Implementação dos *kernels* em CUDA C

O processamento de imagem, comumente, envolve a necessidade de cálculos em grandes quantidades de dados. Desta maneira, a primeira estratégia adotada foi alocar espaço na memória da placa gráfica, em seguida copiar os dados das matrizes a partir da memória do computador (RAM) para a memória da placa gráfica *Graphics Dynamic Random-Access Memory* (GDRAM). Com isso, torna-se possível acessar os dados para realização dos cálculos apresentados no início deste capítulo. A partir deste momento, os dados passam a ser manipulados, processados e armazenados diretamente na GPU.

Os dados de entrada foram mapeados para a memória de textura, e desta maneira, os acessos são realizados por *warps* à cada duas requisições de acesso, ou uma para cada *half-warp*. Os acessos à memória global serão realizados em uma única transação, permitindo a recuperação de um segmento *half-warp* para todas as *threads*, conhecido como "acesso coalescido". Como a placa gráfica utilizada nesta pesquisa possui a capacidade computacional superior a 1.2, as restrições para coalescer os acessos à memória global são menores, possibilitando a realização de transações por segmentos, como por exemplo, ilustrado na Figura 6.4, onde duas transações são realizadas, sendo uma de 32bytes e outra de 64 bytes.

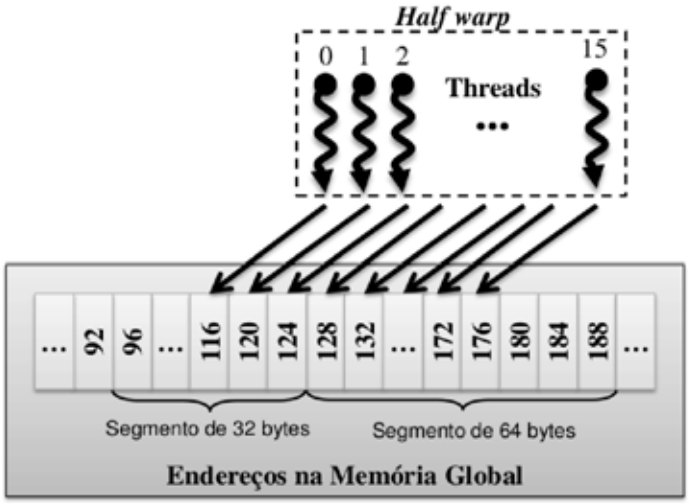


Figura 6.4: *Acesso à memória global.*

O carregamento dos dados foi realizado em segmentos contíguos, o que permitirá um bloco da imagem ser processado por um *thread block* mais eficientemente, como pode ser observado na Figura 6.5.

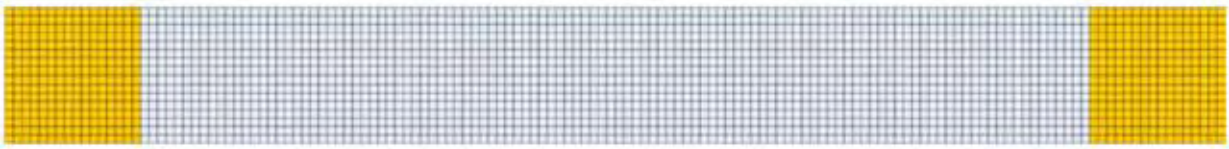


Figura 6.5: *Carregamento de dados na memória em segmentos contíguos.*

A implementação das Equações 6-2, 6-3 e 6-4, resultou nos *kernels* chamados de *kDiFinitas*, *kVariancia* e *kFinal*, respectivamente. Os *threads* do *kernel kDiFinitas* realizam os cálculos representados na Eq. 6-2, em cada elemento da matriz, independentemente. Este *kernel* possui diversos *threads*, sendo que cada um representa um índice da matriz, ou seja, cada um deles processa um determinado elemento. Desta forma é colocada uma condição para cada *thread*, verificando a necessidade de execução do código associado e a permissão para modificação das matrizes, utilizando a lógica do comando "*for*", respeitando delimitadores para início e fim. A Figura 6.6 apresenta o código implementado para o *kernel kDiFinitas*, utilizando a linguagem CUDA C. As linhas de código 3 e 4, de cada Figura de código, realizam o mapeamento dos índices de *threads* e blocos para as coordenadas da imagem processada. É importante destacar neste código, a utilização da memória de textura, por meio da função *tex1Dfetch* no acesso aos elementos de entrada.

```
__global__ void kDiFinitas(int alturaImagem, int larguraImagem, float* DxMaisPorDxMod,
    float* DyMaisPorDyMod, float auxDenominador, float regulador) {
   int j = threadIdx.y + blockIdx.y * blockDim.y;
   int i = threadIdx.x + blockIdx.x * blockDim.x;

   if (i < alturaImagem - 1 && j < larguraImagem - 1 ) {
      if (i == 1) {
         DxMaisPorDxMod[0 * larguraImagem + j] = tex1Dfetch(texMAux, 1 * larguraImagem
             + j);
         DxMaisPorDxMod[(alturaImagem - 1) * larguraImagem + j] = tex1Dfetch(texMAux,
             (alturaImagem - 2) * larguraImagem + j); }
      if (j == 1) {
         DxMaisPorDxMod[i * larguraImagem + 0] = tex1Dfetch(texMAux, i * larguraImagem
             + 1);
         DxMaisPorDxMod[i * larguraImagem + (alturaImagem - 1)] = tex1Dfetch(texMAux,
             i * larguraImagem + (alturaImagem - 2)); }
      DxMaisPorDxMod[i * larguraImagem + j] = (tex1Dfetch(texMAux, (i + 1) *
          larguraImagem + j) - tex1Dfetch(texMAux, i * larguraImagem + j)) / ((sqrtf(
          powf(tex1Dfetch(texMAux, (i + 1) * larguraImagem + j) - tex1Dfetch(texMAux,
          i * larguraImagem + j), 2) + powf(termoM((tex1Dfetch(texMAux, i *
          larguraImagem + (j + 1)) - tex1Dfetch(texMAux, i * larguraImagem + j)), (
          tex1Dfetch(texMAux, i * larguraImagem + j) - tex1Dfetch(texMAux, i *
          larguraImagem + (j - 1)))), 2) + regulador)) + auxDenominador);

      if (i == 1) {
         DyMaisPorDyMod[0 * larguraImagem + j] = tex1Dfetch(texMAux, 1 * larguraImagem
             + j);
         DyMaisPorDyMod[(alturaImagem - 1) * larguraImagem + j] = tex1Dfetch(texMAux,
             (alturaImagem - 2) * larguraImagem + j); }
      if (j == 1) {
         DyMaisPorDyMod[i * larguraImagem + 0] = tex1Dfetch(texMAux, i * larguraImagem
             + 1);
         DyMaisPorDyMod[i * larguraImagem + (alturaImagem - 1)] = tex1Dfetch(texMAux,
             i * larguraImagem + (alturaImagem - 2)); }
      DyMaisPorDyMod[i * larguraImagem + j] = (tex1Dfetch(texMAux, i * larguraImagem +
           (j + 1)) - tex1Dfetch(texMAux, i * larguraImagem + j)) / ((sqrtf(powf((
          tex1Dfetch(texMAux, i * larguraImagem + (j + 1)) - tex1Dfetch(texMAux, i *
          larguraImagem + j)), 2) + powf(termoM((tex1Dfetch(texMAux, (i + 1) *
          larguraImagem + j) - tex1Dfetch(texMAux, i * larguraImagem + j)), (
          tex1Dfetch(texMAux, i * larguraImagem + j) - tex1Dfetch(texMAux, (i - 1) *
          larguraImagem + j))), 2) + regulador)) + auxDenominador); } } ...
```

