Universidade Federal da Paraíba

Centro de Tecnologia

Programa de Pós-Graduação em Engenharia Mecânica

Estudo para Otimização do algoritmo Non-local means visando aplicações em tempo real

Hamilton Soares da Silva

Tese de Doutorado apresentada à Universidade Federal da Paraíba

João Pessoa, Julho de 2014

Hamilton Soares da Silva

Estudo para Otimização do algoritmo Non-local means visando aplicações em tempo real

Tese de Doutorado apresentada ao Programa de Pós-Graduação em Engenharia Mecânica da Universidade Federal da Paraíba, em cumprimento às exigências para obtenção do Grau de Doutor.

Área de concentração: Dinâmica e Sistemas de Controle
Orientador: Professor Doutor Francisco Antonio Belo

João Pessoa, Julho de 2014

S586e Silva, Hamilton Soares da.

Estudo para otimização do algoritmo Non-local means visando aplicações em tempo real / Hamilton Soares da Silva.- João Pessoa, 2014.

92f. : il.

Orientador: Francisco Antonio Belo

Tese (Doutorado) – UFPB/CT

1. Engenharia mecânica. 2. Processamento digital de imagens. 3. Redução de ruído. 4. Computação reconfigurável. 5. Computação paralela. 6. Programação GPU CUDA.

ESTUDO PARA OTIMIZAÇÃO DO ALGORITMO NON-LOCAL MEANS VISANDO APLICAÇÕES EM TEMPO REAL

por

HAMILTON SOARES DA SILVA

Tese aprovada em 25 de julho de 2014

Prof. Dr. FRANCISCO ANTONIO BELO
Orientador

Prof. Dr. CARLOS ANTONIO CABRAL DOS SANTOS
Examinador Interno

Prof. Dr. ABEL CAVALCANTE LIMA FILHO
Examinador Interno

Prof. Dr. JOSÉ ANTONIO GOMES DE LIMA
Examinador Externo

Prof. Dr. LEONARDO VIDAL BATISTA
Examinador Externo

João Pessoa – PB *2014*

Agradecimentos

Ao Professor Doutor Francisco Antonio Belo pela orientação e pela oportunidade de transformar um sonho em realidade.

Ao Professor Doutor Carlos Antonio Cabral dos Santos pela participação na banca e pelas sugestões apresentadas para a melhoria da redação final do trabalho.

Ao Professor Doutor Abel Cavalcante Lima Filho pela participação na banca e pelas sugestões apresentadas para a melhoria da redação final do trabalho.

Ao Professor José Antonio Gomes de Lima pela participação na banca e pela parceria e companheirismo na realização de diversos trabalhos ao longo de mais de quarenta anos de amizade.

Ao Professor Doutor Leonardo Vidal Batista, referência no ensino e pesquisa na área de Processamento de Imagens, pela participação na banca e pelas sugestões apresentadas para o desenvolvimento do trabalho.

À Raimundo, Gustavo, Hélio e demais colegas de trabalho pelo incentivo.

Aos alunos de graduação Bruno Maia de Moraes, Ruan Delgado Gomes, Sidney Pontes Filho e Bruno Mascarenhas Pinto pela importante participação no desenvolvimento do trabalho.

Aos alunos de Pós-Graduação Lucas Lucena Gambarra, Daniel Soares e Marques e Leandro de Figueiredo Alves pela importante participação no desenvolvimento do trabalho.

Aos meus pais, João e Iva, pelo amor, educação e formação proporcionada.

Aos meus irmãos Antonio e Amilcar pelo apoio e amizade ao longa da vida.

Aos meus filhos Renan e Raíssa pelo amor e incentivo.

À Lydia, minha esposa à mais de trinta anos, pelo amor e apoio em todas as minhas atividades.

Resumo

O propósito deste trabalho é estudar o algoritmo *non-local means*(NLM) e propor técnicas para otimizar e implementar o referido algoritmo visando sua aplicação em tempo real. Ao todo são sugeridas duas alternativas de implementação. A primeira trata do desenvolvimento de uma placa aceleradora para computadores que possuam Barramento PCI, contendo um hardware especializado que implementa o Filtro NLM. A segunda implementação utiliza o ambiente densamente multiprocessado GPU, existente nas controladoras de vídeo. As duas propostas aceleraram significativamente o algoritmo NLM, mantendo a mesma qualidade visual das implementações tradicionais em software, tornando possível sua utilização em tempo real. A filtragem de ruídos é uma área importante para o processamento digital de imagens, sendo cada vez mais utilizada devido as melhorias dos novos equipamentos de captação, e o consequente aumento da resolução da imagem, que favorece o aparecimento dessas perturbações. Ela é amplamente estudada nos campos de tratamento de imagens, visão computacional e manutenção preditiva de subestações elétricas, motores, pneus, instalações prediais, tubos e conexões, focando em reduzir os ruídos sem que se remova os detalhes da imagem original. Várias abordagens foram propostas para filtragem de ruídos, uma delas é o método não-local, chamado de *Non-Local Means* (NLM), que não só utiliza as informações locais, mas a imagem inteira, destaca-se como o estado da arte, porém, há um problema neste método, que é a sua alta complexidade computacional, que o torna praticamente inviável de ser utilizado em aplicações em tempo real, até mesmo para imagens pequenas.

Palavras chave Processamento Digital de Imagens; Redução de Ruído; Computação Reconfigurável; Computação Paralela; Programação GPU CUDA.

Abstract

The aim of this work is to study the non-local means algorithm and propose techniques to optimize and implement this algorithm for its application in real-time. Two alternatives are suggested for implementation. The first deals with the development of an accelerator card for computers, which has a PCI bus containing specialized hardware that implements the NLM filter. The second implementation uses densely GPU multiprocessor environment, which exists in the parent video. Both proposals significantly accelerates the NLM algorithm, while maintains the same visual quality of traditional software implementations, enabling real-time use. Image denoising is an important area for digital image processing. Recently, its use is becoming more popular due to improvements of of the new acquisition equipments and, thus, the increase of image resolution that favors the occurrence of such perturbations. It is widely studied in the fields of image processing, computer vision and predictive maintenance of electrical substations, motors, tires, building facilities, pipes and fittings, focusing on reducing the noise without removing details of the original image. Several approaches have been proposed for filtering noise. One of such approaches is the non-local method called Non-Local Means (NLM), which uses the entire image rather than local information and stands out as the state of the art. However, a problem in this method is its high computational complexity, which turns its application almost impossible in real time applications, even for small images.

Key words Digital Image Processing; Image Denoising; Reconfigurable Computing; Parallel Computing; Programming CUDA GPU.

Lista de Equações

Lista de Tabelas

Lista de Figuras

Sumário

Capítulo I: Introdução

O presente trabalho propõe implementações em hardware e software para a otimização do algoritmo de remoção de ruídos pelo método *Non-Local Means*, utilizado em processamento digital de imagens, visando aplicações em tempo real.

1.1. Motivação

A importância da remoção de ruídos em aplicações na área de Engenharia, em especial da Engenharia Mecânica, se deve a natureza dos métodos de aquisição, digitalização e transmissão de sinais digitais de imagens que naturalmente inserem ruídos que devem ser retirados para que se possa trabalhar com o sinal o mais próximo possível da informação original.

A utilização de imagens captadas por câmaras digitais especializadas tem se tornado uma excelente ferramenta na manutenção preditiva de subestações elétricas, motores, pneus, instalações prediais, tubos e conexões, etc. Por exemplo, a Figura 1 mostra a imagem original de uma conexão e, ao seu lado, a imagem captada por uma câmara de infravermelho, que exibe detalhes de aquecimento da peça. Observa-se na imagem uma área de emissividade, ilustrada pela seta verde, que sinaliza para o observador que a conexão apresenta oxidação (Santos, Laerte dos, 2006).

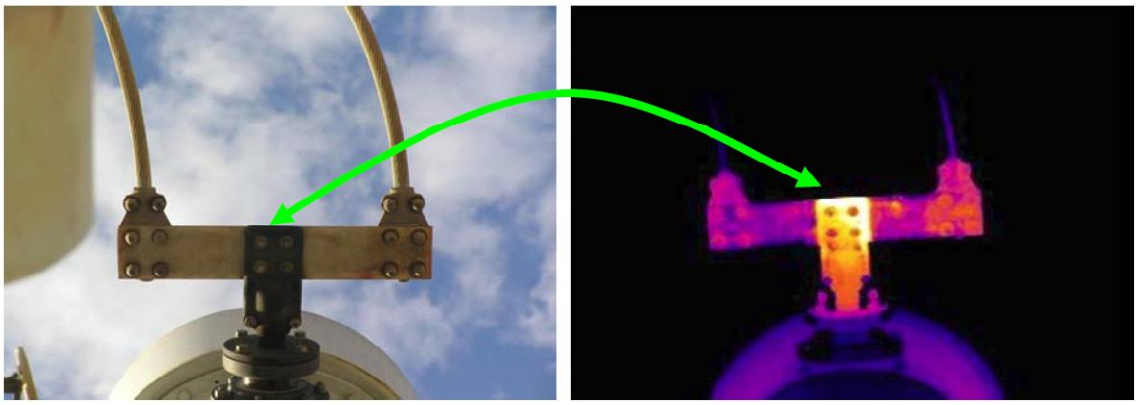

Figura 1: Conexão elétrica apresentando maior emissividade em área oxidada. (Santos, Laerte dos, 2006)

Outro exemplo de uso importante de imagens digitais especializadas é na identificação de infiltrações em lajes como ilustra a Figura 2.

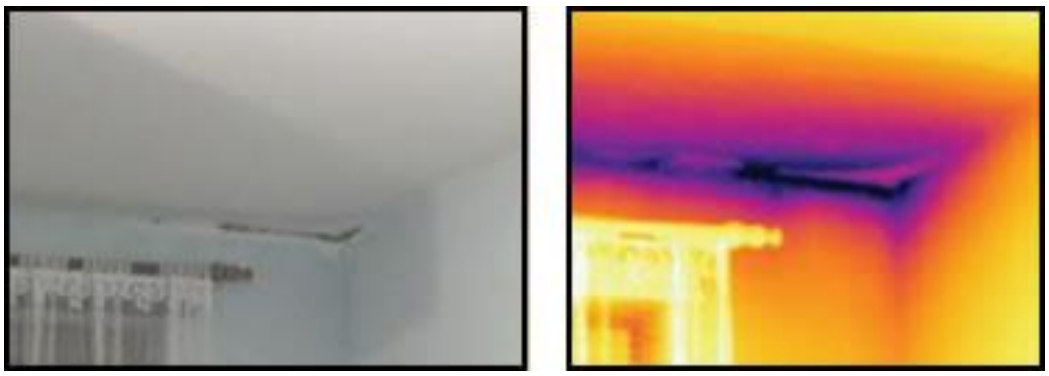

Figura 2: Imagem de termografia de infiltração por água(tons azuis) em laje. Fonte: http://www.gerenciall.com.br.

A aquisição e o processamento de sinais que antes eram feitos apenas através de circuitos eletrônicos analógicos passaram a ser feitos, devido a evolução da microeletrônica, em processadores digitais tornando mais fácil e flexível a sua manipulação. O uso de processadores digitais originou uma série de novos desafios, alguns originados da era digital, outros herdados da era analógica. Um destes desafios consiste em contornar as distorções causadas por ruído, que é um produto inerente a qualquer forma de processamento de sinal, em particular, de imagem. Os ruídos geralmente corrompem a imagem durante a aquisição e transmissão e normalmente degradam sua qualidade. Aplicações relacionadas com imagem necessitam que os ruídos sejam removidos para produzir resultados confiáveis, como por exemplo: na área aeroespacial; análises de imagens médicas; detecção de objetos; análise sísmica; visão computacional; robótica; imagens de tomografia e ultrassonografia para detecção de fluídos. Em todas essas aplicações são utilizadas técnicas de filtragem para remoção de ruídos.

Por outro lado, a filtragem é muitas vezes necessária como um pré-processamento para outras tarefas, tais como: compressão, segmentação e reconhecimento (Coupé, Yger, & Barillot, 2006). Por esses motivos, a filtragem tem sido um dos problemas mais importantes e amplamente estudados em tratamento de

imagem e visão computacional. O objetivo é remover o ruído efetivamente, preservando os detalhes da imagem original tanto quanto possível. O filtro ótimo é aquele que remove o ruído mantendo a estrutura da imagem, como níveis e bordas, onde bordas são as descontinuidades da imagem. Até agora, vários métodos têm sido propostos para remover o ruído e recuperar a imagem verdadeira. Em boa parte deles se observa que a remoção de ruído é feita basicamente calculando-se a média, que pode ser realizada localmente, dentre os quais se destacam:

・Filtro Gaussiano(Lindenbaum et al, 1994) _ É um exemplo de Modelo Linear e tem sido comumente utilizado para reduzir ruído. O Filtro gaussiano se baseia no calculo entre a convolução da imagem e um núcleo gaussiano. Os filtros gaussianos executam bem sua tarefa em regiões planas (ou seja, em que há pouca variação) das imagens. A principal desvantagem é que eles não são capazes de preservar as bordas adequadamente;

・Filtro Anisotrópico(Perona-Malik,1990) _ A remoção de ruído anisotrópica se baseia na preservação de características importantes da imagem como bordas e quinas pela aplicação de suavização dependente da direção. O algoritmo proposto por Perona-Malik se baseia na geometria com o uso dos diversos conceitos de curvatura para estabelecer a detecção e medição da perturbação geométrica de uma imagem. Portanto, a filtragem de ruído em uma imagem é realizada removendo picos de curvatura não desejados enquanto simultaneamente preserva características relevantes da distribuição da curvatura. A vantagem do Filtro é a preservação das bordas e quinas com a desvantagem de degradação da região plana e da textura;

・ Filtro de Variação Total (Rudin et al,1992) _ É um exemplo de Modelo não-linear. Filtros não-lineares podem preservar as bordas bem mais adequadamente que os lineares. O Filtro se baseia na suposição que a imagem é derivada de uma descrição geométrica simples, conjunto de conjuntos conectados, os objetos resultantes são suavizados dentro de cada conjunto, mas com saltos através das fronteiras. Como no

Filtro Anisotrópico as bordas são preservadas, mas existe degradação na região plana e na textura;

- Filtragem Bilateral (Tomasi e Manduchi, 1998) _ É um exemplo de Filtragem por vizinhança (*Neighborhood filtering*) e se baseia na restauração de um ponto, tomando uma média dos valores de seus vizinhos. Ele emprega a continuidade das intensidades para os pontos em uma vizinhança local, esses métodos consideram apenas a influência dos pontos na vizinhança centrada ao redor de um ponto quando o ajustam.

Existem muitos padrões de repetição em imagens naturais, é uma visão simplista considerar apenas a região local. Portanto, não se deve utilizar apenas uma região local pequena para calcular o valor de cada pixel, mas sim a imagem inteira ou a maior parte dela.

Foi pensando dessa maneira que (Buades, Coll, & Morel, 2005) desenvolveram um algoritmo não local para redução de ruídos em imagens, o qual utiliza a informação codificada em toda a imagem para a filtragem. Para calcular o valor final de cada pixel, o algoritmo primeiro calcula a semelhança entre uma vizinhança fixa centrada em torno dele e as vizinhanças centradas em torno dos outros pixels da imagem inteira. Em seguida, toma a semelhança como peso para ajustar esse pixel através de uma média ponderada.

Em (Buades, Coll, & Morel, 2005), pode ser observada uma comparação relativa à qualidade de vários outros métodos com o algoritmo *Non-local means*, as quais ficam mais claras através da comparação da imagem original, Figura 3, com as imagens obtidas da filtragem com outros métodos exibidas na Figura 4.

Figura 3: Imagem original (Buades, Coll, & Morel, 2005).

Como pode ser observado na Figura 4, o *NLM* apresenta resultados convincentes e foi considerado como o mais eficiente na remoção de ruídos (SHAHAM, 2007), mas a eficiência em tempo é baixa. Para uma imagem com N^2 pixels e utilizando vizinhanças de comparação de tamanho M^2, a complexidade computacional do algoritmo original é $O(M^2 x N^2)$. Desta forma, fica evidente que a alta complexidade computacional torna o NLM inviável em aplicações em tempo real, uma vez que muitas aplicações possuem requisitos não funcionais de desempenho, que são aqueles que se referem à velocidade de operação do sistema. De acordo com (Sommerville & Kotonya, 1998) diferentes tipos de requisitos podem ser especificados, são eles:

- Tempo de Resposta, que especifica o tempo de resposta aceitável do ponto de vista do cliente para que alguma operação seja concluída;
- *Throughput*, que especifica a quantidade de dados que precisam ser processados em um intervalo de tempo pré-definido;
- Temporização, que especifica quão rápido o sistema precisa coletar uma entrada proveniente de sensores antes que sejam sobrescritos por outras entradas da mesma natureza, ou quão rápido o sistema precisa realizar o processamento para que seja a entrada de outro subsistema.

Os próprios idealizadores do *NLM* (Buades, Coll, & Morel, 2005) sugeriram utilizar apenas uma janela (*NLM* em janela) de pesquisa em vez de toda a imagem

para realizar a filtragem. Porém, embora reduzindo a complexidade, o tempo de execução continuou elevado.

Nesse contexto, é desejável uma alternativa para reduzir o tempo de execução do *NLM*. Segundo (Gonzales & Woods, 2000), embora a maioria das funções de processamento de imagens possam ser implementadas em *software*, a necessidade de velocidade em algumas aplicações é razão para utilizar *hardware* especializado ou ambientes densamente multiprocessado.

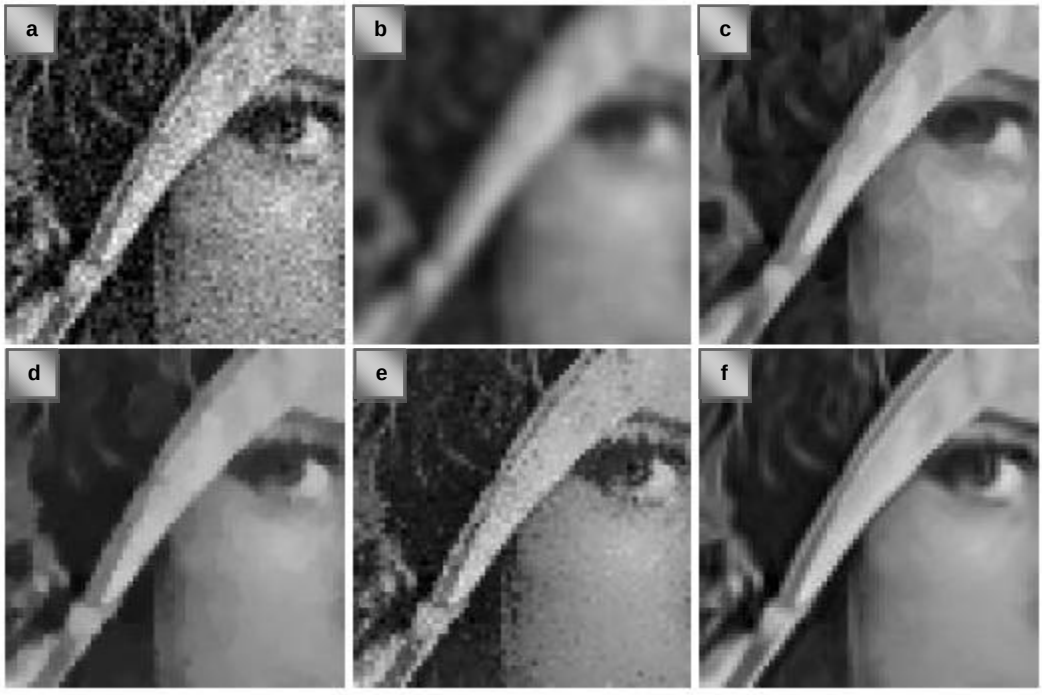

Figura 4: (a) imagem com um ruído com desvio-padrão de 20; (b) filtragen Gaussiana; (c) filtragem anisotrópica; **(d) filtragem por variação total; (e) filtragem Bilateral; (f) algoritmo *NLM*. Comparação entre filtragens em uma imagem digital (Buades, Coll, & Morel, 2005).**

1.2. Objetivos

A proposta deste trabalho é estudar o algoritmo non-local means sugerido em (BUADES, COLL e MOREL,2005) e propor técnicas para otimizar e implementar o referido algoritmo visando sua aplicação em tempo real. Ao todo são sugeridas duas alternativas de implementação.

A primeira trata do desenvolvimento de uma placa aceleradora para computadores que possuam Barramento PCI, O software, presente no computador *host*, fica encarregado de transferir a imagem e passar parâmetros para a Memória da Placa Aceleradora PCI. O mesmo deverá efetuar suas atividades e se comunicar com o hardware especializado para este continuar com o processamento do filtro. O hardware especializado fica embutido na Placa Aceleradora PCI, e sua comunicação com o software se dá mediante a interface PCI do computador *host*. As atividades que o hardware efetua compreende o processamento do algoritmo NLM, bem como a interface com a Memória presente na Placa Aceleradora. A segunda alternativa de implementação é a de utilizar o ambiente densamente multiprocessado CUDA, existente nas controladoras de vídeo da NVIDIA.

1.3. Organização

O presente documento esta organizado da seguinte forma: O capítulo II apresenta a fundamentação teórica necessária para o desenvolvimento do trabalho abordando os seguintes tópicos: Conceitos sobre Imagem digital, Processamento Digital de Imagens, Algoritmo *Non Local Means*, Computação Reconfigurável, Arquiteturas Massivamente Paralelas e GPU's; O capítulo III apresenta o sistema proposto da Placa Aceleradora para Otimização do algoritmo *Non Local Means*; O capítulo IV apresenta a proposta de utilizar placas gráficas com GPU's para otimização do algoritmo *Non Local Means*; O capítulo V apresenta os Resultados Obtidos; Finalmente o capítulo VI apresenta a conclusão da dissertação fazendo considerações sobre o trabalho.

Capítulo II: Fundamentação Teórica

Este capítulo apresenta o embasamento teórico necessário para o desenvolvimento do trabalho apresentando Conceitos sobre Imagem Digital, Processamento Digital de Imagens, Algoritmo *Non Local Means*, Computação Reconfigurável, Arquiteturas Massivamente Paralelas e GPU's.

2.1 Conceitos sobre Imagem Digital

A imagem é considerada como uma representação gráfica de um objeto real, podendo ser considerada como uma distribuição de energia em uma posição espacial ou, matematicamente, pode ser expressa como uma função bidimensional da energia em cada ponto $(x, y) \subset R^2$.

$$f(x, y) = i(x, y)r(x, y) \tag{2.1}$$

onde

$$0 < i(x, y) < \infty \tag{2.2}$$

e

$$0 < r(x, y) < 1 \tag{2.3}$$

Assim $i(x, y)$ é a função de energia (quantidade de luz, radiação, etc., dependendo dos tipos de sensores, que incidem na cena observada) e $r(x, y)$ é a função de reflexão, absorção ou transparência do objeto. A função contínua $f(x, y)$ descreve a energia da imagem na coordenada (x, y).

