UNIVERSIDADE FEDERAL DA PARAÍBA
CENTRO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM INFORMÁTICA

PROPOSTA DE IMPLEMENTAÇÃO EM HARDWARE PARA O ALGORITMO NON-LOCAL MEANS

Lucas Lucena Gambarra

JOÃO PESSOA-PB
Julho-2012

UNIVERSIDADE FEDERAL DA PARAÍBA
CENTRO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM INFORMÁTICA

PROPOSTA DE IMPLEMENTAÇÃO EM HARDWARE PARA O ALGORITMO NON-LOCAL MEANS

Lucas Lucena Gambarra

JOÃO PESSOA-PB
Julho-2012

Lucas Lucena Gambarra

PROPOSTA DE IMPLEMENTAÇÃO EM HARDWARE PARA O ALGORITMO NON-LOCAL MEANS

DISSERTAÇÃO APRESENTADA AO CENTRO DE INFORMÁTICA DA UNIVERSIDADE FEDERAL DA PARAÍBA, COMO REQUISITO PARCIAL PARA OBTENÇÃO DO TÍTULO DE MESTRE EM INFORMÁTICA (SISTEMAS DE COMPUTAÇÃO).

Orientador: Prof. Dr. José Antônio Gomes de Lima

JOÃO PESSOA-PB
Julho-2012

G188p Gambarra, Lucas Lucena.
Proposta de implementação em hardware para o algoritmo non-local means / Lucas Lucena Gambarra.- João Pessoa, 2012.
76f. : il.
Orientador: José Antônio Gomes de Lima
Dissertação (Mestrado) – UFPB/CCEN
1. Informática. 2. Processamento Digital de Imagens. 3. Redução de ruídos. 4. Non-local means. 5. Desempenho. 6. Hardware.

UFPB/BC CDU: 004(043)

Ata da Sessão Pública de Defesa de Dissertação de Mestrado de **Lucas Lucena Gambarra**, candidato ao Título de Mestre em Informática na Área de Sistemas de Computação, realizada em 13 de agosto de 2012.

Ao décimo terceiro dia do mês de agosto do ano dois mil e doze, às dez horas, na sala do REUNI - da Universidade Federal da Paraíba, reuniram-se os membros da Banca Examinadora constituída para examinar o candidato ao grau de Mestre em Informática, na área de "*Sistemas de Computação*", na linha de pesquisa "*Sinais, Sistemas Digitais e Gráficos*", o Sr. **LUCAS LUCENA GAMBARRA**. A comissão examinadora foi composta pelos professores doutores: JOSÉ ANTÔNIO GOMES DE LIMA (PPGI-UFPB), Orientador e Presidente da Banca, LEONARDO VIDAL BATISTA (PPGI-UFPB), examinador interno e ELMAR UWE KURT MELCHER (UFCG) como examinador externo. Dando início aos trabalhos, o professor JOSÉ ANTÔNIO GOMES DE LIMA, cumprimentou os presentes, comunicou aos mesmos a finalidade da reunião e passou a palavra ao candidato para que o mesmo fizesse, oralmente, a exposição do trabalho de dissertação intitulado "*Proposta de Implementação em Hardware para o Algoritmo Non-Local Means*". Concluída a exposição, o candidato foi arguido pela Banca Examinadora que emitiu o seguinte parecer: "aprovado". Assim sendo, deve a Universidade Federal da Paraíba expedir o respectivo diploma de Mestre em Informática na forma da lei e, para constar, eu, professor Alisson Vasconcelos de Brito, coordenador deste programa, servindo de secretário, lavrei a presente ata que vai assinada por mim mesmo e pelos membros da Banca Examinadora. João Pessoa, 13 de agosto de 2012.

Alisson Vasconcelos de Brito

Prof. Dr. José Antônio Gomes de Lima
Orientador (PPGI-UFPB)

Prof. Dr. Leonardo Vidal Batista
Examinador Interno (PPGI-UFPB)

Prof. Dr. Elmar Uwe Kurt Melcher
Examinador Externo (UFCG)

Dedico este trabalho

Aos meus pais, Lúcia e Pedro Gambarra

O sonho do careta é a realidade do maluco.

Raul Seixas

Agradecimentos

Aos companheiros de laboratório do LASID que souberam transformar um ambiente de estudo e trabalho num ambiente agradável, coeso e animado.

Aos colegas Bruno Maia e Ruan Delgado que me forneceram a ideia inicial para o trabalho.

Aos Professores Hamilton Soares e Leonardo Vidal, pelo apoio essencial na conclusão deste trabalho.

Ao meu orientador, Professor José Antônio, pela oportunidade oferecida e pela confiança depositada em mim.

Resumo

Imagem digital é a representação de uma imagem bidimensional usando números binários codificados de modo a permitir seu armazenamento, transferência, impressão ou reprodução, e seu processamento por meios eletrônicos. É formada por um conjunto de pontos definidos por valores numéricos, escala de cinza, no qual cada ponto representa um pixel.

Em qualquer imagem digital em nível de cinza, a medição do valor em cada pixel é sujeita a algumas perturbações. Essas perturbações são devidas à natureza aleatória do processo de contagem de fótons nos sensores usados para captura da imagem. O ruído pode ser amplificado por correções digitais ou por qualquer software de processamento de imagem como, por exemplo, as ferramentas que aumentam o contraste.

A remoção de ruídos cujo objetivo é recuperar, ou estimar a imagem original, é ainda um dos mais fundamentais e amplamente estudados problemas do processamento de imagem. Em diversas áreas, a remoção de ruídos é uma etapa fundamental para melhorar a qualidade dos resultados. Entre as alternativas com essa finalidade, o método proposto por Buades (2005), conhecido como *Non-Local Means* (*NLM)*, representa o estado da arte.

Embora bastante eficaz quanto à remoção de ruídos, o NLM é muito lento para ser realizado de modo prático. Sua complexidade computacional é alta devido à necessidade de cálculo de pesos para todos os pixels da imagem durante o processo de filtragem de cada pixel, resultando numa complexidade quadrática no número de pixels da imagem. Os pesos são obtidos por meio do cálculo da diferença entre as vizinhanças de pixels correspondentes.

Muitas aplicações possuem requisitos de tempo para que seus resultados sejam úteis, e nesse contexto, este trabalho propõe uma implementação em FPGA para o algoritmo *Non-local means* com o objetivo de obter um baixo tempo de execução usando, para isto: *pipelines*, paralelismo em hardware e aproximação linear por partes. A implementação proposta é aproximadamente 290 vezes mais rápida que o algoritmo *Non-local means* em software e apresenta, além disso, resultados semelhantes ao algoritmo original quanto ao erro médio quadrático (MSE) e a qualidade perceptiva de imagem.

Palavras chave Processamento Digital de Imagens, Redução de Ruídos, *Non-local means*, Desempenho, Hardware.

Abstract

A digital image is a representation of a two-dimensional image using binary numbers coded to allow its storage, transfer, printing, reproduction and its processing by electronic means. It is formed by a set of points defined by numerical values (grayscale), in which each point represents a pixel.

In any grayscale digital image, the measurement of the gray level observed in each pixel is subject to alterations. These alterations, called noise, are due to the random nature of the photons counting process by sensors used for image capture. The noise may be amplified by virtue of some digital corrections, or by image processing software such as tools to increase contrast.

Image denoising with the goal to recover or estimate the original image is still one of the most fundamental and widely studied problems related to image processing. In many areas, such as aerospace and medical image analysis, noise removal is a key step to improve the quality of results. Among the alternatives for this purpose, the method proposed by Buades (2005), known as Non-Local Means (NLM), represents the state of the art.

Although quite effective for removing noise, the NLM is too slow to be performed in a practical manner. Its high computational complexity is caused by the need of weights calculated for all the image pixels during the filtering of each pixel, resulting in quadratic complexity relative to the number of the image pixels. The weights are obtained by calculating the difference between the neighborhoods corresponding to each pixel.

Many applications have timing requirements so that their results are useful. This work proposes a hardware implementation for the Non-Local Means algorithm for image denoising with a lower computation time using pipelines, hardware parallelism and piecewise linear approximation. It is about 290 times faster than the original non-local means algorithm, yet produces comparable results in terms of mean-squared error (MSE) and perceptual image quality.

Keywords: Digital Image Processing, Image denoising, Non-local Means, Performance, Hardware.

Sumário

Lista de Equações

Lista de Figuras

Lista de Tabelas

Capítulo I: Introdução

Apresentar uma proposta de implementação em hardware para o filtro *Non-local means* demonstrou ser uma tarefa desafiadora, visto que a maioria das implementações atuais apresentam abordagens em software. Neste capítulo, o presente trabalho é introduzido e sua organização é apresentada.

1.1. Motivação

Durante a aquisição e transmissão, as imagens são muitas vezes deterioradas por ruídos que normalmente degradam sua qualidade. Consequentemente, várias aplicações relacionadas com imagens, como por exemplo, na área aeroespacial, análises de imagens médicas e detecção de objetos, geralmente requerem a filtragem de ruídos para produzir resultados confiáveis.

Além disso, a filtragem é muitas vezes necessária como um pré-processamento para outras tarefas, tais como: compressão, segmentação e reconhecimento (COUPÉ, YGER e 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 e várias abordagens têm sido propostas para a redução do ruído.

Os modelos lineares como, por exemplo, o filtro Gaussiano (LINDENBAUM, FISCHER e AND BRUCKSTEIN, 1994) têm sido comumente utilizados para reduzir ruído; esses métodos executam bem sua tarefa em regiões planas (de natureza contínua) das imagens, mas têm como principal desvantagem não serem capazes de preservar as bordas (descontinuidades na imagem) adequadamente.

Os modelos não-lineares, por sua vez, podem assegurar as bordas bem mais adequadamente que os lineares. Um modelo popular para redução de ruídos não-linear é o Filtro de Variação Total, introduzido por (RUDIN, OSHER e FATEMI, 1992), o qual é apropriado para a preservação de bordas, embora tenha uma tendência a produzir o efeito de *máscara* nas regiões da imagem de entrada que variam suavemente. Existe ainda outro modelo de redução de ruído denominado Filtragem por vizinhança (*Neighborhood filtering*), como a filtragem Bilateral (TOMASI e

MANDUCHI, 1998), que restaura um pixel[1] tomando uma média dos valores de seus vizinhos.

Empregando a continuidade das intensidades para os pixels em uma vizinhança local, esses métodos consideram apenas a influência dos pixels na vizinhança centrada ao redor de um determinado pixel para filtrá-lo. Na verdade, existem muitos padrões de repetição em imagens naturais, e não se deve utilizar apenas uma região local pequena para calcular o valor de cada pixel, mas sim a imagem inteira ou uma parte grande o suficiente.

Foi pensando dessa maneira que (BUADES, COLL e 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 e MOREL, 2005), pode ser observada uma comparação relativa à qualidade de vários outros métodos com o algoritmo *Non-local means*, a qual fica mais evidente através da Figura 1 e da Figura 2.

A Figura 2 apresenta: (A) imagem com um ruído (desvio padrão de 20); (B) filtragem Gaussiana; (C) filtragem anisotrópica, (D) filtragem por variação total, (E) filtragem Bilateral; (F) filtragem pelo algoritmo NLM.

Como pode ser observado na Figura 2, o NLM apresenta o melhor resultado quanto à remoção de ruídos. O grande problema é que seu tempo de processamento é muito alto. Para uma imagem com N^2 pixels e utilizando vizinhanças de comparação de tamanho M^2, a complexidade computacional do algoritmo original é $O(M^2 x N^4)$.

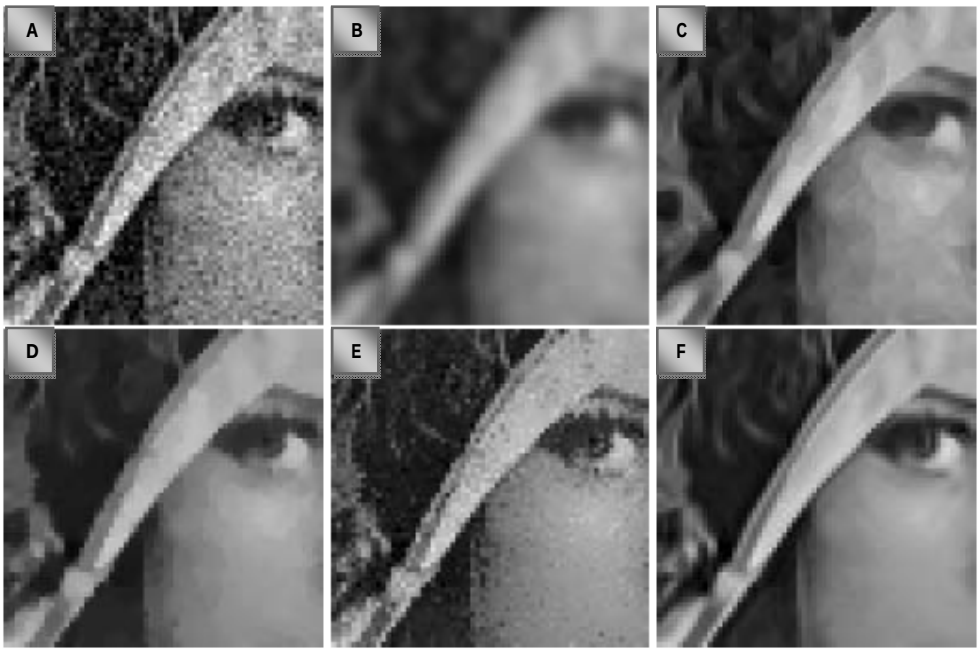
Essa alta complexidade computacional o torna inviável para enfrentar questões práticas, uma vez que muitas aplicações possuem requisitos não-funcionais de desempenho, que são aqueles que se referem à velocidade de operação do sistema.

[1]Pixel (aglutinação de *Picture e Element*, ou seja, elemento de imagem, sendo *Pix* a abreviatura em inglês para Picture). Corresponde ao menor elemento num dispositivo de exibição (como, por exemplo, um monitor), ao qual é possível atribuir-se uma cor. De uma forma mais simples, um pixel é o menor ponto que forma uma imagem digital, sendo que o conjunto de milhares de pixel forma a imagem inteira.

Figura 1- Imagem original.

Fonte: (BUADES, COLL e MOREL, 2005)**.**

Figura 2 - Comparação entre filtragens: (A) imagem com um ruído (desvio padrão de 20); (B) filtragem Gaussiana; (C) filtragem anisotrópica, (D) filtragem por variação total, (E) filtragem Bilateral; (F) filtragem pelo algoritmo NLM.



Fonte: (BUADES, COLL e MOREL, 2005).

De acordo com (SOMMERVILLE e KOTONYA, 1998) diferentes tipos de requisitos podem ser especificados, entre os quais:

- 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 pré-definido de tempo;

Os próprios idealizadores do NLM (BUADES, COLL e 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 e 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.

1.2. Objetivos

Observando o comentário em (GONZALES e WOODS, 2000) sobre a necessidade de hardware especializado para incrementar a velocidade de aplicações em processamento digital de imagens, o objetivo geral do trabalho consiste em construir uma solução para melhorar o tempo de execução do NLM, usando para isso um desenvolvimento em hardware específico. A meta é obter um tempo de execução inferior em relação ao tempo da solução em software.

Para tanto, é possível subdividir o trabalho nos seguintes objetivos específicos:

- Adequar o algoritmo a uma implementação em hardware aumentando seu desempenho, mas mantendo qualidade no resultado;
- Desenvolver um módulo em hardware reconfigurável para calcular o valor final de um pixel, dada como entrada uma janela de pesquisa que contenha esse pixel;
- Desenvolver uma memória dedicada capaz de múltiplas leituras simultâneas para dar suporte ao *hardware* paralelo;
- Efetuar simulações e comparar resultados para validar o trabalho.

1.3. Resultados obtidos

Os resultados obtidos na simulação mostram que o algoritmo é executado em média 290 vezes mais rápido que na execução em software, e os resultados da redução de ruídos são semelhantes tanto em MSE quanto em percepção visual. Empregando-se mais núcleos de filtragem, esses resultados ainda podem melhorados, 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.

1.4. Organização

Este trabalho está organizado da seguinte forma. O Capítulo II apresenta a fundamentação teórica dos temas predominantes deste projeto. Entre outros tópicos, este capítulo descreve o algoritmo NLM; apresenta a tecnologia de hardware reconfigurável; relembra conceitos de paralelismo, e apresenta a abordagem para validar o trabalho proposto. O Capítulo III traz o enfoque usado para melhorar o desempenho do NLM baseado em técnicas de desenvolvimento em hardware. Nele pode ser vista a arquitetura lógica do sistema proposto e são apresentadas informações de simulação e implementação para um FPGA específico. O Capítulo IV apresenta os resultados obtidos tanto em termos de desempenho quanto de qualidade. O Capítulo V contém a conclusão de tudo o que foi apresentado.

Capítulo II: Fundamentação Teórica

Este capítulo apresenta a fundamentação teórica dos temas predominantes desta proposta, definindo os conceitos básicos para elucidação e entendimento do trabalho. Entre outros tópicos, descreve o algoritmo NLM e analisa sua complexidade; apresenta a tecnologia de hardware reconfigurável, com ênfase em dispositivos FPGA (*Field Programmable Gate Array*); relembra conceitos de paralelismo, que são aqui usados para aperfeiçoar o desempenho do NLM; também apresenta as várias abordagens adotadas para verificar a qualidade e validar a funcionalidade da implementação proposta.

2.1. Processamento Digital de Imagens

A utilização dos sistemas digitais vem aumentando em diversas áreas ao longo dos anos devido a sua crescente evolução. Entre estas áreas pode ser citado o processamento digital de imagens que anteriormente tinha todo seu processamento realizado por meios analógicos. Embora as áreas de aplicação sejam diversas, os problemas comumente convergem para a necessidade de métodos capazes de melhorar a informação visual para a análise e interpretação humanas (GONZALES e WOODS, 2000).

Tratando-se de sinais pictóricos, imagens digitais podem ser vistas como um sinal representado por uma função bidimensional e consequentemente, capaz de ser processada por um computador. Dada uma imagem como entrada, ela será processada por procedimentos expressos de forma algorítmica, resultando em 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 e a única razão para utilizar hardware especializado para processamento de imagens é a necessidade de velocidade em algumas aplicações ou para vencer algumas limitações fundamentais da computação.

2.2. Ruídos em imagens digitais

O ruído ocorre em imagens por várias razões, e de um modo um tanto impreciso, define-se ruído como um componente indesejado na imagem. 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.

Para definir ruído de um modo mais formal, 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:

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

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

$$v(i) = u(i)\, x\, m(i) \qquad \textbf{Equação 2}$$

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).