Figura 6.6: *Código fonte do kernel kDiFinitas, em CUDA C.*

Após a execução do *kernel kDiFinitas*, deve ser realizado o cálculo da variável λ, descrito

na Eq. 6-4 e paralelizado no *kernel kVariancia*. A implementação foi realizada utilizando um vetor auxiliar para armazenar os valores das operações de cada iteração, ou seja, cada *thread* calcula o valor de uma iteração.

```
1 __global__ void kVariancia(int alturaImagem, float* resultadoAux, int larguraImagem,
      float* mAux, float* DxMaisPorDxMod, float* DyMaisPorDyMod, float* imagemEntradaD,
      float auxDenominador) {
2     int j = threadIdx.y + blockIdx.y * blockDim.y;
3     int i = threadIdx.x + blockIdx.x * blockDim.x;
4
5     if (i < alturaImagem - 1 && j < larguraImagem - 1)
6         resultadoAux[i * larguraImagem + j] = ((((tex1Dfetch(texDxMaisPorDxMod, i *
              larguraImagem + j)) - ((tex1Dfetch(texDxMaisPorDxMod, (i - 1) *
              larguraImagem + j)))) + ((tex1Dfetch(texDyMaisPorDyMod, i * larguraImagem +
              j)) - ((tex1Dfetch(texDyMaisPorDyMod, i * larguraImagem + (j - 1)))))) * ((
              mAux[i * larguraImagem + j] - imagemEntradaD[i * larguraImagem + j]) * mAux[
              i * larguraImagem + j]) / ((mAux[i * larguraImagem + j] + imagemEntradaD[i *
               larguraImagem + j]) + auxDenominador));
7 }
```

Figura 6.7: *Código fonte do kernel kVariancia, em CUDA C.*

O *kernel kFinal* é invocado para realizar o processamento do parâmetro de peso, utilizado no *kernel kVariancia* e aplicado em cada elemento da matriz, realizando acessos à cache de textura e também acessos coalescidos na memória global. Cada *thread* atribui o valor resultante à posição de memória correspondente, disponibilizando os dados para a soma final de cada elemento da matriz.

```
1 __global__ void kFinal(int alturaImagem, int larguraImagem, float* mAux, float*
      imagemEntradaD, float auxDenominador, float deltaT, float lambda) {
2     int j = threadIdx.y + blockIdx.y * blockDim.y;
3     int i = threadIdx.x + blockIdx.x * blockDim.x;
4
5     if (i < alturaImagem - 1 && j < larguraImagem - 1 )
6         mAux[i * larguraImagem + j] = deltaT * ((tex1Dfetch(texDxMaisPorDxMod, i *
              larguraImagem + j) - tex1Dfetch(texDxMaisPorDxMod, (i - 1) * larguraImagem +
               j)) + (tex1Dfetch(texDyMaisPorDyMod, i * larguraImagem + j) - tex1Dfetch(
              texDyMaisPorDyMod, i * larguraImagem + (j - 1)))) + lambda * (((powf(
              imagemEntradaD[i * larguraImagem + j], 2)) / (powf(mAux[i * larguraImagem +
              j], 2) + auxDenominador)) - 1.0) + mAux[i * larguraImagem + j];
7 }
```

Figura 6.8: *Código fonte do kernel kFinal, em CUDA C.*

Por fim, o *kernel somaElem* é executado para auxiliar no cálculo da soma dos valores do vetor. Para calcular a soma, o vetor foi dividido em duas partes iguais e foram somados os valores de cada parte, e cada *thread* somando dois elementos e armazenando na menor das duas posições. Este processo deve ser repetido até sobrar apenas um elemento no vetor de uma posição, que é o resultado do somatório. Em caso de vetor com número ímpar de elementos, foi utilizado um elemento a mais contendo o valor zero. Este procedimento aproveita-se da estratégia de particionamento da memória global, responsável por otimizar o acesso à esta memória para *warps* ativas as quais foram distribuídas em partições. Este *kernel* é o mais lento da implementação, pois os blocos de memória tornam-se cada vez menos contíguos, na medida em que os elementos são processados. O diagrama da Figura 6.9 ilustra o método de paralelização implementado.

Uma imagem afetada por ruído multiplicativo é utilizada como dado de entrada (passo 1), e após o processamento dos *kernels* (passos $4-8$ descritos anteriormente, o resultado será uma nova imagem, no entanto, sem ruídos. Os passos $4, 6$ e 7 realizam leitura na memória de textura, por outro lado, cada etapa de escrita em memória é realizada na memória global ($1, 4, 6, 7$ e 8), onde a imagem de saída será armazenada.

6.3 Considerações Finais

Este capítulo apresentou os procedimentos realizados na paralelização do método de suavização baseado num modelo variacional. A partir da decomposição da equação apresentada como solução para reduzir ruídos multiplicativos, foi elaborado um roteiro com as etapas utilizadas na programação massivamente paralela, utilizando a arquitetura CUDA e a linguagem CUDA C. Diante da disponibilidade de uma hierarquia própria de memória (GDRAM), o método implementado foi adaptado para incorporar o modelo de execução tradicionalmente disponíem em CUDA, o modelo SIMT, e otimizado para obter o máximo de desempenho utilizando estratégias avançadas de redução de latência em memória global. Com isso, o Modelo de Paralelização do Método de Suavização Variacional torna-se apto para sua validação e avaliação de desempenho, que será realizada no próximo capítulo.