A imagem digital é o resultado da conversão analógico-digital de uma imagem e como tudo que é digitalizado é formada por bits 1 e 0. Para um sistema digital ela consiste em um arranjo bidimensional de valores que representam as características energéticas de objetos em uma cena, sendo cada um destes elementos chamado de *PIXEL* (*picture element*). Em imagens digitais coloridas, cada ponto (*pixel*) é composto por três valores representando a quantidade de cada uma das três cores primárias

vermelho, verde, azul (padrão RGB) ou ciano, magenta e amarelo (padrão CMY), que são combinadas para a formação de uma nova cor. Em imagens monocromáticas, cada *pixel* conterá apenas um valor de 8 bits, representando o nível de cinza ou a intensidade de iluminação. Portanto, é na imagem em sua forma digital que são aplicadas as transformações, que chamamos de Processamento Digital de Imagens. Deste modo, dada uma imagem como entrada, ela será processada por procedimentos geralmente expressos de forma algorítmica e cujo resultado de saída será obtida uma nova imagem, que é a imagem processada. Dessa forma, segundo (GONZALES e WOODS, 2000), a maioria das funções de processamento de imagens pode ser implementada em software. A única razão para se utilizar hardware especializado em processamento de imagens é a necessidade de velocidade em algumas aplicações ou para vencer algumas limitações fundamentais da computação.

A filtragem, ou remoção de ruídos, em imagens é uma área importante do Processamento Digital de Imagens. Mesmo com a tecnologia atual, as imagens muitas vezes são prejudicadas com ruído, por isso a filtragem é necessária para recuperar a qualidade da imagem o mais próximo possível da imagem original.

2.2 Processamento Digital de Imagens

Com a evolução dos sistemas digitais sua utilização foi sendo expandida para diversas áreas. O processamento de sinais, antes efetuado apenas através de métodos analógicos, passou a ser uma das áreas do conhecimento a utilizar o processamento digital na resolução dos seus problemas. Ao utilizarmos o processamento de sinais digitais no tratamento de sinais pictóricos, ou seja, imagens digitais, estamos pondo em evidência um sinal que pode ser representado por uma função bidimensional capaz de ser processada por um computador. Essa área do conhecimento chamamos de Processamento de Imagens e é aplicada a qualquer computação na qual a entrada é uma imagem, que será processada por procedimentos geralmente expressos de forma algorítmica, e cujo resultado de saída

será uma nova imagem que é a imagem processada. Dessa forma, segundo (GONZALES e WOODS, 2000), a maioria das funções de processamento de imagens pode ser implementada em software. A única razão para hardware especializado para processamento de imagens é a necessidade de velocidade em algumas aplicações ou para tornar viáveis outras em tempo real devido a complexidade de alguns. A área de processamento de imagens digitais está evoluindo continuamente e embora as áreas de aplicação sejam diversas, tais problemas comumente convergem para os métodos capazes de melhorar a informação visual para a análise e interpretação humanas (GONZALES e WOODS, 2000).

2.2.1 Ruídos em imagens digitais

O ruído pode ser definido como uma perturbação na imagem. Em imagens reais a captação é responsável por essa perturbação, uma vez que os sensores de imagem devem contar fótons, e sendo o número de fótons contados uma quantidade aleatória, as imagens muitas vezes têm ruído de contagem (ou quantização) de fótons, especialmente em situações de pouca luz. O ruído granular em filmes fotográficos é, por vezes, modelado como uma distribuição Gaussiana e outras vezes como uma distribuição de Poisson. Muitas imagens são corrompidas por ruído sal e pimenta (*salt and pepper*), que é como se alguém tivesse polvilhado pontos pretos e brancos na imagem. O ruído Gaussiano é uma parte de quase todos os sinais.

O ruído pode ser definido da seguinte forma: dada uma imagem $v = \{v(i) \mid i \in I\}$, então ela pode ser decomposta em um componente original u e outra de ruído n. A decomposição mais comum é a aditiva:

$$v(i) = u(i) + n(i) \quad \textbf{(1)}$$

O ruído Gaussiano é frequentemente considerado um componente aditivo. A segunda decomposição mais comum é a multiplicativa:

$$v(i) = u(i) * n(i) \quad \textbf{(2)}$$

O modelo multiplicativo pode ser transformado num modelo aditivo usando logaritmos e um modelo aditivo pode ser transformado em um multiplicativo usando exponenciação (Bovik, 2005).

A redução da componente de ruído na composição aumenta a qualidade da imagem. Grande parte dos métodos para se eliminar ruído de imagens, descritos de forma algorítmica, depende de um parâmetro de filtragem *h* que é determinado como uma função do desvio padrão do ruído. Desse modo, a imagem final poderá ser descrita como a combinação de dois componentes, a imagem suavizada e a componente de ruído (SHAHAM,2007).

$$v = D_h v + n(D_v, v) \quad \textbf{(3)}$$

onde:

• D_h:método de remoção do ruído;

• $n(D_h,v)$: parte da imagem original que o método reconheceu como ruído;

• D_h v: imagem *v* suavizada através do método D_h.

O ideal é que a componente $D_h v$ seja mais suave que *v*. A figura 5 mostra o exemplo de uma imagem (*v*), seu componente de ruído (*n*) e seu componente filtrado ($D_h v$).

(a) (b) (c)

Figura 5: (a) Imagem inicialmente com ruído (v), (b) Componente de ruído da imagem (n), (c) Imagem filtrada ($D_h v$).(Imagem original de 1972 da revista Playboy norte-americana.)

De acordo com (BUADES, COLL e MOREL, 2004), um bom modo para se calcular a quantidade de ruído em uma imagem é utilizar a razão *sinal/ruído* (SNR), que pode ser definida pela Equação (4).

$$\boldsymbol{SNR = \frac{\sigma(v)}{\sigma(n)} onde\ \sigma(v) = \sqrt{\frac{1}{|I|}\sum_{i \epsilon I}(v(i) - \overline{v})^2}} \qquad \textbf{(4)}$$

O melhor modo de se verificar o efeito de um ruído numa imagem digital é adicionar ruído branco gaussiano, cujo *n(i)* (perturbação do ruído no pixel *i*) é identicamente distribuído à variáveis gaussianas reais. Quando σ*(n)*= 3, nenhuma alteração visível é observada. Dessa forma, uma razão SNR = 60/3 = 20 é aproximadamente invisível a ruído. Um dos grandes problemas envolvendo algoritmos de remoção de ruídos em imagens é que os mesmos confundem pequenos detalhes da imagem como sendo o ruído a ser tratado e, portanto, acabam por remover partes da imagem que não são ruídos. Todos os métodos assumem que o ruído na imagem é oscilatório e que a imagem é suave, ou parcialmente suave. Dessa forma, tentam separar o que é suave do que é oscilatório, porém muitas estruturas inerentes à imagem são tão oscilatórias quanto um ruído. Com isso, em muitos casos os algoritmos geram novas distorções na imagem (BUADES, COLL e MOREL, 2004).

2.3 Non-Local Means

O algoritmo non-local means, ao contrário da maioria dos algoritmos convencionais, realiza uma filtragem não local, onde para cada pixel, todos os outros pixels da imagem são levados em consideração para a realização da filtragem. Tal como a maioria dos algoritmos que efetuam a redução de ruídos em imagens, o *Non-local means* também faz uso do cálculo de médias como forma de realizar essa tarefa. A diferença está em que, enquanto a maioria dos algoritmos baseia-se na suposição de que apenas as características que estão próximas entre si tendem a ter valores semelhantes sendo, portanto, usadas para calcular a média; o NLM parte de outra suposição: imagens naturais têm características que se repetem e podem ser

detectadas globalmente, não apenas localmente (Buades, Coll, & Morel, 2005). Sabendo disso, para remover o ruído de um pixel *p*, o algoritmo procura características semelhantes àquelas na vizinhança de *p* por toda a imagem, e atribui um peso a cada pixel de acordo com a semelhança de suas vizinhanças. A filtragem de *p* é, portanto, efetuada através de uma média ponderada de todos os pixels da imagem. Dessa forma, o algoritmo utiliza todos os pixels relevantes para a realização do cálculo, mesmo que estes não estejam na vizinhança do pixel filtrado. A figura 5 mostra um exemplo para ilustrar a semelhança entre pixels e suas vizinhança em uma imagem real. Assim, a Figura 6 mostra quatro regiões denominadas *P1, P2, P3 e P4*. Apesar da região *P2* estar mais próxima da região *P1*, as regiões *P3* e *P4* apresentam características mais relevantes de semelhança, para a filtragem de *P1*. Isso mostra que apesar de existir uma grande probabilidade de se achar pixels semelhantes em uma vizinhança próxima, também há a possibilidade de se encontrar pixels relevantes localizados em uma distância maior.

O algoritmo NLM é definido formalmente da seguinte maneira: Dada uma imagem discreta com ruído $v = \{v(i) \mid i \in I\}$, o valor estimado $NL[v](i)$, para um pixel i, é computado como uma média ponderada de todos os pixels da imagem, definido pela Equação (5).

$$NL[v][i] = \sum_{j \in I} w(i,j)\, v(j), \qquad \textbf{(5)}$$

onde a família de pesos $\{w(i,j)\}_j$, depende da similaridade entre os pixels i e j, e satisfaz a condição $0 \leq w(i,j) \leq 1$ e $\sum_j w(i,j) = 1$.

A similaridade entre dois pixels i e j depende da semelhança entre os vetores (as vizinhanças) de intensidade de nível de cinza $v(V_i)$ e $v(V_j)$, onde V_k denota uma vizinhança quadrada de tamanho fixo centralizada no pixel k. Essa similaridade é mensurada através da distância Euclidiana quadrática $S(i,j)$, ver equação 6.

$$S(i,j) = ||v(V_i) - v(V_j)||^2 \qquad \textbf{(6)}$$

Figura 6: Semelhanças e vizinhanças entre pixels nas regiões P1, P2, P3 e P4.(Imagem original de 1972 da revista Playboy norte-americana).

Os pixels que possuem vizinhanças semelhantes quanto ao nível de cinza $v(V_i)$ têm pesos maiores. Os pesos são definidos como sendo:

$$w(i,j) = \frac{1}{Z(i)} e^{-\frac{S(i,j)}{h^2}} \quad \textbf{(7)}$$

onde $Z(i)$ é o fator de normalização:

$$Z(i) = \sum_j e^{-\frac{S(i,j)}{h^2}} \quad \textbf{(8)}$$

o parâmetro h atua como um fator de filtragem, controlando o decaimento da função exponencial. A Figura 7 mostra o efeito do fator de decaimento *h* na imagem filtrada. A Figura 7.a mostra uma imagem com ruído para $\sigma = 10$; as três outras imagens são resultados da aplicação do filtro *NLM* com fatores de decaimento diferentes. A Figura 7.b possui $h^2 = 4$; a Figura 7.c, $h^2 = 2.8$; e a Figura 7.d, na parte de baixo à direita, $h^2 = 8$. A imagem na Figura 7.c está filtrada brandamente e apresenta um considerável nível de ruído. A imagem na Figura 7.d está intensamente filtrada e vários detalhes finos desapareceram junto com o ruído. A imagem no topo à direita,

Figura 7.b, filtrada de acordo com a recomendação de $h^2 = 0.40\sigma$, encontrada em (Buades, Coll, & Morel, 2011), apresenta um bom equilíbrio entre perda de detalhes e ruído residual.

É importante salientar que o *NLM* não compara apenas o nível de cinza em um pixel, mas sim a configuração geométrica em uma vizinhança inteira, o que o faz mais robusto que outros filtros de ruído.

Figura 7: Efeito do fator de decaimento *h* na imagem filtrada: a_ Imagem com ruído; b_ Imagem filtrada para h^2 =200; c_ Imagem filtrada para h^2 =100; d_ Imagem filtrada para h^2 =2000.(Imagem original de 1972 da revista Playboy norte-americana).

2.3.1 Complexidade do NLM

Um grande problema que impede o uso do *NLM* em aplicações em tempo real em processamento de imagens é a sua alta complexidade computacional, e mesmo considerando o *NLM* como o estado da arte para remoção de ruído, o método é muito lento para ser utilizado em aplicações práticas.

A complexidade computacional é alta devido ao custo de cálculo dos pesos para todos os pixels da imagem durante o processo de filtragem. Para cada pixel a ser processado, toda imagem é pesquisada e as diferenças entre as vizinhanças correspondentes são computadas. A complexidade é então quadrática no número de pixels da imagem (Sapiro & Mahmoudi, 2005).

Admitindo N^2 como o número de pixels da imagem, e utilizando vizinhanças de dimensão quadrática (M^2), a complexidade do algoritmo é $M^2 * N^4$. De uma forma mais simples, essa complexidade se deve ao fato de que para filtrar cada pixel da imagem (N^2) devemos varrer todos os outros pixels (N^2) e calcular a distância euclidiana quadrática (M^2). Tal complexidade é tão alta que torna seu uso impraticável em simples imagens de 3 *MPixels* devido ao longo tempo de execução (Shaham, 2007).

2.3.2 Comparando o NLM com outros métodos de filtragem

Em (Buades, Coll, & Morel, 2004) podem ser vistos comentários sobre outros algoritmos para filtragem, assim como também uma comparação entre tais métodos e o NLM. Os algoritmos abordados, descritos resumidamente na seção 1.1, são:

- Filtro gaussiano (GF);
- Filtro anisotrópico (AF);
- Variação total (TV);
- Variação total iterada (TV Iter);
- Filtragem de Vizinhança, ou *Yaroslavsky neighborhood filters* (YNF);
- *Translation invariant wavelet thresholding* (TIWT);
- *DCT empirical Wiener filter* (EWF).
- Deslocamento de Curvatura Média;

São apresentados os seguintes resultados quanto ao MSE (Tabela 1), usando 5 tipos diferentes de imagens, e ao ruído do método (Figura 8), usando a tradicional imagem de Lena, para as seguintes técnicas de Filtragem: (a) Filtragem Gaussiana,

(b) Deslocamento de Curvatura Média, (c) Variação Total, (d) Variação Total Iterada, (e) Filtragem de Vizinhança,(f) *Hard TIWT* , (g) *Soft TIWT*, (h) DCT e (i) *NLM*.

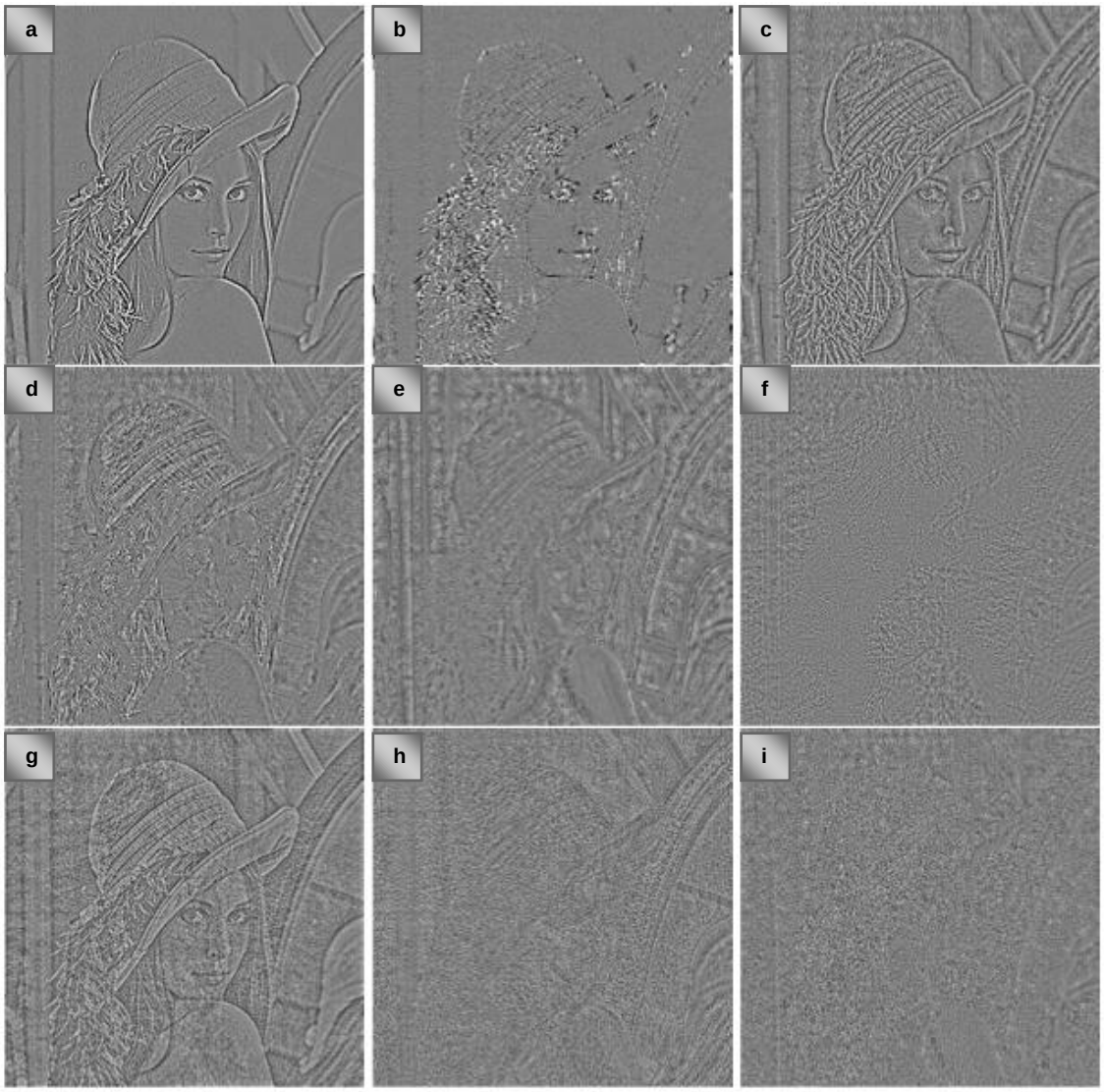

Figura 8: Imagens dos Ruídos dos Métodos Utilizados: (a) Filtragem Gaussiana, (b) Deslocamento de Curvatura Média, (c) Variação Total, (d) Variação Total Iterada (e) Filtragem de Vizinhança, (f) *Hard TiWT*, (g) *Soft TIWT*, (h) DCT, (i) NLM. (Buades, Coll, & Morel, 2004).

Como é possível observar pelos resultados alcançados demonstrados tanto no cálculo do erro quadrático médio (Tabela 1), quanto na imagem de ruído de método (Figura 6), o NLM foi o método que obteve melhores resultados. A única desvantagem associada ao NLM é o alto tempo de cálculo do processamento do método de filtragem.

IMAGEM	σ	GF	AF	TV	TV Iter	YNF	EWF	TIHWT	NL-MEANS
BARCO	8	53	38	39	29	39	33	28	23
LENA	20	120	114	110	82	129	105	81	68
BÁRBARA	25	220	216	186	189	176	111	135	72
BABUÍNO	35	507	418	365	364	381	396	365	292
MURO	35	580	660	721	715	598	325	172	59

Tabela 1: Erros Médios Quadráticos produzidos por cada Método de Filtragem

2.4 Validação da Proposta

A validação da solução proposta consiste em mostrar que a implementação gasta um menor tempo de execução para o NLM, sem perda de qualidade.

A medição para mostrar que a implementação do algoritmo em hardware é mais rápida que a implementação em software é feita comparando-se os tempos gastos em cada uma das implementações.

A captura e transmissão da imagem podem introduzir alguma distorção de modo que a avaliação da qualidade é um problema importante. Para medir este parâmetro dois métodos são utilizados:

1. Erro médio quadrático (*mean square error - MSE*). Abordagem estatística na qual é estimado o erro médio quadrático entre a imagem filtrada e a imagem original (Zhou, 2006). Um valor menor de MSE indica que a imagem filtrada está mais próxima da original.

2. Comparação entre imagens com base no *ruído do método* introduzido pelos autores do NLM em (Buades, Coll, & Morel, 2005). A diferença entre a imagem original e a imagem filtrada mostra o ruído removido pelo algoritmo. Esse resíduo é chamado ruído do método. A Figura 9 mostra um exemplo do uso do ruído do método: à imagem original (*a*), é adicionado ruído branco com $\sigma = 10$ (*b*), e então suavizada com um filtro Gaussiano (*c*) e pelo *NLM* (*d*). Em princípio, o ruído do método deve parecer como um branco. Se apenas ruído for filtrado da imagem, espera-se que o ruído de

método seja semelhante ao ruído branco que, diferentemente da situação do filtro Gaussiano (*e*), é, aproximadamente, o caso do filtro *NLM* (*f*).

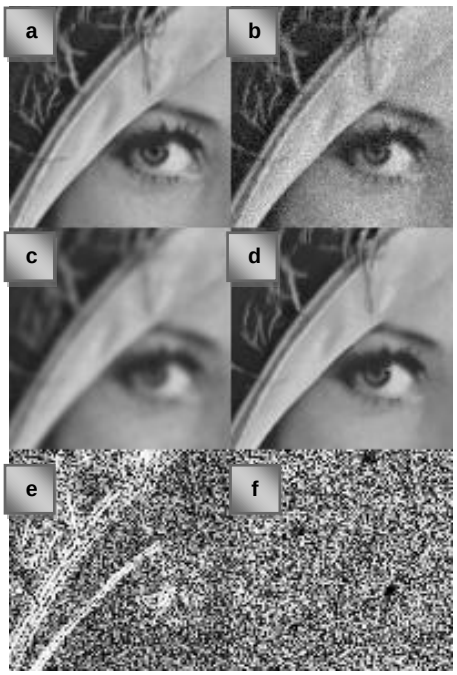

Figura 9: Exemplo de ruído do método.(Buades, Coll, & Morel, 2004).

2.5 Computação Reconfigurável

A microeletrônica tem evoluído bastante nos últimos anos e oferece uma vasta gama de dispositivos que podem ser configurados de acordo com o projeto do usuário, que define a lógica a ser implementada através do uso de uma linguagem de descrição de Hardware. Atualmente é comum nos depararmos com uma variedade de dispositivos eletrônicos cada vez mais velozes, com capacidades de armazenamento de dados maiores, com consumo de energia e custos de mercado mais baixos. O balanço entre velocidade de operação e generalidade desses sistemas tem se tornado um desafio para os projetistas de computadores e sistemas dedicados (Villasenor e Smith, 1997).

A resolução de problemas através da computação é feita utilizando implementações em *hardware* e *software*, ou em muitos casos a combinação dos dois.

A implementação diretamente em circuito, apresenta um maior desempenho em relação à sua implementação em software utilizando computadores de uso geral. Porém, o desenvolvimento de hardware possui maior custo e requer maior tempo de desenvolvimento, além de oferecer menor flexibilidade. Portanto, o desenvolvimento em software, utilizando computadores ou microprocessadores de uso geral, apresenta além de uma maior flexibilidade, um menor custo, além de requerer menor tempo para o desenvolvimento. Porém, o desempenho é baixo em relação à sua implementação em hardware específico.

O projeto de processadores em arquiteturas de computadores depende das aplicações e do ambiente de uso e vão desde máquinas paralelas para alta *performance* onde o foco está concentrado em altas frequências de *clock,* que consomem bastante potência, até sistemas dedicados, nos quais o preço final do equipamento é o fator principal durante o desenvolvimento e o consumo de energia, deve ser otimizado para permitir que ele tenha a maior autonomia possível, considerando o provável uso de baterias.