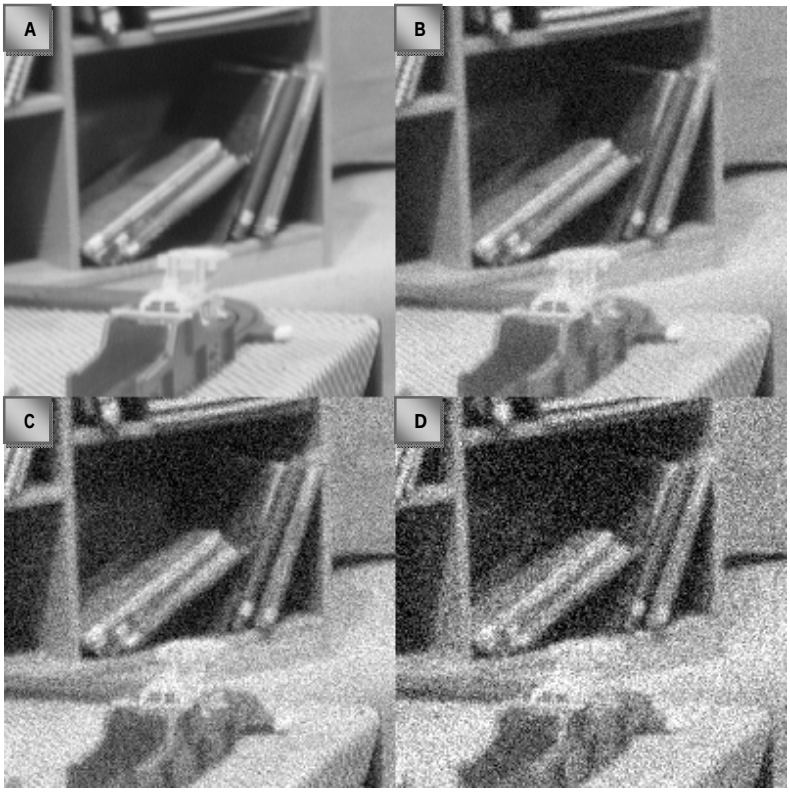
A Figura 3 apresenta uma imagem acrescida de diferentes níveis de ruído: a Figura 3.A mostra a imagem original, supostamente sem ruídos. As Figuras 3.B, 3.C e 3.D apresentam essa mesma imagem acrescida de ruído branco com desvios-padrão de 10, de 20 e de 30, respectivamente.

Reduzir o componente de ruído melhora a qualidade da imagem. A maioria dos algoritmos para eliminação de ruído em imagens depende de um parâmetro de filtragem (h) que é usualmente uma função do desvio padrão do ruído.

2.3. Sobre a apresentação das imagens

O leitor deve ficar ciente que as imagens apresentadas neste documento são mais adequadamente visualizadas na tela do computador do que em papel impresso. A perda de resolução e a adição de meio-tom, *dot-gain* e outros artifícios podem provocar a perda de detalhes sutis das imagens que são demonstrados em vários exemplos. Isso ocorre uma vez que vários destes artifícios mascaram o ruído que foi adicionado à grande parte das imagens. O uso da opção de *zoom in* do software de visualização é sugerido para observar os detalhes finos das imagens que escapam quando visualizados na resolução normal.

Figura 3 - Exemplo de ruído: as imagens B, C e D correspondem à imagem A acrescida de ruído com desvio-padrão de 10, de 20 e de 30, respectivamente.



2.4. O algoritmo *Non-Local Means*

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 e 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. A questão de semelhança é mais bem explicada na Figura 4.

Figura 4 - Vizinhanças: *P1, P2, P3 e P4*.

Na Figura 4 podem ser visualizadas 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, no quesito 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 oportunidade de se encontrar pixels relevantes localizados a uma maior distância.

Feitas essas considerações sobre a ideia de funcionamento do *NLM,* seu algoritmo pode ser formalmente definido. 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,

$$NL[v][i] = \sum_{j \in I} w(i,j)\, v(j), \qquad \textbf{Equação 3}$$

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 $S(i,j)$.

$$S(i,j) = ||v(V_i) - v(V_j)||^2$$ **Equação 4**

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

$$w(i,j) = \frac{1}{Z(i)} e^{-\frac{S(i,j)}{h^2}}$$ **Equação 5**

onde $Z(i)$ é a fator de normalização

$$Z(i) = \sum_j e^{-\frac{S(i,j)}{h^2}}$$ **Equação 6**

e o parâmetro h atua como um fator de filtragem, controlando o decaimento da função exponencial. A Figura 5 demonstra o efeito do fator de decaimento na imagem filtrada. A Figura 5.A contém uma imagem com ruído para σ = 10; as três outras imagens são resultados da aplicação do filtro *NLM* com fatores de decaimento diferentes. A Figura 5.B possui h^2 = 4; a Figura 5.C, h^2 = 2.8; e a Figura 5.D, h^2 = 8. A imagem na Figura 5.C está filtrada brandamente e apresenta um considerável nível de ruído. A imagem na Figura 5.D está intensamente filtrada e vários detalhes finos desapareceram junto com o ruído. A imagem na Figura 5.B, filtrada de acordo com a seguinte recomendação, h^2 = 0.40σ , encontrada em (BUADES, COLL e MOREL, 2011), apresenta um bom equilíbrio entre perda de detalhes e ruído residual.

É importante perceber 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 eficaz que outros filtros de ruído.

2.4.1. Complexidade do NLM

Embora a qualidade dos resultados do NLM represente o estado da arte em redução de ruídos em imagens, o método é muito lento para ser realizável de modo prático. 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 (MAHMOUDI e SAPIRO, 2005).

Admitindo N^2 como o número de pixels da imagem, e utilizando vizinhanças de dimensão quadrática (M^2), a complexidade do algoritmo é $M^2 x\, N^4$. De uma forma

mais simples, essa complexidade se deve ao fato de que para filtrar cada pixel da imagem (N^2) é necessário varrer todos os outros pixels (N^2) e calcular a distância euclidiana quadrática (M^2).

Figura 5 - Efeito do fator de decaimento (h) na imagem filtrada: imagem com ruído (A), imagens filtradas para $h^2 = 4$ (B), $h^2 = 2.8$ (C) e $h^2 = 8$(D), respectivamente.

2.5. Avaliação de desempenho

Para validar a solução proposta é necessário mensurar, avaliar e comparar os resultados alcançados. Para o trabalho proposto, o critério de sucesso consiste em alcançar uma melhor razão de custo/desempenho para execução NLM. Em outras palavras, deseja-se atingir um menor tempo de execução para o NLM, sem, no entanto, perda notável de qualidade.

A aceleração pode ser mensurada de forma simples comparando o tempo de execução em hardware ao tempo de execução do algoritmo em software.

A percepção de perda de qualidade de imagem é um parâmetro mais problemático de medir. Como foi dito, a captura e transmissão podem introduzir alguma distorção na imagem de modo que a avaliação da qualidade é um problema importante. Para mensurar este parâmetro são utilizados:

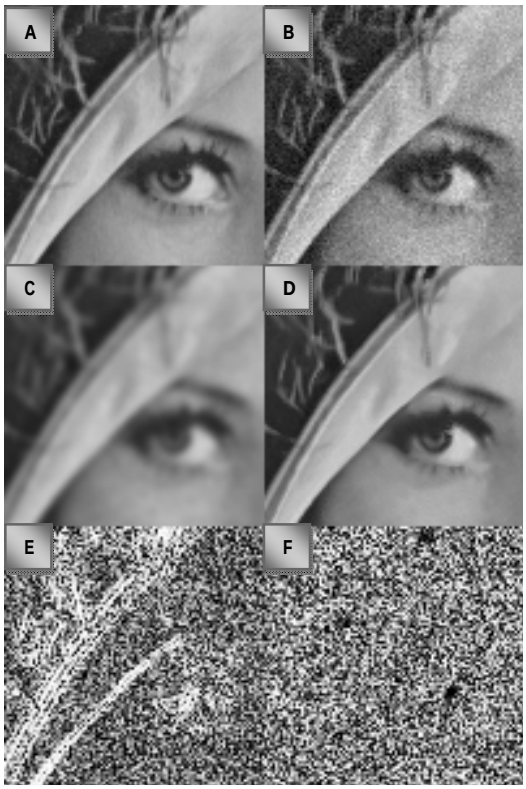
1. Erro médio quadrático (*mean square error - MSE*). Abordagem estatística na qual é estimado o erro médio quadrático entre a imagem filtrada e a imagem

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 de método* introduzido pelos autores do NLM em (BUADES, COLL e MOREL, 2005). A diferença absoluta entre a imagem com ruído e a imagem filtrada é a imagem obtida pelo ruído de método, que mostra o ruído removido pelo método.

A medida baseada no ruído de método pode ser entendida através do exemplo mostrado no exemplo da Figura 6: À imagem original (A), é adicionado ruído branco com σ = 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 6 - Exemplo de ruído do método.



2.6. Comparando o NLM com outros métodos de filtragem

Em (BUADES, COLL e 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 são:

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

São apresentados os resultados quanto ao MSE na Tabela 1, assim como os relativos ao ruído do método, na Figura 7, para as seguintes técnicas de filtragem: Gaussiana (Figura 7.A), Deslocamento de Curvatura Média (Figura 7.B), Variação Total (Figura 7.C), Variação Total Iterada (Figura 7.D), de Vizinhança (Figura 7.E), Hard TIWT (Figura 7.F), Soft TIWT (Figura 7.G), DCT empirical Wiener filter (Figura 7.H) e NLM (Figura 7.I).

A filtragem Gaussiana de ruídos destaca todas as características importantes da imagem, como textura, contornos e detalhes. O método de redução de ruídos da Filtragem Anisotrópica destaca os cantos e as características de alta frequência, em detrimento das bordas retas que têm uma curvatura reduzida. O método da Variação Total modifica a maioria das estruturas e características da imagem, e mesmo as bordas retas não são bem preservadas. Porém, através do refinamento dado pelo método da Variação Total Iterativo, a qualidade da filtragem é melhorada de modo que as bordas retas e os detalhes são mais bem preservados. A filtragem por Vizinhança preserva objetos planos e bordas com maior contraste, enquanto as bordas com um contraste baixo não são mantidas, em geral, o contorno, textura e detalhes parecem ser bem preservados.

O método *TIWT (Translation invariant wavelet thresholding*) usa um limiar para filtragem e concentra-se nas margens e nas características de alta frequência. No Hard TWIT, é utilizado um limiar rígido de modo que as estruturas que conduzem a coeficientes de valor suficientemente grande, porém menor que o limiar, são removidas pelo algoritmo. Já o método *Soft TIWT* (limiar flexível) mostra-se bem mais estruturado que o *Hard TIWT.* Com efeito, a filtragem de ruídos não se baseia apenas

nos coeficientes pequenos, mas também sobre uma atenuação dos coeficientes grandes, conduzindo a uma alteração maior da imagem original.

Para o método *DCT empirical Wiener filter* é difícil encontrar uma estrutura notável, apenas alguns contornos são perceptíveis. Em geral, este filtro parece funcionar muito melhor do que todos os filtros de suavização locais e filtros de outros domínios de frequência.

Tabela 1 - Erros Médios Quadráticos produzidos por cada método de filtragem.

Imagem	Σ	GF	AF	TV	TV Iter	YNF	EWF	TIHWT	*NLM*
Boat	8	53	38	39	29	39	33	28	23
Lena	20	120	114	110	82	129	105	81	68
Barbara	25	220	216	186	189	176	111	135	72
Baboon	35	507	418	385	364	381	396	365	292
Wall	35	580	660	721	715	598	325	712	59

2.7. Trabalhos relacionados à otimização em tempo do *NLM* em software

Diversos métodos têm sido propostos para acelerar o *Non-local means*, tais como: pré-seleção das vizinhanças que serão usadas para filtrar cada pixel com base no valor médio e gradiente (MAHMOUDI e SAPIRO, 2005); através do uso de média e variância (COUPÉ, PRIMA, *et al.*, 2008); métodos usando gradientes e quantização vetorial (SHAHAM, 2007); por meios de estatísticas de ordem superior (*higher-order statistical moments*) (DAUWE, GOOSSENS, *et al.*, 2008); empregando arranjo de clusters em árvore (*cluster tree arrangement*) (BROX, KLEINSCHMIDT e CREMERS, 2008) e (DINESH, GOVINDAN e MATHEW, 2009); aceleração usando valores médios em diferentes resoluções (KARNATI, ULIYAR e DEY, 2009) ou finalização antecipada probabilística (*probabilistic early termination*) (VIGNESH, OH e KUO, 2010). Além desses métodos, o cálculo da distância entre diferentes vizinhanças pode ser otimizado usando o filtro de média móvel (GOOSSENS, LUONG, *et al.*, 2008). Adams *et al*. propôs o uso de estruturas de dados inteligentes para executar a suavização multidimensional, (CONDAT, 2010) incluindo a filtragem do NLM (ADAMS, GELFAND, *et al.*, 2009), (ADAMS, BAEK e DAVIS, 2010). Em (CONDAT, 2010) é apresentada uma versão do NLM usando rotinas de convolução que o aceleram significativamente em comparação com sua implementação clássica.

Todas as fontes relacionadas acima apresentam uma abordagem de melhoria em software através de modificações no algoritmo original. 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 que acelera o método significativamente em comparação com a implementação clássica. Apesar da indicação em (GONZALES e WOODS, 2000) para utilização de implementação em hardware para aceleração de processamento de imagens, esta observação não foi feita até agora na literatura para o *NLM*, ao menos ao conhecimento do autor.

2.8. Computação Reconfigurável

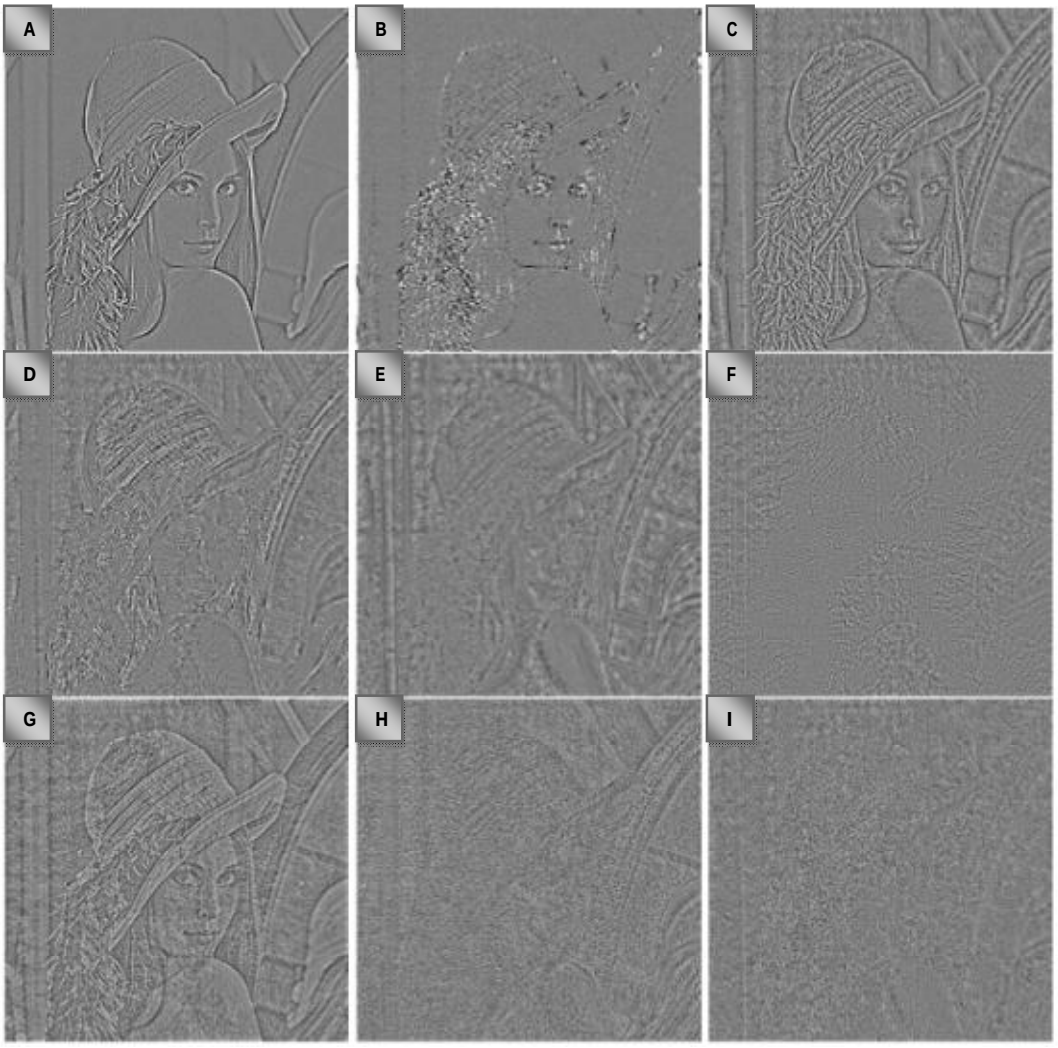
A microeletrônica evoluiu muito nas últimas décadas, possibilitando a construção de uma diversidade de dispositivos eletrônicos mais velozes, com maior capacidade para armazenamento de dados, menor consumo de energia e a custos decrescentes.

É possível projetar dispositivos de uso geral para uma grande diversidade de funções e reprogramá-los de forma bastante flexível através de software. Entretanto, a velocidade de processamento desses dispositivos pode ser lenta quando comparada aos dispositivos projetados para uma finalidade específica, e que executam apenas conjuntos limitados de instruções (HAUCK, 2008). Nessa segunda categoria, destacam-se os circuitos com hardware personalizados, conhecidos como *Applications-Specific Integrated Circuits* (ASIC). Os ASICs, apesar dos benefícios de desempenho, não apresentam a flexibilidade da programação por software.

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 o *hardware* dos ASICs, os FPGA*s* implementam computações espacialmente, 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, esses cálculos são programados (ou reprogramados) no *chip*, não tendo sua funcionalidade estabelecida definitivamente pelo processo de fabricação. Isso significa que um sistema baseado em FPGA pode ser reprogramado muitas vezes (HAUCK, 2008)**.**

FPGAs fornecem quase todos os benefícios da flexibilidade e dos modelos de desenvolvimento de software, e quase todos os benefícios da eficiência de hardware, mas não completamente.

Figura 7 - Ruídos de Método para as filtragens: Gaussiana (A), Deslocamento de Curvatura Média (B), Variação Total (C), Variação Total Iterada (D), de Vizinhança (E), *Hard TIWT* (F), *Soft TIWT* (G), DCT empirical Wiener filter (H) e *NLM* (I).



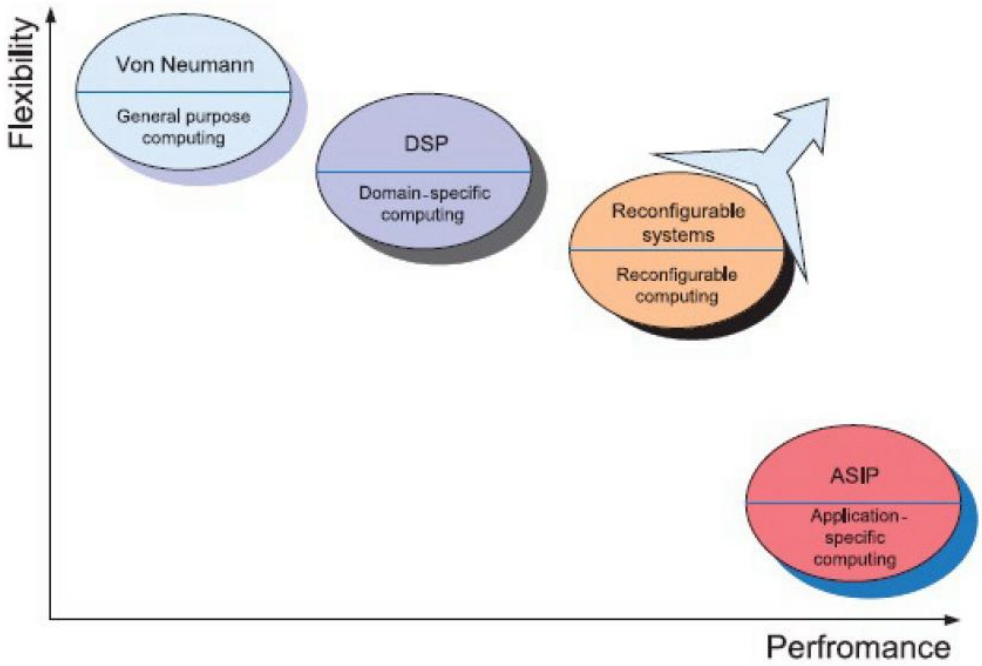
Fonte: (BUADES, COLL e MOREL, 2004).