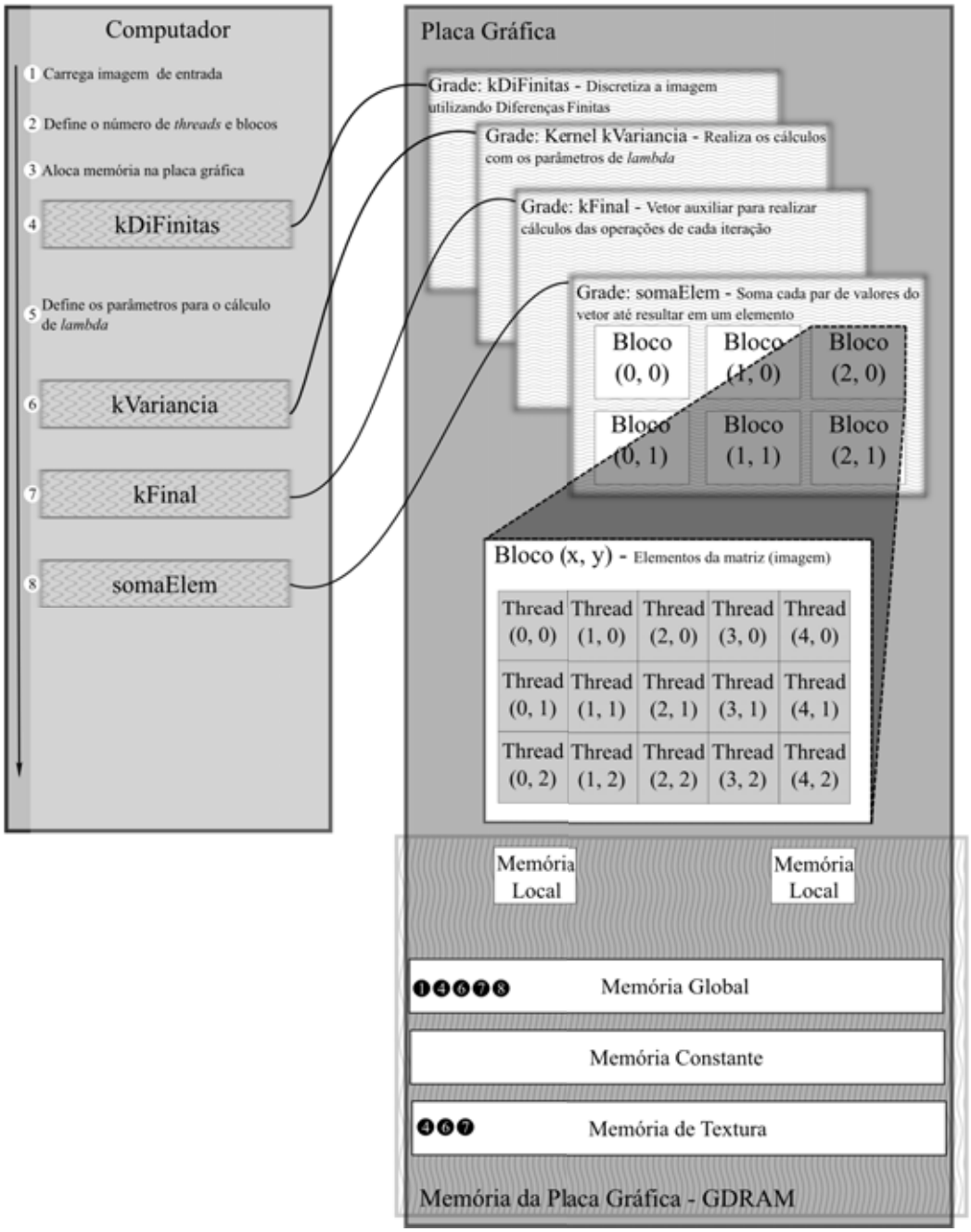


Figura 6.9: *Modelo de Paralelização do Método de Suavização Variacional.*

CAPÍTULO 7

Experimentos e Análise dos Resultados

Este capítulo descreve a infraestrutura de *hardware* e *software* utilizados para realização dos experimentos, bem como ilustra os resultados obtidos em experimentos por meio de gráficos e tabelas.

7.1 Considerações Iniciais

A métrica utilizada para avaliar o desempenho da solução paralelizada foi o *speedup*, descrito na seção 3.1, utilizando comparações envolvendo o tempo de processamento entre a implementações sequencial e paralelizada do método de suavização baseada em um modelo variacional. O tempo de processamento foi medido utilizando o recurso *timer*, disponível nativamente na linguagem CUDA C.

O procedimento de suavização de imagem, utilizando abordagem local, requer a realização de operações aritméticas pixel a pixel, e no caso de imagem com alta resolução, o maior impacto está no tempo de processamento. O desenvolvimento desta pesquisa utilizou o método hipotético-dedutivo (SEVERINO, 1996) na validação dos resultados encontrados, a partir da hipótese formulada sobre a possibilidade de obtenção de maior desempenho empregando a arquitetura massivamente paralela em GPGPU, na etapa de pré-processamento em imagens contaminadas com ruído multiplicativo.

7.2 Infraestrutura de Testes

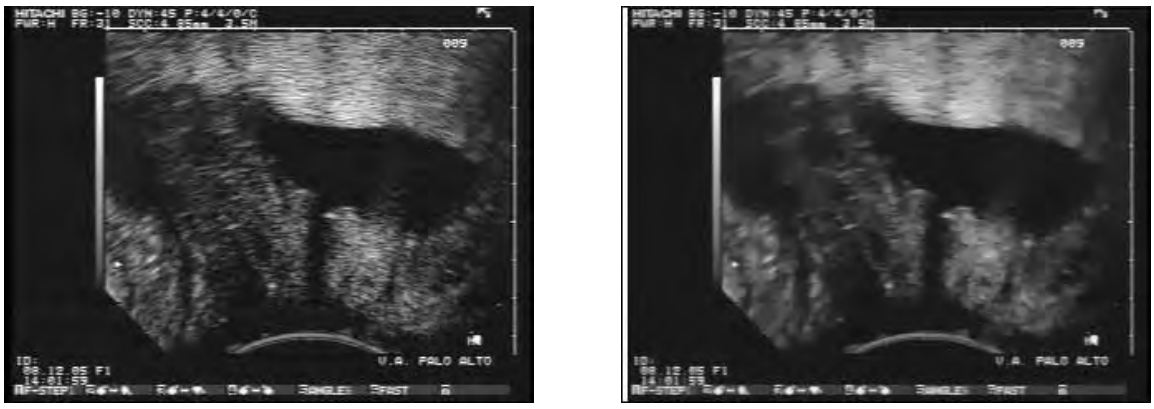
A implantação da infraestrutura de testes, foi composta pela instalação do compilador e *suite* de desenvolvimento CUDA, disponível para *download* no repositório da NVIDIA. Utilizou-se uma placa gráfica GeForce GT 540M (NVidia) com 96 CUDA *cores* e 2GB de memória GDRAM, computador portátil Dell XPS 15.4", equipado com processamento Intel i7-2630QM de 2GHz, com 8GB de memória RAM (DDR3 1333 MHz), sistema operacional Linux Ubuntu 12.04 - 64 bits, compilador CUDA nvcc 4.2 *release*, compilador g++ versão 4.6.3, e o *software* utilizado como ambiente de desenvolvimento foi o Netbeans 7.1.

