Co-projeto hardware e software para o cálculo de fluxo ótico

Tiago Mendonça Lobo

Co-projeto hardware e software para o cálculo de fluxo ótico

Tiago Mendonça Lobo

Orientador: **Prof. Dr. Eduardo Marques**

Dissertação apresentada ao Instituto de Ciências Matemáticas e de Computação - ICMC-USP, como parte dos requisitos para obtenção do título de Mestre em Ciências - Ciências de Computação e Matemática Computacional. *VERSÃO REVISADA*

USP – São Carlos
Agosto de 2013

Ficha catalográfica elaborada pela Biblioteca Prof. Achille Bassi
e Seção Técnica de Informática, ICMC/USP,
com os dados fornecidos pelo(a) autor(a)

Lobo, Tiago
L799c Co-projeto hardware e software para o cálculo de fluxo ótico / Tiago Lobo; orientador Eduardo Marques. -- São Carlos, 2013.
72 p.

Tese (Doutorado - Programa de Pós-Graduação em Ciências de Computação e Matemática Computacional) -- Instituto de Ciências Matemáticas e de Computação, Universidade de São Paulo, 2013.

1. Fluxo ótico. 2. FPGA. 3. Processamento de Imagem. 4. Visão Computacional. I. Marques, Eduardo, orient. II. Título.

"Só existem dois dias da sua vida que não se pode fazer nada por voce: O Ontem e o Amanhã"

Dalai Lama

Dedico este trabalho

Aos meus queridos pais Antonio Veira e Maria da Conceição, que sempre me incentivam e apoiam em todas as fases de minha vida.

E aos meus avós Antônio Mendonça e Raimundo Lobo aos quais eu devo, respectivamente, grande parte da minha criação e o conceito de homem de caráter impecável, que Deus os tenha em boas mãos.

Agradecimentos

Em primeiro lugar gostaria de agradecer aos meus pais e meu irmão pelo o apoio incondicional que eu sempre recebi deles. Obrigado "Madre", "Macho Réi"e "Joey".

Em segundo lugar, agradecer meus amigos que sempre me deram apoio, conversando ou discutindo, para seguir em frente.

Um agradecimento especial ao meu orientador Prof. Dr. Eduardo Marques que me guiou para a realização deste trabalho; sem sua orientação a realização deste trabalho teria sido mais difícil.

Obrigado ao Prof. Dr. João Cardoso por ter me proporcionado a oportunidade única da participação como investigador no projeto REFLECT na Faculdade de Engenharia do Porto em Portugal.

Agradecer a todos os integrantes do laboratório LCR que me apoiaram nessa jornada seja conversando ou em um café no final de tarde. Muito obrigado Leandro, Jecel, Hell, Marci, Antonio, Bruno, Cris, Valéria, Sergiones, Perina, Jean, Eva e Erinaldo.

Agradecimentos as instituições de fomento CAPES e FAPESP pelo auxílio durante a realização deste trabalho.

Gostaria também de agradecer a Universidade de São Paulo, em especial o Instituto de Ciências Matemáticas e de Computação, por ter me proporcionado toda infra-estrutura e a base teórica necessária à realização deste trabalho.

Resumo

O cálculo dos vetores de movimento é utilizado em vários processos na área de visão computacional. Problemas como estabelecer rotas de colisão e movimentação da câmera (egomotion) utilizam os vetores como entrada de algoritmos complexos e que demandam muitos recursos computacionais e consequentemente um consumo maior de energia. O fluxo ótico é uma aproximação do campo gerado pelos vetores de movimento. Porém, para aplicações móveis e de baixo consumo de energia se torna inviável o uso de computadores de uso geral. Um sistema embarcado é definido como um computador desenvolvido com um propósito específico referente à aplicação na qual está inserido. O objetivo principal deste trabalho foi elaborar um módulo em sistema embarcado que realiza o cálculo do fluxo ótico. Foi elaborado um co-projeto de hardware e software dedicado e implementados em FPGAs Cyclone II e Stratix IV para a prototipação do sistema. Desta forma, a implementação de um projeto que auxilia a detecção e medição do movimento é importante não só como aplicação isolada, mas para servir de base no desenvolvimento de outras aplicações como tracking, compressão de vídeos, predição de colisão, etc.