Comparados a um microprocessador de uso geral, esses dispositivos apresentam um desempenho significantemente melhor, 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, podem ter um desempenho de 5 a 25 vezes inferior (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**.**

As arquiteturas atuais podem ser categorizadas em três grupos dominantes (ROSSI, 2012) de acordo com o seu grau de flexibilidade: grupo de computação 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, dois pontos principais podem ser identificados 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*, pois não podem executar tarefas em paralelo. 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. Considerando 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 8.

Figura 8 - Flexibilidade versus *Performance*.



Fonte: (BOBDA, 2007).

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. 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.

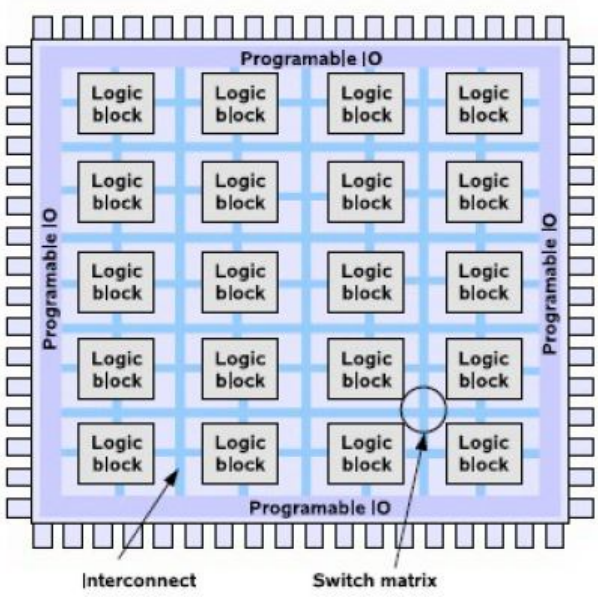
O progresso da computação reconfigurável tem sido extraordinário nas duas últimas décadas (ROSSI, 2012). Isto se deve principalmente pela larga aceitação dos Dispositivos Lógicos Programáveis (FPGAs) que estão agora se estabelecendo como os dispositivos mais usados entre os dispositivos reconfiguráveis.

2.8.1. Mais sobre FPGA (*Field Programmable Gate Array*)

Um FPGA é um circuito integrado que contém um grande número (na ordem de milhares) de unidades lógicas. Um FPGA é um dispositivo que não programa a função lógica no método tradicional, que são as portas dedicadas (ROSSI, 2012). Na verdade, cada elemento lógico é uma pequena matriz de memória que é programada diretamente com a função desejada, a partir da tabela verdade (BERGER, 2012). Essa matriz de memória está localizada nos blocos lógicos contidos no FPGA. A estrutura geral do FPGA pode ser visualizada na Figura 9, que apresenta seus três recursos básicos: os blocos de *I/O* para conectar tanto os pinos de entrada como os de saída do FPGA; os blocos lógicos dispostos num arranjo bidimensional e um conjunto de chaves de interconexões organizadas como canais de roteamento horizontal e vertical entre as linhas e colunas dos blocos lógicos.

Além dos recursos padrões, um FPGA pode disponibilizar no arranjo bidimensional recursos bastante sofisticados, tais como multiplicadores dedicados, MACs programáveis (Multiplicador e Acumulador, também denominados de *slice XtremeDSP*), blocos de memória, DCM (*Digital Clock Manager* ou Gerenciador de Relógio Digital, utilizados para multiplicar ou dividir a frequência de um sinal de *clock*), microcontroladores (IBM PowerPC, Nios, etc.) e trans-receptores multi-giga bit (que servem para implementar interfaces seriais para transferência de bits em altíssima velocidade). A utilização de tais recursos embarcados possibilita otimizar o consumo de área e também desenvolver projetos mais eficientes (ROSSI, 2012).

Figura 9 - Estrutura de um FPGA.



Fonte: (BOBDA, 2007).

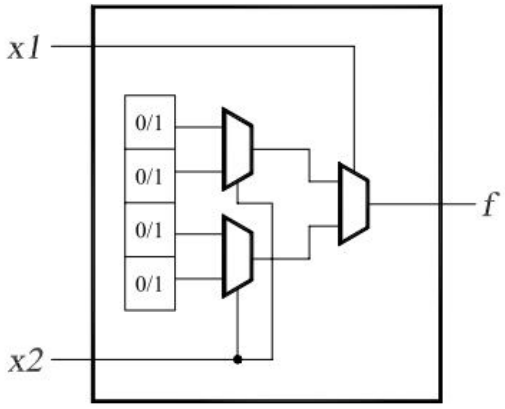
Os canais de conexão entre os blocos lógicos são constituídos por condutores internos e pelas chaves existentes no FPGA, que são configurados conforme a necessidade de comunicação. No FPGA, em geral, cada bloco lógico contém algumas entradas e uma saída, e são baseados em LUTs (*LookUp Table*), as quais contém células de armazenamento que são usadas para implementar uma função lógica simples. Cada célula é capaz de armazenar um valor lógico simples 0 ou 1. O tamanho de uma LUTs é definido pelo número de entradas, de modo que LUTs de vários tamanhos podem ser criadas. A Figura 10 apresenta a estrutura de uma pequena LUT com duas entradas x1 e x2, e uma saída f, que é capaz de programar qualquer função de duas variáveis.

Devido à tabela verdade de duas variáveis possuir quatro combinações possíveis, esta LUT tem quatro células de armazenamento. As variáveis de entrada x1 e x2 são usadas como seletores de entrada dos três multiplexadores, que de acordo com os valores de x1 e x2 selecionam o conteúdo de uma das quatro células como saída da LUT.

Quando um circuito lógico *é* desenvolvido em um FPGA, os blocos lógicos são configurados para realizarem as funções necessárias, e as chaves de interconexão são selecionadas para definir o canal de comunicação entre estes blocos lógicos.

As células de armazenamento nas LUT em um FPGA são voláteis, deste modo, todas as vezes que o dispositivo for energizado deverá ter suas *LUTs* programadas.

Figura 10 - Estrutura de uma LUT de duas entradas.



Fonte: (BROWN e VRANESIC, 2000).

Frequentemente uma pequena memória para armazenar estes dados, chamada de PROM (*Programable Read Only*), é incluída no circuito da placa onde se encontra o FPGA (VANDERLEI, 2004). As células no FPGA são carregadas automaticamente da *PROM* quando o chip é energizado.

2.9. Paralelismo

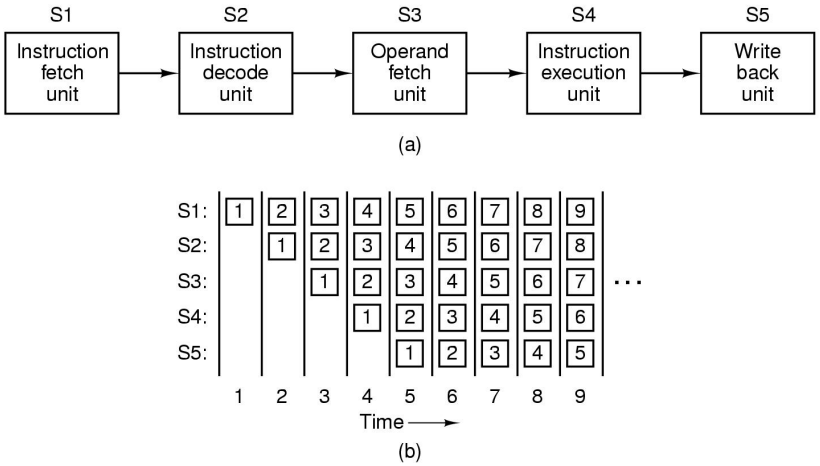
Arquitetos de computadores estão sempre se esforçando para projetar máquinas com melhor desempenho. Fazer com que chips funcionem com uma velocidade de relógio maior é uma possibilidade, mas há sempre um limite para cada época da história (TANENBAUN, 2007). Como alternativa, busca-se o paralelismo como meio para se conseguir um desempenho maior para um dado ciclo de relógio. A arquitetura dos processadores modernos baseia-se fortemente na capacidade em executar várias instruções em cada ciclo de relógio, o que normalmente se designa por paralelismo ao nível da instrução. Esta designação surge porque as instruções são executadas em paralelo. Duas técnicas são frequentemente utilizadas para a execução de instruções em paralelo:

1. Execução encadeada (*pipeline*) de instruções – cada instrução é executada em várias fases, sendo a execução de várias instruções sobreposta, como numa linha de montagem; as várias instruções executam em paralelo, mas em fases diferentes;

2. Execução superescalar de instruções – as instruções são executadas em paralelo, o que envolve a duplicação de unidades funcionais para suportar a combinação de instruções pretendida.

Um pipeline, Figura 11, pode ser definido como a execução de uma instrução sendo dividida em várias partes, cada uma delas manipulada por uma parte dedicada de hardware e todas elas sendo executadas em paralelo.

Figura 11 - Pipelines.



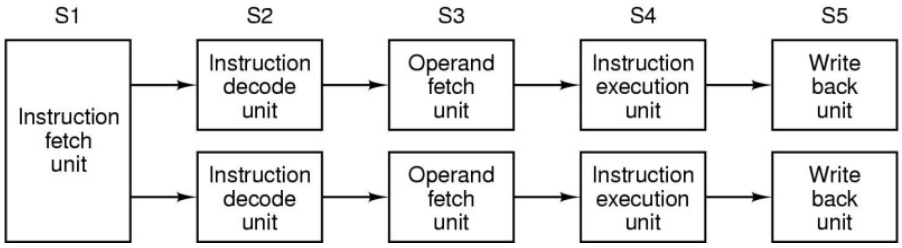
Fonte: (TANENBAUN, 2007).

A Figura 11.a ilustra um pipeline com cinco unidades, também denominados estágios. O estágio 1 (S1) busca a instrução na memória e a coloca em um buffer até que ela seja necessária. O estágio 2 (S2) decodifica a instrução, determina seu tipo e de quais operandos ela necessita. O estágio 3 (S3) localiza e busca os operandos, seja nos registradores, seja na memória. O estágio 4 (S4) é que realiza o trabalho de executar a instrução. Por fim, o estágio 5 (S5) escreve o resultado no registrador adequado.

Na Figura 11.b, é mostrado como o pipeline funciona em função do tempo. Durante o ciclo de relógio 1, o estágio 1 está trabalhando na instrução 1, buscando-a na memória. Durante o ciclo 2, o estágio S2 decodifica a instrução 1, enquanto o estágio S1 busca a instrução 2. Durante o ciclo 3, o estágio S3 busca os operandos da instrução 1, o estágio S2 decodifica a instrução 2, e estágio S1 busca a instrução 3. Durante o ciclo 4, o estágio S4 executa a instrução 1, S3 busca os operandos da instrução 2, S2 decodifica a instrução 3 e S1 busca instrução 4. Por fim, durante o ciclo 5, S5 escreve o resultado da instrução 1 de volta no registrador, enquanto os demais estágios trabalham nas instruções seguintes.

Em arquiteturas superescalares múltiplas instruções são executadas em único ciclo de relógio (TANENBAUN, 2007). Por exemplo, na Figura 12 são mostrados pipelines duplos de cinco estágios com uma unidade de busca em comum. Para poder executar em paralelo, as instruções não devem ter conflito de utilização de recursos (por exemplo, registradores) e nenhuma deve depender do resultado da outra.

Figura 12 - Arquiteturas superescalares.



Fonte: (TANENBAUN, 2007).

Observa-se, portanto, computação paralela envolvendo diferentes unidades funcionais que geram e consomem resultados intermediários. A diferença está na computação totalmente temporal/sequencial das tarefas versos computação totalmente espacial/paralela.

A execução de instruções em *pipeline* permite reduzir a duração do ciclo de relógio do processador, ou seja, aumentar a sua frequência. Por outro lado, a execução superescalar de instruções aumenta o número de instruções realizadas em cada ciclo. Note que qualquer uma destas duas alternativas permite reduzir o tempo de execução da aplicação.

2.10. Aproximação linear por partes

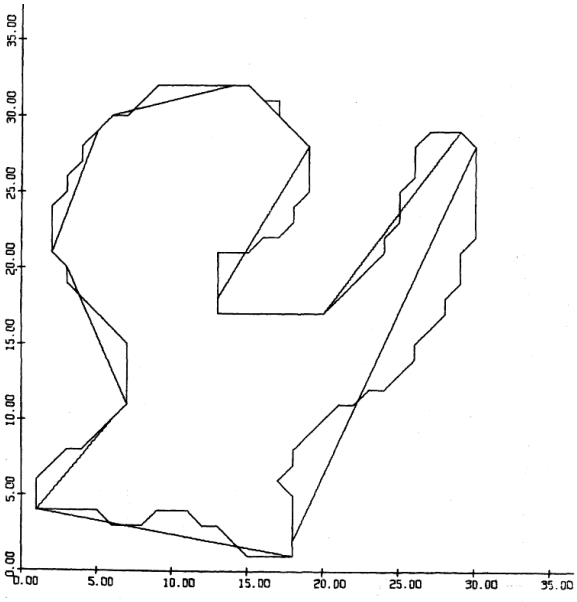
A aproximação linear por partes de uma curva digital bidimensional, Figura 13, é uma representação muitas vezes compacta e eficaz para a análise de forma, padrão e classificação, (PAVLIDIS, 1978) e (PAVLIDIS, 1980). Tais representações facilitam a extração de características numéricas para uso em cálculos diversos ou classificação da curva dada. Nesse trabalho, a aproximação linear por partes é utilizada para extrair características de funções que apresentam um decaimento exponencial.

2.11. Verificação Funcional

O objetivo da verificação é validar a implementação em HDL de acordo com o que se espera ser o seu funcionamento correto. Na verificação funcional, estímulos de

entrada são inseridos em uma implementação de alto nível (modelo de referência) e no módulo de hardware a ser verificado (DUV – *Design Under Verification*). As saídas de cada um deles são confrontadas e certos valores de saída são monitorados para que determinada condição seja atingida e a verificação seja finalizada (cobertura).

Figura 13 - Aproximação linear por partes de uma curva poligonal.



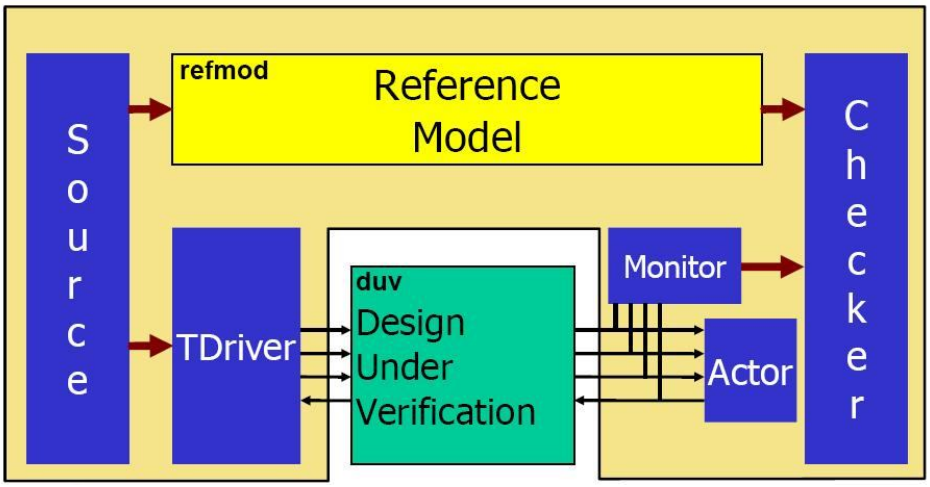
Fonte: (RAMER, 1972).

A verificação funcional aqui apresentada baseia-se na BVM (*Brazil-IP Verification Methodology*) (ARAÚJO, 2010). Nela, os *testbenches*, que nada mais são que os ambientes virtuais onde ocorre todo o processo da verificação, são implementados em nível de transação em contraste com o DUV que é implementado em nível de sinais. Transações são operações finitas formadas por um conjunto de instruções e/ou dados para realizar determinada operação. A visão geral de um *testbench* é mostrada abaixo na Figura 14.

O *Source* é responsável por enviar a mesma transação de entrada para o *TDriver* e para o modelo de referência. O modelo de referência processa a transação de entrada e produz a respectiva transação de saída. Enquanto isso, o *TDriver* converte a transação de entrada em sinais que estimulam o DUV a produzir sinais de saída (através da interação com o *Actor)*, os quais são observados pelo *Monitor* para a produção de transações equivalentes. Tais transações chegam ao *Checker* juntamente com as transações do modelo de referência a fim de serem comparadas.

As transações que fluem do *Source* são geradas aleatoriamente de acordo com uma distribuição de probabilidade direcionada. Tanto as transações aleatórias como as transações resultantes que chegam ao *Checker* são observadas com o intuito de determinar o momento de parar o processo de verificação. A tal condição de parada se dá o nome de cobertura.

Figura 14 - Visão geral de um *testbench*.



Dentre algumas opções de cobertura, cita-se a principal delas como a cobertura funcional, a qual observa sinais e transações e conta o número de ocorrência de determinadas situações especificadas.

2.12. Linguagem de Descrição de Hardware (HDL) SystemVerilog

SystemVerilog IEEE 1800™ é o primeiro padrão unificado da indústria para linguagem de descrição de hardware e verificação. SystemVerilog é uma grande extensão da estabelecida linguagem Verilog IEEE 1364™ (SYSTEMVERILOG, 2012). Foi desenvolvido originalmente pela Accellera e mostrou-se capaz de aumentar a produtividade projetos de circuito integrado. SystemVerilog é destinada principalmente à implementação de chips e ao fluxo de verificação.

A linguagem tem sido adotada pelas principais empresas de projeto de semicondutores ao redor do mundo e suportada por quase todas as ferramentas disponíveis no mercado de software de eletrônica.

O poder de SystemVerilog é visível pelos recursos que ela oferece, pois suporta descrições de módulos em nível de sinais ou RTL (*Register Transfer Level*), em nível de transações ou TL (*Transaction Level)* (com suporte a orientação a

objetos), descrição estrutural e descrição comportamental. Além disso, permite expressar ações concorrentes; expressar tempos (atrasos e *clock*); mesclar diferentes visões de diferentes subsistemas; simulação, síntese e verificação. Por fim, ela é destacada por ser fácil e segura de se usar.

Por oferecer as características apresentadas, SystemVerilog foi a linguagem de descrição de hardware e verificação escolhida para desenvolvimento da versão proposta neste trabalho para o NLM.