7.3 Análise dos Resultados

Nesta seção são apresentados os resultados obtidos a partir da paralelização do método de suavização adotado e utilizado em imagens com ruído multiplicativo. A avaliação de desempenho do método de suavização paralelizado, foi realizada comparando-se o tempo de processamento entre as implementações sequencial e paralelizada.

Inicialmente, foram realizados experimentos utilizando um vídeo de ultrassom, com resolução de 320x240 pixels, e posteriormente, foram analisadas imagens com diversas resoluções: 128x128, 256x256, 512x512, 1024x1024, 2048x2048 e 4096x4096 pixels. Para demonstrar que os resultados do algoritmo paralelizado são iguais aos resultados do algoritmo sequencial, foram adotados os índices PSNR e SSIM. Como já descritas no Capítulo 6, estas ferramentas de análise numérica permitem validar a eficiência do filtro utilizado. Os índices indicam a variação de erro e similaridade existentes entre duas imagens.

No experimento realizado com o vídeo de ultrassom, o algoritmo foi executado 50 vezes para realizar a suavização de 255 *frames* do vídeo. O tempo médio de execução foi, 140 segundos para a versão sequencial e 39 segundos para a versão paralelizada em CUDA. Desta maneira, o tempo de processamento do algoritmo paralelizado foi, cerca de 3,5 vezes menor em relação ao algoritmo sequencial para processar o mesmo vídeo (GULO et al., 2012). O algoritmo de suavização foi executado utilizando 120 iterações e $\Delta t = 0.01$. O resultado da suavização em 1 *frame*, retirado aleatoriamente do vídeo, segue apresentado na Figura 7.1(a). Os índices PSNR e SSIM foram +19.84209 e 0.83345, respectivamente.

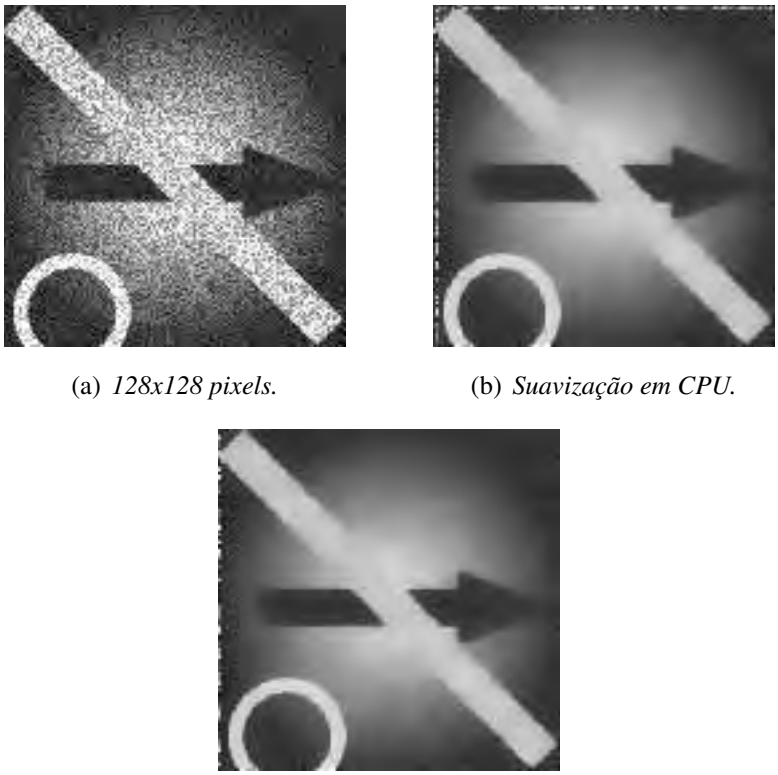


(a) *Imagem original.* (b) *Suavização em GPGPU.*

Figura 7.1: *Experimentos utilizando o método de suavização.*

Na Figura 7.2, são apresentados os resultados da aplicação dos métodos sequencial e paralelo em uma imagem, no entanto, os experimentos foram realizados em todas as imagens descritas anteriormente. Na mesma figura apresenta-se, da esquerda para a direita, a imagem afetada pelo ruído multiplicativo, o resultado da suavização sequencial, e o resultado da

suavização paralelizada. Ambos os testes foram realizados adotando $\Delta t = 0.8$, 15, 25 e 50 iterações, como ajustes de parâmetros para o método de suavização (JIN; YANG, 2011).



(a) *128x128 pixels.* (b) *Suavização em CPU.*

(c) *Suavização em GPGPU.*

Figura 7.2: *Experimentos utilizando o método de suavização, para 50 iterações.*

Os cálculos dos índices PSNR e SSIM, representados na Tabela 7.1, confirmam que as imagens resultantes dos algoritmos sequencial e paralelo são de fato iguais, pois os valores do SSIM são muito parecidos, com pequena vantagem para a implementação paralelizada, já que os valores resultam em maior aproximação a 1 (imagem original, sem ruídos).