Palavras-chave: Fluxo Ótico, FPGA, Nios II, multiprocessamento

Abstract

The motion vectors calculation is used in many processes in the area of computer vision. Problems such as establishing collision routes and the movement of the camera (egomotion) use this vectors as input for complexes algorithms that require many computational and energy resources. The optical flow is an approximation of the field generated by the motion vectors. However, for mobile, low power consumption applications becomes infeasible to use general-purpose computers. An embedded system is defined as a computer designed with a specific purpose related to the application in which it is inserted. The main objective of this work is to implement a hardware and software co-design to assist the optical flow field calculation using the CycloneII and Stratix IV FPGAs. Sad that, it is easily to see that the implementation of a project to help the detection and measurement of the movement can be the base to the development of others applications like tracking, video compression and collision detection.

Keywords: Optical Flow, FPGA, Nios II, multiprocessing

Lista de Figuras

Lista de Tabelas

Sumário

Capítulo 1

Introdução

Sendo a visão um dos mais importantes sentidos do ser humano, o processamento de imagens e a visão computacional são importantes áreas da computação no que diz respeito a retirar informações do meio através de imagens. As imagens podem ser processadas para ressaltar seu conteúdo ou para extrair informações importantes.[15].

Atualmente, existe um número grande de aplicações que utilizam visão computacional como forma de perceber o ambiente e inferir informações sobre o mesmo[16, 17, 9].

Em aplicações como detecção de obstáculos [17] e seguimento objetos em movimento (*object tracking*) é necessário à priori o cálculo do fluxo ótico das imagens[16]. O fluxo ótico é utilizado por estes métodos como uma aproximação do campo de vetores de movimento. Um dos primeiros estudos sobre vetores de movimento foi descrito por Horn e Schunck em 1981 [18].

1.1 Motivação

O custo computacional dos algoritmos de visão computacional é muito alto, o que gera a necessidade de sistemas de computação de uso geral mais poderosos. Sua utilização se torna um problema quando inseridos em aplicações com restrições de custo e tamanho[9].

Diferente dos computadores de uso geral, um sistema embarcado é desenvolvido com um propósito específico referente à sua aplicação [19]. Normalmente, possuem recursos mais limitados do que os computadores de uso geral, porém podem apresentar vantagens no que diz respeito ao consumo de energia, tamanho e tempo de execução[19].

Com o crescimento da capacidade de transistores que cabe em um único *chip* dobrar a cada 18 anos aumentou a complexidade de suas implementações. Devido a grande procura por sistemas cada vez mais adequados as novas tecnologias a

mudança de produtos e soluções no mercado são constantes. Essa rápida mudança em projetos já vigentes gera uma necessidade de flexibilidade nos mesmos que pode ser atingida através da programação de microcontroladores ou microprocessadores. Porém, o uso destes limita quantidade de entradas e saídas que podem ser utilizadas.

Novas tecnologias que visam a computação reconfigurável, como os FPGA´s (*Field Programable Gate Arrays*), têm marcado o desenvolvimento de *hardware* nos últimos anos [20]. O uso de FPGA´s possibilita que projetos inteiros de sistemas de hardware sejam prototipados em uma única placa. Por ser reprogramável a FPGA possibilita que erros de projeto sejam identificados e corrigidos em tempo menor do que se o sistema tivesse sido implementado em ASIC.

Outra de suas utilizações é a implementação de processadores *soft-core*. Tais processadores são escritos em linguagem de alto nível para a descrição de *hardware*, como exemplo destas linguagens tem-se VHDL (VHSIC *Hardware Description Language*), Verilog, SystemVerilog, Bluespec, dentre outras [20]. O uso de tal tipo de processador tem como vantagem a possibilidade de customização de instruções. Nesta linha tem-se o processador Nios II [21], um processador baseado na arquitetura RISC (*Reduced Instruction Set Computer*).