2.13. Linguagem Java

Java é uma linguagem de programação orientada a objeto desenvolvida na empresa *Sun Microsystems* e atualmente mantida pela Oracle (ORACLE, 2012). Diferentemente das linguagens convencionais, que são compiladas para código nativo, a linguagem Java é compilada para um *bytecode* que é executado por uma máquina virtual. A linguagem de programação Java é a linguagem convencional da Plataforma Java.

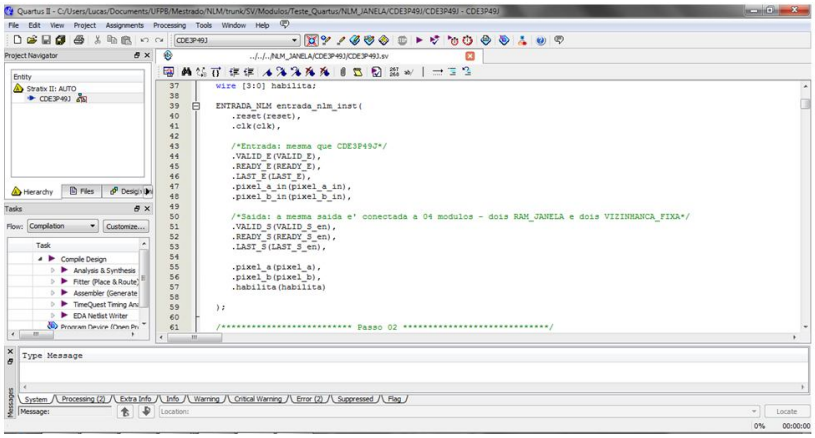
A maioria das imagens neste trabalho é apresentada por meio da linguagem Java, visto que ela oferece facilidades para a apresentação e manipulação de imagens devido à existência de várias APIs (*Application programming interface*), nativas da linguagem, para tratar imagens. Além disto, o NLM (versão para software) também foi programado na linguagem Java para ter seus resultados de desempenho comparados ao do desenvolvimento proposto em hardware para o NLM.

2.14. Quartus® II Web Edition Software

Altera Quartus® II, Figura 15, é software para desenvolvimento de dispositivos de lógica programável da Altera (ALTERA, 2012), que possui as seguintes características:

- Uma implementação de VHDL, Verilog e SystemVerilog para descrição de hardware.
- Edição visual de circuitos lógicos.
- Simulação por forma de onda.

Figura 15 - Quartus II.

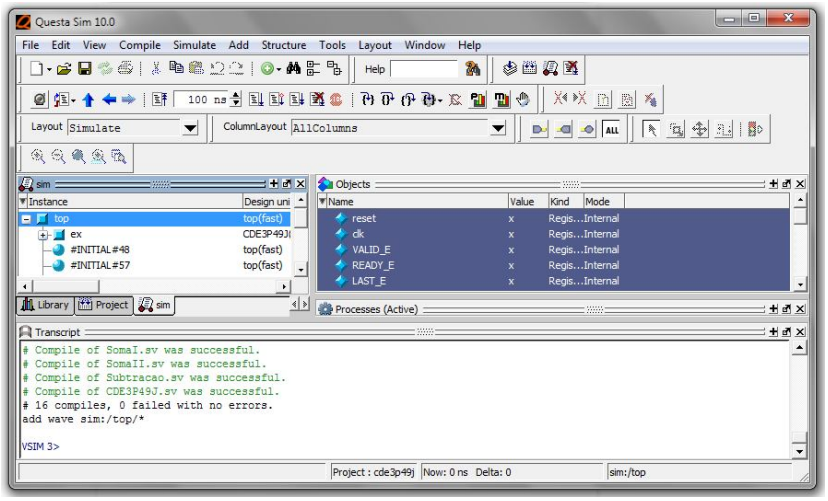


O Quartus II é utilizado para obtenção de resultados de síntese em FPGA como frequência, aréa ocupada e memória utilizada.

2.15. Questa® Advanced Simulator

O Questa® Advanced Simulator, Figura 16, combina um ambiente de simulação com recursos unificados de depuração para o mais completo suporte nativo as linguagens Verilog, SystemVerilog, VHDL, SystemC, entre outras. O Questa® Advanced Simulator é o núcleo de simulação e o motor de depuração da Plataforma Verificação Questa; essa plataforma é capaz de reduzir o risco de validar implementações resultantes de projetos de hardware complexos (GRAPHICS, 2012). Os *testbenches* codificados em SystemVerilog foram executados na ferramenta QuestaSim.

Figura 16 - Questa® Advanced Simulator.



Capítulo III: Proposta de otimização do *NLM*

Este capítulo apresenta enfoque para otimizar o desempenho do NLM baseado em várias técnicas de implementação em FPGA. Primeiro são mostrados os trabalhos relacionados à otimização do NLM. Depois é apresentada a arquitetura lógica do sistema proposto. Ao final são apresentadas informações de simulação e de implementação para um FPGA específico.

3.1. *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 e 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 17, 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).

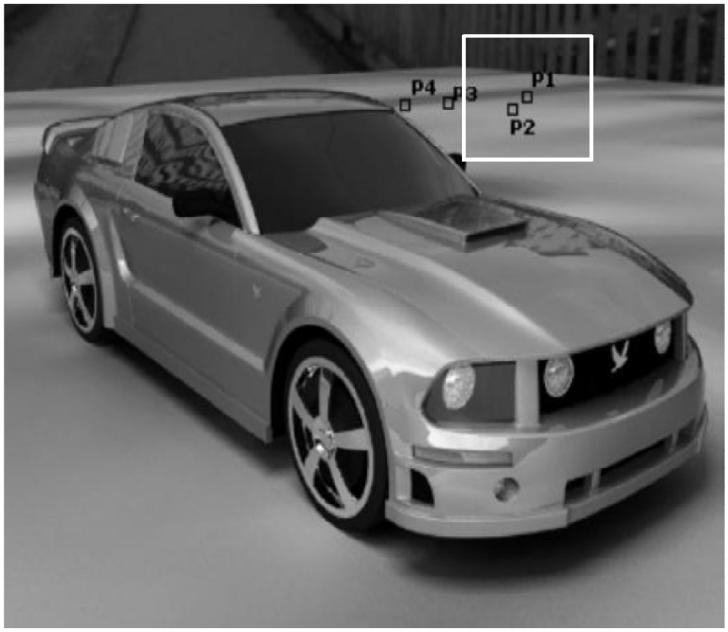
Restringindo a busca por pixels semelhantes a uma janela de W^2 pixels a complexidade final do algoritmo passa a ser $O(M^2\ x\ W^2\ x\ N^2)$ o que é bem menor que a complexidade original uma vez que o valor de *W* é menor que *N*.

3.2. 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 .

O desenvolvimento em hardware utiliza *pipelines*, paralelização de instruções, assim como a construção de memórias especializadas que permitem buscar rapidamente os dados necessários para efetuar os cálculos.

Figura 17 - Janela de semelhança.



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.

3.2.1. Parâmetros

Utilizando valores pequenos para M e para W, conforme sugestão dos criadores do *NLM* (BUADES, COLL e MOREL, 2005), o tempo de execução para o processamento em software não é reduzido a níveis razoáveis. Muitas abordagens que usam desta prática geralmente provocam distorções na imagem filtrada por falta de informação para calcular os pesos (SHAHAM, 2007), uma vez que usam valores arbitrários pequenos para M e W. Entretanto, os autores do *NLM* em (BUADES, COLL e 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. À medida que σ cresce são necessárias janelas de pesquisa W^2 maiores para realizar uma comparação mais exata. Ao mesmo tempo, é preciso estender o tamanho da vizinha M^2 de comparação para ampliar a capacidade de remoção de ruídos do algoritmo através da pesquisa de píxeis 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.

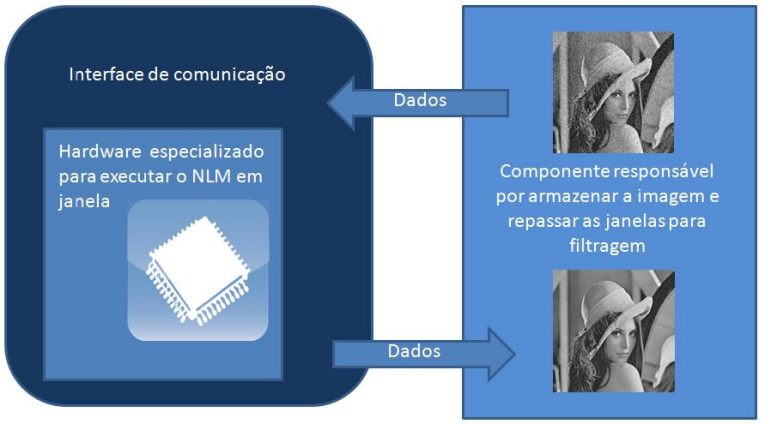
3.2.2. Visão do sistema

O sistema idealizado, Figura 18, é composto por um hardware para a otimização do desempenho de execução do NLM. O mesmo será responsável por efetuar a filtragem de um pixel, dadas como entradas: o pixel e sua respectiva vizinhança, e a janela de pesquisa utilizada para filtrar esse pixel. Note que o hardware proposto não é responsável pelo armazenamento da imagem inteira a ser filtrada, mas somente pela janela de busca sendo utilizada para filtragem do pixel atual. Desta forma, outro componente ficará encarregado do armazenamento da imagem e de enviar os dados necessários para o módulo de filtragem proposto. Esse outro componente, que poderá ser um computador de uso geral ou até mesmo um hardware específico que ficará responsável ainda por armazenar a imagem e se comunicar com o hardware especializado para este continuar com o processamento do filtro. Retirar a carga do módulo desenvolvido de armazenar a imagem completa é uma vantagem que possibilta flexibilidade, pois poderá trabalhar com qualquer tamanho de imagem independente de sua capacidade de armazenamento, que é geralmente pequena.

3.2.3. Arquitetura lógica

A Figura 19 mostra o fluxo da filtragem utilizando o NLM em hardware. Adiante, é detalhada cada fase do fluxo apresentado.

Figura 18 - Visão externa do sistema.



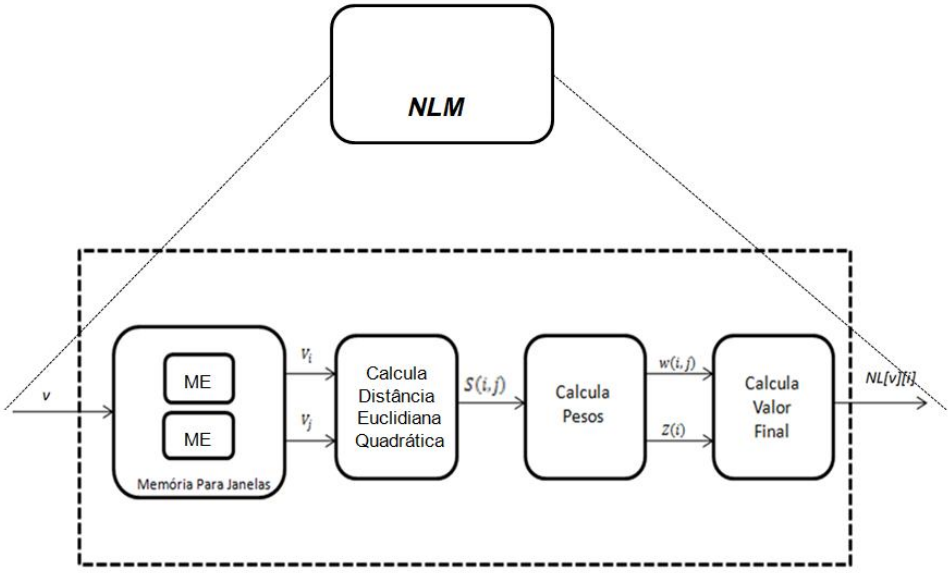
3.2.3.1. Paralelização do cálculo da distância Euclidiana quadrática

Para melhorar o desempenho do NLM, foi elaborada uma forma de acelerar o cálculo da distância Euclidiana quadrática,

$$S(i,j) = ||v(V_i) - v(V_j)||^2 \qquad \textbf{Equação 4}$$

na qual é mensurado 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.

Figura 19 - Fluxo da filtragem de ruídos para o NLM em FPGA.



Analisando a equação, percebe-se que os M^2 cálculos são independentes entre si, possibilitando realizá-los em paralelo. Além disto, cada operação (subtração, potenciação e soma dos resultados gerados) realizada nos cálculos pode ser separada em uma sequência de etapas para uma implementação utilizando *pipeline*.

Isso permite que, após o devido preenchimento do *pipeline*, seja realizado um cálculo de distância Euclidiana quadrática por ciclo de relógio. Por exemplo, se forem usadas vizinhanças com *7x7* pixels (M = 7), em uma estrutura puramente sequencial são necessários *3x7x7*, ou seja, 147 ciclos; enquanto que usando uma abordagem com paralelismo é possível ter a vazão de uma distância Euclidiana quadrática calculada a cada período de relógio.

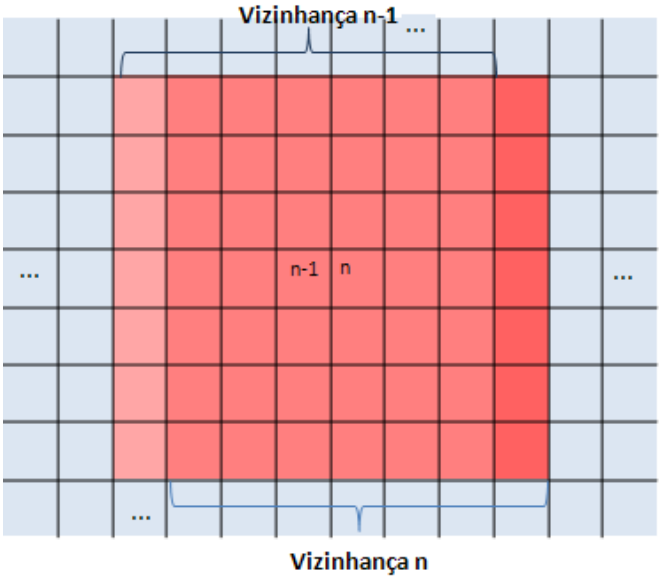
3.2.3.2. Memória especializada (ME) para vizinhanças

Após a construção do componente responsável por efetuar o cálculo da distância Euclidiana quadrática, foi observado que o mesmo não seria efetivamente utilizado caso não houvesse informações sobre os M^2 píxeis a cada ciclo de relógio. Se assim fosse, a execução continuaria sequencial, pois seria preciso aguardar os dados vindos da memória (sequencial) para, só então, operar. Uma memória especializada com suporte para acesso simultâneo a vários endereços mostrou-se indispensável.

Uma alternativa seria replicar a janela em cálculo por $M^2/2$ memórias (supondo-as capazes de permitirem duas leituras por ciclo de relógio). A Figura 20 permite ver a maneira como são acessadas as posições de memória ao passar de uma vizinhança para outra, vizinhança n-1, centrada no pixel n-1; vizinhança n, centrada no pixel n: os pixels em rosa claro e em rosa pertencem à vizinhança n-1; ao passar para a vizinhança de n os pixels em rosa claro são descartados e os pixels em rosa escuro passam a compor a vizinhança n.

Concluí-se então que, na passagem de uma vizinhança para outra, apenas M dos M^2 pixels são realmente novos (os demais precisam apenas de realocação). Essa observação viabilizou a construção de uma solução mais eficiente.

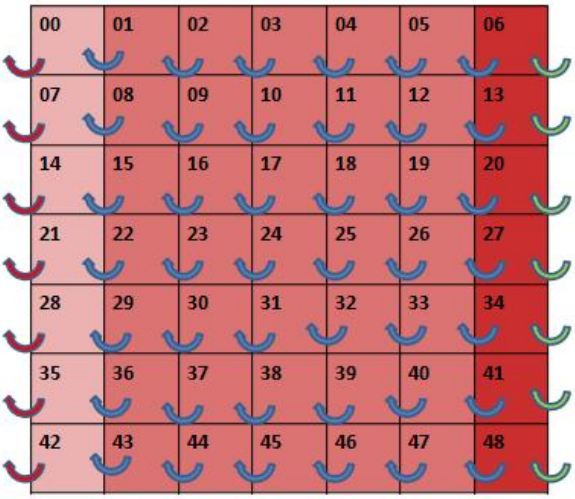
Figura 20 - Duas vizinhanças consecutivas: n-1 e n.



O modo como os pixels são trocados na passagem de uma vizinhança para outra pode ser observado na Figura 21: setas azuis, os pixels que são trocados entre registradores; setas vermelhas, os que são descartados; e, finalmente, setas verdes,

posições que recebem sete novos pixels relativos à nova vizinhança. Fica claro que apenas M buscas simultâneas são realmente necessárias, desde que sejam trocados de forma ágil os demais pixels da vizinhança anterior, garantindo que as informações necessárias para calcular a distância Euclidiana quadrática estarão sempre disponíveis a tempo. Para isto, foi desenvolvida uma memória especializada (ME) para permitir o acesso simultâneo às M posições de memória. É suficiente replicar a janela em cálculo por apenas $M/2$ memórias. Visto que cada uma pode ter dois valores lidos (ou escritos) ao mesmo tempo, é possível ler as M posições simultaneamente.

Figura 21 - Troca de pixels de uma vizinhança para a seguinte.



3.2.3.3. Memória para janelas

Para filtrar um pixel são necessárias uma vizinhança (M^2 pixels) e uma janela estendida, ambas centralizadas nesse pixel. Janela estendida é um artifício comum usado em processamento de imagens onde a borda da mesma é replicada para que até mesmo os pixels nessa região possuam vizinhanças válidas. Observando adequadamente, é suficiente replicar as bordas por $M/2$ pixels. Dessa forma o número de pixels numa janela estendida é dado por $\left(W + \frac{M}{2}\right) x\ (W + M/2)$. Para filtrar um pixel são necessárias uma vizinhança e uma janela estendida, ambas centralizadas neste pixel.

Para cada pixel da janela (não estendida) é calculado seu peso em relação à vizinhança do pixel a ser filtrado. Então, um total de W^2 ciclos são gastos nesta etapa. Durante o cálculo dos pesos, a memória sendo utilizada não deve ser alterada. Para

melhorar o tempo de execução, foi acrescentada outra memória especializada (ME) para que, enquanto uma estiver sendo utilizada para calcular os pesos a outra possa ser preenchida simultaneamente, a fim de que sempre haja dados disponíveis para calcular os pesos, mantendo o *pipeline* preenchido.

Uma janela é composta por $\left(W + \frac{M}{2}\right) x (W + M/2)$ pixels e são gastos W^2 ciclos calculando pesos. Supondo uma memória onde são possíveis duas leituras e escritas simultâneas, são necessárias $\left[\left(W + \frac{M}{2}\right) x \left(W + \frac{M}{2}\right)\right] / 2$ ciclos para preencher uma ME como os dados de janela.

A versão do NLM proposta no presente trabalho utiliza os parâmetros $W = 21$ e $M = 7$; valores adequados de acordo com (BUADES, COLL e MOREL, 2011), para filtrar imagens com ruído branco com desvio padrão até 30. De acordo com os cálculos apresentados obtém-se uma janela estendida com 729 pixels. São necessários 441 ciclos de relógio para filtrar cada pixel e 389 (metade de 729 mais 49) ciclos para preencher a memória (ME).