As arquiteturas podem ser categorizadas em três grupos dominantes (Rossi, 2012), de acordo com o seu grau de flexibilidade: computação de uso geral, que é baseado no paradigma de Von Neumann (VN); processadores de domínio específico, feitos para uma classe de aplicações que possui características bem variadas; e processadores para aplicações específicas, projetados para apenas uma aplicação. Considerando estes três grupos de arquiteturas, podemos identificar dois pontos principais para caracterização dos processadores: flexibilidade e *performance* (Bobda, 2007). Os computadores que utilizam a arquitetura Von Neumann, chamado de grupo geral, são muito flexíveis, pois são aptos a realizar qualquer tipo de tarefa. Esta é a razão da terminologia GPP (Processadores de Propósito Geral) ser usada para a máquina Von Neumann. Eles não possuem muita performance na resolução de um problema específico mas oferecem bom desempenho para a maioria das aplicações

utilizadas pelos usuários em geral. O enfoque dessas arquiteturas é a execução simultânea de *threads*, ou seja execução simultânea de sequências independentes de instruções, é só ver a evolução das arquiteturas de processadores *Hiperthread*(HT) ou *Multicore* oferecidas pelos grandes fabricantes de processadores de uso geral INTEL e AMD. Por sua vez, o grupo dos processadores de domínio específico possuem muita performance porque são otimizados para uma aplicação particular. O *set* de instruções requerido para uma dada aplicação pode ser construído em um chip. A alta performance é possível devido ao *hardware* ser sempre adaptado à aplicação. Se considerarmos duas escalas, uma para a *performance*, e outra para flexibilidade, os computadores Von Neumann podem ser colocados em uma extremidade e os de Domínio Específico em outra, conforme é ilustrado na Figura 10.

Atualmente é possível se projetar sistemas mais gerais em chips, que executam grandes quantidades de cálculos diferentes. Entretanto, o desempenho de tais dispositivos parece lento quando comparado a dispositivos que efetuam aplicações específicas que executam apenas um conjunto limitado de instruções. Nesta segunda modalidade de circuitos, é válido destacar os sistemas em hardware personalizado, também conhecidos como *Applications Specific Integrated Circuits*(ASICs) – Circuitos Integrados com Aplicações Específicas. Devido a estas características específicas, os projetistas conseguem desenvolver chips cada vez menores, com desempenho melhor e que possuem um consumo energético menor que um processador de uso geral.

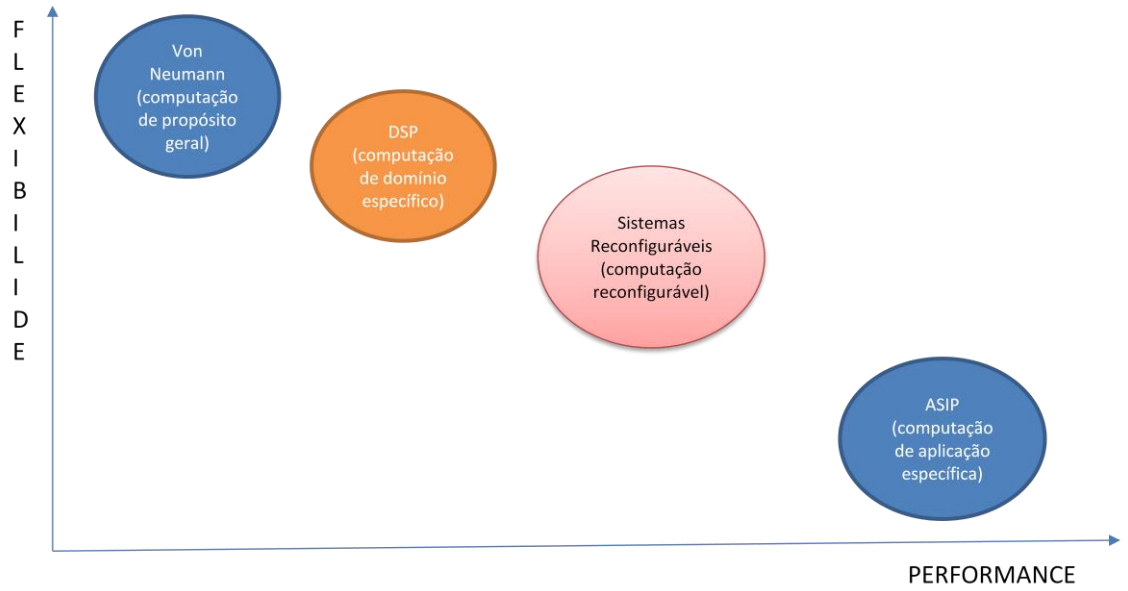

Figura 10: Flexibilidade versus Performance de classes de processadores (Bobda, 2007).

Os dispositivos em hardware *ASICs*, *application-specific integrated circuits*, são circuitos integrados projetados especialmente para uma determinada função, fabricados em uma fábrica com todos os conjuntos de mascaras de metal, polissilicio, dopagem no silicio, etc. Os *ASICs* são fabricados em duas tecnologias:

• *Full-custom* (Implementação a nível de transistor).

• Standard cell (Implementação baseada em biblioteca de células já caracterizadas).

Os Circuitos Integrados *ASICs* oferecem recursos altamente otimizados para a rápida execução de tarefas críticas (Villasenor, 1997), mas é permanentemente configurado para apenas uma aplicação através de um projeto de custo elevado. A implementação em software fornece a flexibilidade para alterar os aplicativos e realizar um grande número de tarefas diferentes, entretanto em termos de desempenho, eficiência em uso de área de silício e em uso de energia, é pior que implementações em *ASICs*.

Antes de se converter um projeto de circuito digital em um Circuito Integrado, a indústria requer algumas semanas ou até meses para se chegar ao projeto finalizado no silício. Entretanto, durante as fases intermediárias do projeto, é recomendada a

utilização de alguma tecnologia de circuitos integrados reconfigurável (Vahid, 2008). Assim, testes práticos podem ser efetuados sobre o projeto do dispositivo implementado antes do mesmo ser enviado para a indústria de circuitos integrados para a finalização do sistema numa pastilha de silício. A tecnologia reconfigurável de circuitos integrados também pode ser utilizada diretamente em aplicações como sistemas dedicados, sem haver a necessidade da produção do chip em silício. Para o caso de desenvolvimento de uma simples solução, que não seja produzida em larga escala, recomenda-se o uso de dispositivos reconfiguráveis, tendo em vista que o custo de produção de um único chip pode se tornar inviável, tornando atrativo a utilização de circuitos reprogramáveis como alternativa para utilização do sistema. Os atributos desejáveis de soluções que utilizam sistemas computacionais reconfiguráveis, são: regularidade de comportamento, flexibilidade, desempenho, generalidade, eficácia e custo (Souza, 2008). A principal característica da computação reconfigurável é a presença de um circuito em hardware que pode ser reconfigurado para implementar uma funcionalidade específica mais apropriada e sob medida, e não um processador de propósito geral. Sistemas de computação reconfigurável unem os microprocessadores e o hardware programável com a finalidade de combinar o potencial do hardware e do software e ser utilizado em aplicações que vão desde um sistema embarcado a sistemas de alta performance computacional (Athanas e Silverman, 1993), (Olukotun, Helaihel, *et al.*, 1994).

2.5.1 Tipos de Circuitos Reconfiguráveis

Os Circuitos Reconfiguráveis possuem a seguinte estrutura:

• A tecnologia de programação pode ser implementada utilizando os seguintes itens: anti-fusível, memórias EPROM/EEPROM/FLASH ou memória SRAM.

• Os Blocos lógicos básicos customizáveis podem ser: multiplexadores também conhecido como Lookup Table - LUT, arranjos OR e AND, etc.

• As Conexões são programáveis e devem oferecer Ferramentas para projeto e síntese no componente programável.

De acordo com o bloco lógico básico e a estrutura das conexões programáveis os Circuitos Reconfiguráveis podem ser do tipo FPGAs (*Field Programmable Gate Array*) – Arranjo de Portas Programável em Campo ou CPLDs(Complex Programmable Logic Devices) – Dispositivos Lógicos Complexos Programáveis. Os CPLDs normalmente tem um conjunto de arranjos AND e OR lembrando muito os antigos PLAs, enquanto os FPGAs são arranjos regulares de multiplexadores separados em colunas e linhas, o que aumenta a densidade.

Os FPGAs (*Field Programmable Gate Array*) são chips reconfiguráveis, que dependendo da maneira como são implementados, podem ser configurados somente uma vez, enquanto outros podem ser diversas vezes reconfigurados (Maxfield, 2004). Uma FPGA em diagrama de blocos é mostrado na Figura 11. Apesar de geralmente estarem presentes em placas de kit de prototipagem de circuitos integrados, como será exemplificado logo mais, os chips FPGAs também podem ser comprados separadamente do kit para serem acoplados a uma placa de configuração e programados para implementar o circuito desejado. Engenheiros de projetos podem reconfigurar estes dispositivos para uma enorme variedade de tarefas.

Os CPLDs(*Complex Programmable Logic Devices*) são circuitos integrados cuja arquitetura interna é predeterminada na fabricação, mas são desenvolvidas de tal forma que os engenheiros de projeto possam configura-las para executarem variadas funções. Um CPLD em diagrama de blocos é mostrado na Figura 12. Comparando-se com os dispositivos FPGAs, os CPLDs contêm um número resumido de portas lógicas, desse modo as funções que eles podem implementar são geralmente menos complexas.

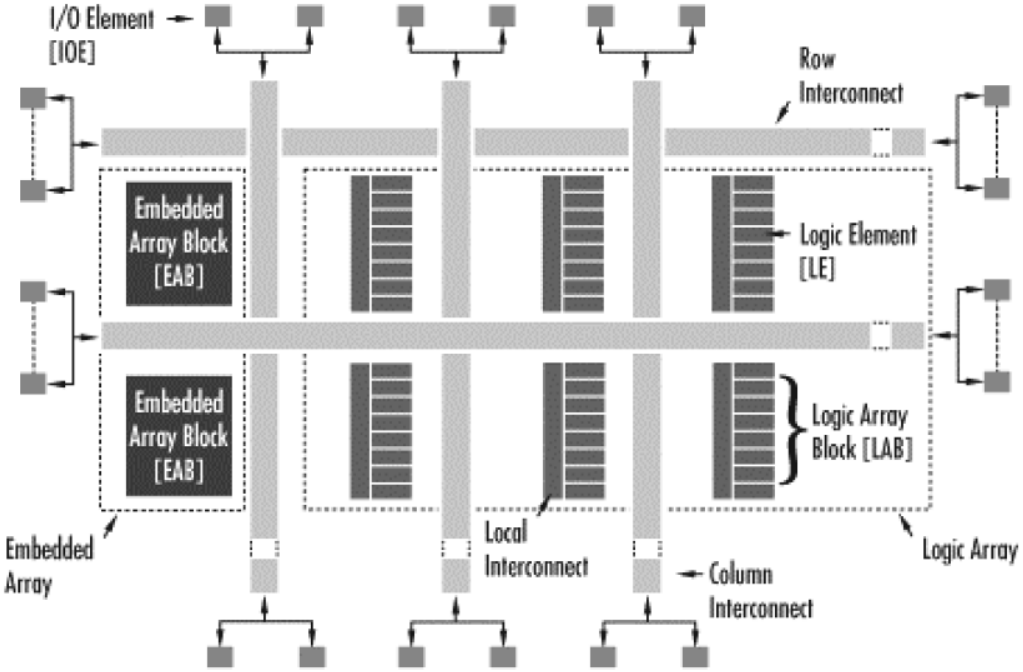

Figura 11: FPGA FLEX 10K da ALTERA (Altera. (2012)).

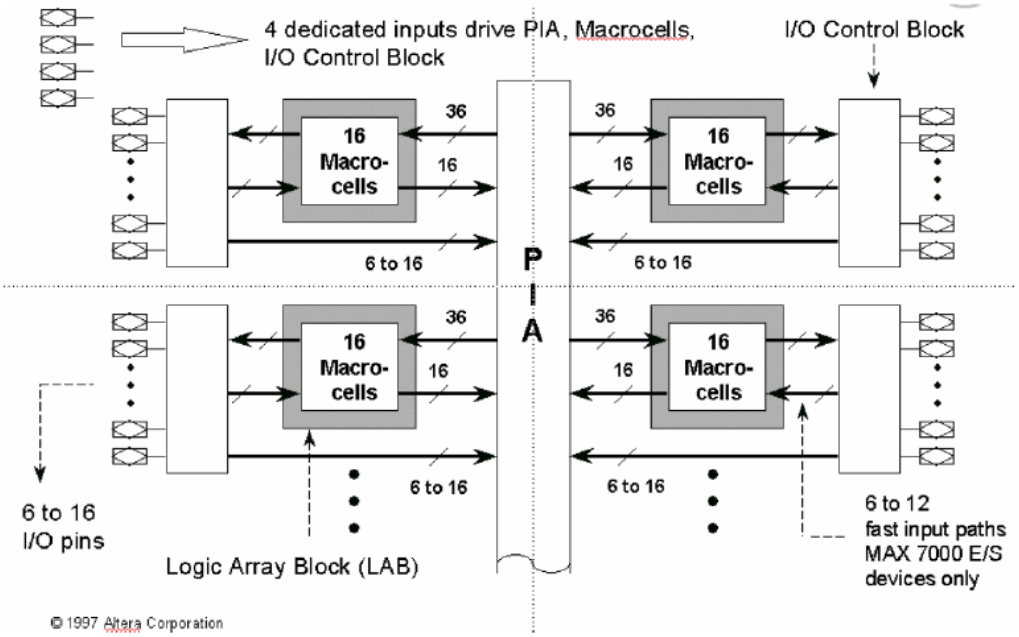

Figura 12: CPLD MAX 7000 da ALTERA Altera. (2012)

Podemos afirmar que um FPGA está integrado a um grupo de dispositivos entre os ASICs e os CPLDs, pois suas funcionalidades podem ser customizadas como

um CPLD e podem possuir milhões de portas lógicas que possibilitam o seu uso na implementação de grandes e complexas funções que só seriam possíveis utilizando ASICs. Os custos em um projeto desenvolvido em FPGAs são muito mais baixos que dos ASICs, embora estes sejam mais baratos para produções em larga escala e ofereçam um maior desempenho, complexidade e tamanho (número de transistores). Contudo, a implementação do FPGA pode ser mudada diversas vezes no local de trabalho onde está sendo desenvolvido o projeto do sistema, enquanto que projetar e construir um ASIC é um processo mais complexo, demorado e caro, acrescentado a desvantagem de que o processo final é customizado em uma pastilha de silício e não pode ser modificada sem a criação de um novo dispositivo (MAXFIELD, 2004).

Os arranjos de portas programável em campo*, Field-programmable gate arrays* (*FPGA*), são dispositivos que misturam os benefícios de ambos, *hardware* e *software*. Assim como os dispositivos em *hardware* ASICs, os dispositivos *FPGAs* implementam computações através da replicação de recursos, realizando simultaneamente milhões de operações nos recursos distribuídos através de um *chip* de silício. Tais sistemas podem ser centenas de vezes mais rápidos que projetos baseados em software e executados em microprocessadores de uso geral. No entanto, ao contrário dos *ASICs*, estes cálculos são programados no *chip*, não tendo sua funcionalidade sido implementada permanentemente pelo processo de fabricação. Isto significa que um sistema baseado em *FPGA* pode ser reprogramado muitas vezes (Hauck, 2008)**.**

Os dispositivos *FPGAs* fornecem quase todos os benefícios da flexibilidade e dos modelos de desenvolvimento de software, e quase todos os benefícios da eficiência de hardware. Quando comparados a um microprocessador, estes dispositivos são tipicamente mais rápidos e mais eficientes em consumo de energia, mas a criação de programas eficientes para eles é mais complexa. Normalmente, os *FPGAs* são úteis para operações que processam grandes fluxos de dados, tais como processamento de sinais, redes, e assim por diante. Comparado com *ASICs*, eles

podem ser de 5 a 25 vezes pior em termos de área, desempenho, e atraso (*delay*) (Hauck, 2008). No entanto, enquanto um projeto *ASIC* pode levar meses ou anos para ser desenvolvido e ter um custo elevado, um projeto de *FPGA* pode levar apenas dias e custar bem menos**.**

Os dispositivos reconfiguráveis têm emergido como uma tecnologia capaz de prover alta *performance* computacional em diversas aplicações, incluindo processamento de sinais, processamento de imagem, simulações aceleradas e computação científica. Em muitos casos, aplicações que utilizam computação reconfigurável superam o equivalente a dezenas ou centenas de microprocessadores contemporâneos ou processadores digitais de sinais (*DSP*) (Rossi, 2012). A alta *performance* é atingida através da construção de operadores customizados dinamicamente, dutos (*pipelines*) de dados e transmissores projetados especificamente para a tarefa a ser realizada. Com esta abordagem, características de uma aplicação particular, como paralelismo, localidade e resolução de dados podem ser exploradas. Portanto, máquinas reconfiguráveis oferecem benefícios de *performance* computacional de circuitos de aplicações específicas (*ASICs*), e ainda possuem a flexibilidade e a reconfigurabilidade de microprocessadores de propósito geral.

Idealmente, é desejável ter a flexibilidade dos GPP (Processadores de Propósito Geral) e a *performance* de um *ASIC* (Processadores de Aplicação Específica) no mesmo circuito. Gostaríamos de ter um dispositivo capaz de se adaptar à aplicação durante o funcionamento (Bobda, 2007). Este dispositivo é chamado de dispositivo de hardware reconfigurável ou dispositivo reconfigurável, ou Unidade de Processamento reconfigurável (RPU) por analogia a Unidade de Processamento Central (CPU). A Computação Reconfigurável é definida como o estudo da computação usando dispositivos reconfiguráveis. Para uma dada aplicação, em um determinado momento, a estrutura espacial do dispositivo é modificada para atingir o

melhor rendimento para acelerar a aplicação. Se uma nova aplicação necessita ser computada, a estrutura do dispositivo é modificada novamente para realizar a nova tarefa. Ao contrário dos computadores que seguem a arquitetura Von Neuman, que são programados através de um *set* de instruções a serem executadas sequencialmente, a estrutura dos dispositivos reconfiguráveis é modificada através da mudança total ou parcial do hardware no momento da compilação ou no momento da execução, pela reprogramação do dispositivo. O progresso da computação reconfigurável tem sido extraordinário nas duas últimas décadas (Rossi, 2012). Isto se deve principalmente pela larga aceitação dos Dispositivos Lógicos Programáveis (FPGAs) que estão agora se estabelecendo como os dispositivos mais utilizados entre os dispositivos reconfiguráveis.

2.6 Processamento Vetorial

O avanço da tecnologia de semicondutores possibilitou a implementação de arquiteturas vetoriais em processadores de aplicação específica comumente utilizadas para processamento de vídeos, imagens e atualmente amplamente utilizadas também para processamento vetorial. Essas arquiteturas são conhecidas como *GPU*'s, ou *Graphic Processing Unit*'s.

Com o objetivo de utilizar o potencial das *GPU's*, a plataforma de computação paralela CUDA (*Compute Unified Device Architecture) (NVIDIA, 2012)* oferece recursos para que o usuário possa desenvolver sua própria aplicação utilizando todo o recurso de hardware vetorial disponível pelas *GPU's*.

Impulsionado pela alta demanda por tempo real, alta definição e gráficos 3D, as unidades de GPU se desenvolveram em uma arquitetura massivamente paralela, com vários núcleos(core) de processamento com um enorme poder computacional e alta largura de banda, alcançando desempenho muito superior à CPUs, em aplicações vetoriais ou matriciais, ou seja naquelas aplicações em que uma mesma operação deve ser executada sobre uma grande massa de dados.

A diferença de desempenho entre uma CPU “multicore” para uma GPU com múltiplos núcleos na execução de tarefas que envolvem paralelismo está na filosofia de projeto de cada tipo de processador. O projeto de uma CPU é otimizado para aumentar a performance de códigos sequenciais. Ele faz uso de uma sofisticada lógica de controle que permite uma linha de execução ser suspensa em qualquer instante e ter sua execução reiniciada a partir do ponto no qual houve a interrupção temporária, sem que isso afete o fluxo de execução desta tarefa. Esses mecanismos auxiliam os sistemas operacionais a fazerem o escalonamento de tarefas. Outro fator importante é o uso de memórias *cache* com capacidade relativamente alta, que são usadas para reduzir latências durante os acessos a memória.

No projeto das GPUs o que prevalece é o uso de um elevado número de pequenos núcleos para serem alocados por um número massivo de sequencias independentes de instruções, chamados de *“threads”*. Parte dos *“threads”* são postos em execução enquanto outros esperam pelo acesso à memórias *cache* de baixa latência, dessa forma reduzindo a lógica de controle requerida para a execução de cada *“thread”*. A Figura 13 mostra as diferenças de arquitetura entre CPU e GPU.

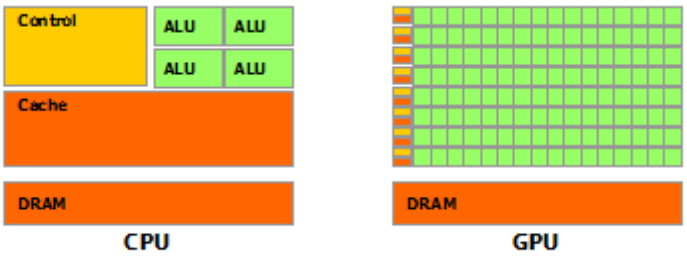

Figura 13: Diferenças na arquitetura de *CPU* e GPU (NVIDIA, 2012)

Entretanto, para que a GPU possa atingir o desempenho esperado é necessário que os problemas a ela submetidos possuam computação de dados paralelos, e em situação ideal, a computação deva ser a mesma, de modo que todos os *cores* executem o mesmo conjunto de operações e nenhum *core* fique em estado de latência.

Em problemas relacionados com imagem, soluções implementadas em *GPU* oferecem grandes ganhos de desempenho, tornando este tipo de implementação uma excelente opção para filtragem de imagens.

2.6.1 Arquitetura GPU habilitada com CUDA

A arquitetura de uma típica GPU habilitada com CUDA é composta de um conjunto de "Streaming Multiprocessors" (**SMs**), ver Figura 14 que apresenta a Arquitetura da GPU *fermi*, altamente paralelizada, que por sua vez, executam grupos de *thread* chamados *warp*. Por outro lado, cada SM possui vários núcleos, chamados de CUDA *cores*, e cada CUDA core possui pipelines completos de operações lógicas e aritméticas (ULA - Unidade Lógica e Aritmética) e de pontos flutuantes (FPU - *Floating Pount Unit*). Por sua vez o SM é constituído de uma memória *cache* L1 comum somente aos núcleos de um SM, e todos os *cores* de um SM têm acesso a uma memória global (*cache* L2). A Figura 15 mostra o SM da arquitetura Fermi. Note que boa parte do SM corresponde aos CUDA cores, caracterizando sua especialidade em operações que envolvem muitos cálculos, ao contrário das CPUs, onde a *cache* é predominante.