Com a customização de instruções dedicadas ao projeto, combinada ao uso de *hardware* dedicado, sequências de instruções complexas podem ser reduzidas à uma única instrução melhorando o desempenho do algoritmo em *software*[21].

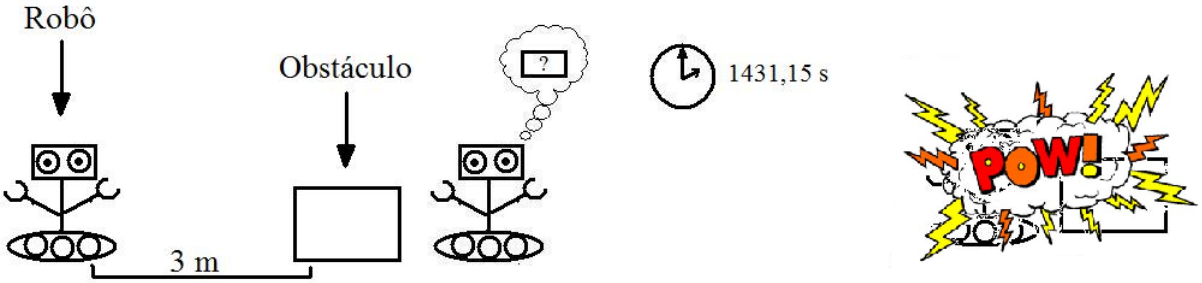
O crescimento do número de transistores por área de *chip* devido a Lei de Moore fez com que o desenvolvimento de *hardware* tenha atingido barreiras físicas[20, 22]. O aumento do número de transistores disponíveis tem feito com que os projetistas explorem cada vez mais o paralelismo do *software* implementando, por exemplo, arquiteturas superescalares [20]. Embora o uso de tais arquiteturas aumente a capacidade computacional, o custo (principalmente no que diz respeito ao consumo) ainda permanece elevado. Tal fato tem influencia no crescimento da computação paralela[20].

Embora o uso de multiprocessadores explore o paralelismo inerente a algumas aplicações, a implementação pura de hardware pode explorar ainda mais o potencial paralelo. O uso de FPGA´s , por facilitar a implementação e prototipação de hardware, cria um ambiente propício para o desenvolvimento de um co-projeto hardware e software onde pode-se explorar as vantagens dos dois tipos de abordagem.

1.2 Objetivo

Desta forma, este trabalho teve como objetivo criar um co-projeto de hardware e software para o auxílio no cálculo do fluxo ótico utilizando-se FPGA´s. Para tal, à princípio implementou-se uma arquitetura multiprocessada prototipada em uma FPGA Cyclone II para inversões de matrizes. Porém, com o desenvolvimento do projeto e a aquisição de uma placa mais moderna, decidiu-se terminar o Co-projeto hardware e software utilizando a FPGA Stratix IV e com o sistema feito como um bloco de hardware.

Como a robótica móvel é uma aplicação típica de sistemas embarcados, para a avaliação dos tempos de execução deste projeto foi levado em consideração a distância percorrida pelo robô Pioneer 3 dx (que se move na velocidade de 1,6 m/s) durante o tempo de processamento do algoritmo de fluxo ótico. Desta forma, além de avaliar o processamento, o projeto também é avaliado no contexto de uma aplicação de tempo real. Na Figura 1.1 ilustra-se uma aplicação de detecção de obstáculos que utiliza fluxo ótico e que tem um tempo de processamento insatisfatório de 1.431,15 segundos, calculados pelo trabalho desenvolvido em [23]. No trabalho [23] foi implementado o fluxo ótico totalmente em software, sendo o algoritmo executado 100% no processador *soft-core* NiosII. Obviamente, o resultado não atende as características de tempo real do robô, fazendo com que o mesmo colida com o objeto, pois com esse tempo de processamento o robô percorre 2 km até detectar o objeto em sua trajetória. Este resultado motivou o desenvolvimento deste mestrado, visando implementar uma nova versão do fluxo ótico, adotando-se a filosofia de co-projeto de hardware e software.