Uma vez que são gastos *441* ciclos calculando pesos, e são necessários apenas 389 ciclos para armazenar os dados necessários na memória, conclui-se que há tempo suficiente para preencher a memória extra enquanto a anterior está sendo processada, garantindo que sempre haverá dados para preencher o pipeline.

3.2.3.4. Aproximação da função exponencial

Como pode ser observado na Equação 5, usada para calcular os pesos

$$w(i,j) = \frac{1}{Z(i)} e^{-\frac{S(i,j)}{h^2}} \qquad \textbf{Equação 5}$$

e na Equação 6, onde $Z(i)$ é a fator de normalização

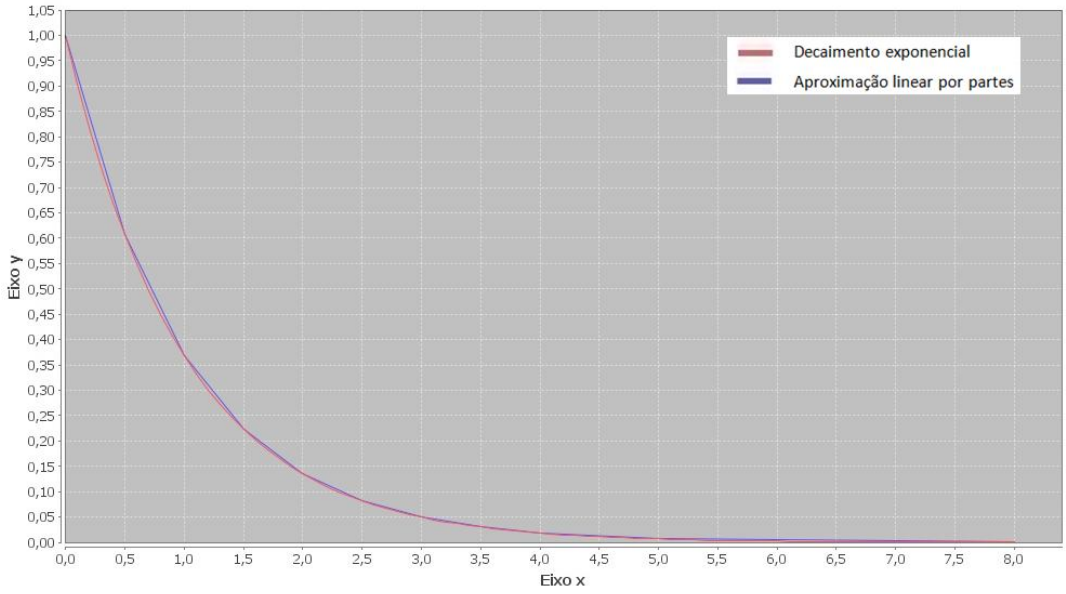
$$Z(i) = \sum_j e^{-\frac{S(i,j)}{h^2}} \qquad \textbf{Equação 6}$$

o NLM faz uso intensivo do cálculo de exponenciais. Entretanto, muitos processadores não possuem uma unidade de exponenciação em hardware. A operação é implementada em software, sendo realizada por meio da combinação de tabelas de busca, multiplicações de ponto flutuante e adições. A implementação em software da

exponenciação é muito lenta quando comparada com uma implementação em hardware (POTTATHUPARAMBIL e SASS, 2008).

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 e ROTH, 1969) para a função exponencial. Na Figura 22 são utilizadas apenas dez retas para aproximar o decaimento exponencial.

Figura 22 - Aproximação do decaimento usando dez retas.



Usando a aproximação linear por partes, o decaimento da função é aproximado por retas, das quais os pontos podem ser calculados pela equação da reta:

$$y = mx + b \qquad \textbf{Equação 7}$$

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, *et al.*, 2004) e (TAKAGI, KADOWAKI e TAKAGI, 2005).

Obtidos os pesos relativos a cada pixel, resta calcular o valor final do pixel. De acordo com a Equação 3

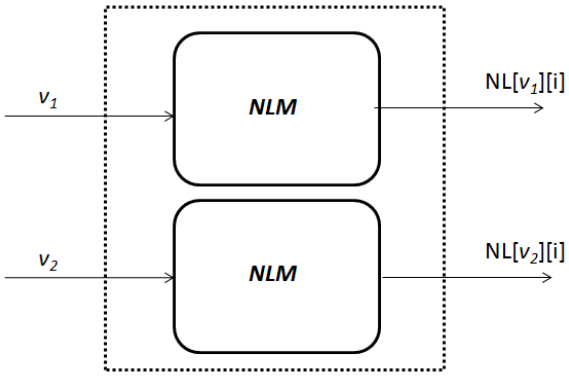
$$NL[v][i] = \sum_{j \,\epsilon\, I} w(i,j)\, v(j), \qquad \textbf{Equação 3}$$

basta calcular a média ponderada de todos os pixels da janela, o que é feito novamente com uso de *pipelines*.

3.3. Processamento Paralelo

Outra 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 os limites de E/S). A Figura 23 mostra dois núcleos NLM independentes usados para acelerar a filtragem.

Figura 23 - Dois núcleos *NLM* para filtragens independentes.



3.4. Implementação e Simulação

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

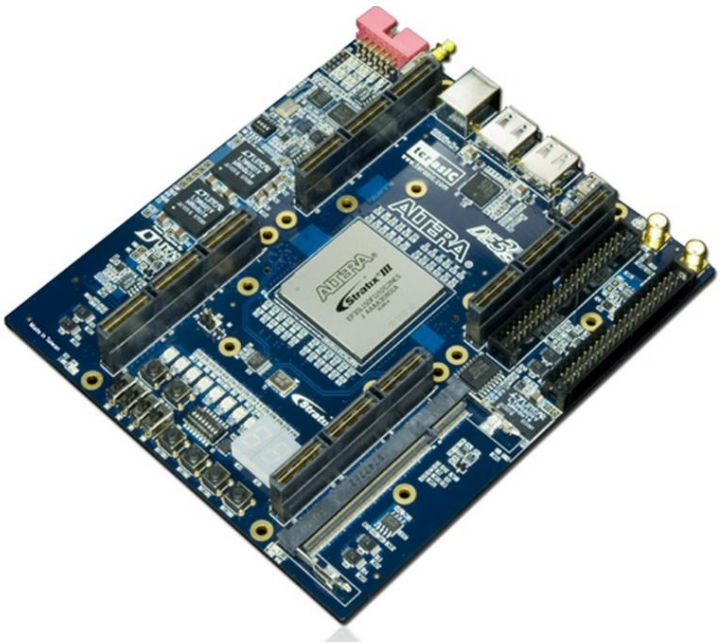
O Stratix III FPGA Development Kit, Figura 24, oferece vários recursos, entre os quais: provê dispositivo FPGA EP3SL50F484C2, 142.500 elementos lógicos, 744 pinos de *I/O*, 384 multiplicadores 18x18, memórias flash, RAM estática e dinâmica, interfaces Ethernet, CompactFlash, JTAG, osciladores de 50 e 125 MHz, LEDs, displays de sete segmentos, entre outros.

Como já foi mencionada, a verificação funcional aqui apresentada baseia-se na metodologia BVM (*Brazil-IP Verification Methodology*) (ARAÚJO, 2010). Para a realização da verificação funcional, este trabalho utilizou a linguagem C para implementação do modelo de referência e a linguagem SystemVerilog para a implementação do *design under verification* (DUV) e dos *testbenches* propriamente

ditos. A linguagem C foi escolhida devido a existência de um API (*Application Programming Interface*) própria do SystemVerilog usada para integrar código escrito em ambas as linguagens.

Vale ressaltar a praticidade da linguagem SystemVerilog ao permitir integrar, em ambiente único, entidades de alto nível, transações, DUV, sinais e componentes do *testbench*, através dos recursos oferecidos pela linguagem para esses diversos fins. Os *testbenches* codificados em SystemVerilog foram executados na ferramenta QuestaSim versão 10.0 da Mentor Graphics.

Figura 24 - Stratix III.



O modelo de referência (*Reference Model*) para filtro NLM proposto foi desenvolvido na linguagem C, pois o uso de uma linguagem de mais alto nível para criação do modelo referência permite obter de forma mais simples uma implementação correta do DUV. Para modelos escritos com um interface simples nessa linguagem a *Direct Programming Interface (*DPI) pode ser usada para executar o código *de dentro* do *SystemVerilog***.**

A DPI consiste de um API (*Application Programming Interface*) própria do SystemVerilog que visa fornecer um forma simples para se comunicar com o C (e por extensão com o C++). Uma vez que uma rotina C é importada com a declaração *import* ela pode ser chamada como qualquer outra rotina *SystemVerilog*.

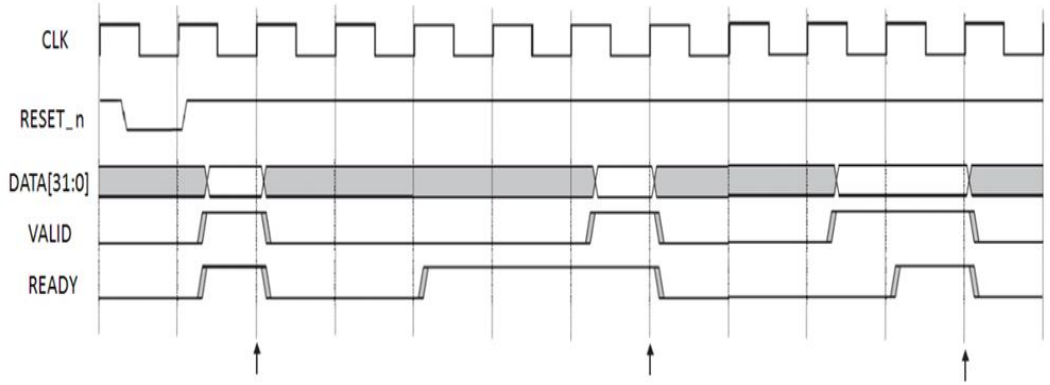
3.4.1. Protocolo de sinais

Para facilitar o fluxo de dados entre os módulos, foi implementado um protocolo de sinais simples baseado nos sinais *ready* e *valid*. Trata-se de um protocolo síncrono composto por quatro sinais básico: *clk*, *data*, *valid* e *ready*. Além deles, ainda existem os sinais *reset*. Abaixo, segue a descrição de cada um deles:

- *clk*: *clock* global do sistema;
- *data*: dados a serem transferidos (variam em quantidade e largura de acordo com o módulo);
- *valid:* sinaliza a existência de dados válidos a serem transferidos;
- *ready:*indica que o módulo está pronto para realizar a transação;
- *reset:reset* assíncrono do sistema.

A regra fundamental do protocolo é: dados são transferidos na borda de subida do *clock* quando *valid* e *ready* estão ambos em nível lógico alto. A Figura 25 mostra as transações envolvendo os sinais apresentados. As setas indicam os momentos exatos de cada transferência.

Figura 25 - Transferências de dados usando os sinais do protocolo.



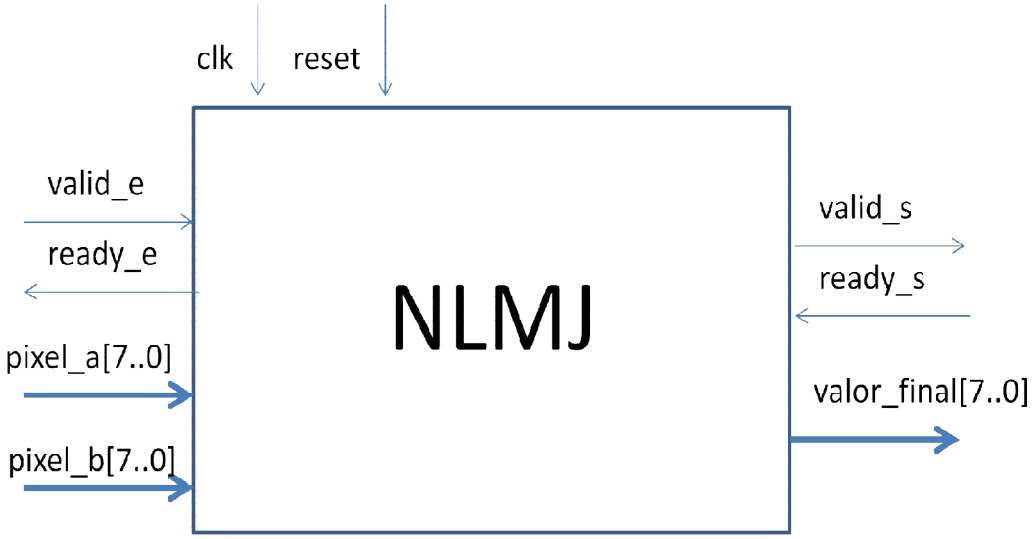
Para diferenciar os termos do protocolo usados nos fluxos de entrada e de saída, *valid* e *ready* são denominados *valid_e* e *ready_e* no primeiro caso e *valid_s* e *ready_s* no segundo.

3.4.2. Arquitetura da implementação

Diante fluxo da filtragem apresentado na arquitetura lógica (Figura 19, página 43), é possível ilustrar a disposição de blocos da implementação, suas conexões e sinais envolvidos em 2 níveis. Na Figura 26, é visto, no primeiro nível, o diagrama de bloco do filtro proposto, o qual é denominado NLMJ.

A entrada para o NLMJ é constituída dos valores referentes aos pixels da janela que será usada para filtrar o pixel central dessa mesma janela, sendo transmitidos de dois em dois pixels por vez (já que a memória utilizada é capaz de receber duas escritas simultâneas). Os dois pixels de entrada, *pilxel_a* e *pixel_b*, recebem valores no intervalo de 0 a 255 cada um, referentes à escala de cinza. Para cada uma dessas entradas é usado um barramento de 8 bits para transportar esses valores.

Figura 26 - Diagrama de bloco do NLMJ.



A saída do módulo NLMJ, valor_final, é o pixel central da janela recebida como entrada após ser filtrado pelo módulo, sendo formado também por 8 bits. No segundo nível, ilustrado na Figura 27, temos dois módulos, sendo que o primeiro, Figura 27.a, tem a mesma entrada do NLMJ, e é responsável pelo armazenamento da janela e cálculo da distância Euclidiana quadrática, correspondendo as duas primeiras etapas do fluxo mostrado na Figura 19 (página 43): (Memória para janelas e Calcula distância Euclidiana Quadrática). Esse módulo é denominado CDE2P49J, e tem como saídas um pixel (8 bits) e valor da distancia Euclidiana quadrática (deq) correspondente, que está no intervalo de 0 a 3186225 (limites de valores da Distância Euclidiana quadrática - Equação 4), para os quais é suficiente um barramento de 22 bits. Recorde que, para cada pixel da janela, é calculado o seu peso usando a configuração de vizinhança da qual é o pixel central. A primeira etapa para essa tarefa é o cálculo da distância Euclidiana quadrática.

Também nesse segundo nível, a saída do módulo CDE2P49J é transmitida para o módulo CP, Figura 27.b, que é responsável por calcular, depois normalizar os

pesos e, por fim, calcular o valor final do pixel filtrado. São as duas últimas etapas do fluxo mostrado na Figura 19: Calcula Pesos e Calcula Valor Final. A saída é a mesma do NLMJ.

Resumindo, o NLMJ funciona da seguinte forma: o módulo CDE3P49J recebe a janela referente ao pixel a ser filtrado e para cada pixel calculo a distância Euclidiana quadrática. A saída de CDE3P49J, pixel_central e deq, é comunicada para o módulo CP que é responsável por calcular e normalizar os pesos e, finalmente, calcular o valor final do pixel filtrado.

3.4.3. Verificação funcional

Como foi dito na Seção 2.11 (página 35), o objetivo da verificação é validar o modelo HDL do filtro. Na verificação funcional, estímulos de entrada são inseridos em uma implementação de alto nível (modelo de referência) e no módulo de hardware a ser verificado, *Design Under Verification* (DUV). As saídas de cada um deles são confrontadas e certos valores de saída são monitorados para que determinada condição seja atingida e a verificação seja finalizada.

Seguida a metodologia apresentada, ou seja, após a construção e validação, foi obtido no primeiro nível o *testbench* para verificação do NLMJ, que é apresentado na Figura 28.

Como tanto os dados de entrada como os de saída são pixels, que correspondem a valores que variam de 0 a 255, para atingir a cobertura da verificação foram estabelecidas as seguintes condições:

1. Cada valor de pixel de entrada nessa faixa deve aparecer pelo menos 10 vezes.
2. Cada valor final do pixel de saída, também nessa faixa, deve aparecer pelo menos 3 vezes.

A maior cobertura nos pixels de entrada está de acordo com as características próprias do *NLM*, uma vez que uma grande quantidade de pixels de entrada (*pixel_a_in* e *pixel_b_in*) é utilizada para filtrar um único pixel, que corresponde à saída (*valor_filnal*). Na Figura 29 é mostrado, a título de exemplo, parte do código usado para configurar a cobertura da entrada do módulo. A cobertura de saída é feita de modo semelhante.

A Figura 28 representa o testbench desenvolvido para validar a implementação proposta para NLM. O *refmod* corresponde a implementação do NLM em linguagem C e o *duv* ao sua implementação em *SystemVerilog,* a linguagem usada para a descrição do hardware.

Figura 27 - visão de blocos para o segundo nível.