As Arquiteturas GPUs da NVIDIA apresentam a seguinte evolução:

- **G80**_ A oitava geração de placas gráficas da NVIDIA foi introduzida em novembro de 2006. A GeForce 8800 foi a primeira GPU com suporte a linguagem C, introduzindo a tecnologia CUDA. A placa gráfica introduziu a arquitetura de *shaders* unificada, que são pequenos programas utilizados pelos jogos e aplicativos de renderização recentes para executar operações específicas dentro das imagens, com 128 CUDA cores, distribuídos entre 8 SMs [Beyond 3D Nvidia g80: Architecture and GPU analysis].

Figura 14: Arquitetura de uma típica GPU *Fermi* habilitada com CUDA. (NVIDIA, 2012)

- **G200**_ Lançada em 2008, esta arquitetura da NVIDIA trouxe novas melhorias para o CUDA. Pela primeira vez foi implementado o suporte para computação com 64-bits de precisão dupla [Beyond 3D Nvidia g200: Architecture and GPU analysis].
- **Fermi**_ Lançada em abril de 2010, esta arquitetura trouxe suporte para novas instruções para programas em C++, como alocação dinâmica de objeto e tratamento de exceções. Cada SM de um processador Fermi possui 32 CUDA cores. Até 16 operações de precisão dupla por SM podem ser executadas em cada ciclo de clock. Além disso, cada SM possui: **Dezesseis unidades de *load* e *store***, possibilitando que o endereço de fonte e destino possam ser calculados para dezesseis *threads* por clock; **Quatro Unidades de Funções Especiais(SFU)**, que executam instruções para calculo de seno, cosseno, raiz quadrada, etc. Cada SFU executa uma instrução por thread por ciclo. O

pipeline da SFU é desacoplado da *dispatch unit*, permitindo que esta possa realizar o despacho de outra instrução enquanto a SFU está ocupada.

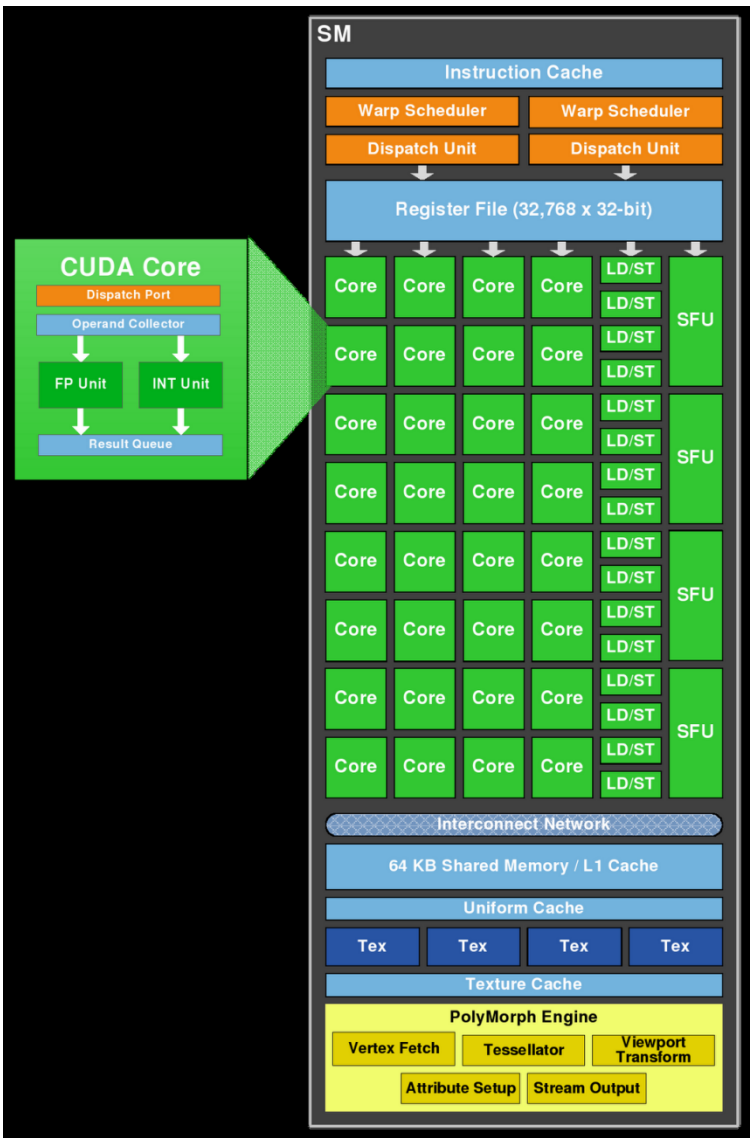

Figura 15: SM da Arquitetura *Fermi*.(NVIDIA, 2012)

- **Kepler**_ Lançada em 2012, a mais nova arquitetura da NVIDIA introduziu um novo modelo de SM, chamado de SMX, que possui 192 CUDA cores, totalizando 1536 cores no chip (8 SMXs). No processo de criação da arquitetura Kepler, um dos principais objetivos era obter uma melhor eficiência energética. Os transistores de 28nm foram importantes para a redução no consumo de energia, mas a alteração da arquitetura, com o uso dos SMX, foi a principal responsável pela maior redução no consumo de energia. Com essa

alteração foi possível executar os CUDA cores em frequência maior e ao mesmo tempo reduzir o consumo de energia da GPU, fazendo com que ela aqueça menos. A Figura 16 mostra o SMX da arquitetura Kepler. Além dos 192 CUDA cores de precisão simples, o SMX possui 64 unidades de precisão dupla, 32 unidades de funções especiais(SFU) e 32 unidades de *load* e *store*. O paralelismo dinâmico no Kepler reutiliza as *threads* dinamicamente através da adaptação de novos dados, sem a necessidade de retornar os dados intermediários para a CPU. Isso permite que os programas sejam executados diretamente na GPU, já que agora os *kernels* possuem a habilidade de executar cargas de trabalho adicionais independentemente, como mostra a Figura 17.

Figura 16: SMX da Arquitetura *Kepler*. (NVIDIA Corporation. Kepler gk110 architecture whitepaper)

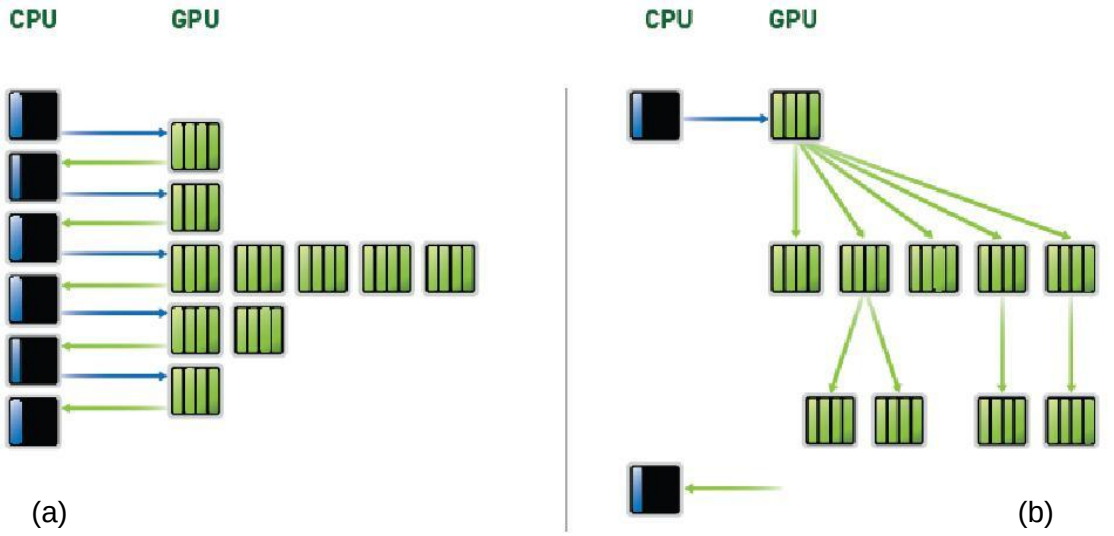

Figura 17: (a) Sem o Paralelismo Dinâmico, a CPU executa cada *kernel* na GPU. (b) Com o paralelismo dinâmico, a GPU Kepler GK110 pode agora executar *kernels* aninhados, eliminado a necessidade da comunicação com a CPU. (NVIDIA Corporation. Kepler gk110 architecture whitepaper).

2.6.2 Programando em CUDA

O modelo de programação CUDA é baseado na linguagem de programação C, possuindo um conjunto mínimo de extensões para possibilitar amplo acesso a toda arquitetura das *GPU's*, como memórias compartilhadas e sincronismo (NVIDIA, 2012). O CUDA C é uma extensão C de modo que o programador possa definir funções em C, chamadas de *kernels*, que são executadas N vezes em N diferentes CUDA *threads*.

Os programas CUDA podem ter partes do código cuja execução é estritamente serial e outras partes que permitem paralelismo no processamento de dados. Dependendo da riqueza do paralelismo, as fases de execução dos programas são implementadas para executarem na GPU. As funções que devem ser enviados para a GPU são marcadas com uso de palavras chaves que fazem parte da extensão CUDA feita sobre as linguagens e são chamadas de "kernels". Essas palavras chaves são reconhecidas pelos compiladores fornecidos pela NVIDIA, como é o caso do **NVIDIA ® C Compiler** (**nvcc**). Os segmentos de código que não são marcados são processados por compiladores padrão e são executados como processos ordinários na CPU.

Quando funções “kernel” são invocadas, a execução é movida para a GPU, onde tipicamente é gerado um número elevado de “threads” para explorar o paralelismo. Esse número de “threads” pode ser muito maior que o número de núcleos, significando que alguns das “threads” irão aguardar a conclusão de outras “threads”. Outro evento que pode provocar a ativação de “threads” é o acesso á memória global. Em razão da alta latência dessa memória, as “threads” que tentam acessá-la têm sua execução suspensa temporariamente. A Figura 18 ilustra em diagrama de blocos como um programa CUDA é executado.

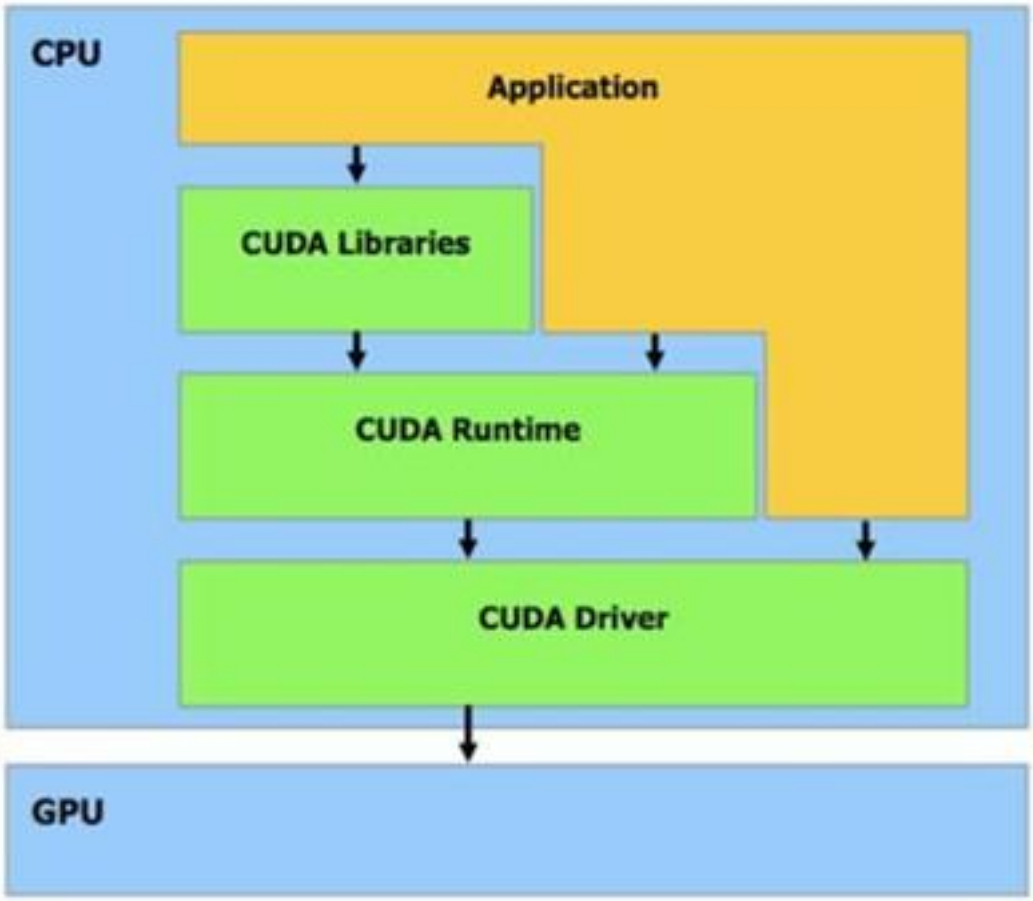

Figura 18: Diagrama de Blocos de Execução de um Programa escrito em CUDA.(NVIDIA,2012).

Um *kernel* corresponde a um *grid* de Blocos, cada Bloco é composto por threads e todas as threads do mesmo bloco compartilham a mesma área de memória. Assim as *threads* de um mesmo bloco podem compartilhar dados umas com as outras,

enquanto *threads* de blocos diferentes não podem compartilhar memória entre si, como ilustra a Figura 19.

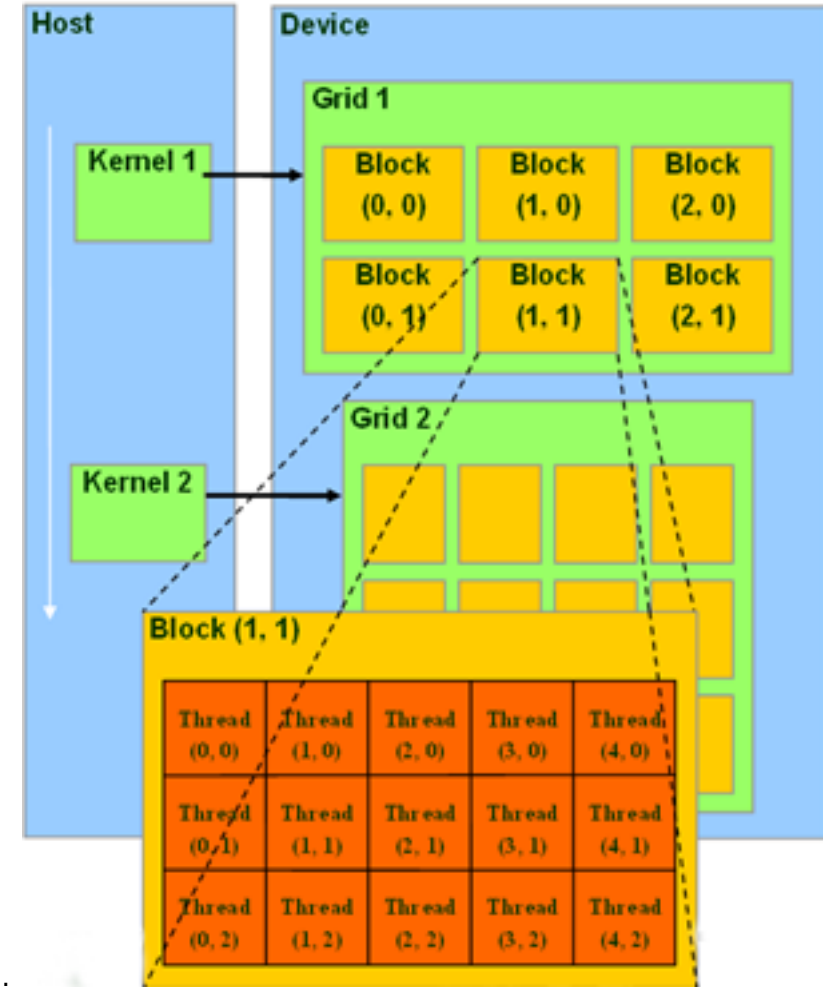

Figura 19: Organização *Grid,* blocos e *threads* (NVIDIA, 2012)

Em CUDA, **host** (CPU) e **device** (GPU) têm memórias separadas. Quando uma aplicação tem um conjunto de dados e quer que eles sejam processados pela GPU, esses blocos de dados têm que ser copiados da memória principal da CPU para a memória dedicada no "device". Essa memória alvo é a memória global do "device". Existem outras memórias nos "devices" habilitados com CUDA, como é ilustrado na Figura 20. Assim cada thread possui acesso a três níveis diferentes de memória, sendo cada um mais lento que o outro. Em primeiro lugar está a memória privada da thread, que não pode ser compartilhada entre threads e representa tanto trechos da

memória interna de um multiprocessador quanto registradores. Em segundo plano, está a memória compartilhada por bloco. Todas as threads em um mesmo bloco possuem acesso a essa memória que fica no multiprocessador, sendo portanto bastante rápida. No terceiro nível está a memória global, que é bastante lenta. Sua utilização deve ser feita somente em caso de extrema necessidade.

Ainda existem dois tipos especiais de memória de leitura: a memória constante e a memória de textura. Elas também são globais e podem ser lidas por todas as *threads* e são otimizadas para acesso rápido, sendo a memória de textura capaz de oferecer como única especialidade um modo de endereço diferenciado para rápido acesso a dados locais, como por exemplo dados próximos em diferentes dimensões de determinado elemento em um dado multidimensional, tais como texturas e imagens. Estas memórias especiais recebem e mantém os dados antes do processamento do *kernel* na mesma aplicação.

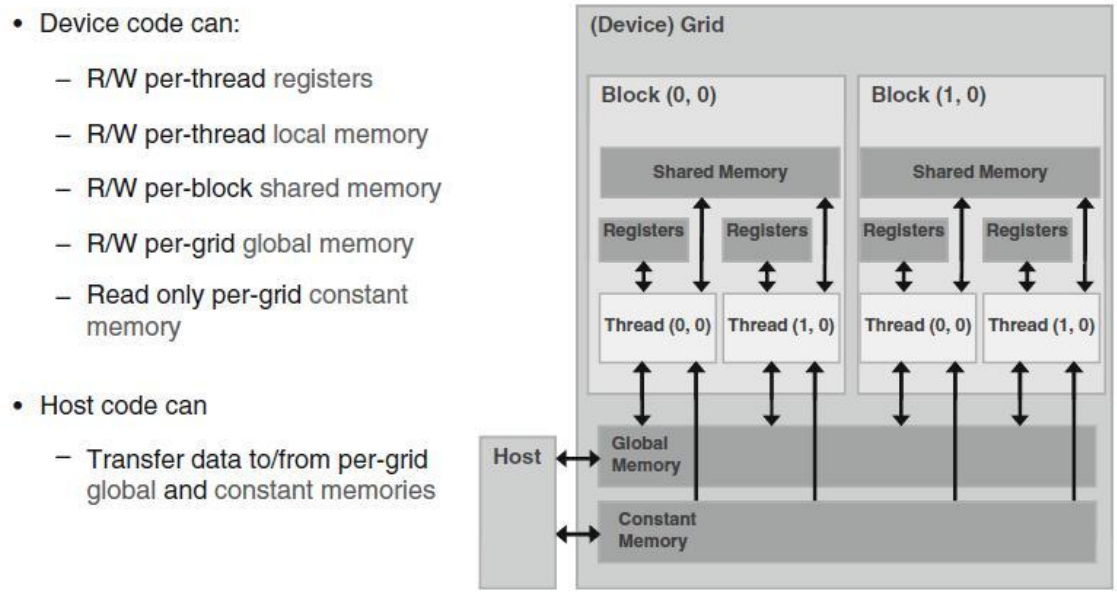

Figura 20: Uso da Memória em uma GPU.(NVIDIA, 2012)

As threads são agrupadas por blocos, que atualmente podem conter até 1024 threads residindo no mesmo multiprocessador, e também podem ser identificadas pela variável *blockIdx*, igualmente com possibilidade de até três dimensionalidades.

Por fim, os blocos são organizados em *grids* unidimensionais, bidimensionais ou tridimensionais. O número de blocos em um *grid* é usualmente ditado pelo tamanho do dado a ser processado ou o número de processadores no sistema, como por exemplo, no processamento de imagens, cada *thread* pode executar o processamento de um *pixel*, logo o tamanho total da imagem definirá o tamanho final do *grid* a depender do tamanho dos blocos de *threads.*

Os blocos gerados ao chamar um *kernel* são entregues ao gerenciador dentro de um multiprocessador, sendo distribuídos entre os multiprocessadores pelo gerenciador da GPU para manter o nível de carga igual. Internamente, os blocos ainda são separados em *warps,* cada um com 32 *threads*, que são controlados pelo gerenciador dentro do multiprocessador. Essa hierarquia é mostrada na Figura 21.

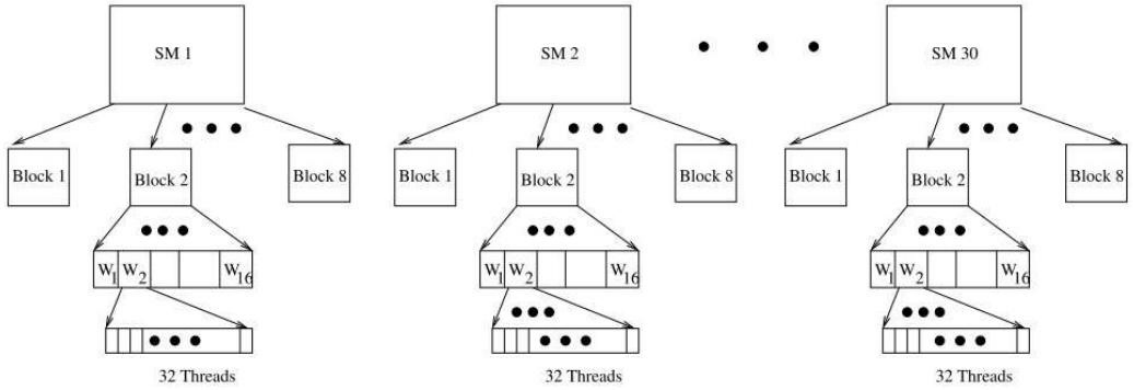

Figura 21: Hierarquia das threads segundo o modelo CUDA. (NVIDIA, 2012).

Um programa da GPU chama-se *kernel*. A Programação de um *kernel* exige: Reservar espaço na memória da placa gráfica; Copiar dados para a memória da placa; Chamar o código a ser executado na GPU; Copiar os resultados de volta da GPU.

2.7 Técnicas de Paralelismo

Os projetistas de computadores estão sempre procurando projetar máquinas que apresentem melhor desempenho, fazer circuitos integrados com um ciclo de relógio menor, que implica em uma maior frequência de trabalho, é uma possibilidade,

entretanto sempre há um limite para cada época da história((Tanembaum, 2007). Uma outra maneira de melhorar o desempenho é a de utilizar o paralelismo como meio para se conseguir um desempenho maior para um dado ciclo de relógio. Desta forma o paralelismo permite que várias operações possam ser executadas simultaneamente em cada ciclo de relógio. O paralelismo é implementado através de duas técnicas comumente usadas: Paralelismo Temporal e Paralelismo Espacial.