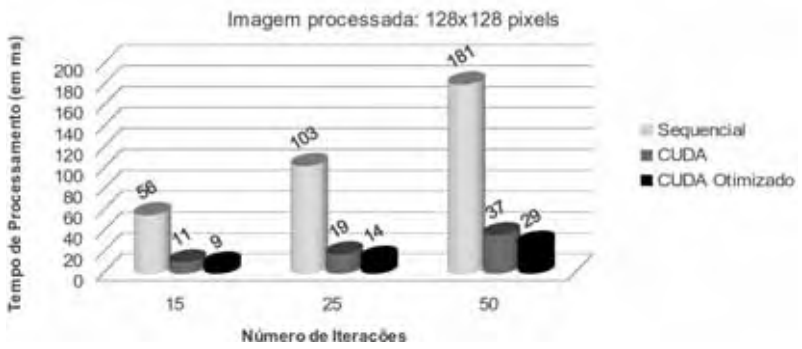
Tabela 7.1: *Índices estruturais das imagens utilizadas nos experimentos.*

Imagens	PSNR		SSIM	
	GPGPU	CPU	GPGPU	CPU
128x128	+25.24591	+23.36234	0.91494	0.90694
256x256	+19.54312	+19.30549	0.81498	0.81155
512x512	+12.26620	+12.24426	0.81723	0.81713
1024x1024	+14.80541	+14.79272	0.84877	0.84875
2048x2048	+14.05741	+14.05253	0.82687	0.82686
4096x4096	+13.67509	+13.67332	0.82081	0.82079

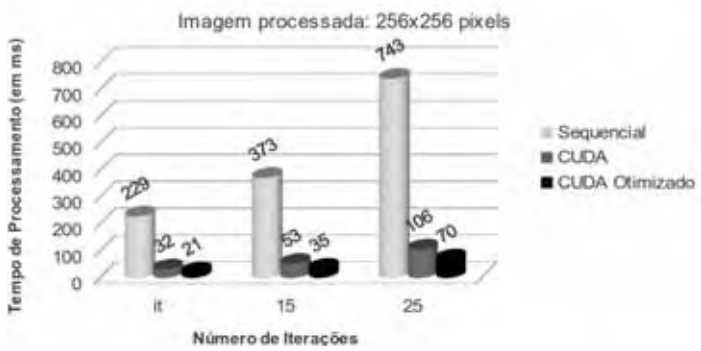
Os índices PSNR também apresentam resultados que indicam a similaridade entre as implementações, indicando que os valores maiores representam mais proximidade com a imagem original. Isto indica que as imagens comparadas, de maneira geral, não apresentam alterações nas informações estruturais, bem como não foi detectado nenhum erro entre as mesmas.

A primeira implementação paralelizada do método considerou apenas o uso da memória global, no entanto, o método foi otimizado e passou a utilizar a memória de textura para obter um ganho de desempenho ainda maior. A otimização do código, definida nesta pesquisa como CUDA Otimizado, permitiu um ganho acima 1,5 vezes em relação à implementação com apenas memória global, devido ao acesso coalescido dos dados e a menor frequência de acesso aos dados armazenados na memória de textura, que também possui um tempo de acesso muito menor do que a memória global.

As imagens foram criadas a partir de um editor de imagens, as quais foi inserido aleatoriamente ruído multiplicativo sintético com variância igual a 0.3. Nas Figuras 7.3(a), 7.3(b), 7.4(a), 7.4(b), 7.4(c) e 7.4(d), são apresentados os tempos de execução do algoritmo para as respectivas imagens com dimensões entre 128x128 e 4096x4096 pixels. O tempo de leitura da imagem em disco rígido e o carregamento da imagem na memória da placa gráfica não foram considerados nestes experimentos.

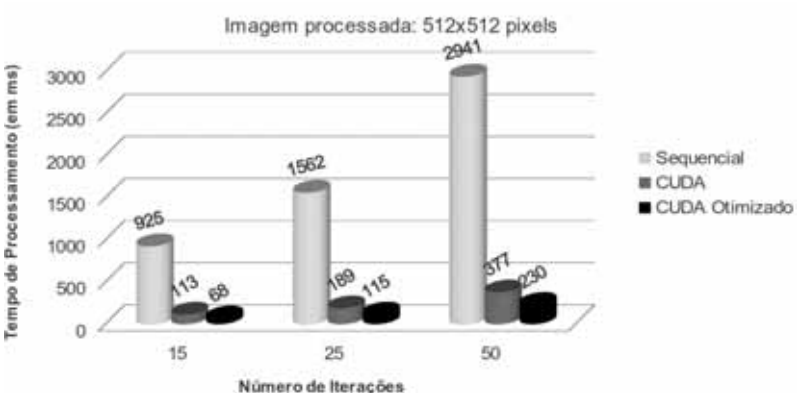


(a)

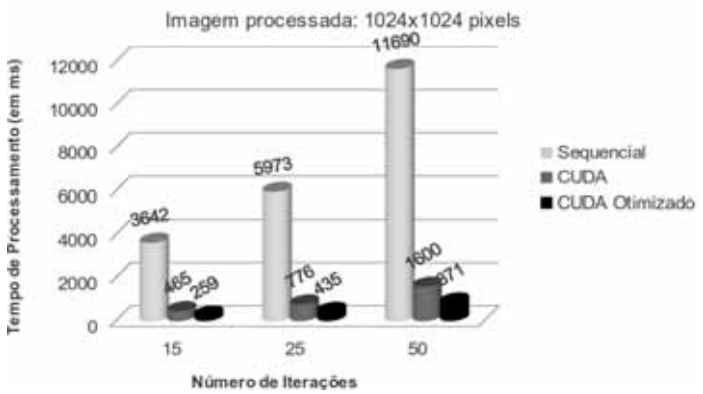


(b)

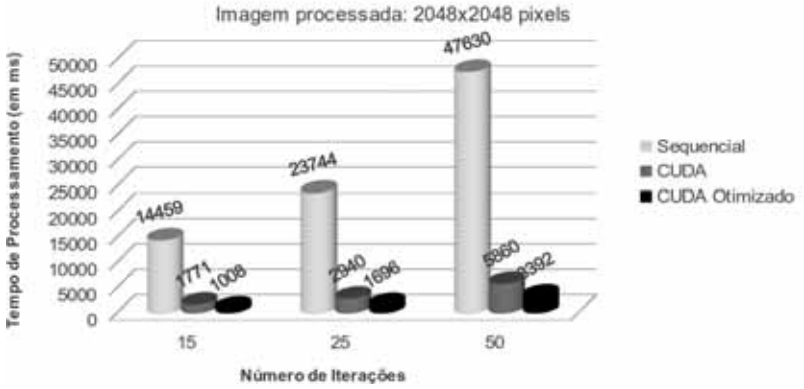
Figura 7.3: *Comparação de Desempenho entre os métodos sequencial e paralelizado.*



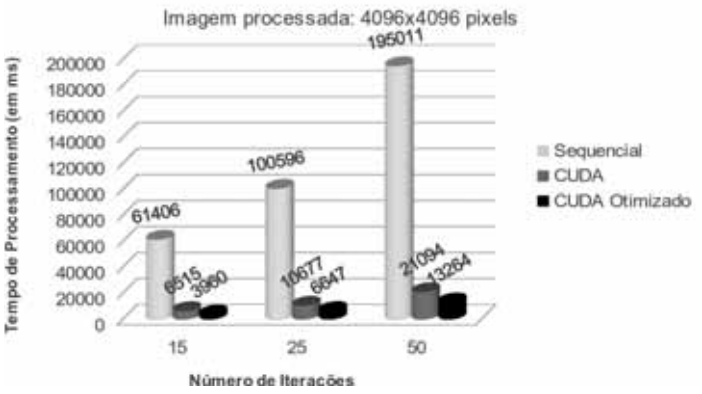
(a)



(b)



(c)



(d)

Figura 7.4: *Comparação de Desempenho entre os métodos sequencial e paralelizado.*

Nota-se pelas figuras, que houve uma redução significativa no tempo de processamento das imagens quando as mesmas foram processadas em GPGPU, principalmente quando a quantidade de iterações foi elevada. As Tabelas 7.2 e 7.3 permitem perceber o *speedup* entre as implementações Sequencial x CUDA e Sequencial x CUDA Otimizado, respectivamente.