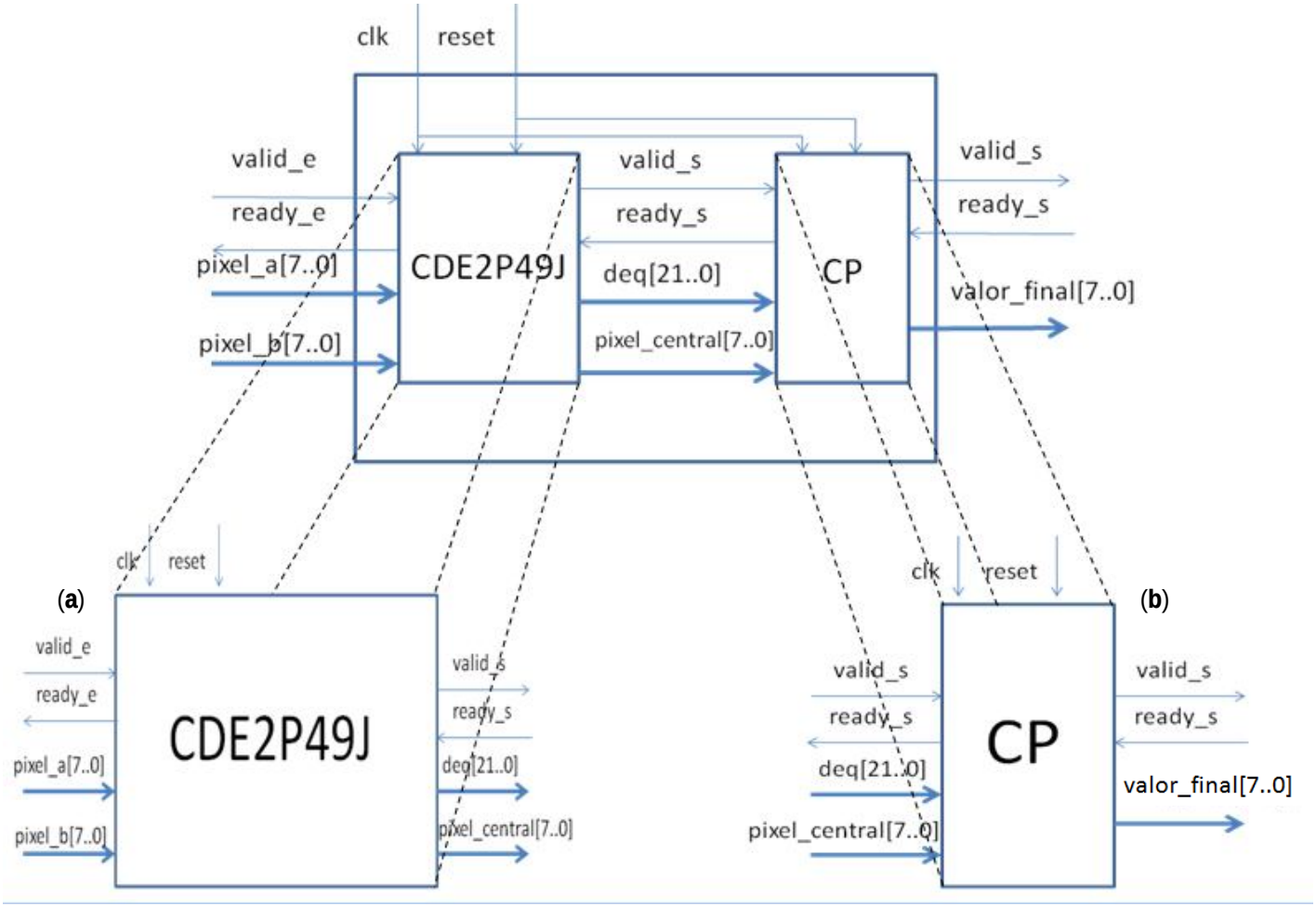
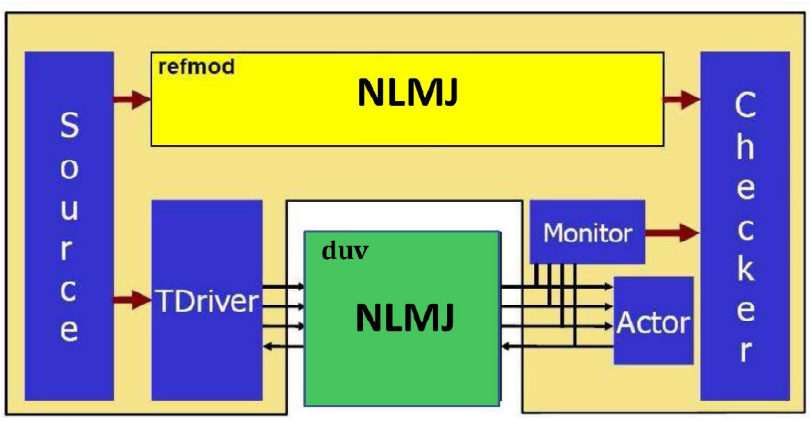
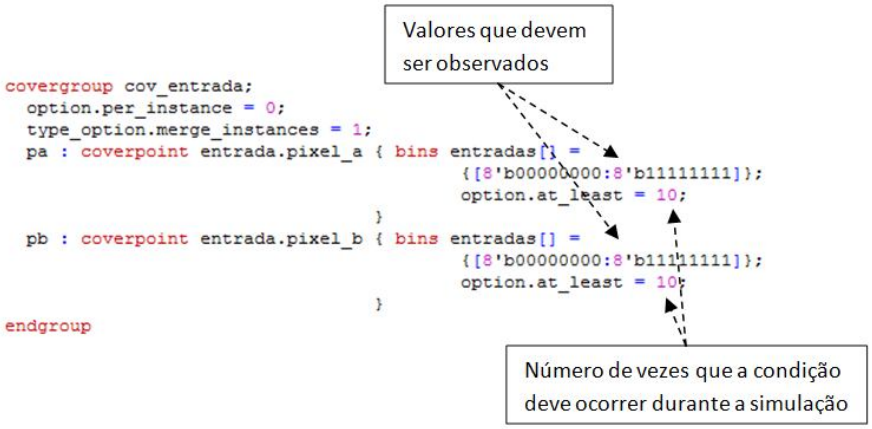


Figura 28 - Testbench para filtro NLM proposto.



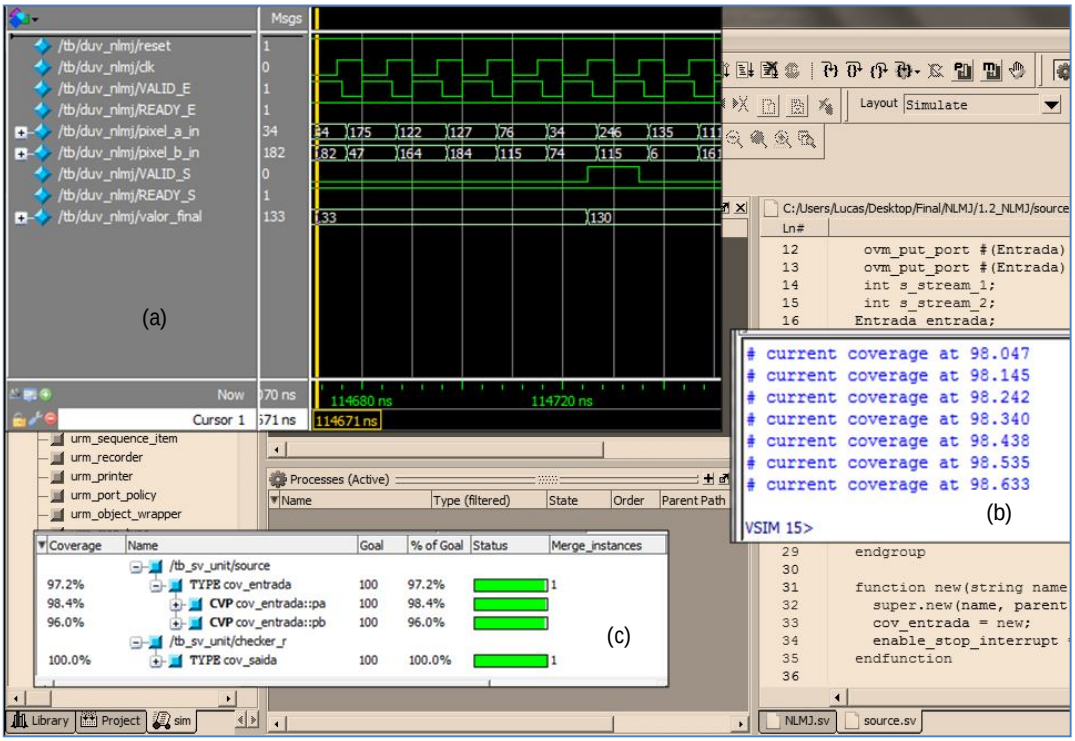
A Figura 30 ilustra a execução do testbench no ambiente de simulação disponível na ferramenta Questa Advanced Simulator. Na Figura 30.a são mostrados os sinais da simulação. Na Figura 30.b temos o acompanhamento geral da cobertura, ou seja tanto da entrada quanto da saída. Na Figura 30.c temos a cobertura mais detalhada, entrada e saída separadamente, que torna possível observar minuciosamente o andamento da cobertura.

Figura 29 - Configuração de cobertura.



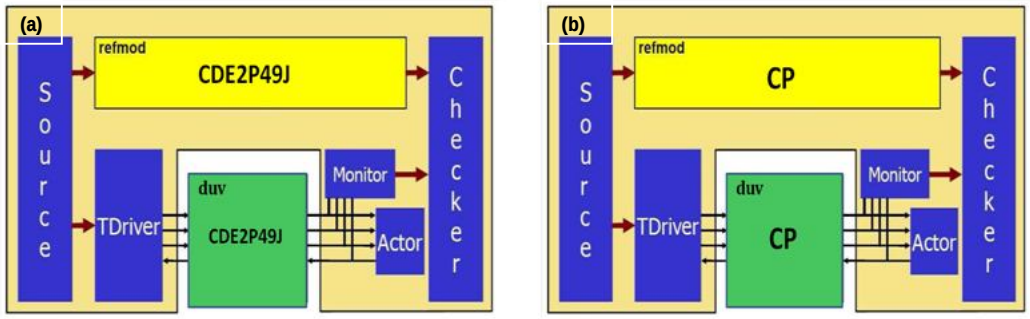
Uma das grandes vantagens do *testbench* é verificar automaticamente quando é atingida a cobertura.

Figura 30 - Cobertura da verificação funcional.



Seguindo a metodologia de verificação, também foram construídos *testbenchs* para os blocos do segundo nível do NLMJ: CDE3P49J visto na Figura 31.a e CP, Figura 31.b. Para ambos, CDE3P49J e CP, foram executados testes similares aos efetuados para o NLMJ.

Figura 31 - Testbenches para módulo do segundo nível.



Com o ambiente de simulação preparado antes do início da implementação do módulo em HDL, foram obtidos os componentes da verificação que permitiram avaliar e corrigir quando necessário o comportamento do filtro de ruídos quanto a sua

funcionalidade, tanto nos submódulos como no sistema como um todo, produzindo um núcleo de filtragem em hardware validado.

Capítulo IV: Resultados

Nesta sessão são apresentados os resultados de síntese em FPGA, assim como os resultados e as comparações do desempenho e da qualidade entre a versão em FPGA e a versão em software do NLM em janela.

4.1. Área e frequência

Para 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*).

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). Esses resultados são resumidos na Tabela 2.

Tabela 2 - Resultados de síntese.

	Lógica Utilizada	Memória Utilizada	Frequência (MHz)
Um núcleo	34%	2%	104,78
Dois núcleos	68%	4%	96,26

4.2. Qualidade da filtragem

A seguir é apresentada uma sequência de figuras que permitem uma comparação da percepção de qualidade entre resultados das filtragens realizadas pelo NLM em janela executando em software e sua versão em SystemVerilog.

A Figura 32 apresenta a imagem, supostamente sem ruído, na qual é inserido ruído aleatório com diferentes valores de desvio padrão.

O resultado dos diferentes níveis de ruídos inseridos, 5, 10, 15 e 20, pode ser visualizado nas Figura 33.A, Figura 33.B, Figura 33.C e Figura 33.D, respectivamente. Como pode ser notado, quanto maior o desvio padrão, maior é a quantidade de ruído inserida na imagem.

Figura 32 - Imagem sem ruído.

Figura 33 - Diferentes níveis de ruído: 5 (A), 10 (B),15 (C) e 20 (D).

Como objetivo de observar se a versão proposta do NLM para hardware mantém uma qualidade comparada à versão em software, as imagens degradadas da Figura 33, foram filtradas pelas duas versões do NLM. Para ambas as versões foram utilizados fatores de decaimento (h) de 2, 4, 6 e 8 para filtrar os ruídos com desvios

padrão 5, 10, 15 e 20, respectivamente, conforme (BUADES, COLL e MOREL, 2011). O fator de decaimento funciona como um fator de filtragem e esses valores implicam um bom equilíbrio entre perda de detalhes e ruído residual.

As imagens resultantes da filtragem pelo NLM em software são apresentadas na Figura 34 e resultantes da filtragem pelo NLM em hardware são mostradas na Figura 35.

Figura 34 - Resultado da filtragem pelo NLM em software.

Através de uma observação simples, isto é, feita a olho nu, os resultados obtidos na Figura 34 e na Figura 35 não apresentam diferenças, apontando que a versão em hardware mantém a qualidade da versão em software.

Ainda para comparar esses resultados foi utilizado o *ruído do método* (BUADES, COLL e MOREL, 2005). Esse método efetua a diferença absoluta entre a imagem com ruído e a imagem filtrada, mostrando o ruído removido pelo algoritmo. É oportuno relembrar que se a técnica utilizada remover apenas o ruído da imagem, o ruído de método será semelhante ao ruído branco.

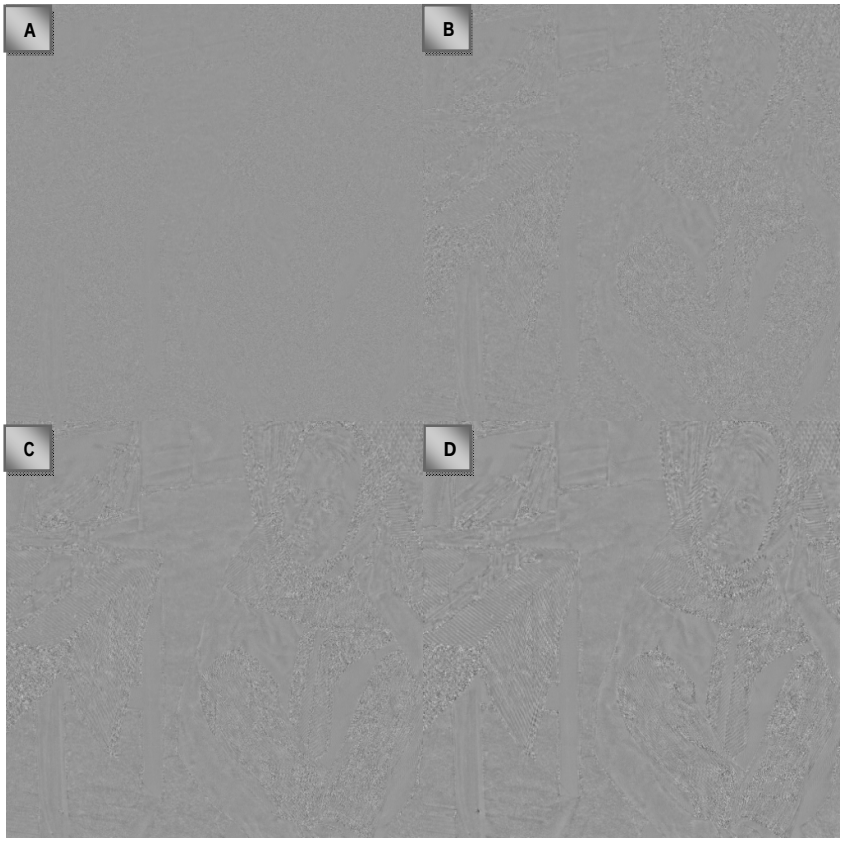
Figura 35 - Resultado da filtragem pelo NLM em hardware.

Para obter o ruído de método referente ao NLM em software cada imagem da Figura 34 foi subtraída, pixel a pixel nas posições equivalentes, da imagem sem ruído da Figura 32, resultando nas imagens: Figura 36.A, Figura 36.B, Figura 36.C e Figura 36.D.

O mesmo processo é aplicado a cada imagem da Figura 35 para obter o ruído método para o filtro em hardware obtendo-se as imagens: Figura 37.A, Figura 37.B, Figura 37.C e Figura 37.D.

Comparando os resultados da aplicação do ruído do método tanto para o NLM em software, Figura 36, quanto para NLM em hardware, Figura 37, observa-se que os resultados são quase idênticos, mostrando mais uma vez que os resultados das versões em hardware e software são muito próximos.

Figura 36 - Ruído do método para o NLM em software.



O último método utilizado para mensurar comparativamente o resultado da qualidade da versão em hardware proposta para NLM é o Erro médio quadrático (*mean square error* - *MSE*) (ZHOU, 2006). É uma abordagem estatística na qual um valor menor de MSE indica que a imagem filtrada está mais próxima da original.

A Tabela 3 e a Figura 38 mostram uma comparação entre o algoritmo NLM em FPGA e o NLM em software com relação ao MSE para ruídos com diferentes valores de desvios padrão.

Os resultados mostram que a o filtro proposto em hardware obteve resultados comparáveis ao NLM em software em termos do erro quadrático médio MSE, percepção visual e comparação por ruído do método.

Note que as pequenas diferenças encontradas durante a as comparações se devem ao uso de aproximações de números fracionários por inteiros e ao uso da aproximação linear por partes da função exponencial.

Figura 37 - Ruído do método para o NLM em hardware.

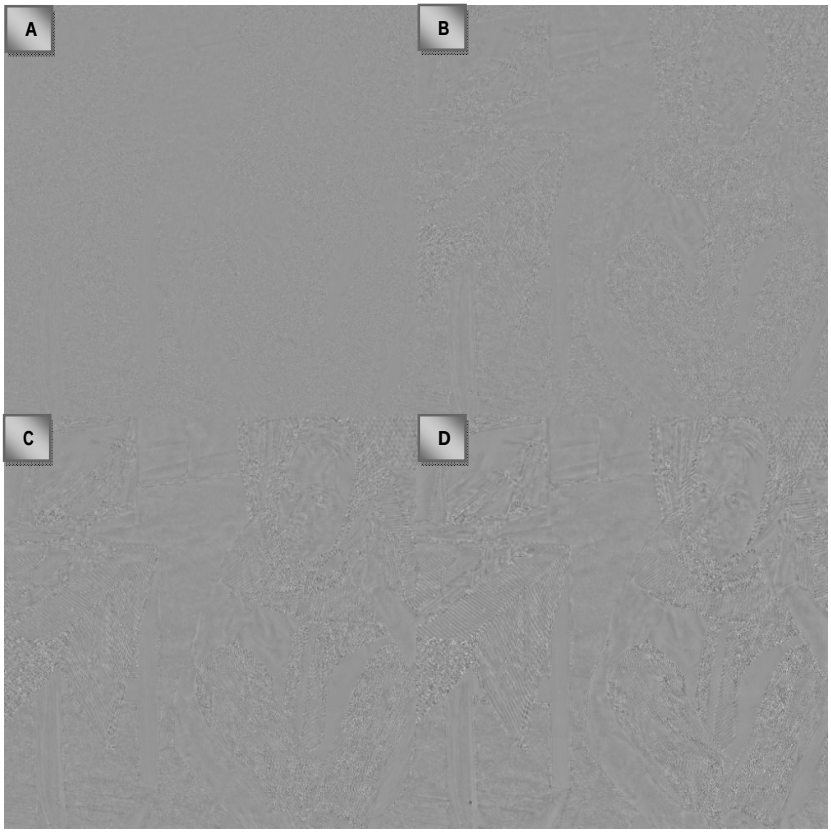


Tabela 3 - Comparação de MSE

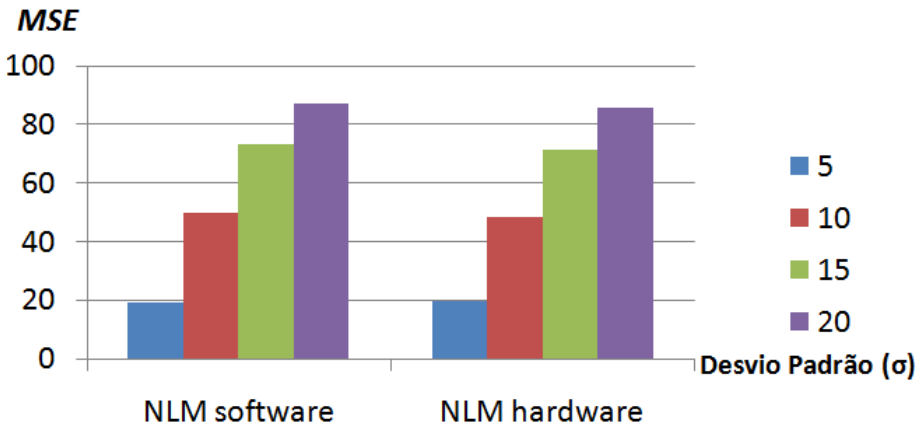
Método utilizado	Desvio Padrão (σ)			
	5	***10***	***15***	***20***
NLM em software	19,15	49,88	73,29	86,96
NML em FPGA	19,45	48,47	71,15	85,96

4.2. Resultados de desempenho

Conforme sugerido por (BUADES, COLL e MOREL, 2005), foi utilizado M = 7, e uma janela de pesquisa de 21x21 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 período de relógio de T (9,45*ns* obtidos na simulação) permite inferir o tempo despendido para filtrar uma imagem, sendo que, para uma imagem com N^2 pixels são gastos 441xN^2xT.