O Paralelismo Temporal, também chamado de *pipeline*, é a técnica ideal para agilizar tarefas que por natureza são sequenciais, o procedimento é quebrar a tarefa em sub-tarefas diferentes, chamadas de estágios, de modo que se tenha a execução de várias tarefas sobrepostas, como numa linha de montagem; as várias tarefas são executadas em paralelo, mas em fases diferentes, como ilustra a Figura 22.

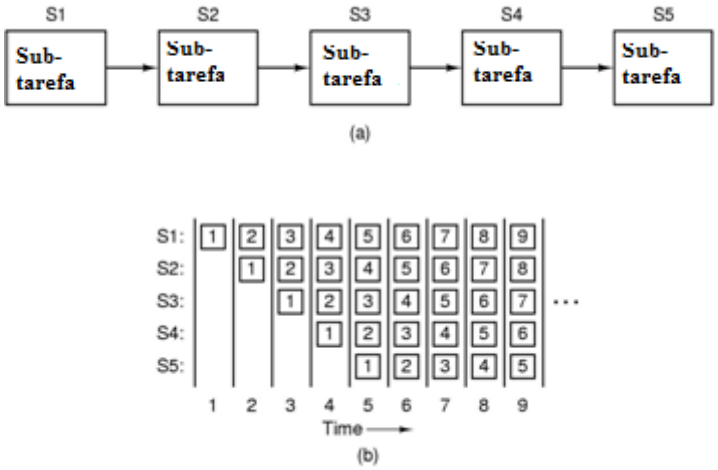

Figura 22: (a) Paralelismo Temporal, também chamado de *pipeline*. Exemplo com 5 estágios.(b) Diagrama mostra no tempo a execução de tarefas. (Tanenbaun, A. S., 2007)

O Paralelismo Espacial trata da replicação de recursos, ou seja da execução em paralelo de duas tarefas e é a técnica adequada para agilizar tarefas que podem ser executadas simultaneamente, como ilustra a Figura 23. Outra forma de paralelismo espacial é a de manter em *pipeline* a parte da tarefa que é sequencial e criar blocos especializados com a parte da tarefa que pode ser executada simultaneamente, como mostra a Figura 24(Tanembaum, 2007).

Restrições de desenvolvimento em hardware tais como: a utilização de valores fracionários e cálculos envolvendo a função exponencial; puderam ser contornadas utilizando aproximações por inteiros e aproximações lineares por partes, respectivamente.

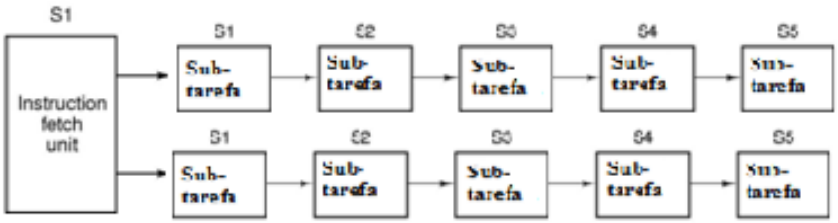

Figura 23: Paralelismo Espacial. Exemplo com replicação de dois *pipelines*. (Tanenbaun, A. S., 2007)

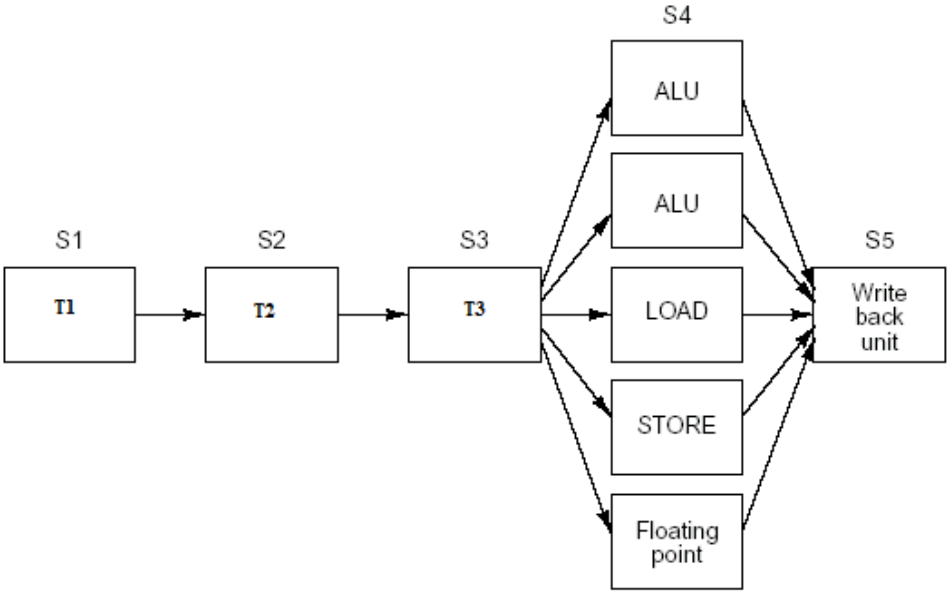

Figura 24: Paralelismo Espacial. Exemplo com blocos especializados. (Tanenbaun, A. S., 2007)

Capítulo III: Placa Aceleradora para Otimização do Algoritmo *Non Local Means*

Este capítulo apresenta a implementação de uma Placa Aceleradora para barramento PCI para otimização do Algoritmo *Non Local Means*. Inicialmente é apresentado os trabalhos relacionados à otimização do NLM, em seguida é mostrado a arquitetura da Placa Aceleradora.

3.1 Trabalhos relacionados à otimização em tempo do *NLM* em software

O NLM se trata de um algoritmo de alta complexidade, porém com excelentes resultados. Deste modo, a literatura científica apresenta uma vasta bibliografia referente a trabalhos que procuram otimizar o algoritmo *NLM,* onde o principal objetivo é tratar de maneira mais eficiente o cálculo de comparação das médias dos *pixels*; que na maioria das vezes se dá reduzindo a quantidade de *pixels* que serão computados para a média.

Dentre as propostas apresentadas na literatura científica destacamos os seguintes métodos para acelerar o *Non-local Means*:

- (SAPIRO e MAHMOUDI, 2005) propõe uma aceleração para o NLM através da introdução de filtros que eliminam vizinhanças sem correlação durante o cálculo da *Distância* L^2 na janela de busca. Estes filtros são baseados em médias locais, dos valores dos *pixels* e gradientes, gerando uma pré-classificação das vizinhanças a serem utilizadas que reduz a complexidade quadrática do tempo de execução do *NLM* para uma complexidade linear. Esta abordagem implica na redução da influência de áreas com grau de semelhança baixo durante a filtragem de um dado *pixel*.
- (WANG, GUO, YING, LIU e PENG, 2006) propõe o uso de *Summed Square Image (SSI)* e transformada rápida de Fourier (*FFT*), realizando apenas convoluções e somas.

- (COUPÉ, YGER e BARILOT, 2006) com o objetivo de remoção de ruídos em imagens médicas 3D propõe também uma seleção de *pixels*, porém baseado na média e desvio padrão, focando imagens com alto nível de ruído. Apesar de obter bons resultados com imagens de ressonância magnética - onde existem altos níveis de ruído - este método tende a ser falho por suavizar muito a imagem, eliminando também algumas características de texturas comumente existentes em diversos outros tipos de imagem.
- (BROX, KLEINSCHMIDT e CREMERS, 2008) relata a aceleração do método usando subamostragem espacial usando clusters de computadores, que torna os resultados mais significativos, uma vez que o tempo de execução permanece praticamente constante ao filtrar imagens com maiores resoluções, desde que aumente proporcionalmente a quantidade de computadores incorporados ao cluster.
- (BILCU e VEHVILAINEN, 2008) propõe uma otimização (*ICINLM)* baseada num método denominado *Interseção de Intervalos de Confiança* (*Intersection of the Confidence Intervals*), que procura manter o desempenho de filtragem do *NLM* convencional, mas com baixo custo computacional através de cálculos baseados em aproximação polinomial local. Nos testes definidos neste trabalho, são apresentadas melhorias consideráveis de tempo de execução e de qualidade da filtragem quando comparado com o proposto em (BUADES, COLL e MOREL, 2004) e (SAPIRO e MAHMOUDI, 2005).
- (AVANAKI, DIYANAT & SODAGARI, 2008) promete encontrar melhorias no cálculo do *NLM* com relação ao *fator de decaimento* (*h*) que conduzem a otimizações na performance de filtragem. Neste trabalho há vários resultados de testes sobre imagens, com índices de tempo de execução e qualidade de imagem expostos, porém também não cita informações sobre imagens de teste.

- (PANG et al., 2009) propõe uma pré-seleção de *pixels* para o cálculo da similaridade entre os *pixels* – também apresenta bons resultados de otimização, inclusive quando comparado com o trabalho desenvolvido em (BILCU e VEHVILAINEN, 2008).
- (DAUWE, GOOSSENS, ET AL., 2008) propõe uma abordagem utilizando estatísticas de ordem superior, o resultado é um baixo desempenho em tempo de execução.
- (BHUJLE e CHAUDHURI, 2012) propõe a utilização de dicionários de imagem para remover o ruído, são procuradas no dicionário vizinhanças semelhantes do *pixel* de interesse. O critério de agrupamento é a distância euclidiana quadrática. Este método proposto tem como vantagem a não dependência do tamanho da janela da vizinhança e oferece um bom desempenho, aproximadamente 10x comparado com o *NLM* por janela, mas como resultado apresenta imagens relativamente borradas com bordas granuladas.
- (HEDJAM, MOGHADDAM e CHERRIET, 2009) propõe que os *pixels* sejam selecionados para os cálculos dos pesos através da utilização do particionamento de grafos, utilizando a Clusterização de Markov em um Grafo de Adjacência de *Pixels* (em inglês PAG). Este método apresenta melhores níveis de PSNR, porém não cita informações de desempenho.
- (LAI e DOU, 2010) propõe otimizações no *Kernel Gaussiano* e uma abordagem de cálculo de pré-classificação da similaridade da vizinhança entre *pixels*, resultando na melhoria da filtragem e preservação dos detalhes da imagem. Apesar de mostrar valores consideráveis de qualidade de filtragem, por meio do índice *PSNR*, e afirmar que o tempo da computação é semelhante ao demonstrado em (Sapiro & Mahmoudi, 2005), não são citadas informações sobre as imagens de teste utilizadas.

- (DINESH e RAMYA, 2012) propõe que os *pixels* sejam agrupados baseado na intensidade de cinza usando *k-means* na imagem após uma pré-filtragem com o filtro gaussiano. Após este agrupamento que resulta em uma segmentação da imagem, é avaliado uma função de semelhança proposta utilizando *mask function*. O método apresenta melhores resultados no índice PSNR do que o *NLM* original, oferecendo melhor qualidade na filtragem de regiões de textura na imagem, porém também não apresenta resultados de tempo de execução.

Portanto, constata-se que, de acordo com a pesquisa bibliográfica acima relacionada, todas apresentam uma abordagem de melhoria em software através de modificações no algoritmo original. Apenas o estudo feito no laboratório de pesquisa LASID introduziu uma implementação em hardware para aceleração do NLM que foi apresentado no artigo *"Otimização do Algoritmo Non-local Means utilizando uma implementação em FPGA"* (Gambarra, L., Lima, J.A.G., Silva, H.S., Batista, L. V., Marques, D. S., 2012) e nas teses de mestrado de Lucas Gambarra e Daniel Marques, também desenvolvidas no laboratório LASID no ano de 2012. Nesta proposta, é mostrado que, a não ser pelas aproximações de valores fracionários e da função exponencial, existe uma forma quase exata para a execução do *NLM* em janela usando uma implementação em hardware reconfigurável em uma Placa Aceleradora para barramento PCI que acelera o método significativamente em comparação com a implementação clássica.

Apesar da indicação em (Gonzales & Woods, 2000) para utilização de implementação em hardware para aceleração de processamento de imagens, este tipo de implementação não foi apresentada na literatura para o NLM, ao menos no conhecimento do autor, até o inicio dos nossos estudos.

3.2. *NLM* em janela

Imagens naturais tendem a ter grandes áreas uniformes, com variância geralmente pequena entre pixels próximos. Essa conclusão pode ser alcançada,

também, analisando os resultados em (Mahmoudi & Sapiro, 2005), que até mesmo reportam melhor desempenho em áreas uniformes quando comparados com o algoritmo NLM original. Com base nessa suposição, define-se uma janela de busca dentro da qual é realizada a procura por vizinhanças semelhantes reduzindo a complexidade do NLM. Note que usando uma janela de busca em lugar de pesquisar toda a imagem, o algoritmo modificado poupa a ideia de filtrar características semelhantes na imagem que estão a uma distância maior que a janela de busca. Com uma janela de busca, o algoritmo aproxima-se de um filtro de vizinhança local tomando como medida de similaridade a distância Euclidiana quadrática.

Na Figura 25, ao filtrar P1, o valor de P3 é usado na filtragem, mas não o valor de P4, pois ele esta fora da janela de busca (indicada pelo quadrado branco em volta de P1).

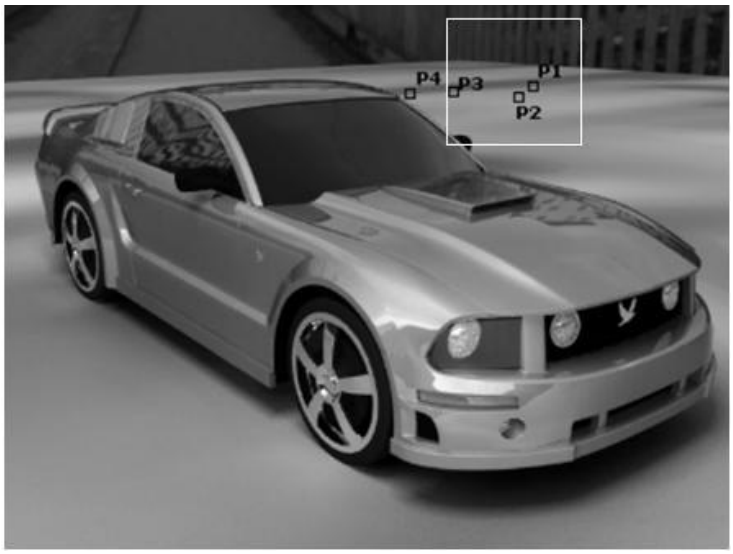

Figura 25: Janelas de semelhança.(Gambarra, L. L., 2012)

Restringindo a busca por pixels semelhantes a uma janela de W^2 pixels a complexidade final do algoritmo passa a ser $O(M^2 * W^2 * N^2)$ o que é empiricamente menor considerando que na maioria dos casos *W* é bem menor que *N*.

3.3 Otimização do *NLM* em *FPGA*

Para tornar o algoritmo *NLM* rápido o suficiente para permitir sua utilização em aplicações com requisitos de performance (tempo de resposta, *throughput* e, ou temporização), o presente trabalho propõe uma implementação em hardware da versão em janela do *NLM*, utilizando um dispositivo *FPGA* (*Field-Programmable Gate Array*) e colocado em uma placa PCI para utilizar um computador de uso geral como *HOST*.

Na proposta de desenvolvimento em hardware do algoritmo *NLM* estão sendo utilizadas técnicas de paralelismo temporal e espacial, bem como é proposto uma organização de memórias especializadas com o objetivo de buscar rapidamente os dados para manter o bloco em *hardware* do NLM em execução.

3.3.1. Parâmetros

Os criadores do *NLM* (Buades, Coll, & Morel, 2005) sugerem que sejam utilizados valores pequenos para M e para W, mesmo assim, o tempo de execução para o processamento em software não é reduzido a níveis razoáveis.

A utilização de valores arbitrários pequenos para M e W geralmente provocam distorções na imagem filtrada por falta de informação para calcular os pesos (Shaham, 2007). Entretanto, os autores do *NLM* em (Buades, Coll, & Morel, 2011) mostram como escolher adequadamente os valores para esses parâmetros de modo a garantir a qualidade da filtragem. Os tamanhos da vizinhança de comparação e da janela de pesquisa dependem do valor do desvio padrão σ. Deste modo, a medida que σ cresce são necessárias janelas de pesquisa W^2 maiores para realizar uma comparação mais precisa, e ao mesmo tempo, é preciso estender o tamanho da vizinha M^2 de comparação para ampliar a capacidade de remoção de ruídos do algoritmo, que é realizada através da pesquisa de *píxels* mais similares. O valor do parâmetro de filtragem é definido como $h = k\sigma$. O valor de k diminui à proporção que o tamanho da janela de comparação aumenta. Para tamanhos maiores, a distância entre duas

vizinhanças com ruído natural se concentra mais em torno de $2\sigma^2$ e, portanto, um menor valor de k pode ser usado para filtragem.

3.3.2. Implementação em *FPGA* do *NLM* em Janela

A Figura 26 mostra o fluxo da filtragem utilizando o *NLM* em hardware. Adiante, é detalhada cada fase do fluxo apresentado.

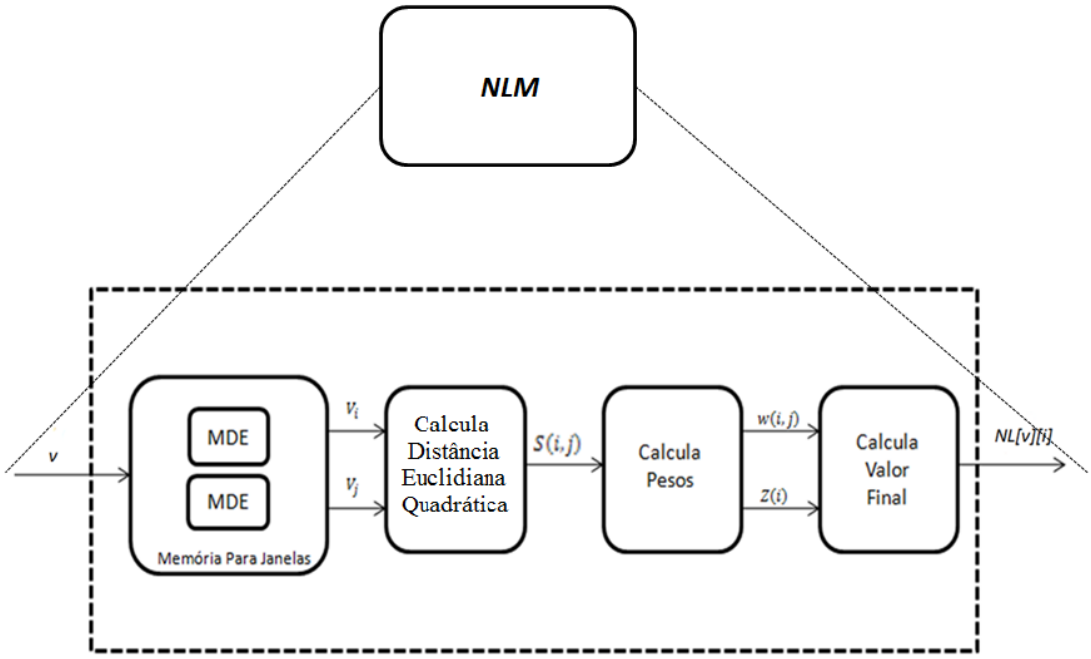

Figura 26: Fluxo da filtragem de ruídos para o NLM em FPGA.

3.3.2.1. Paralelização do cálculo da distância Euclidiana quadrática

Para melhorar o desempenho do NLM, foi elaborada uma forma de acelerar o cálculo da distância Euclidiana quadrática, observe a Equação 9, na qual é avaliado o grau de semelhança entre duas vizinhanças. O parâmetro que permite mensurar a quantidade de operações efetuadas nesse cálculo é o tamanho da vizinhança, ou seja, M^2. Para cada pixel presente na vizinhança são realizadas três operações: subtração, potenciação e soma dos resultados gerados. As operações são independentes entre si, portanto podem ser operadas em paralelo. Por outro lado, cada operação pode ser realizada em *pipeline*. Assim, após o devido preenchimento do *pipeline*, pode-se realizar um cálculo de distância Euclidiana quadrática por ciclo de relógio.

$$S(i,j) = ||v(V_i) - v(V_j)||^2 \quad (9)$$

Como exemplo, se forem usadas vizinhanças com $7 * 7$ pixels ($M = 7$), em uma estrutura puramente sequencial são necessários $3 * 7 * 7$, ou seja, 147 ciclos; enquanto que usando a abordagem proposta que emprega técnicas de paralelismo é possível ter a vazão de uma distância Euclidiana quadrática calculada a cada período de relógio.

3.3.2.2. Memória dinâmica especializada (MDE) para vizinhanças

Com a finalidade de manter o componente Calculo da Distância Euclidiana Quadrática sempre em funcionamento foi projetada uma Memória especializada com suporte para acesso simultâneo a vários endereços chamada de Memória Dinâmica Especializada (MDE).

Observando a Fig. 27 percebemos a maneira como são acessadas as posições de memória ao passarmos de uma vizinhança para outra: os pixels em vermelho claro e em vermelho pertencem à vizinhança $n - 1$; ao passar para a vizinhança de n os pixels em vermelho claros são descartados e os pixels em vermelho escuro passam a compor a vizinhança n. Concluímos então que, na passagem de uma vizinhança para outra, apenas sete pontos dos 49 são realmente novos, os demais 42 precisam apenas serem realocados.

A Figura 28 mostra como os pixels são trocados na passagem de uma vizinhança para outra: os pixels que são trocados entre registradores, setas azuis; os que são descartados , setas vermelhas; e, finalmente, os que recebem sete novos pixels da nova vizinhança, setas verdes. Então apenas sete buscas simultâneas são realmente necessárias, desde que troquemos de forma ágil os demais 42 pixels da vizinhança anterior. A Lógica do componente MDE foi desenvolvida de modo a permitir o acesso simultâneo às sete posições de memória, ao mesmo tempo em que a janela em cálculo é replicada por quatro memórias. Considerando que cada uma

pode ter dois valores lidos (ou escritos) ao mesmo tempo, tornou-se possível ler as sete posições simultaneamente.

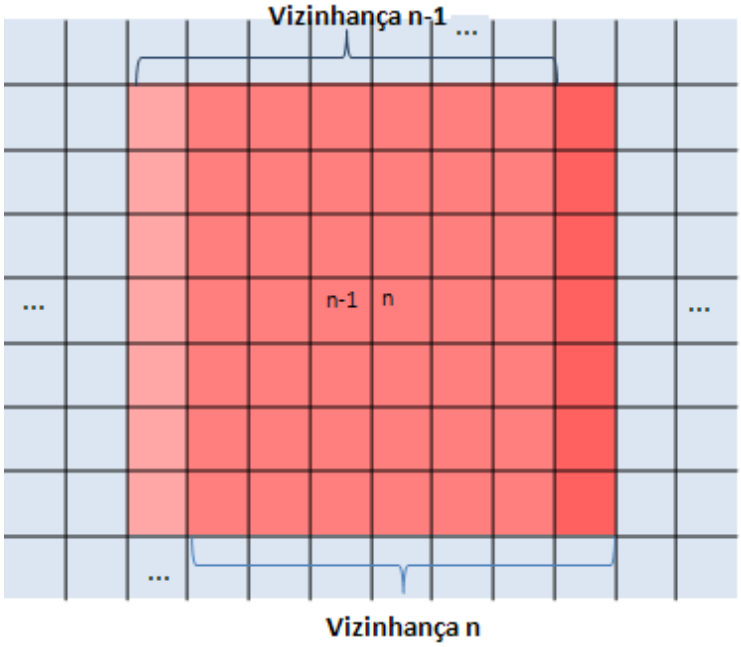

Figura 27: Duas vizinhanças consecutivas: vizinhança *n-1,* centrada no pixel *n-1;* vizinhança *n*, centrada no pixel *n.*

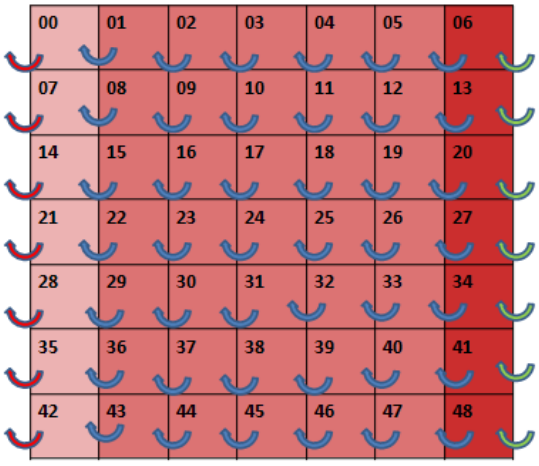

Figura 28: Troca de pixels de uma vizinhança para a seguinte.