Tabela 7.2: *Speedup dos algoritmos: Sequencial x CUDA.*

No. de	Dimensões das Imagens					
Iterações	128x128	256x256	512x512	1024x1024	2048x2048	4096x4096
5	5,0	7,2	8,2	7,9	8,4	9,7
25	5,6	7,1	8,3	7,7	8,2	9,6
50	5,0	7,1	7,8	7,6	8,3	9,3

Os experimentos utilizando otimização em memória de textura demonstram uma redução ainda maior no tempo de processamento do método de suavização paralelizado em GPGPU em relação à implementação sequencial, confirmando a expectativa prevista inicialmente. Os dados são apresentados na Tabela 7.3.

Tabela 7.3: *Speedup dos algoritmos: Sequencial x CUDA Otimizado.*

No. de	Dimensões das Imagens					
Iterações	128x128	256x256	512x512	1024x1024	2048x2048	4096x4096
5	6,4	10,8	13,6	14,1	14,4	15,7
25	7,2	10,7	13,7	13,8	14,1	15,2
50	6,3	10,7	12,8	13,4	14,1	14,8

A utilização da arquitetura CUDA como infraestrutura para a etapa de suavização de imagem mostrou-se uma alternativa viável e capaz de disponibilizar processamento de alto desempenho, possibilitando inclusive, o processamento de aplicações de tempo real com baixo custo (PARK et al., 2011).

7.4 Considerações Finais

O ganho de desempenho do método paralelizado, inicialmente, demonstrou a alta capacidade de processamento disponível na arquitetura adotada. Os recursos disponibilizados nestas placas gráficas, em termos de quantidade de núcleos e memória GDRAM, o modelo de paralelização SIMT associado às técnicas de otimização de memória potencializam o ganho de desempenho quando explorados de maneira eficiente. A combinação adequada deste conjunto

de *features* proporcionou um ganho crescente de desempenho, iniciando em cinco vezes na versão preliminar, e alcançando até quinze vezes na versão otimizada.

A vantagem da versão otimizada é totalmente justificada, pois, cada redução no tempo de processamento, permite minimizar ou até eliminar, limitações impostas em razão do tempo de espera de aplicações das mais variadas áreas. Contudo, ressalta-se a importância de *investir* os esforços exigidos na otimização de implementações paralelizadas, especialmente em arquitetura CUDA.

CAPÍTULO 8

Conclusões e Trabalhos Futuros

A presente pesquisa partiu de um levantamento bibliográfico, sendo realizado estudos e análises sobre arquiteturas paralelas, modelos e técnicas de programação paralela, direcionando a pesquisa para arquiteturas massivamente paralelas em GPGPU e modelos de programação paralela SIMT. Foram realizados estudos acerca de tipos comuns de métodos de suavização de imagens médicas paralelizados, desta maneira, a pesquisa concentrou-se na paralelização em GPGPU de um método de suavização baseado num modelo variacional.

O desenvolvimento do projeto foi realizado em duas etapas, sendo a primeira utilizando a paralelização em GPGPU e memória global, e a última etapa foi a otimização da implementação paralelizada incluindo o uso da memória de textura e acesso coalescido. A proposta foi validada por meio de experimentos, comparando o desempenho entre as implementações paralelizadas em CUDA e sequencial, bem como, foram adotados métodos de análise numérica para aferir a eficiência do método de suavização adotado e implementado.

Técnicas de computação paralela são empregadas no intuito de explorar ao máximo a capacidade de processamento de arquiteturas multiprocessador, no entanto, o custo financeiro para aquisição de *hardware* para computação de alto desempenho não é muito baixo, implicando na busca de alternativas para sua utilização. Desta maneira, a arquitetura GPGPU mostrou-se uma opção de baixo custo sem comprometer o poder de desempenho. A paralelização do método de suavização de imagens baseado em um modelo variacional utilizando arquitetura CUDA, permitiu reduzir cerca de até quinze vezes seu tempo de processamento, em relação à implementação sequencial.

Os experimentos apresentaram o ganho de desempenho do algoritmo implementado em CUDA aplicado em um conjunto de imagens com diferentes resoluções. Os resultados podem ser considerados de grande relevância, permitindo a utilização do método paralelizado em problemas que exigem o processamento em tempo real, bem como imagens com altas resoluções, como demonstrado na seção 7.3.

As pesquisas realizadas por Zheng, Xu e Mueller (2011) e Lourenco (2011) utilizaram a arquitetura CUDA, no entanto, não é possível realizar uma comparação entre as implementação e este projeto em razão de cada uma utilizar estratégias diferentes de otimização de memórias CUDA, bem como, utilizar diferentes tipos memórias em diferentes etapas da

paralelização, influenciando diretamente no desempenho da aplicação.

Diante da variedade de conhecimentos abordados nesta pesquisa, percebeu-se a necessidade de estudos relacionados à técnicas de programação paralela, pois foi primeiro contato com este tema. É importante destacar a popularização de dispositivos equipados com GPUs, e o custo relativamente baixo de placas gráficas com GPGPU, o que pode caracterizar um segmento de mercado promissor para desenvolvedores de *software*, e ainda, continuar impulsionando o desenvolvimento de pesquisas científicas que exigem alto desempenho.

Foram duas as principais contribuições do presente trabalho. A primeira contribuição diz respeito à abordagem de paralelização descrita no Capítulo 7, e ressaltando, possui escalabilidade transparente e portável, graças ao modelo SIMT (NVIDIA, 2010; KIRK; HWU, 2010). A segunda contribuição consiste em disponibilizar um método de redução de ruídos multiplicativo paralelizado em CUDA, oferecendo uma alternativa de alto desempenho para aplicações que utilizam sensores suscetíveis a este tipo de ruído. A implementação do problema abordado neste trabalho, combinado com a arquitetura CUDA, demonstrou a valiosa ferramenta para o processamento de alto desempenho que pode atender não só a comunidade científica mas também, aplicações comerciais.