Na Tabela 4 experimentos em um PC com um processador Core I5 2.53GHz, 6GB de RAM e Sistema Operacional Windows 7 Professional, demonstraram que o NLM em software gasta aproximadamente 3440,5 segundos para filtrar uma imagem de 2592*x*1944 pixels em uma execução *monothread*, e gasta 1617,8 segundos em uma execução *multithread* (no caso quatro threads de execução). Já a versão em FPGA aqui proposta gasta 21,2 segundos aproximadamente para filtrar a mesma imagem usando um núcleo de filtragem e 11,56 segundos utilizando dois núcleos.

Figura 38 - Gráfico de comparação de MSE (relativo à Tabela 3).

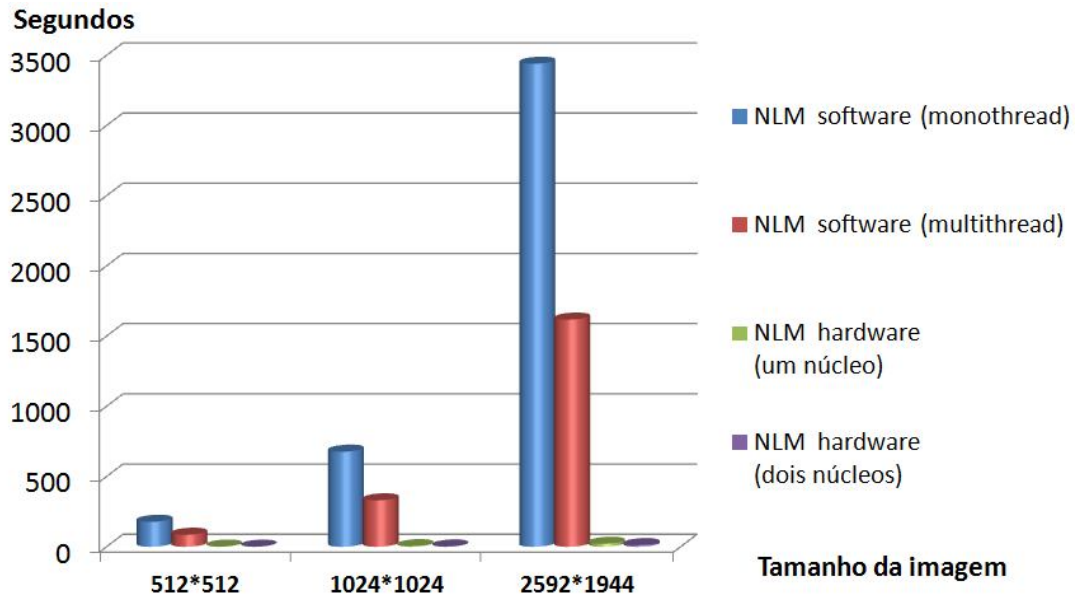


Conforme demonstrado na Tabela 4 e na Figura 39, comparado a implementação monothread em software, o resultado do trabalho proposto é aproximadamente 160 vezes mais rápido usando apenas um núcleo de filtragem, e cerca de 290 vezes mais rápido quando dois núcleos são utilizados.

Tabela 4 - Resultados de desempenho.

Tamanho da imagem	**Método utilizado (tempo em segundos)**			
	NLM software (monothread)	***NLM software (multithread)***	***NLM hardware (um núcleo)***	***NLM hardware (dois núcleos)***
512x512	177,7	85,5	1,10	0,60
1024x1024	677,2	330,7	4,41	2,40
2592x1944	3440,50	1617,8	21,2	11,56

Figura 39 - Gráfico com resultados de desempenho (relativo à Tabela 4).



4.3. Comparação com outros trabalhos e discussão dos resultados

A Tabela 5 mostra os resultados do sistema proposto e de outros trabalhos relacionados à melhoria do desempenho do NLM relativa ao tempo de gasto na execução da filtragem em imagens de $512 \, x \, 512$ pixels. As comparações são apresentadas apenas em termos de tempo, pois em relação à qualidade visual os resultados diferem de forma significativa entre os trabalhos até mesmo quando se referem ao uso da versão original do NLM e usam a mesma imagem. Isso provavelmente ocorre porque, embora usem imagens clássicas, como a de *Lena*, as imagens possuem diferentes níveis de ruído.

Em (MAHMOUDI e SAPIRO, 2005), que utiliza pré-seleção das vizinhanças com base no valor médio e gradiente para acelerar o NLM não foram apresentados resultados diretos, e por isso não aparecem na Tabela 5. Foi dito apenas que a aceleração se encontrava entre 7 e 24 vezes dependendo dos parâmetros escolhidos.

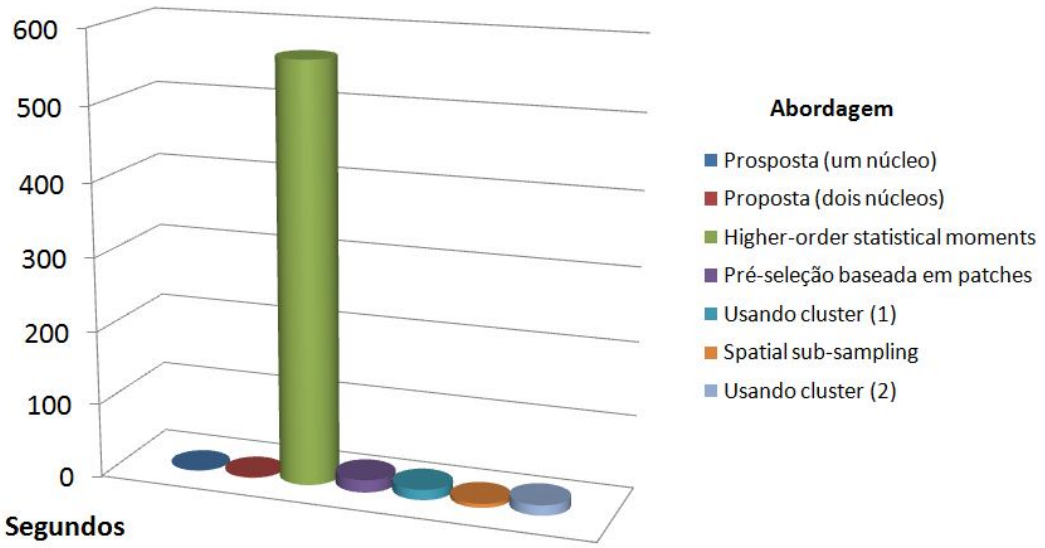
Em (BROX, KLEINSCHMIDT e CREMERS, 2008) é relatado que a aceleração do método usando subamostragem espacial desfoca a imagem. Nesse mesmo trabalho e em (DINESH, GOVINDAN e MATHEW, 2009) ao fazerem o uso de clusters de computadores, os resultados são mais significativos, uma vez que o tempo de execução permanece praticamente constante ao filtrar imagens com maiores dimensões ao custo de aumentar os recursos computacionais. A abordagem utilizando

estatísticas de ordem superior (DAUWE, GOOSSENS, *et al.*, 2008) foi a que consumiu mais tempo de processamento para realizar a filtragem.

Tabela 5 - Comparação de desempenho com outras abordagens.

Trabalho	Tempo (segundo)	Abordagem
Proposto	1,10	execução em FPGA (um núcleo)
Proposto	0,60	execução em FPGA (dois núcleos)
(DAUWE, GOOSSENS, *et al.*, 2008)	565	por meios de estatísticas de ordem superior *(higher-order statistical moments)*
(DINESH, GOVINDAN e MATHEW, 2009)	17	pré-seleção baseada em *patches* estruturalmente semelhantes
	14	Usando cluster (1)
(BROX, KLEINSCHMIDT e CREMERS, 2008)	5,4	pré-seleção de vizinhanças usando subamostragem espacial (*Spatial sub-sampling*)
	14	Usando cluster (2)

Figura 40 - Gráfico desempenho para várias abordagens.



Os resultados podem ser mais bem analisados observando a Figura 40 na qual os dados da Tabela 5 são apresentados num gráfico de colunas e mostram que o filtro

proposto em hardware obteve os melhores resultados, ou seja, menos tempo para executar a filtragem usando o NLM, tanto para o caso onde é usado apenas um núcleo de processamento quanto para o caso usando dois núcleos.

Capítulo 5 Conclusão

Neste trabalho foram investigados métodos para acelerar o algoritmo *Non-Local means* em FPGA. O enfoque dado para encontrar uma melhor taxa de custo-desempenho para o algoritmo concentrou-se em imagens naturais; custo em termos de perda de qualidade, e desempenho em termos de ganhos de aceleração.

Buscou-se manter os resultados perto daqueles da versão para software do algoritmo NLM em termos de MSE e *ruído do método*, embora seja difícil definir indicadores na área de qualidade de imagens visto que indicadores matemáticos nem sempre refletem a apreciação subjetiva do observador.

O trabalho resultou numa implementação para o algoritmo NLM em janela usando um *hardware* de propósito específico, no qual uma alta *performance* é atingida através da construção de memórias customizadas, *pipelines* e paralelismo projetados especificamente para a tarefa a ser realizada.

Os resultados obtidos na simulação mostram que o algoritmo é executado em média 290 vezes mais rápido, usando apenas dois núcleos, 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.

Os resultados da redução de ruídos são semelhantes ao algoritmo original tanto em *MSE* quanto em percepção visual.

Ao mesmo tempo, a utilização de uma fase de verificação funcional como a aqui apresentada é um ponto positivo para o trabalho, pois favorece a confiabilidade do filtro desenvolvido.

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 XVIII Workshop IBERCHIP-2012 entre 29 de Fevereiro e 2 de Março de 2012 em Playa del Carmen, México.

O presente trabalho ainda pode ser explorado na intenção de se apresentar soluções para suas desvantagens e melhorar seu desempenho.

Como o tempo de execução do sistema proposto é diretamente vinculado ao período de relógio, seria válido estudar outras formas desenvolvimento buscando-se uma maior frequência de operação e, por conseguinte uma melhor taxa de desempenho.

Por outro lado, pode-se ainda buscar alternativas de desenvolvimento que resultem em um menor consumo de elementos lógicos, o que possibilitaria, através do emprego de mais núcleos de filtragem, melhorar ainda mais a taxa de desempenho.

Outra alternativa seria experimentar modificações no sistema para observar como as variações nas aproximações usadas para o desenvolvimento em hardware, como a aproximação linear por partes, podem influenciar tanto o tempo de processamento quanto a qualidade da filtragem. Isso é importante uma vez que áreas de aplicação diferentes podem requerer diferentes níveis de filtragem e desempenho.

Referências Bibliográficas

ADAMS, A. et al. Gaussian KD-trees for fast high-dimensional filtering. **Proc. of ACM SIGGRAPH**, 2009.

ADAMS, A.; BAEK, J.; DAVIS, A. Fast high-dimensional filtering using the permutohedral lattice. **Proc. of EuroGraphics**, 2010.

ALTERA. Quartus II Web Edition Software, 2012. Disponivel em: <https://www.altera.com/download/software/quartus-ii-we>. Acesso em: 10 jun. 2012.

ARAÚJO, H. F. D. O. **BVM:** Reformulação da metodologia de verificação funcional VeriSC. Campina Grande: Universidade Federal de Campina Grande, 2010. Dissertação de Mestrado em Ciência da Computação.

BELLMAN, R.; ROTH, R. Curve Fitting by Segmented Straight Lines. **Journal of the American Statistical Association**, Setembro 1969. 1079-1084.

BERGER, A. **Embedded systems design**. Kansas: CMP Books, 2012.

BOBDA, C. **Introduction to Reconfigurable Computing:** Architectures, Algorithms, and Applications. [S.l.]: Springer, 2007.

BOVIK, A. C. **Handbook of image and video processing**. San Diego: Elsevier, 2005.

BROWN, S.; VRANESIC, Z. **Fundamentals of digital logic with vhdl design.** Toronto: Mc Mac Hill, 2000.

BROX, T.; KLEINSCHMIDT, O.; CREMERS, D. Efcient nonlocal means for denoising of textural patterns. **IEEE Trans. Image Process**, p. 1083-1092, 2008.

BUADES, A.; COLL, B.; MOREL, J. M. On image denoising methods. **CMLA. [S.l.]**, 2004. 40.

BUADES, A.; COLL, B.; MOREL, J. M. A non-local algorithm for image denoising. **Proc. IEEE CVPR, vol. 2**, 2005. 60-65.

BUADES, A.; COLL, B.; MOREL, J.-M. IPOL: Non-local Means Denoising, 05 Outubro 2011. Disponivel em: <http://www.ipol.im/pub/algo/bcm_non_local_means_denoising/>. Acesso em: 26 Janeiro 2012.

CONDAT, L. A Simple Trick to Speed Up and Improve the Non-Local Means. **Laurent Condat - Homepage**, Agosto 2010. Disponivel em: <http://www.greyc.ensicaen.fr/~lcondat/publications.html>. Acesso em: 16 Julho 2012.

COUPÉ, P. et al. An optimized blockwise nonlocal means denoising lter for 3-D magnetic resonance images. **IEEE Trans. Med. Imag**, p. 425-441, 2008.

COUPÉ, P.; YGER, P.; BARILLOT, C. Fast Non Local Means Denoising for 3D MR Images. **SpringerLink**, 2006.

DAUWE, A. et al. A fast non-local mage denoising algorithm. **Proc. of SPIE Electronic Imaging**, Janeiro 2008. 1331-1334.

DINESH, J. P.; GOVINDAN, V. K.; MATHEW, T. Robust estimation approach for nonlocal-means denoising based on structurally similar patches. **Int. J. Open Problems Compt. Math.**, 2009.

GONZALES, R. C.; WOODS, R. E. **Processamento de Imagens Digitais**. São Paulo: Edgard Blücher, 2000.

GOOSSENS, B. et al. An improved non-local denoising algorithm. **Proc. of LNLA**, 2008.

GRAPHICS, M. Questa Advanced Simulator, 2012. Disponivel em: <http://www.mentor.com/products/fv/questa/>. Acesso em: 10 jun. 2012.

HAUCK, S. **Reconfigurable computing the theory and practice of fpga based computation.** [S.l.]: Elsevier, 2008.

KARNATI, V.; ULIYAR, M.; DEY, S. Fast non-local algorithm for image denoising. **Proc. of IEEE ICIP**, 2009.

LINDENBAUM, M.; FISCHER, M.; AND BRUCKSTEIN, A. On gabor contribution to image enhancemen. **Pattern Recognition, vol. 27**, 1994. 1–8.

MAHMOUDI, M.; SAPIRO, G. Fast image and video denoising via nonlocal means of similar neighborhoods. **Signal Processing Letters**, 2005. 839-842.

ORACLE. Oracle Technology Netwoerk for Java Developers , 2012. Disponivel em: <http://www.oracle.com/technetwork/java/index.html>. Acesso em: 10 jun. 2012.

PANATO, A. et al. Design of very deep pipelined multipliers for FPGAs. **Design, Automation and Test in Europe Conference and Exhibition**, 2004.

PAVLIDIS, T. A review of algorithms for shape analysis. **Comput.Graphics Image Processing**, 1978. 243-258.

PAVLIDIS, T. Algorithms for shape analysis of contours and waveforms. **IEEE Trans. Pattern Anal. Mach. Intell.**, 1980. 301-302.

POTTATHUPARAMBIL, R.; SASS, R. Implementation of a CORDIC-based Double Precision Exponential Core on an FPGA. **Proceedings of the Fourth Annual Reconfigurable Systems Summer Institute**, p. 7-9, 2008.

RAMER, U. An iterative procedure for the polygonal approximation of plane curves. **J. Comput. Graphwcs and Image Processing**, v. 1, p. 244-256, 1972.

ROSSI, D. L. **Tese de Doutorado: Projeto de um controlador PID para controle de ganho de uma câmera com sensor CMOS utilizando computação reconfigurável**. USP. São Carlos. 2012.

RUDIN, L.; OSHER, S.; FATEMI, E. Nonlinear total variation based noise removal algorithm. **Physica D, vol.60**, 1992. 259–268.

SHAHAM, N. **Dissertação de Mestrado: Métodos para aceleração do "non-local means" algoritmo de redução de ruído**. PUC. Rio de Janeiro. 2007.

SOMMERVILLE, I.; KOTONYA, G. **Requirements Engineering**. Wiley: [s.n.], 1998.

SYSTEMVERILOG. SystemVerilog - Overview, 2012. Disponivel em: <http://www.systemverilog.org/overview/overview.html>. Acesso em: 10 jun. 2012.

TAKAGI, N.; KADOWAKI, S.; TAKAGI, K. A hardware algorithm for integer division. **17th IEEE Symposium on Computer Arithmetic**, p. 140,146,27-29, 2005.

TANENBAUN, A. S. **Organização Estruturada de Computadores**. São Paulo: Pearson, 2007.

TOMASI, C.; MANDUCHI, R. Bilateral filtering for gray and color image. **Proc. IEEE ICCV**, 1998. 839–846.

VANDERLEI, B. **Dissertação de Mestardo: Projeto de um módulo de aquisição e pré-processamento de imagem colorida baseado em computação reconfigurável e aplicado a robôs móveis.** USP. São Carlos. 2004.

VIGNESH, R.; OH, B. T.; KUO, C.-C. J. “Fast non-local means (NLM) computation with probabilistic early termination. **IEEE Signal Process.Lett.**, p. 277-280, 2010.

ZHOU, M. Estimators, Mean Square Error, and Consistency, 20 Janeiro 2006. Disponivel em: <http://www.ms.uky.edu/~mai/sta321/mse.pdf>. Acesso em: 22 Janeiro 2012.

APÊNDICE: TRABALHO PUBLICADO

Otimização do algoritmo Non-local means utilizando uma implementação em FPGA

Lucas Lucena Gambarra, José Antônio Gomes de Lima, Hamilton Soares da Silva, Leonardo Vidal Batista, Daniel Soares e Marques

Centro de Informática
Universidade Federal da Paraíba
João Pessoa, Brasil
{lucas, jose, hamilton, leonardo}@di.ufpb.br danielsmarx@gmail.com

***Abstract*— This paper proposes a hardware implementation for the non-local means algorithm for image denoising with a lower computation time using pipelines, hardware parallelism and piecewise linear approximation. It is about 170 times faster than the original non-local means algorithm, yet produces comparable results in terms of mean-squared error (MSE) and perceptual image quality.**

INTRODUÇÃO

Filtragem de ruído em imagem é 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. A redução de ruídos é muitas vezes necessária como um pré-processamento para outras tarefas, tais como: compressão, segmentação e reconhecimento [1].