3.3.2.3. Duplicação da Memória

A versão do NLM proposta no presente trabalho utiliza os parâmetros $W = 21$ e $M = 7$; valores adequados de acordo com (Buades, Coll, & Morel, 2011), para filtrar imagens com ruído branco com desvio padrão até 30. O nosso sistema implementa a versão do NLM para uma janela $21\text{x}21$ (ou 441) pixels. Para que todos os pixels possuam vizinhanças $7\text{x}7$ válidas, optamos por usar janelas estendidas, com 27x27

(ou 729) pixels. Para filtrar um pixel são necessárias uma vizinhança (49) e uma janela estendida (729), ambas centralizadas neste pixel. A entrada para nosso sistema é composta por pares de pixels. Então são necessários 389 (metade de 729 mais 49) para filtrar um pixel. Para cada pixel da janela (não estendida) é calculado seu peso em relação à vizinhança do pixel que se deseja filtrar o ruído. Assim um total de 441 ciclos são gastos nesta etapa. Durante o cálculo dos pesos, a memória utilizada não deve ser alterada. Pensando nisto, duplicamos a memória dinâmica especializada (MDE), ver o fluxo completo de remoção de ruídos na Figura 29, para que, enquanto uma estiver sendo utilizada para calcular os pesos a outra possa ser preenchida, a fim de que sempre tenhamos dados para calcular os pesos(Gambarra, L. L., 2012) e (Gambarra, L.L., Lima, J.A.G., Silva, H.S.,Batista, L.V., Marques, D.S., 2012).

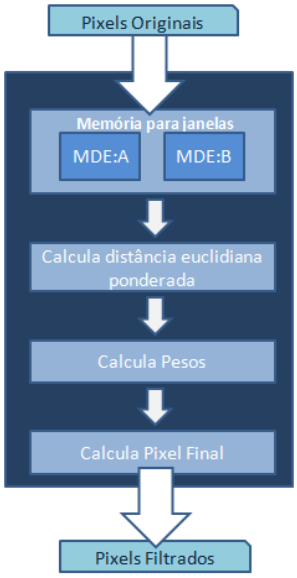

Figura 29: Fluxo da redução de ruídos para o NLM em FPGA.

3.3.2.4. Aproximação da função exponencial

O NLM faz uso intensivo do cálculo de exponenciais, como mostram as equações 10, usada para calcular os pesos, e 11, onde $Z(i)$ é a fator de normalização. A implementação de uma unidade de exponenciação em hardware não é comum em muitos processadores comerciais, o que leva a se implementar essa operação em software, através da combinação de tabelas de busca, multiplicações e adições em

ponto flutuante. Esses algoritmos são muito lentos quando comparados a uma implementação em hardware (Pottathuparambil & Sass, 2008).

$$w(i,j) = \frac{1}{Z(i)} e^{-\frac{S(i,j)}{h^2}} \quad \textbf{(10)}$$

$$Z(i) = \sum_j e^{-\frac{S(i,j)}{h^2}} \quad (\mathbf{11})$$

Analisando as características do algoritmo é possível constatar que, para o cálculo do NLM, a precisão da exponenciação não é essencial, mas sim sua característica de decaimento em função da distância Euclidiana quadrática. Essa observação permitiu o uso de uma aproximação linear por partes (Bellman & Roth, 1969) para a função exponencial. A Figura 30 permite visualizar essa aproximação feita com a utilização de dez retas para aproximar o decaimento exponencial.

Na aproximação linear por partes a curva da função é aproximada por retas, cujos pontos podem ser calculados pela equação da reta, ver a equação 12, onde m é o coeficiente angular e b o coeficiente linear. Além das operações de soma e subtração, a multiplicação e a divisão podem ser implementadas com *pipelines*, (Panato, Silva, Wagner, Johann, Reis, & Bampi, 2004) e (Takagi, Kadowaki, & Takagi, 2005).

$$y = mx + b \quad (\mathbf{12})$$

Obtidos os pesos relativos a cada *pixel* calcula-se o valor final do pixel, de acordo com a Equação 13, novamente com o uso de *pipelines*.

$$NL[v][i] = \sum_{j\,\epsilon\,I} w(i,j)\, v(j), \quad (\mathbf{13})$$

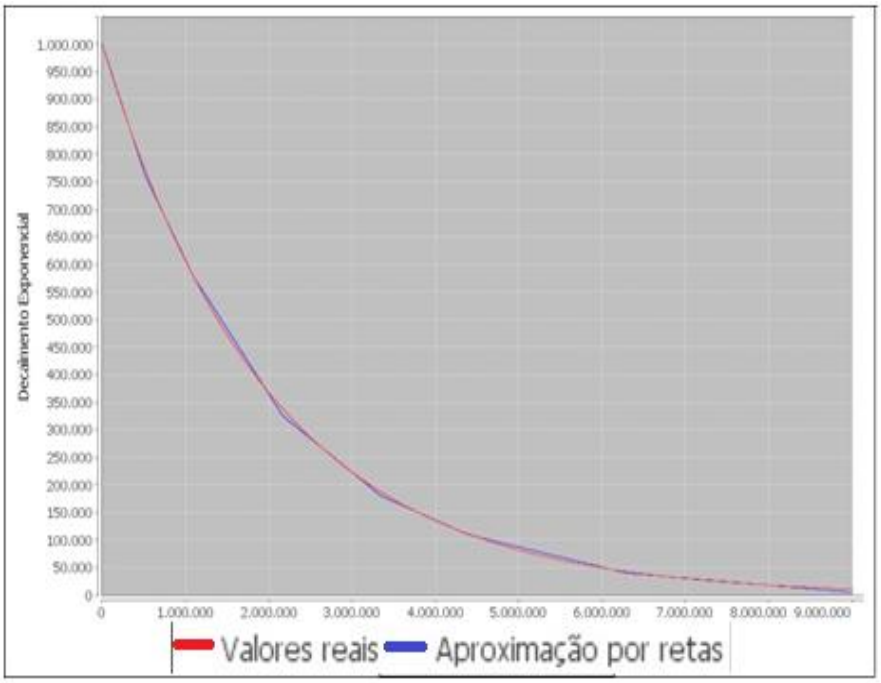

Figura 30: Aproximação do decaimento usando dez retas.

3.4 Processamento Paralelo

Uma característica importante do *NLM* é que a filtragem de um pixel é independente da filtragem dos demais. Essa propriedade permite que cada pixel possa ser calculado por um núcleo NLM de filtragem diferente multiplicando a capacidade de processamento até atingir o limite da interface de E/S. A Figura 31 mostra dois núcleos *NLM* independentes usados para acelerar a filtragem.

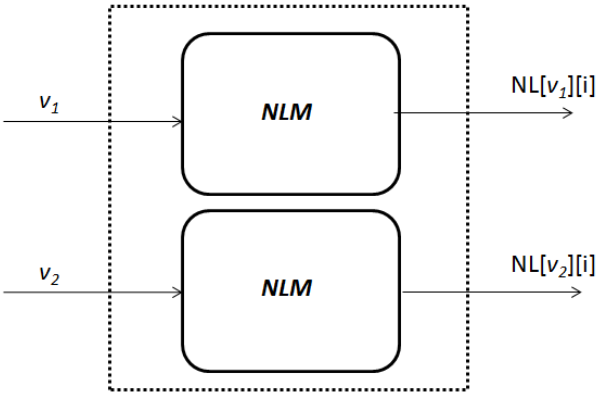

Figura 31: Dois núcleos *NLM* para filtragens independentes.

3.5. Implementação e Simulação

O filtro proposto foi descrito usando a linguagem de descrição de hardware (HDL) SystemVerilog. A descrição foi compilada e sintetizada para o dispositivo EP3SL50F484C2 da família Stratix III usando o software Quartus II v.10.1 da Altera.

Para validar a funcionalidade, simulações de filtragem de ruído foram realizadas utilizando o software QuestaSim v.10.0 da Mentor Graphics. Pixels de uma mesma imagem (com ruído) foram enviados tanto para o filtro descrito em SystemVerilog, quanto para outro descrito em linguagem Java, supostamente correto. Os pixels resultantes de ambos são comparados para verificar a correção. Este processo pode ser visto na Figura 31 (Gambarra, L.L., 2012) e (Gambarra, L. L., Lima, J.A.G., Silva, H.S.,Batista, L.V., Marques, D.S., 2012).

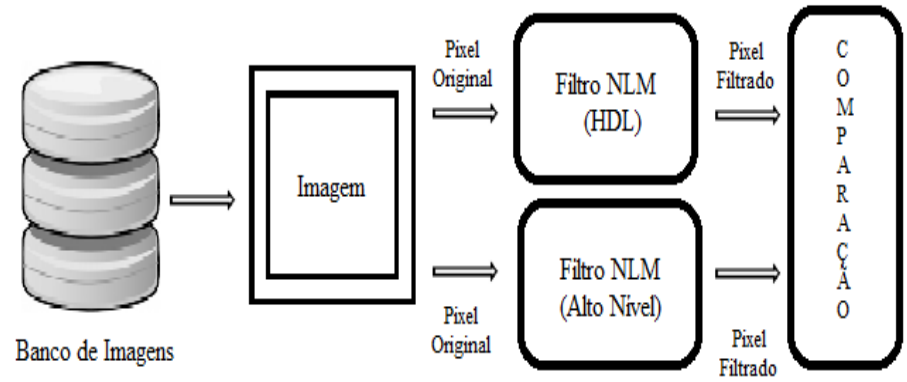

Figura 31: Metodologia de simulação.

3.6. Visão do sistema

O sistema idealizado é composto de uma combinação em software e hardware para a implementação do algoritmo NLM, como mostra a Figura 32.

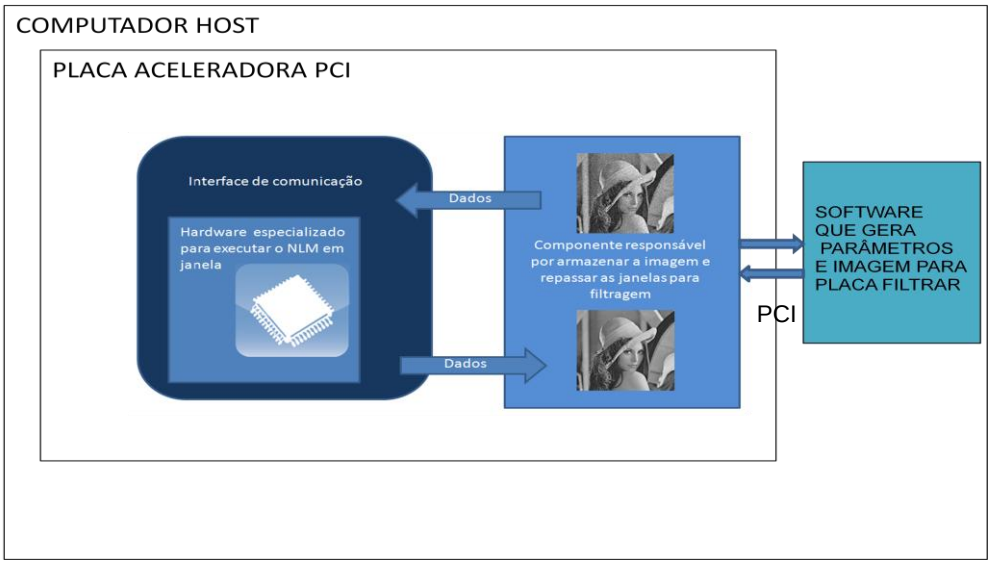

Figura 32: Visão externa do sistema.

O software, presente no computador *host*, fica encarregado de transferir a imagem e passar parâmetros para a Memória da Placa Aceleradora PCI. O mesmo deverá efetuar suas atividades e se comunicar com o hardware especializado para este continuar com o processamento do filtro. O hardware especializado fica embutido no Kit de prototipagem *PCI Development Board* da Altera, ver Figura 33, e sua comunicação com o software se dá mediante a interface PCI do computador *host*. As atividades que o hardware efetua compreende o processamento do algoritmo NLM, bem como a interface com a Memória presente na Placa Aceleradora. O bloco **Componente responsável por armazenar a imagem e repassar as janelas para filtragem** é implementado na Placa Aceleradora PCI e fará interface, via barramento PCI com o bloco **Software que gera Parâmetros e Imagem para a Placa Filtrar**. Este bloco é constituído de um ***driver*** especifico para controlar a Placa Aceleradora PCI mais os programas da aplicação que manipulam a imagem a ser filtrada, como mostra em diagrama de blocos a Figura 34.

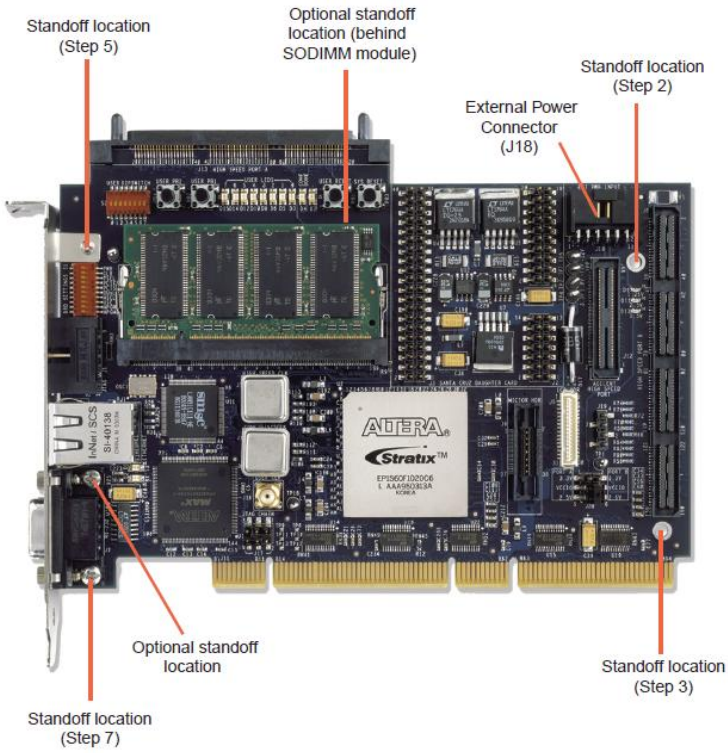

Figura 33: Kit de prototipagem *PCI Development Board* da Altera.

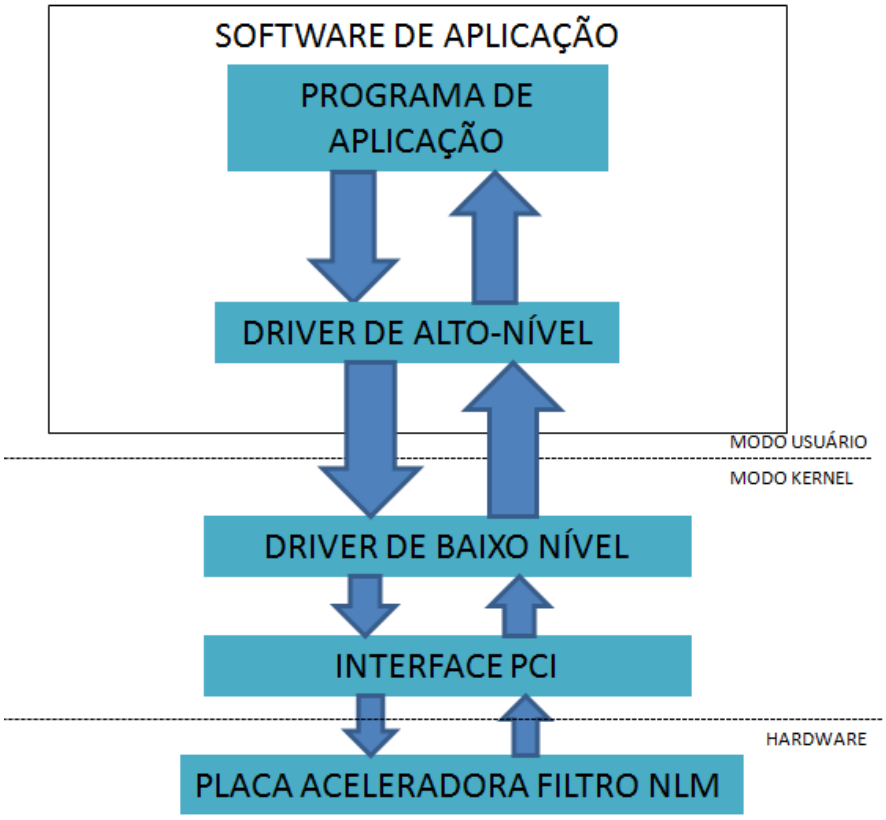

Figura 34: Software em Diagrama de Blocos.

3.6.1 Kit de Desenvolvimento *PCI High-Speed* da Altera

A solução proposta para a otimização do algoritmo *NLM* foi implementada utilizando o kit de desenvolvimento oferecido pela Altera, ver Figura 32. A utilização destes kits de prototipagem de projetos de circuitos integrados facilitam o desenvolvimentos de projetos em FPGA. Os kits de desenvolvimento são placas que possuem, além do dispositivo FPGA, interfaces de E/S para testes do sistema programado a ser embutido dentro da FPGA. O kit de prototipagem *Statix PCI* é uma plataforma de avaliação e desenvolvimento para interfaces de alta velocidade incluindo *PCI*, *PCI-X*, com suporte a dispositivos *FPGA* EP1S60F1020 DA FAMÍLIA *Statix*(Altera, 2003), uma memória DDR-SDRAM de 256 MBytes para uso nas aplicações desenvolvidas. O dispositivo FPGA EP1S60F1020 é conectado a todos os componentes através de interfaces adequadas *on-chip* e circuitos na placa, como mostra a Figura 35. O filtro NLM proposto escrito na linguagem de descrição de *hardware verilog* é programado no dispositivo EP1S60F1020 através da interface USB por meio do cabo *ByteBlaster I*. A Figura 36 mostra em diagrama de blocos o projeto de referência da placa aceleradora. O kit de desenvolvimento oferece implementado em seu dispositivo FPGA controle para a memória DDR-SDRAM, tanto para a transferência de dados do *HOST*, via barramento PCI, para a memória DDR do kit, como da memória DDR do *kit* via barramento PCI, para o *HOST*. Deste modo, a imagem a ser filtrada é transferida do HOST para a DDR do kit, em seguida é disparada a filtragem, implementada no dispositivo FPGA, para ao fim da filtragem a imagem ser transferida para o HOST.

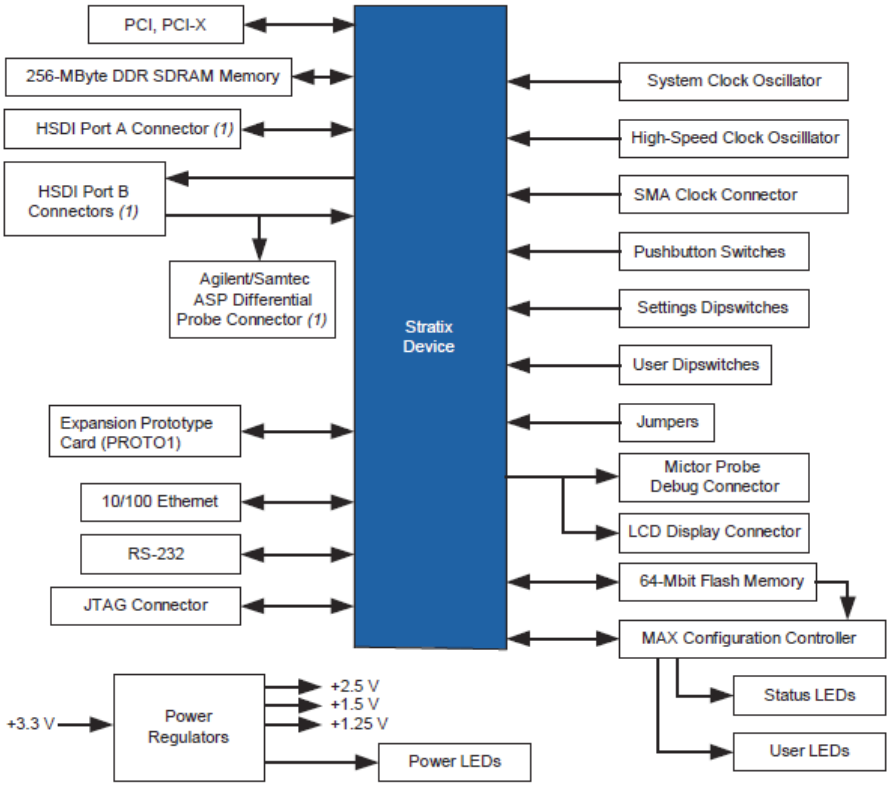

Figura 35: Diagrama de blocos das conexões internas Kit de prototipagem *PCI* da Altera.

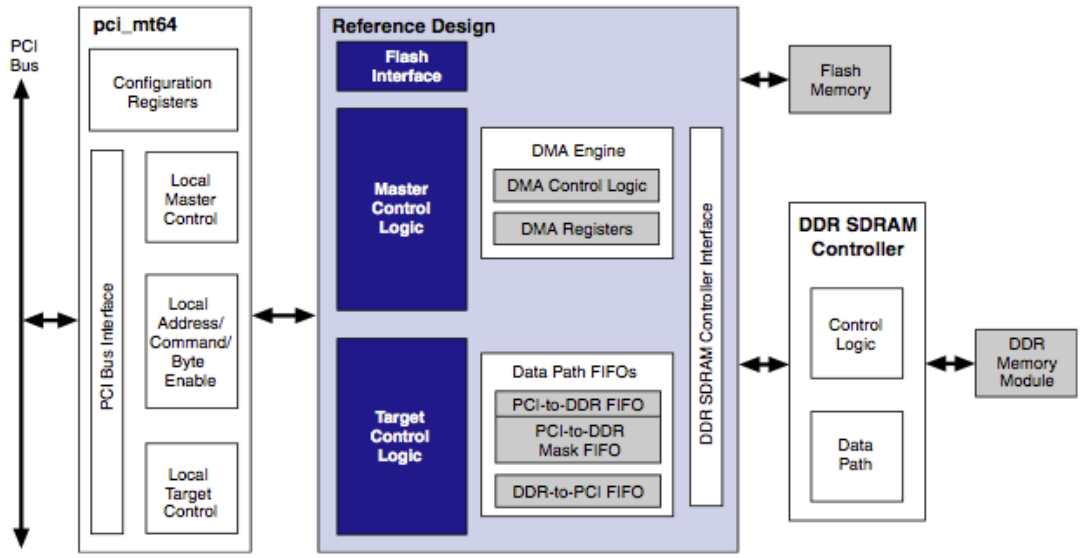

Figura 36: Projeto de referência do Kit de prototipagem *PCI* da Altera.