A produção de ações para divulgação e transferência de informação, tais como publicação de resultados em conferências e revistas nacionais e internacionais foram realizadas, gerando os seguintes trabalhos:

- Método de suavização de imagem baseado num modelo variacional paralelizado em arquitetura CUDA. In: XXXII CSBC. GPU Computing Developer Forum. Curitiba, Paraná, Brasil, 2012.
- O uso de técnicas de paralelização em GPGPU e Multirresolução para a Suavização de Imagem. In: XXV SIBGRAPI - Conference on Graphics, Patterns and Images. Ouro Preto, Minas Gerais, Brasil, 2012.

A presente pesquisa foi desenvolvida como *ponto de partida*, no Grupo de Pesquisas de Sistemas de Tempo Real-UNESP, Câmpus de Bauru, produzindo conhecimentos inerentes à combinação das áreas de Processamento de Imagens e Computação de Alto Desempenho. Como continuidade desta pesquisa, em seguida são apresentadas algumas sugestões quanto a trabalhos futuros:

- desenvolver um *framework* utilizando o modelo proposto;
- a combinação do método paralelizado com aplicações envolvendo a extração de contornos de objetos em tempo real, capturados por sensores do tipo *kinect*;
- otimizar o método para explorar seu funcionamento em múltiplas GPUs;

- utilizar outras tecnologias como OpenMP, MPI, mCUDA ou OpenCL para ampliar as possibilidades de paralelismo do presente método, tornando-o com maior desempenho para outras aplicações.

Referências Bibliográficas

ARAS, Rifat; SHEN, Yuzhong. GPU accelerated stylistic augmented reality. In: *Modeling and Simulation Capstone Conference*. [S.l.: s.n.], 2010.

AUBERT, Gilles; AUJOL, Jean-Francois. A variational approach to removing multiplicative noise. *SIAM Journal on Applied Mathematics*, SIAM, v. 68, n. 4, p. 925–946, 2008. Disponível em: <http://link.aip.org/link/?SMM/68/925/1>.

BANDEIRA, Carlos Igor Ramos. *Análise Comparativa de Filtros Adaptativos de Ruído Speckle*. Monografia de Graduação. Fortaleza, Dezembro 2009.

BORGO, Rita; BRODLIE, Ken. *State of the Art Report on GPUs*. School of Computing, February 2009. Visualization and Virtual Reality Research Group.

CHAPMAN, Barbara et al. *Parallel Computing: From Multicores and GPUs to Petascale*. IOS Press BV, 2010. 760 p. ISBN 978-1-60750-530-3. Disponível em: <http://www.booksonline-.iospress.nl/>.

CHEN, L.D.; ZHANG, M.J.; XIONG, Z.H. Series-parallel pipeline architecture for high-resolution catadioptric panoramic unwrapping. *IET-Image Processing*, v. 4, n. 5, p. 403–412, 2010.

CHEN, Qiang; SUN, Quan sen; XIA, De shen. Homogeneity similarity based image denoising. *Pattern Recognition*, v. 43, n. 12, p. 4089 – 4100, 2010. ISSN 0031-3203. Disponível em: <http://www.sciencedirect.com/science/article/pii/S0031320310003419>.

DASH, Ratnakar; SA, Pankaj Kumar; MAJHI, Banshidhar. Restoration of images corrupted with blur and impulse noise. In: *Proceedings of the 2011 International Conference on Communication, Computing & Security*. New York, NY, USA: ACM, 2011. (ICCCS '11), p. 377–382. ISBN 978-1-4503-0464-1. Disponível em: <http://doi.acm.org/10.1145/1947940-.1948019>.

DUNCAN, Ralph. A survey of parallel computer architectures. *Computer*, v. 23, n. 2, p. 5 –16, feb 1990. ISSN 0018-9162.

ECKERT, Michael P.; BRADLEY, Andrew P. Perceptual quality metrics applied to still image compression. *Signal Processing*, Elsevier North-Holland, Inc., v. 70, n. 3, p. 177 – 200,

1998. ISSN 0165-1684. Disponível em: <http://www.sciencedirect.com/science/article/pii-/S0165168498001248>.

EL-REWINI, Hesham; ABD-EL-BARR, Mostafa. *Advanced Computer Architecture and Parallel Processing*. [S.l.]: Wiley-Interscience, 2005. (Wiley Series on Parallel and Distributed Computing, ISBN 0-471-46740-5).

GEBALI, Fayez. *Algorithms and Parallel Computing*. [S.l.]: John Wiley & Sons, 2011. 365 p. ISBN 978-0-470-90210-3.

GHITA, Ovidiu; WHELAN, Paul F. A new GVF-based image enhancement formulation for use in the presence of mixed noise. *Pattern Recognition*, v. 43, n. 8, p. 2646 – 2658, 2010. ISSN 0031-3203. Disponível em: <http://www.sciencedirect.com/science/article/pii-/S0031320310001020>.

GONZALEZ, Rafael; WOODS, Richard. *Digital Image Processing*. 2nd. ed. [S.l.]: Prentice Hall, 2001. ISBN 0-201-18075-8.

GULO, C. A. S. J. et al. Método de suavização de imagem baseado num modelo variacional paralelizado em arquitetura CUDA. In: XXXII CSBC. *GPU Computing Developer Forum*. Curitiba, Paraná, Brasil, 2012.

GURD, J.R. A taxonomy of parallel computer architectures. In: *Design and Application of Parallel Digital Processors, 1988., International Specialist Seminar on the*. [S.l.: s.n.], 1988. p. 57 –61.

HENNESSY, John L.; PATTERSON, David A. *Computer Architecture A Quantitative Approach*. 4th. ed. [S.l.]: Elsevier, 2007. 705 p.

HILL, Mark D.; MARTY, Michael R. Amdahl's law in the multicore era. *Computer*, v. 41, n. 7, p. 33 –38, july 2008. ISSN 0018-9162.

HOFMANN, Markus; BINNA, Tobias. *Massive Parallel Image Processing*. [S.l.], 2010. Monografia de Graduação.

HUANG, Yu-Mei; NG, Michael K.; WEN, You-Wei. A new total variation method for multiplicative noise removal. *SIAM J. Img. Sci.*, Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, v. 2, n. 1, p. 20–40, jan. 2009. ISSN 1936-4954. Disponível em: <http://dx.doi.org/10.1137/080712593>.

HWANG, Kai. Advanced parallel processing with supercomputer architectures. *Proceedings of the IEEE*, v. 75, n. 10, p. 1348 – 1379, oct. 1987. ISSN 0018-9219.