Em 2005, Buades [2] apresentou um algoritmo inovador para este propósito, conhecido como non-local means (NLM). Anteriormente ao NLM, várias abordagens haviam sido propostas para eliminar o ruído. Tais métodos sempre operam localmente na imagem. O NLM, por sua vez, utiliza a informação codificada em toda a imagem para remover o ruído, sendo notavelmente mais eficaz quando comparado a algoritmos mais antigos. Entretanto, apresenta alta complexidade computacional, dificultando sua utilização para enfrentar questões práticas, até mesmo para imagens relativamente pequenas. O próprio Buades [2] sugeriu utilizar apenas uma janela (NLM em janela) de pesquisa, em vez de toda a imagem, para realizar a filtragem. Porém, mesmo reduzindo bastante o tempo de execução, a complexidade continuou elevada.

Neste contexto, seria desejável uma alternativa para reduzir o tempo de execução do NLM. É com este propósito que o presente trabalho propõe uma implementação em hardware específico para o NLM em janela.

A Seção II introduz a teoria sobre o NLM e analisa a complexidade. A Seção III mostra o fluxo da filtragem de ruídos e a estratégia para a implementação em hardware. Informações sobre síntese e a metodologia de simulação são explicadas na Seção IV. A Seção V expõe os resultados alcançados em hardware e compara com a implementação em software. Finalmente, a Seção VI apresenta as conclusões.

ALGORITMO PARA O NON-LOCAL MEANS

Primeiro, é introduzido o algoritmo NLM original. E, em seguida, é apresentada a versão que utiliza janelas de busca [2]. Uma análise sobre a complexidade computacional de cada versão é feita no final.

A. Non-local means[2]

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,

$$NL[v][i] = \sum_{j \in I} w(i,j)\, v(j), \qquad (1)$$

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 de intensidade de nível de cinza $v(V_i)$ e $v(V_j)$, onde V_k denota uma vizinhança quadrada de tamanho fixo (MxM) centralizada no pixel k. Esta similaridade é mensurada através da distância Euclidiana

ponderada $||v(V_i) - v(V_j)||^2_{2,a}$, onde $a > 0$ é o desvio padrão do kernel Gaussiano[3].

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

$$w(i,j) = \frac{1}{Z(i)} e^{-\frac{||v(V_i)-v(V_j)||^2_{2,a}}{h^2}}, \quad (2)$$

onde $Z(i)$ é a constante de normalização

$$Z(i) = \sum_j e^{-\frac{||v(V_i)-v(V_j)||^2_{2,a}}{h^2}}, \quad (3)$$

e o parâmetro h atua como um fator de filtragem, controlando o decaimento da função exponencial.

O NLM não compara apenas o nível de cinza em apenas um ponto, mas sim a configuração geométrica em uma vizinhança inteira, o que o faz mais robusto que outros filtros de ruído.

B. NLM em janela e complexidade

Admitindo NxN como o número de pixels da imagem, e utilizando vizinhanças de dimensão MxM, a complexidade do algoritmo é M^2xN^4. Entretanto, imagens naturais tendem a ter grandes áreas uniformes, com variância geralmente pequena entre pixels próximos. Esta conclusão pode ser alcançada, também, analisando os resultados em [4], que até mesmo reportam melhor desempenho em áreas uniformes quando comparados com o algoritmo NLM original. Restringindo a busca por pixels semelhantes a uma janela de WxW pixels e a complexidade final do algoritmo passa a ser $M^2xW^2xN^2$.

ACELERAÇÃO DO NLM EM FPGA

Utilizando o valor 7 para M e 21 para W (conforme sugestão dos criadores do NLM para filtragem de imagens em níveis de cinza [2]), o tempo de execução para o processamento em software não é reduzido a níveis razoáveis. Outra forma imediata de reduzir a complexidade seria diminuir o tamanho da vizinhança, ou seja, o valor de M (para 5, ou até mesmo 3). Muitas abordagens que usam desta prática geralmente provocam distorções na imagem filtrada por falta de informação para calcular os pesos [5].

Para tornar o algoritmo rápido o suficiente para permitir sua utilização em muitas aplicações, o presente trabalho propõe uma implementação em hardware, utilizando um dispositivo FPGA (Field-Programmable Gate Array), da versão em janela do NLM. O desenvolvimento em hardware possibilita utilizar pipelines, paralelização de instruções, assim como a construção de memórias especializadas que permitem buscar prontamente os dados necessários para efetuar os cálculos.

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.

A Fig. 1 mostra o fluxo da filtragem utilizando o NLM em hardware. Adiante, é detalhada cada fase do fluxo apresentado.

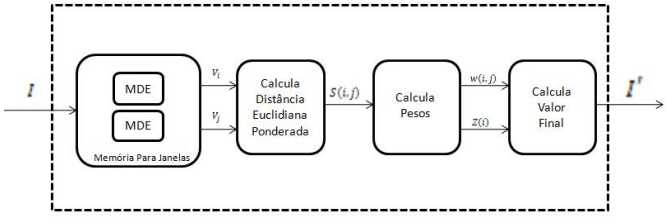


Figura 1. Fluxo da filtragem de ruídos para o NLM em FPGA.

C. Paralelização do cálculo da distância Euclidiana quadrática ponderada

Para melhorar o desempenho do NLM, foi elaborada uma forma de paralelizar o cálculo da distância Euclidiana ponderada.

Nessa abordagem, são usadas vizinhanças com $7x7$ pixels ($M = 7$), implicando 49 operações para calcular a distância Euclidiana ponderada. Analisando este cálculo, percebe-se que as 49 operações são independentes entre si, possibilitando realizá-las em paralelo. Além disto, cada operação pode ser separada em uma sequência de etapas para implementação em pipeline. Isto permite que, após o devido preenchimento do pipeline, seja realizado um cálculo de distância Euclidiana ponderada por ciclo de relógio

D. Memória dinâmica especializada (MDE) para vizinhanças

Após a construção do componente responsável por efetuar o cálculo da distância Euclidiana ponderada, foi observado que o mesmo não seria efetivamente utilizado caso não houvesse informações sobre os 49 pixels a cada ciclo de relógio. Desta forma, a execução continuaria sequencial, pois seria preciso aguardar os dados vindos da memória (sequencial) para, só então, operar. Uma memória especializada com suporte para acesso simultâneo a vários endereços mostrou-se indispensável.

Uma alternativa seria replicar a janela em cálculo por 25 memórias (supondo-as capazes de permitirem duas leituras por ciclo de relógio). Entretanto, a Fig. 2 permite ver a maneira como são acessadas as posições de memória ao passar 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. Concluí-se então que, na passagem de uma vizinhança para outra, apenas sete dos 49 pixels são realmente novos (os demais 42 precisam apenas ser realocados). Esta observação viabilizou a construção de uma solução mais eficiente.

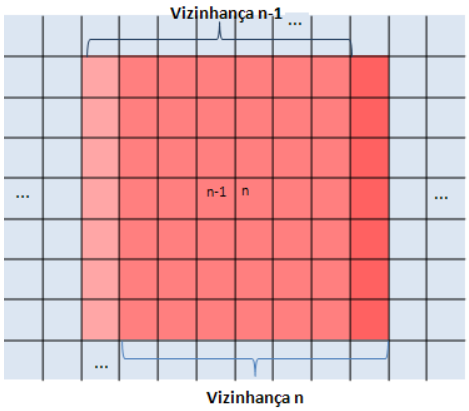


Figura 2. Duas vizinahanças consecutivas: vizinhança *n-1*, centrada no pixel *n-1;* vizinhança *n*, centrada no pixel *n*.

O modo como os pixels são trocados na passagem de uma vizinhança para outra pode ser observado na Fig. 3: setas azuis, os pixels que são trocados entre registradores; setas vermelhas, os que são descartados; e, finalmente, setas verdes, posições que recebem sete novos pixels relativos à nova vizinhança. Fica claro que apenas sete buscas simultâneas são realmente necessárias, desde que sejam trocados de forma ágil os demais 42 pixels da vizinhança anterior, garantindo que as informações necessárias para calcular a distância Euclidiana ponderada estarão sempre disponíveis a tempo. Para isto, foi desenvolvida uma memória dinâmica especializada (MDE) para permitir o acesso simultâneo às sete posições de memória. Foi suficiente replicar a janela em cálculo por apenas quatro memórias. Visto que cada uma pode ter dois valores lidos (ou escritos) ao mesmo tempo, é possível ler as sete posições simultaneamente.

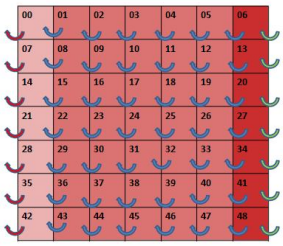


Figura 3. Troca de pixels de uma vizinahança para a seguinte.

E. Memória para janelas

A versão do NLM proposta no presente trabalho utiliza janelas de $21x21$ (ou 441) pixels. Para que todos os pixels possuam vizinhanças $7x7$ válidas, mesmo nas bordas da janela, foram usadas janelas estendidas, com *27x27* (ou 729) pixels. Para filtrar um pixel são necessárias uma vizinhança (49 pixels) e uma janela estendida (729 pixels), ambas centralizadas neste pixel. A entrada 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 a ser processado. Então, um total de 441 ciclos são gastos nesta etapa. Durante o cálculo dos pesos, a memória sendo utilizada não deve ser alterada. Para melhorar o tempo de execução, foi acrescentada outra memória dinâmica especializada (MDE), para que, enquanto uma estiver sendo utilizada para calcular os pesos a outra possa ser preenchida, a fim de que sempre haja dados disponíveis para calcular os pesos, mantendo o pipeline preenchido.

Uma vez que são gastos 441 ciclos calculando pesos, e são necessários apenas 389 ciclos para armazenar os dados necessários na memória, conclui-se que há tempo suficiente para preencher a memória extra, garantindo que sempre haverá dados para preencher o pipeline.

F. Aproximação da função exponencial

Como pode ser observado nas Equações (2) e (3) o NLM faz uso intensivo do cálculo de exponenciais. Entretanto, muitos processadores não possuem uma unidade de exponenciação em hardware. A operação é implementada em software sendo realizada por meio da combinação de tabelas de busca, multiplicações de ponto flutuante e adições. Esses algoritmos são muito lentos quando comparados com uma implementação em hardware [6].

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. Dado um valor de h^2 (Equações 2 e 3), é possível usar algumas retas, como mostrado em [7], para aproximar o decaimento exponencial, ver Fig. 4.

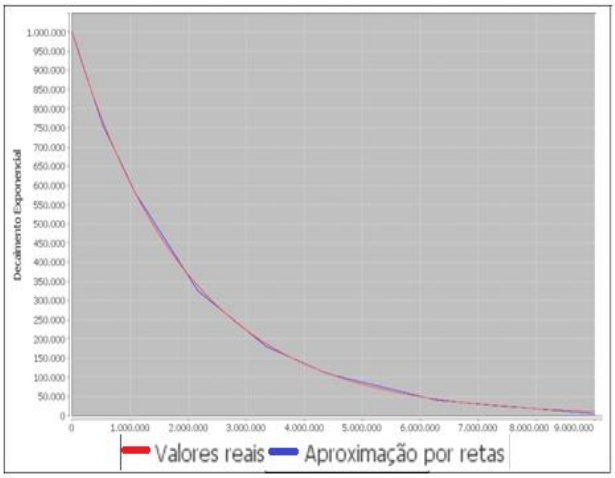


Figura 4. Aproximação do decaimento para h^2=200 usando dez retas.

Os pontos de uma reta podem ser calculados por

$$y = mx + b, \qquad (6)$$

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 [8,9].

G. *Cálculo do valor final do pixels*

Obtidos os pesos relativos a cada pixel, resta calcular o valor final do pixel. De acordo com (1), basta calcular a média ponderada de todos os pixels da janela, o que é feito novamente com uso de pipelines.

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 .

H. *Simulação*

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 Fig. 5.

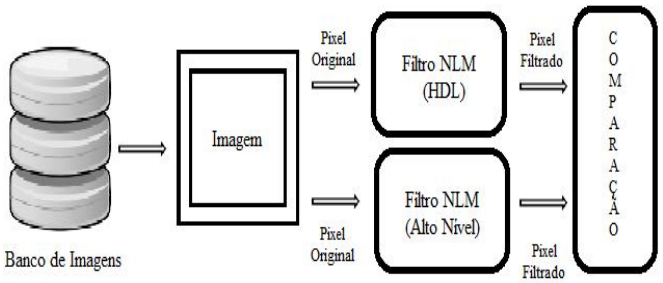


Figura 5. Metodologia de simulação.

RESULTADOS

Nesta sessão são apresentados os resultados de síntese em FPGA, assim como os resultados e as comparações do tempo de execução e da qualidade entre a versão em FPGA e a versão em software do NLM em janela.

I. *Área e frequência*

Para o dispositivo EP3SL50F484C2 da família *Stratix III* da *Altera*, o 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*).

J. *Qualidade da filtragem*

O filtro proposto em hardware obteve resultados comparáveis ao NLM em software em termos do erro quadrático médio MSE. Observando a Fig. 6 pode-se comparar visualmente as versões do NLM em janela em software e em FPGA.

Figura 6. Da esquerda para a direita e de cima para baixo: imagem original; imagem com adição de ruído; resultado do NLM em software (MSE 28,36) e NLM em hardware (MSE 28,10).

A Tabela I e a Fig. 7 mostram uma comparação entre o algoritmo NLM em FPGA e o NLM em software com relação ao MSE, para ruídos com diferentes desvios padrão. Note que a diferença se deve ao uso de retas para aproximar o decaimento exponencial, e aproximação de números fracionários por inteiros.

TABELA I. COMPARAÇÕES DE MSE

Método utilizado	**Desvio Padrão**			
	5	***10***	***15***	***20***
NLM em software	8,91	24,17	40,56	113,61
NML em FPGA	8,89	24,68	41,42	113,71

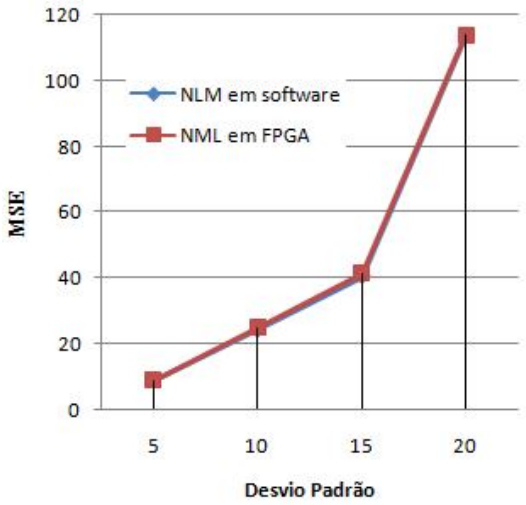


Figura 7. Gráfico relativo à Tabela I.

K. *Performance em tempo*

Conforme sugerido por Buades [2], foi utilizado M=7, e uma janela de pesquisa de $21x21$ pixels. Como as operações relativas ao cálculo da distância Euclidiana ponderada são efetuadas em hardware paralelo.

São gastos 441 ciclos calculando pesos para filtrar um pixel e todas as outras operações são realizadas em pipeline. Essa informação juntamente com período de relógio de T (9,45ns obtidos na simulação) permite inferir o tempo despendido para filtrar uma imagem. Para uma imagem com N^2 pixels são gastos $441xN^2xT$.

Na Tabela II, experimentos em um PC com um processador Core I5 2.53GHz e 6GB de RAM demonstraram que o NLM em software gasta aproximadamente 12 minutes para filtrar uma imagem de $1024x1024$ pixels, enquanto que a versão em FPGA aqui proposta gasta 22 segundos aproximadamente para filtrar a mesma imagem.

Comparado ao algoritmo original, o resultado do trabalho proposto é aproximadamente 170 vezes mais rápido, conforme demonstrado na Tabela II e na Fig. 8.

TABELA II. RESULTADOS DE DESEMPENHO

Tamanho da imagem	**Método utilizado**		
	NLM em software	***NLM em FPGA***	***Razão***
512*512	198,32s	1,10s	180,3
1024*1024	740,12s	4,41s	167,8
2592*1944	3740,50s	21,2s	176,4

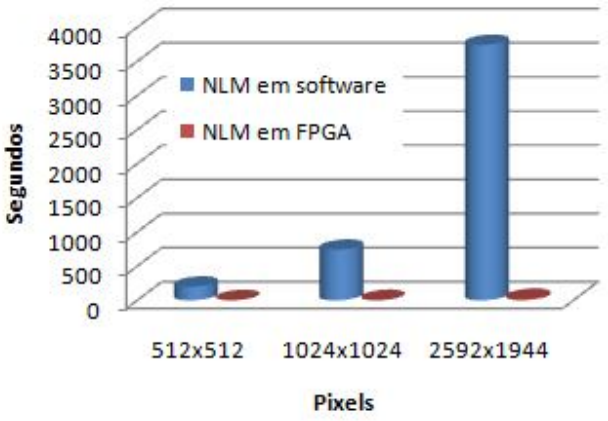


Figura 8. Gráfico relativo a Tabela II

CONCLUSÃO

Este trabalho propõe uma implementação para o algoritmo NLM em janela usando um hardware de propósito especifico. Os resultados obtidos na simulação mostram que o algoritmo é executado em média 170 vezes mais rápido que na implementação em software, e os resultados da redução de ruídos são semelhantes tanto em MSE quanto em percepção visual.

REFERÊNCIAS

[1] P. Coupé, P. Yger, C. Barillot., Fast Non Local Means Denoising for 3D MR Images: SpringerLink, 2006.

[2] A. Buades, B. Coll, and J Morel. A non-local algorithm for image denoising. IEEE International Conference on Computer Vision and Pattern Recognition, 2005.

[3] Sítio http://www.stat.wisc.edu/~mchung/teaching/MIA/reading/diffusion.gaussian.kernel.pdf.pdf

[4] M. Mahmoudi, G. Sapiro, "Fast image and video denoising via nonlocal means of similar neighborhoods". Signal Processing Letters, 12(12):839.842, 2005.

[5] N. SHAHAM, Métodos para aceleração do "non---local means" algoritmo de redução de ruído - Dissertação (Mestrado em Informática). 1.ed. Rio de Janeiro: Pontifícia Universidade Católica do Rio de Janeiro, v.I, 2007.

[6] R. Pottathuparambil, R. Sass, "Implementation of a CORDIC-based Double Precision Exponential Core on an FPGA", Proceedings of the Fourth Annual Reconfigurable Systems Summer Institute (RSSI '08), Urbana, Illinois, USA, July 7-9, 2008.

[7] R. Bellman, R. Roth, "Curve Fitting by Segmented Straight Lines", Journal of the American Statistical Association Vol. 64, No. 327 (Sep., 1969), pp. 1079-1084.

[8] A. Panato, S. Silva, F. Wagner, M. Johann,R. Reis, S. Bampi, "Design of very deep pipelined multipliers for FPGAs," Design, Automation and Test in Europe Conference and Exhibition, 2004.

[9] N. Takagi, S. Kadowaki, K. Takagi, "A hardware algorithm for integer division," Computer Arithmetic, 2005. ARITH-17 2005. 17th IEEE Symposium on , vol., no., pp. 140- 146, 27-29 June 2005.