Capítulo IV: Implementação em GPU CUDA para o Algoritmo *Non Local Means*

Este capítulo apresenta a proposta de uma implementação no ambiente densamente multiprocessado CUDA para otimização do Algoritmo *Non Local Means*. Inicialmente é apresentado o modelo de programação NVIDIA CUDA e em seguida é mostrada a versão otimizada para GPU do NLM.

4.1 Implementação do NLM em GPU

A linguagem utilizada foi CUDA C, a implementação foi feita para que a GPU processe todo o NLM. A versão utilizada na implementação é a do NLM em janela proposto por (BUADES, COLL e MOREL, 2005), a mesma usada na implementação em FPGA, a qual diminui a complexidade do *NLM*, e define a utilização de uma janela de busca limitada dentro da qual a busca por vizinhanças semelhantes será efetuada. Com esta abordagem, a complexidade de tempo é reduzida para *O(nsw)*, sendo *s* o tamanho da janela de busca, *w* o tamanho da vizinhança referente ao *Kernel Gaussiano* bidimensional, e n o tamanho da imagem em *pixels*. Esta otimização mostrou-se também eficiente na redução de ruído além do óbvio ganho de desempenho.

O segredo de se programar em CUDA é o de se preocupar com o uso da memória, uma vez que os acessos a memória global se tornam um gargalo para o procedimento, pois é a memória mais lenta, de 400 a 600 ciclos de relógio, enquanto a memória local compartilhada por bloco gasta cerca de 20 a 40 ciclos de relógio. Portanto, quanto mais se usar as memórias de acesso local, como a compartilhada, menor será a ociosidade das *threads*.

A implementação em CUDA, realizada no nosso laboratório de pesquisa LASID e apresentada no exame de qualificação da tese de mestrado de Leandro(Leandro

Figueiredo, exame de qualificação mestrado, UFPB, 2013) e na monografia de conclusão de curso de Sidney(Sidney Pontes Filho, monografia de conclusão do Curso de Bacharelado em Ciência da Computação, UFPB, 2013), é feita armazenando a imagem a ser filtrada na memória global da Placa de Vídeo, em seguida a janela de busca é colocada na memória compartilhada de cada bloco da Placa de Vídeo. A Figura 37 mostra como a memória compartilhada é manipulada, de forma que os quadrados centrais representam os *pixels* de cada *thread* dentro do bloco. Os *pixels* de borda precisam também ler da memória global os elementos de sua vizinhança, deste modo tem-se uma borda em torno do bloco, de tamanho relacionado com a janela de busca e a vizinhança pesquisada dos *pixels*. Com esta configuração, cada *pixel* é lido uma vez da memória global, e todo o cálculo de similaridade de vizinhanças é feito com leitura na memória compartilhada.

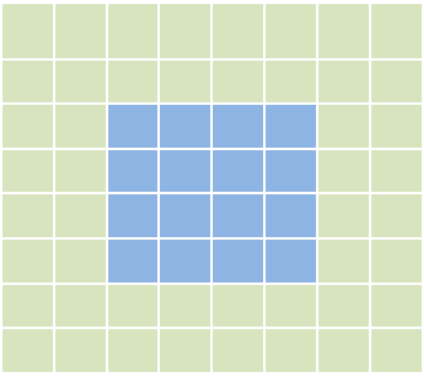

Figura 37: Memória compartilhada para armazenar janela de busca

Na implementação em CUDA cada *thread* é responsável pelo processamento de um único *pixel* que são limitados pela janela de busca armazenada na memória compartilhada. Esta memória armazena todas as janelas de busca dos *pixels* do bloco de *threads* responsável e mais uma borda que inclui os *pixels* da janela de vizinhança, isto quando se está comparando os *pixels* que ficam perto do limite da janela de busca. No código, o tamanho do bloco da janela de busca é de 441 (21x21) *pixels* e da janela de vizinhança é de 49 (7x7) *pixels*, esses tamanhos são sugeridos pelos criadores do NLM. Considerando a placa gráfica utilizada, o tamanho do bloco de

threads é de 256 (16x16) *threads*, assim, a quantidade armazenada na memória compartilhada é de 1764 (42x42) *pixels*, ou seja, 1764 *bytes*, que é facilmente armazenado, pois as placas atuais tem capacidade de até 48 KB nesta memória.

A Figura 38 ilustra um exemplo do armazenamento na memória compartilhada e como funciona a definição do seu tamanho. Neste exemplo o tamanho da janela de busca é de 81 (9x9) *pixels*, da janela de vizinhança é de 9 (3x3) *pixels* e do bloco é de 25 (5x5) *threads*, sendo assim, precisa-se armazenar 225 (15x15) *pixels* na memória compartilhada.

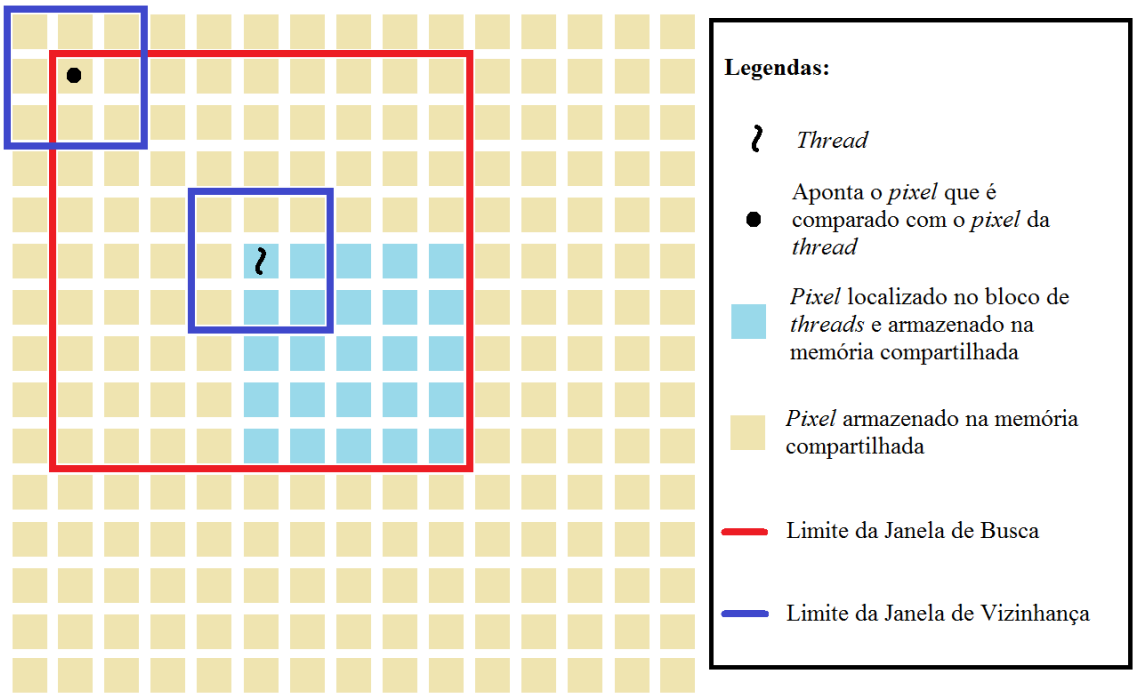

Figura 38 - Armazenamento na Memória compartilhada e definição do seu tamanho

A sequência de operações de cada pixel, que define uma *thread*, obedece a mesma lógica utilizada nas implementações realizadas na linguagem de programação *java*, para computadores de uso geral, e na linguagem de descrição de *hardware verilog*, para circuitos *FPGA*. Assim, cada *thread* é executada por um núcleo GPU e processa um *pixel*, realizando as seguintes operações:

1. Calculo da Distância Euclidiana Quadrática.
2. Calculo dos Pesos.

3. Calculo do Valor Final.

Considerando que o processamento de cada pixel é independente, pode-se filtrar simultaneamente tantos *pixels* quanto a quantidade de núcleos CUDA existentes na placa. Assim, se a placa de vídeo tiver 192 núcleos CUDA, serão filtrados simultaneamente 192 pixels da imagem.

4.2 Avaliação de Desempenho

O tempo de processamento das imagens em CUDA será comparado com os tempos em software e em hardware de (Gambarra, L., Lima, J.A.G., Silva, H.S., Batista, L. V., Marques, D. S., 2012), no qual se utilizou imagens de tamanho 512x512 e 1024x1024, somente com uma banda, usando janela de busca de tamanho 21x21 e janela de vizinhança de tamanho 7x7, que são as mesmas características implementadas em CUDA.

Para se realizar os testes, foi utilizado um computador com a seguinte configuração:

- CPU: Intel® Core™ i3-540
    - Número de *Cores*: 2
    - Número de *Threads*: 4
    - Frequência: 3.06 GHz
    - Intel® Smart Cache: 4 MB
- Memória RAM: 8 GB DDR3
- Sistema operacional: Windows 7 Professional 64-bit
- GPU: NVIDIA GeForce GTX 550 Ti
    - NÚCLEOS CUDA: 192
    - Frequência gráfica: 900 MHz
    - Frequência do processador: 1800 MHz
    - Memória de vídeo dedicada: 1024 MB GDDR5
    - Frequência da memória: 4100 MHz

- Interface da memória: 192-bit
- Largura de banda da memória: 98.4 GB/s
- Versão do CUDA: 5.0

Capítulo V: Resultados

Nesta sessão são apresentados os resultados da síntese em FPGA do algoritmo NLM embutido na Placa Aceleradora PCI, os resultados da implementação em GPU CUDA, bem como os resultados e as comparações do tempo de execução e da qualidade da filtragem entre a versão em FPGA, a versão em software do NLM em janela e a implementada em GPU CUDA. Bem como, os resultados da aquisição de imagens reais obtidas com uma câmara digital infravermelho.

5.1. Resultados da Implementação em FPGA

A validação da proposta foi feita através da implementação utilizando o dispositivo EP3SL50F484C2 da família *Stratix III* da *Altera*, um núcleo do filtro de ruídos desenvolvido ocupa 34% da lógica e 2% da memória. Foi alcançada uma frequência máxima de operação de 104,78 *MHz* (período de relógio de aproximadamente 9,54*ns*) (Gambarra, L.L., 2012] e [Gambarra, L. L., Lima, J.A.G., Silva, H.S.,Batista, L.V., Marques, D.S., 2012).

Para o mesmo dispositivo, dois núcleos ocupam 68% da lógica e utilizam 4% da memória, sendo que nesse caso foi alcançada uma frequência máxima de operação de 96,26 *MHz (*período de relógio de aproximadamente 10.38ns (Gambarra, L.L., 2012) e (Gambarra, L. L., Lima, J.A.G., Silva, H.S.,Batista, L.V., Marques, D.S., 2012).

5.1.1 Qualidade da filtragem

O filtro proposto em hardware obteve resultados comparáveis ao NLM original em janela em termos do erro quadrático médio MSE. A Figura 39 mostra a imagem original, a imagem com adição de ruído, a imagem após o processamento do NLM em software e a imagem após o processamento da versão proposta em FPGA., a qual permite uma percepção visual entre a versão do NLM em janela (software) e a versão

proposta em FPGA (Gambarra, L.L., 2012) e (Gambarra, L. L., Lima, J.A.G., Silva, H.S.,Batista, L.V., Marques, D.S., 2012).

A Tabela 2 mostra uma comparação entre a proposta de implementação do NLM em FPGA e a implementação do NLM em software com relação ao erro quadrático médio (MSE), para ruídos com diferentes desvios padrão. Note que a leve diferença no MSE, ver Figura 40, se deve ao uso de aproximações de números fracionários por inteiros e ao uso da aproximação linear por partes da função exponencial (Gambarra, L.L., 2012) e (Gambarra, L. L., Lima, J.A.G., Silva, H.S.,Batista, L.V., Marques, D.S., 2012).

***Figura* 39: Da esquerda para a direita e de cima para baixo: imagem original, imagem com adição de ruído o resultado do NLM em software(MSE 16,4) e resultado da solução em *hardware* proposta(MSE 15,4).(Gambarra,L.L.,2012).**

Método utilizado	**Desvio Padrão**			
	5	***10***	***15***	***20***
NLM em janela (software)	8,97	23,5	39,97	109,5
NML em FPGA	8,89	24,7	41,4	110,5

Tabela 2: Comparações de MSE

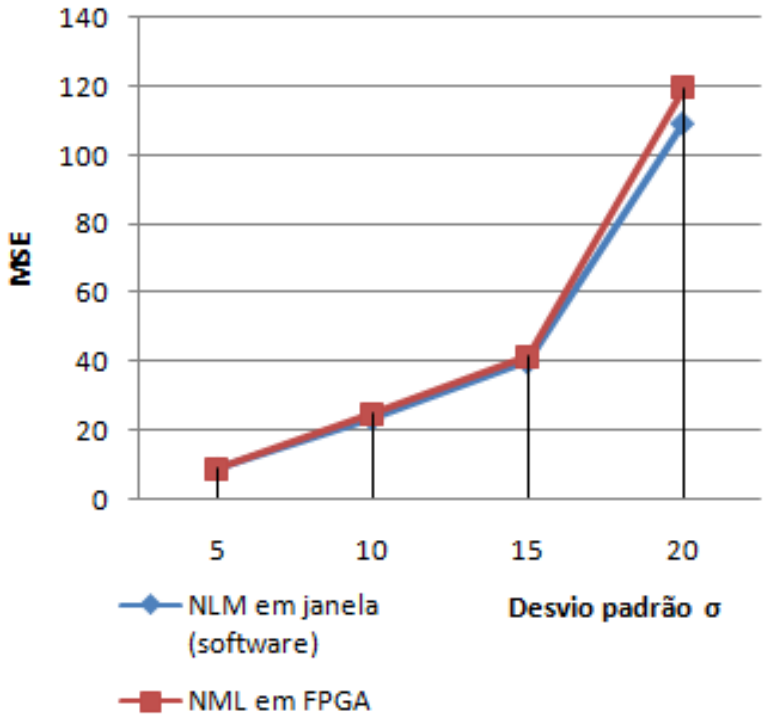

Figura 40: Gráfico correspondente da Tabela 2.

5.1.2 Medidas de Desempenho

Seguindo a sugestão de (Buades, Coll e Morel, 2005) utilizamos 7 como valor de M, e uma janela de pesquisa de $21 * 21$ pixels. As operações relativas ao cálculo da distância Euclidiana quadrática são efetuadas em hardware paralelo e todas as outras operações são realizadas em *pipeline*, com exceção dos 441 ciclos gastos calculando pesos para filtrar um pixel que continuam sequenciais uma vez que possuem dependência. Essa informação juntamente com o período de relógio T (9,45*ns* obtidos na simulação) permite determinar o tempo gasto para filtrar uma imagem. Assim, para uma imagem com N^2 pixels são gastos $441 * N^2 * T$. É importante destacar que como o algoritmo foi implementado em função do período T, se utilizarmos dispositivos FPGA ou *Standard Cell* com tecnologia mais rápida, o tempo de processamento de um *pixel* diminui e teremos resultados melhores que o apresentado na Tabela 3. A Tabela 3 e a Figura 41, apresenta os experimentos realizados em um PC com um processador Core I5 2.53GHz e 6GB de RAM que processou o NLM em software para filtrar imagens de tamanho 512x512, 1024x1024 e

2592x1944 comparando os tempos obtidos com os tempos da implementação proposta em FPGA, usando um núcleo, dois núcleos e quatro núcleos para dispositivos com relógio T=9,45 *ns*.

Tamanho da imagem	Método utilizado (tempos em segundos)			
	NLM em software (monothread)	***NLM em FPGA (um núcleo)***	***NLM em FPGA (dois núcleos)***	***NLM em FPGA (quatro núcleos)***
512x512	177,7	1,1	0,6	0,18
1024x1024	677,2	4,41	2,4	0,39
2592x1944	3440,5	21,2	11,56	1,87

Tabela 3: Medidas de desempenho para T=9,45*ns*.

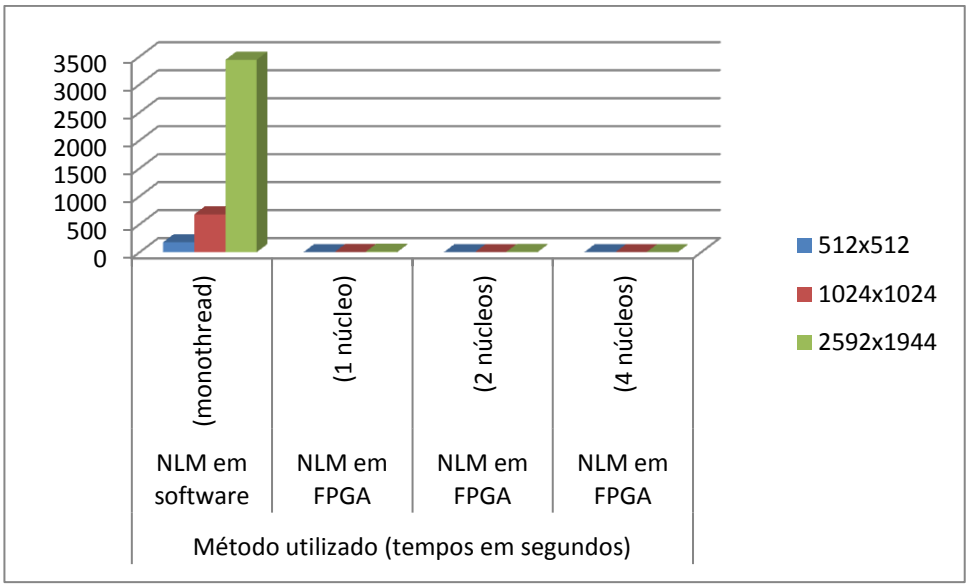

Figura 41: Desempenho em tempo para implementações em software versus FPGA com relógio T=9,45.

Os resultados obtidos na simulação mostram que o algoritmo para uma imagem monocromática de tamanho 512x512, é executado em média 481 vezes mais rápido do que a implementação em software, os quais ainda podem ser melhorados empregando-se mais núcleos de filtragem, uma vez que cada pixel pode ser calculado por um núcleo *NLM* de filtragem diferente, multiplicando a capacidade de processamento ao custo de um maior consumo de recursos.

Os resultados da redução de ruídos são semelhantes ao algoritmo original tanto em *MSE* quanto em percepção visual. A confirmação da validade do trabalho aqui exposto é reforçada pela publicação do artigo intitulado "*Otimização do algoritmo Non-local means utilizando uma implementação em FPGA*" aceito para apresentação no Workshop XVIII IBERCHIP-2012 entre 29 de Fevereiro e 2 de Março de 2012 em Playa del Carmen, México.

A Tabela 4 apresenta resultados de filtragem para imagens de tamanho 512x512, 1024x1024 e 2592x1944 considerando uma implementação do algoritmo em dispositivos de tecnologia mais rápida que determinam um período T de 4*ns*. Os resultados obtidos mostram que o algoritmo para uma imagem monocromática de tamanho 512x512, é executado em média 1211 vezes mais rápido, que podem ser melhorados usando dispositivos com relógio mais rápido ou aumentando-se a quantidade de núcleos de filtragem NLM.

Tamanho da imagem	**Método utilizado (tempos em segundos)**			
	NLM em software (monothread)	***NLM em FPGA (1 núcleo)***	***NLM em FPGA (2 núcleos)***	***NLM em FPGA (4 núcleos)***
512x512	177,7	0,46	0,25	0,07
1024x1024	677,2	1,85	1	0,3
2592x1944	3440,5	8,89	4,85	1,44

Tabela 4: Medidas de desempenho para T=4*ns*.

5.2 Resultados da Implementação em GPU CUDA

Os testes foram realizados utilizando a placa de vídeo NVIDIA GeForce GTX 550 Ti, que é uma placa simples com 192 CUDA núcleos, enquanto as mais potentes têm mais de 3.500. A abordagem utilizada na implementação em CUDA foi o algoritmo *Non-Local Means* que utiliza janela de busca, que além de reduzir a complexidade de tempo, melhora a filtragem de ruídos em imagens naturais. Essa abordagem é excelente para a implementação em CUDA, pois todas as operações do algoritmo foram realizadas utilizando uma das memórias mais rápidas da GPU, que é a

compartilhada. O que confirma o estudo de (Cope, B., Peter, Y.K., Howes, L., Luk W., 2010) onde implementações em GPU oferecem desempenho superior nas aplicações que utilizam reuso de variáveis, ou seja, no nosso caso várias operações com a mesma janela de busca.

Apesar dos dados exibidos na Tabela 5 mostrarem que a implementação em CUDA em média gasta mais tempo de processamento que a implementação em *hardware*, tem-se a vantagem em CUDA de uma maior facilidade, flexibilidade e menor tempo de implementação. Mesmo assim, não foi utilizado uma placa gráfica de grande porte, pois essas placas podem ter mais de 3.500 núcleos CUDA, enquanto a que foi utilizada nos testes tinha 192, a cada ano as GPUs vão ganhando mais núcleos e consequentemente mais desempenho.

Mesmo utilizando uma GPU de porte mediano, o tempo em CUDA ficou melhor que em hardware FPGA com um núcleo e T=9,45 ns. A Tabela 5 e a Figura 40 apresentam o tempo de processamento gasto na implementação em CUDA com o da implementação em *software(java)*, e o da Implementação em FPGA com período T= 4,0 *ns*.

Tamanho da imagem	Método utilizado(tempos em segundos)				
	NLM software (monothread)	*NLM em FPGA (1 núcleo)*	*NLM em FPGA (2 núcleos)*	*NLM em FPGA (4 núcleos)*	NLM CUDA (192 núcleos)
512x512	177,7s	0,46	0,25	0,07	0,46s
1024x1024	677,7s	1,85	1	0,3	1,82 s

Tabela 5: Comparação em tempo das implementações.

A Figura 42 mostra um gráfico que compara o desempenho em tempo das implementações em CUDA e FPGA com 1,2 e 4 núcleos todos com relógio T= 4,0 *ns*.