HWU, Wen mei. *GPU Computing GEMS*. Emerald edition. [S.l.]: Morgan Kaufmann and NVIDIA, 2011. 889 p. ISBN 978-0-12-384988-5.

JANG, Honghoon; PARK, Anjin; JUNG, Keechul. Neural network implementation using CUDA and OpenMP. In: *Proceedings of the 2008 Digital Image Computing: Techniques and Applications*. Washington, DC, USA: IEEE Computer Society, 2008. p. 155–161. ISBN 978-0-7695-3456-5. Disponível em: <http://portal.acm.org/citation.cfm?id=1469127.1470322>.

JIN, Zhengmeng; YANG, Xiaoping. A variational model to remove the multiplicative noise in ultrasound images. *Journal of Mathematical Imaging and Vision*, Kluwer Academic Publishers, v. 39, n. 1, p. 62–74, 2011.

KIRK, David; HWU, Wen-Mei. *Programming Massively Parallel Processors: A Hands-on Approach*. [S.l.]: Elsevier, 2010. 75 p. ISBN 978-0-12-381472-2.

KRISSIAN, K. et al. Speckle-constrained filtering of ultrasound images. In: *Computer Vision and Pattern Recognition, 2005. CVPR 2005. IEEE Computer Society Conference on*. [S.l.: s.n.], 2005. v. 2, p. 547 – 552 vol. 2. ISSN 1063-6919.

KURZAK, Jakub; BADER, David A.; DONGARRA, Jack. *Scientific Computing with Multicore and Accelerators*. [S.l.]: CRC Press, 2010. 474 p. (Computational Science, ISBN 978-1-4398-2536-5).

LOURENCO, Luis Henrique Alves. *Paralelização do detector de bordas Canny para a biblioteca ITK utilizando CUDA*. Dissertação (Mestrado) — Universidade Federal do Paraná, Curitiba, Abril 2011.

MERIGOT, Alain; PETROSINO, Alfredo. Parallel processing for image and video processing: Issues and challenges. In: *Parallel Computing*. [S.l.]: Else, 2008. p. 694–699.

NVIDIA. *GPU Tutorial: Build environment, Debugging/Profiling, Fermi, Optimization/-CUDA 3.1 and Fermi advice*. [S.l.], 2010.

PAGE, Daniel. *A Practical Introduction to Computer Architecture*. [S.l.]: Springer, 2009. ISBN 978-1-84882-255-9.

PARHAMI, Behrooz. *Introduction to Parallel Processing: Algorithms and Architectures*. [S.l.]: Kluwer Academic Publishers, 2002. (Plenun Series in Computer Science, ISBN: 0-306-45970-1).

PARK, In Kyu et al. Design and performance evaluation of image processing algorithms on GPUs. *IEEE Transactions on Parallel and Distributed Systems*, IEEE Press, Piscataway, NJ, USA, v. 22, n. 1, p. 91–104, jan. 2011. ISSN 1045-9219. Disponível em: <http://dx.doi.org-/10.1109/TPDS.2010.115>.

PODLOZHNYUK, Victor. *Image convolution with CUDA*. [S.l.], June 2007. Disponível em: <developer.download.nvidia.com/assets/cuda/files/convolutionSeparable.pdf>.

RAMPONI, Giovanni et al. Nonlinear unsharp masking methods for image contrast enhancement. *J. Electronic Imaging*, v. 5, n. 3, p. 353–366, 1996.

RUDIN, Leonid I.; OSHER, Stanley; FATEMI, Emad. Nonlinear total variation based noise removal algorithms. *Physica D: Nonlinear Phenomena*, v. 60, n. 1 - 4, p. 259 – 268, 1992. ISSN 0167-2789. Disponível em: <http://www.sciencedirect.com/science/article/pii-/016727899290242F>.

SANDERS, Jason; KANDROT, Edward. *CUDA by example: An introduction to General-Purpose GPU programming*. [S.l.]: Addison-Wesley, 2011. 311 p. ISBN 978-0-13-138768-3.

SCHMEISSER, Martin et al. Parallel, distributed and GPU computing technologies in single-particle electron microscopy. *Acta Crystallographica Section D*, v. 65, n. 7, p. 659–671, Jul 2009. Disponível em: <http://dx.doi.org/10.1107/S0907444909011433>.

SCHNORR, Lucas Mello. Paralelização do filtro de mediana. Pesquisa em Disciplina de Pós-Graduação. 2012.

SCHONUNG, Gabriel. Visual geometric environment modeling for augmented reality and robotics. *Advances in Media Technology*, 2011.

SEVERINO, Antonio Joaquim. *Metodologia do Trabalho Científico*. 23. ed. São Paulo: Cortez, 1996. 304 p.

STALLINGS, William. *Computer Organization and Architecture: Designing for Performance*. 6th. ed. [S.l.]: Pearson Education International, 2003. 826 p. ISBN 0-13-049307-4.

TANENBAUM, Andrew S. *Sistemas Operacionais Modernos*. [S.l.]: Pearson Education do Brasil, 2010. 672 p. ISBN 9788576052371.

VADJA, Andras. *Programming Many-Core Chips*. [S.l.]: Springer, 2011. 241 p. ISBN: 978-1-4419-9738-8.

WANG, Zhou; BOVIK; LU, Ligang. Why is image quality assessment so difficult? In: . [s.n.], 2002. v. 4, p. IV. Disponível em: <http://dx.doi.org/10.1109/ICASSP.2002% -.1004620>.

WANG, Zu et al. Image quality assessment: from error visibility to structural similarity. *IEEE Transactions on Image Processing*, v. 13, n. 4, p. 600–612, 2004.

WURM, Kai M. et al. OctoMap: A probabilistic, flexible, and compact 3D Map representation for robotic systems. In: *In Proc. of the ICRA 2010 workshop*. [S.l.: s.n.], 2010.

YANG, J. et al. Image and video denoising using adaptative dual-tree discrete wavelet packets. *IEEE Transactions on Circuits and Systems for Video Technology*, v. 19, n. 5, p. 642–655, 2009.

YI, Jin Jing. *Avaliação de desempenho de filtros redutores de speckle em imagens de ultra-som*. [S.l.], 1999. Monografia de Graduação. Disponível em: <http://di.ufpe.br/~tg/1999-2>.

ZHENG, Ziyi; XU, Wei; MUELLER, Klaus. Performance tuning for CUDA-accelerated neighborhood denoising filters. In: *Workshop on High-Performance Image Reconstruction (HPIR)*. [S.l.: s.n.], 2011.