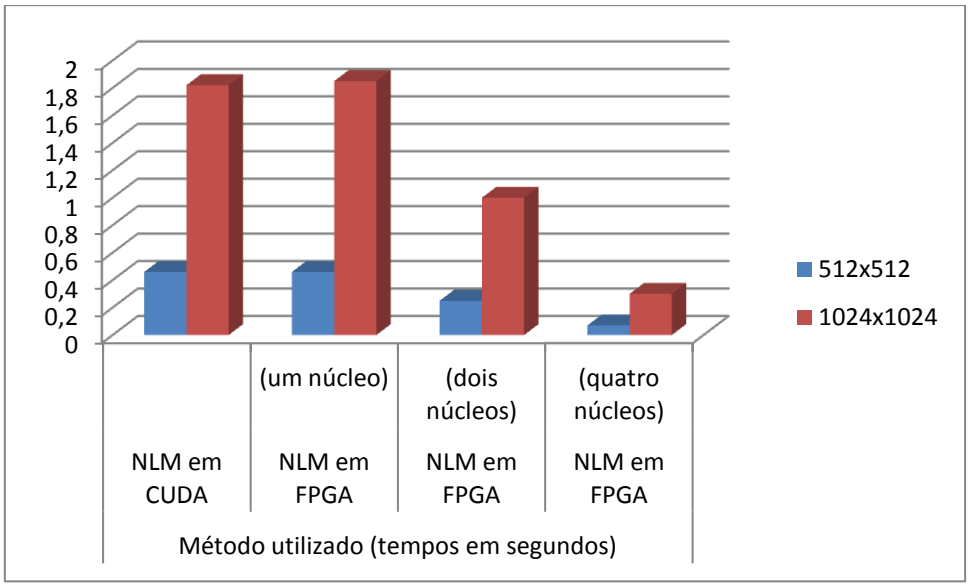

Figura 42: Desempenho em tempo implementações em CUDA versus FPGA

5.3 Comparação com outros trabalhos relacionados

Comparamos os resultados em tempo obtidos nas duas abordagens propostas com trabalhos já validados na literatura, e descritos resumidamente na seção 3.1: (Dinesh, Govindan e Mathew, 2009); (Brox, Kleinschmidt e Cremers, 2008); (Dauwe, Goossens, et al., 2008). A avaliação foi feita com relação ao tempo gasto para filtrar imagens de resolução 512x512, a Tabela 6 mostra os resultados obtidos por cada um dos trabalhos citados. Utilizamos os resultados de simulação das implementações propostas em FPGA, com quatro núcleos de filtragem, e em GPU CUDA com 192 núcleos CUDA para comparar com os trabalhos citados. Os tempos utilizados nas propostas de (Brox, Kleinschmidt e Cremers, 2008) e (Dinesh, Govindan e Mathew, 2009) usadas para comparação são os obtidos na implementação em cluster de computadores. A Figura 43 mostra o gráfico em colunas correspondente à Tabela 6, ressaltando os tempos obtidos por cada trabalho. As abordagens propostas em hardware(FPGA) e CUDA obtiveram os melhores resultados, ou seja, menos tempo para executar a filtragem NLM.

TRABALHO	TEMPO (S)	ABORDAGEM
PROPOSTO	0,07	EXECUÇÃO EM FPGA COM 4 NUCLEOS E T=4,0 *ns*
PROPOSTO	0,46	GPU CUDA
(DAUWE,GOOSSENS, ET AL)	565	ESTATISCAS DE ORDEM SUPERIOR
(DINESH, GOVINDAN e MATHEW, 2009)	14	PRÉ-SELEÇÃO BASEADA EM PATCHES DE ORDEM SUPERIOR
(BROX, KLEINSCHMIDT e CREMERS)	5,4	PRÉ-SELEÇÃO DE VIZINHANÇAS USANDO SUB-AMOSTRAGEM ESPACIAL

Tabela 6: Comparação em tempo com outras abordagens.

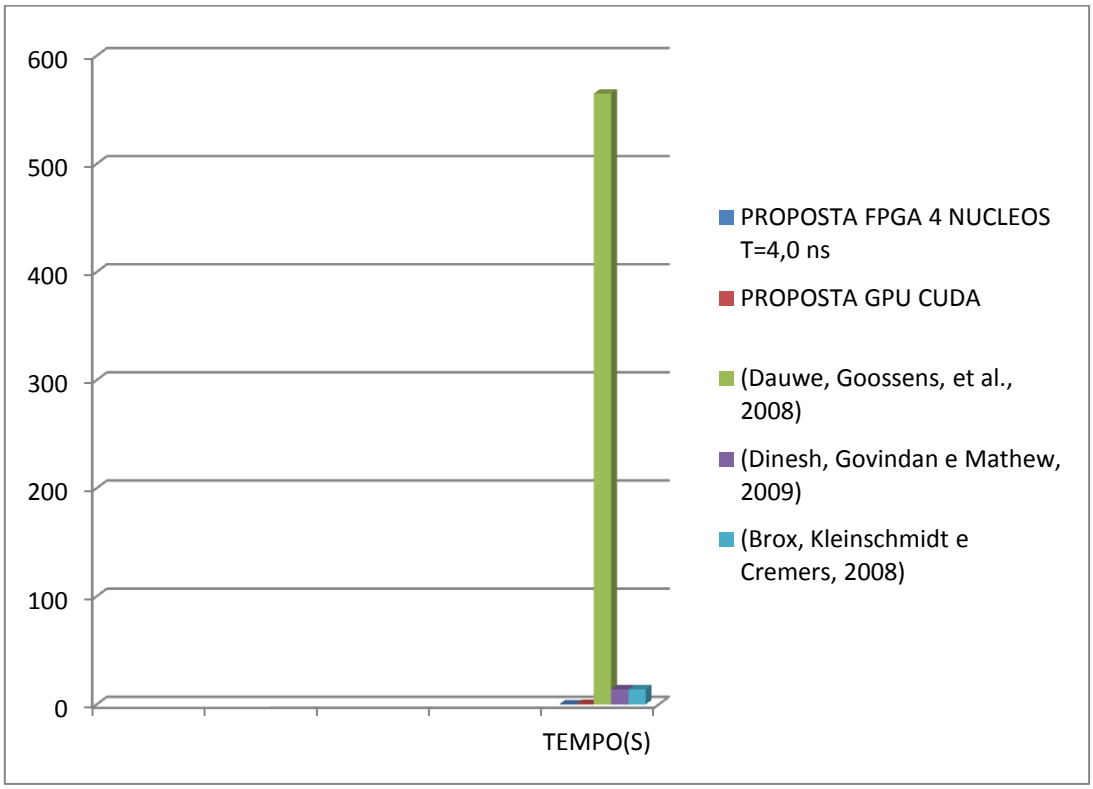

Figura 43: Comparação em tempo com outras abordagens validadas na literatura

5.4 Filtragem de Imagens Reais

A título de exemplo de utilização de filtragem de imagens em tempo real, resolvemos fotografar um celular NOKIA com sensor de 41 Megapixels enquanto filmava durante 10 minutos, com resolução *Full HD* de 1080x1920, com o objetivo de observar o aquecimento de seus circuitos internos. As imagens em infravermelho foram captadas utilizando a Câmara FLIR A600 colorida com resolução de 640x480 *pixels*. A Figura 40 mostra a parte interna do celular NOKIA com os componentes a serem observados.

Figura 44: Parte Interna do celular NOKIA (Fonte NOKIA).

A Figura 45 mostra a lado a lado a imagem original captada pela Câmara FLIR A600 e a imagem após o processamento da Filtragem NLM. A filtragem foi feita utilizando as três bandas RGB, e usamos a implementação em núcleos CUDA. As fotos foram tiradas em intervalos regulares de aproximadamente 2 minutos. A filtragem NLM de cada imagem durou 1,79s. Observa-se o crescimento da emissão de calor ao longo de 10 minutos de captação da imagem. Constata-se que mesmo captando as imagens em condições ótimas em uma sala de laboratório com ar-condicionado, isolada de interferências externas é possível notar que as imagens filtradas apresentam detalhes mais nítidos.

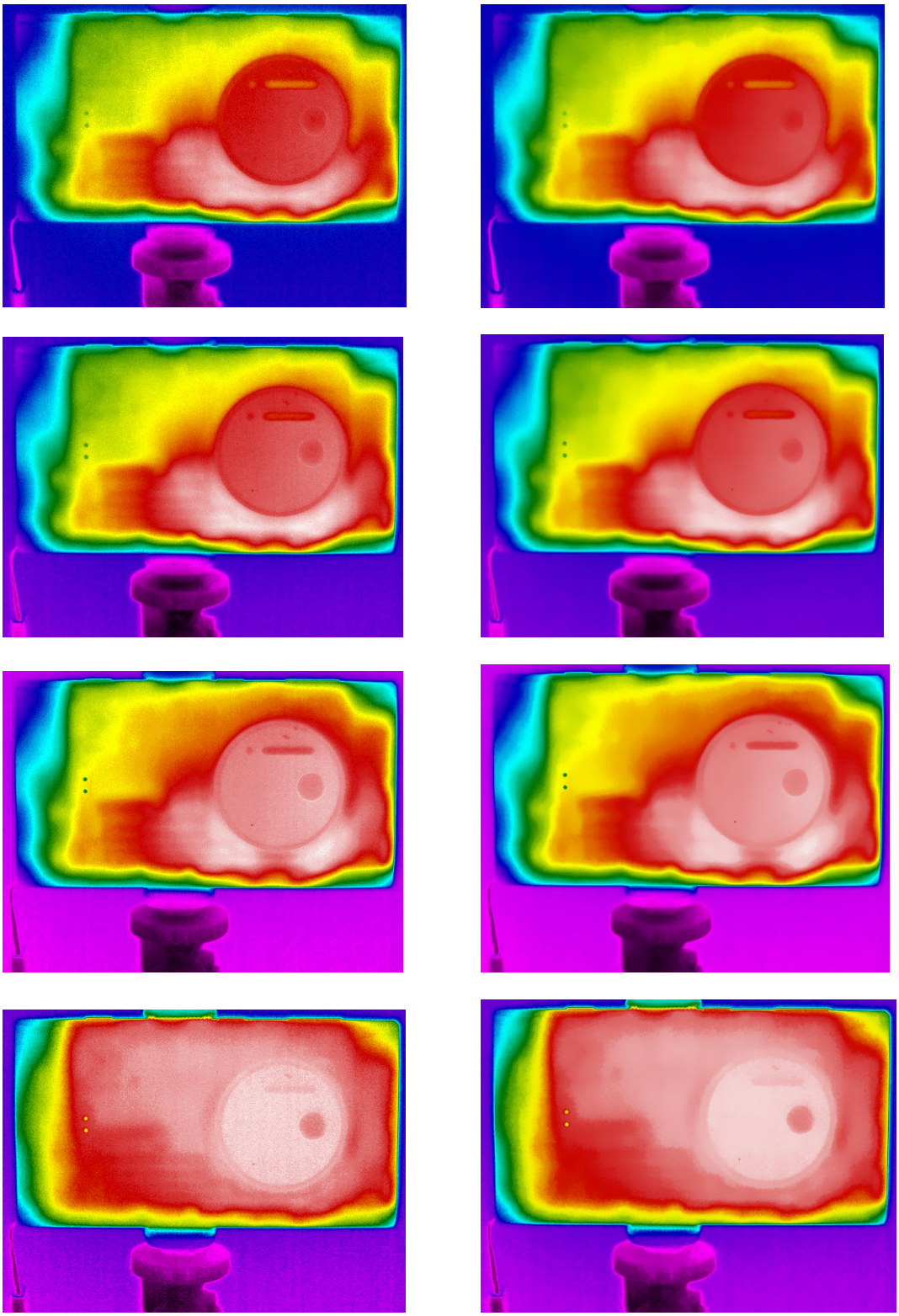

Figura 45: Imagem original captada com a Câmara FLIR A600 e ao lado a imagem filtrada.

Capítulo VI: Conclusão

O presente trabalho apresentou duas propostas para otimizar e implementar o algoritmo de remoção de ruídos *Non Local de Means* (NLM) visando sua aplicação em tempo real. Procurou-se manter os resultados perto daqueles do algoritmo original em termos de *MSE* e *ruído do método*, embora seja difícil definir indicadores na área de qualidade de imagens considerando que indicadores matemáticos nem sempre refletem a apreciação subjetiva do observador.

A abordagem de referência usada para a otimização do NLM é a filtragem através de janelas de buscas sugerida por (Mahmoudi & Sapiro, 2005), cujos resultados reportam melhor desempenho em áreas uniformes quando comparados com o algoritmo NLM original. Com base nessa suposição, define-se uma janela de busca dentro da qual é realizada a procura por vizinhanças semelhantes, reduzindo a complexidade do NLM.

A primeira proposta trata do desenvolvimento de uma placa aceleradora para computadores que possuam Barramento PCI; o software, presente no computador *host*, fica encarregado de transferir a imagem e passar parâmetros para a Memória da Placa Aceleradora PCI. O mesmo deverá efetuar suas atividades e se comunicar com o hardware especializado para este continuar com o processamento do filtro. O hardware especializado fica embutido na Placa Aceleradora PCI, e sua comunicação com o software se dá mediante a interface PCI do computador *host*. As atividades que o hardware efetua compreende o processamento do algoritmo NLM em janelas, bem como a interface com a Memória presente na Placa Aceleradora.

Uma característica importante do *NLM* é que a filtragem de um pixel é independente da filtragem dos demais. Essa propriedade permite que cada pixel possa ser calculado por um núcleo NLM de filtragem diferente multiplicando a capacidade de processamento até atingir o limite da interface de E/S. Portanto, o desempenho do

filtro melhora na medida aumentando a quantidade de núcleos NLM. Por outro lado, o núcleo de filtragem NLM foi desenvolvido utilizando técnicas de paralelismo temporal, também chamado pipeline, que condicionou a filtragem de um pixel em função do ciclo de relógio do pipeline T. Deste modo, se utilizarmos dispositivos FPGA ou *Standard Cell* com tecnologia mais rápida, o tempo de processamento de um *pixel* diminui. Portanto, aumentando-se a quantidade de núcleos NLM e/ou utilizando dispositivos FPGA ou *Standard Cell* com tecnologia mais rápida teremos resultados melhores que os obtidos na simulação apresentada no capítulo 5.

A segunda proposta trata de utilizar o ambiente densamente multiprocessado CUDA, existente nas controladoras de vídeo da NVIDIA, para o processamento do algoritmo NLM em janelas. Essa abordagem é excelente para a implementação em CUDA, pois todas as operações do algoritmo foram realizadas utilizando uma das memórias mais rápidas da GPU, que é a compartilhada. O que confirma o estudo de (Cope, B., Peter, Y.K., Howes, L., Luk W., 2010) onde implementações em GPU oferecem desempenho superior nas aplicações que utilizam reuso de variáveis, ou seja, no nosso caso várias operações com a mesma janela de busca.

Comparando as duas propostas concluímos que em termos de desempenho de tempo a implementação em CUDA em média gasta mais tempo de processamento que a implementação em *hardware* (FPGA ou *standard cell*), por outro lado tem-se a vantagem em CUDA de uma maior facilidade, flexibilidade e menor tempo de implementação. Em termos de qualidade de filtragem as duas são iguais.

Sugerimos para trabalhos futuros procurar implementar o núcleo NLM em *Standard Cell* e colocar em um único circuito integrado um número alto de núcleos NLM. Com respeito a qualidade da filtragem, sugerimos estudar uma alternativa ao NLM em janela, que seria procurar uma maneira de usar de forma rápida uma pesquisa em toda a imagem.

Referências Bibliográficas

- Adams, A., Baek, J., & Davis, A. (2010). Fast high-dimensional filtering using the permutohedral lattice. *Proc. of EuroGraphics* .
- Adams, A., Gelfand, N., Dolson, J., & Levoy, M. (2009). Gaussian KD-trees for fast high-dimensional filtering. *Proc. of ACM SIGGRAPH* .
- Altera. (2012). *SystemVerilog- Overview*. Acesso em 10 de 06 de 2012, disponível em http://www.systemverilog.org/overview/overview.html
- Araújo, H. F. (2010). *BVM: Reformulação da metodologia de verificação funcional VeriSC.* Campina Grande, Paraíba: Universidade Federal de Campina Grande.
- Bellman, R., & Roth, R. (Setembro de 1969). Curve Fitting by Segmented Straight Lines. *Journal of the American Statistical Association* , pp. 1079-1084.
- Beyond 3D Nvidia g200: Architecture and GPU analysis, 2013
- Bobda, C. (2007). Introduction to reconfigurable computing.
- Bovik, A. C. (2005). *Handbook of image and video processing.* San Diego: Elsevier.
- Brox, T., Kleinschmidt, O., & Cremers, D. (2008). Efcient nonlocal means for denoising of textural patterns. *IEEE Trans. Image Process* , 1083-1092.
- Buades, A., Coll, B., & Morel, J. M. (2005). A non-local algorithm for image denoising. *Proc. IEEE CVPR, vol. 2* , pp. 60-65.
- Buades, A., Coll, B., & Morel, J. M. (2004). On image denoising methods. *CMLA. [S.l.]* , p. 40.
- Buades, A., Coll, B., & Morel, J.-M. (05 de Outubro de 2011). *IPOL: Non-local Means Denoising*. Acesso em 26 de Janeiro de 2012, disponível em http://www.ipol.im/pub/algo/bcm_non_local_means_denoising/
- Condat, L. (Agosto de 2010). A Simple Trick to Speed Up and Improve the Non-Local Means. *Research Report hal-00512801* . Caen, Franca.

- Cope, B., Cheung, P., Luk, W., Howes, L. (April, 2010). *Proformance Comparison of Graphics Processors to Reconfigurable Logic: A Case Study. IEEE Transactions on Computers, Vol. 59, N0. 4.*

- Coupé, P., Prima, S., Hellier, P., Kervrann, C., & Barillot, C. (2008). An optimized blockwise nonlocal means denoising lter for 3-D magnetic resonance images. *IEEE Trans. Med. Imag* , 425-441.

- Coupé, P., Yger, P., & Barillot, C. (2006). Fast Non Local Means Denoising for 3D MR Images. *SpringerLink* .

- Dauwe, A., Goossens, B., Luong, H., & Philips, W. (Janeiro de 2008). A fast non-local image denoising algorithm. *Proc. of SPIE Electronic Imaging* , pp. 1331-1334.

- Dehon, A. (1996). Reconfigurable Architecture for General-Purpose Computing. *Ph.D. thesis, Massachusetts Institute of Technology* .

- Dinesh, J. P., Govindan, V., & Mathew, T. (2009). Robust estimation approach for nonlocal-means denoising based on structurally similar patches. *Int. J. Open Problems Compt. Math.*

- Figueiredo, Leandro (2013). Filtragem de Imagens Baseado no Non Local Means em Ambientes Multiprocessados, Exame de qualificação do Mestrado em Informática, UFPB.

- Filho, Sidney P. (2013). Otimização do Algoritmo Non-Local Means em GPU, monografia de conclusão do Curso de Bacharelado em Ciência da Computação, UFPB.

- Gambarra, L. L., (2012). Proposta de Implementação em Hardware para o Algoritmo Non-Local Means, Tese de mestrado em Informática, UFPB

- Gambarra, L. L., Lima, J.A.G., Silva, H.S., Batista, L.V. Marques, D.S.(2012). Otimização do Algoritmo Non-Local means utilizando uma implementação em FPGA. Iberchip 2012 Procedings, pp.72-76.

- Gonzales, R. C., & Woods, R. E. (2000). *Processamento de Imagens Digitais.* São Paulo: Edgard Blücher.
- Goossens, B., Luong, H., Pizurica, A., & Philips, W. (2008). An improved non-local denoising algorithm. *Proc. of LNLA* .
- Hauck, S. (2008). *Reconfigurable computing the theory and practice of fpga based computation.* Elsevier.
- Karnati, V., Uliyar, M., & Dey, S. (2009). Fast non-local algorithm for image denoising. *Proc. of IEEE ICIP* .
- Lindenbaum, M., Fischer, M., & and Bruckstein, A. (1994). On gabor contribution to image enhancemen. *Pattern Recognition, vol. 27* , pp. 1–8.
- Mahmoudi, M., & Sapiro, G. (2005). Fast image and video denoising via nonlocal means of similar neighborhoods. *Signal Processing Letters* , pp. 839-842
- Marques, D. S., (2012). Sistema Misto Reconfigurável Aplicado à Interface PCI para Otimização do Algoritmo Non-Local Means, Tese de mestrado em Informática, UFPB.
- NVIDIA**,** CUDA C Programming Guide 4.2, 2012. Disponível em <http://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf> . Último acesso em: 10 jul. 2013
- Panato, A., Silva, S., Wagner, F., Johann, M., Reis, R., & Bampi, S. (2004). Design of very deep pipelined multipliers for FPGAs. *Design, Automation and Test in Europe Conference and Exhibition* .
- Pavlidis, T. (1978). A review of algorithms for shape analysis. *Comput.Graphics Image Processing* , pp. 243-258.
- Pavlidis, T. (1980). Algorithms for shape analysis of contours and waveforms. *IEEE Trans. Pattern Anal. Mach. Intell.* , pp. 301-302.
- Pottathuparambil, R., & Sass, R. (2008). Implementation of a CORDIC-based Double Precision Exponential Core on an FPGA. *Proceedings of the Fourth Annual Reconfigurable Systems Summer Institute* , 7-9.

- Ramer, U. (1972). An iterative procedure for the polygonal approximation of plane curves. *J. Comput. Graphwcs and Image Processing , 1*, 244-256.
- Rossi, D. L. (2012). Projeto de um controlador PID para controle de ganho de uma câmera com sensor CMOS utilizando computação reconfigurável. *Tese de Doutorado* . Brasil.
- Rudin, L., Osher, S., & Fatemi, E. (1992). Nonlinear total variation based noise removal algorithm. *Physica D, vol.60* , pp. 259–268.
- Rudin, L.; Osher, S. (Nov. 1994). Total variation based image restoration with free local constraints, in roc. Of IEEE International Conference on Image Processing (ICIP), vol. 1, pp. 31–35
- Santos, Laerte dos (2006). Termografia Infravermelha em Subestações de Alta Tensão Desabrigadas, Tese de Mestrado, Pós-Graduação em Energia, Universidade Federal de Itajubá.
- Shaham, N. (2007). Métodos para aceleração do "non--local means" algoritmo de redução de ruído. *Dissertação (Mestrado em Informática)* . Rio de Janeiro, Rio de Janeiro, Brasil: Pontifícia Universidade Católica do Rio de Janeiro.
- Sommerville, I., & Kotonya, G. (1998). *Requirements Engineering.* Wiley.
- SYSTEMVERILOG. SystemVerilog - Overview, 2012. Disponivel em: <http://www.systemverilog.org/overview/overview.html>. Último acesso em: 10 jul. 2013.
- Takagi, N., Kadowaki, S., & Takagi, K. (2005). A hardware algorithm for integer division. *17th IEEE Symposium on Computer Arithmetic* , 140,146,27-29.
- Tanenbaun, A. S. (2007). *Organização Estruturada de Computadores.* São Paulo: Pearson.
- Tomasi, C., & Manduchi, R. (1998). Bilateral filtering for gray and color image. *Proc. IEEE ICCV* , pp. 839–846.

- Vignesh, R., Oh, B. T., & Kuo, C.-C. J. (2010). “Fast non-local means (NLM) computation with probabilistic early termination. *IEEE Signal Process.Lett.* , 277-280.
- Villasenor, J. &. (1997). Configurable Computing. *Scientific American* .
- Zhou, M. (20 de Janeiro de 2006). *Estimators, Mean Square Error, and Consistency*. Acesso em 22 de Janeiro de 2012, disponível em http://www.ms.uky.edu/~mai/sta321/mse.pdf