(a) na configuração inicial: o robô com um obstáculo à 3 m.

(b) 1431,15 s para processar enquanto o robô continua percorrendo 2 km.

(c) resultado final: colisão.

Figura 1.1: Importância da distância percorrida.

1.3 Justificativa

O crescimento da necessidade cada vez maior de sistemas embarcados, em diversos lugares a todo momento, gera a necessidade de aplicações antes pensadas apenas para computadores de grande porte se adaptem para restrições de energia,tamanho, peso, etc.

A importância da visão computacional cresce cada vez mais devido ao aumento da inteligência artificial de aparelhos e na robótica móvel. A necessidade de aferir informações de vídeos e imagens é cada vez maior. O movimento é uma das informações mais importantes adquiridas através da visão.

Desta forma, é possível perceber que a implementação de um projeto que venha auxiliar na detecção e medição do movimento é importante não só como informação em si, mas como servir de base para o desenvolvimento de outras aplicações como tracking, compressão de vídeos, predição de colisão, etc.

1.4 Delimitação do trabalho

Embora este trabalho trate do assunto de visão computacional, o mesmo não está integrado com a câmera CMOS presente na placa de FPGA. O trabalho utiliza como dados de entrada duas imagens previamente guardadas em uma memória interna no FPGA. Sendo assim, não considera a comunicação com memórias externas ao *chip* da FPGA (RAM´s, SRAM´s, FLASH, etc) e nem com a câmera .

1.5 Organização do trabalho

Este trabalho está organizado em seis capítulos. O primeiro inicia com uma pequena contextualização sobre visão computacional e uma justificativa do porquê foi escolhida a implementação em FPGA do sistema.

No segundo capítulo consta a revisão bibliográfica como forma de mostrar conceitos tanto na área de processamento de imagem e visão computacional, como de computação reconfigurável e co-projeto de hardware e software. Finalmente, é apresentado alguns trabalhos relacionados a este.

No terceiro capítulo são explicadas ferramentas e técnicas utilizadas no decorrer do desenvolvimento do trabalho. Embora inicialmente expostas de forma mais superficial, no fim do capítulo é realizado detalhamento de como foram utilizadas em conjunto.

No quarto capítulo são descritas todas as abordagens utilizadas para a realização do presente trabalho. Inicializando com a abordagem multiprocessada, que

posteriormente foi substituída pela implementação de uma instrução customizada e sendo finalizada pelo desenvolvimento de um bloco puramente em *hardware*.

No quinto capítulo são expostas as conclusões finais do trabalho levando em consideração as dificuldades e resultados obtidos. Neste capítulo também constam algumas sugestões para trabalhos futuros.

Por fim no último capítulo constam as referências que embasam os conceitos explicitados na revisão bibliográfica.

Capítulo 2

Revisão Bibliográfica

2.1 Aquisição e Digitalização de Imagens

Nesta seção serão apresentados técnicas e conceitos referentes à aquisição e digitalização de imagem, com o objetivo de contextualizar as técnicas de processamento descritas nos capítulos posteriores. Além de explicitar tais tópicos faz-se necessário a explicitação de alguns conceitos e convenções sobre imagens monocromáticas. Uma imagem monocromática pode ser descrita como uma matriz de tons de cinza onde cada coordenada (x, y) tem um valor determinado por uma função $f(x, y)$ que determina um valor numérico proporcional a intensidade luminosa do ponto na imagem. Para esse trabalho será adotada a convenção dos eixos ilustrada na Figura 2.1, note que a orientação dos mesmos é diferente dos adotados na geometria analítica.

Figura 2.1: Convenção utilizada para os eixos x e y da imagem digital

A função $f(x, y)$ citada anteriormente é, na realidade, o produto entre a quan-

tidade de luz que incide sobre o objeto (iluminância), dado por $i(x, y)$, e as propriedades de refletância do objeto, que podem ser descritas pela função $r(x, y)$ cujo valor representa a fração da luz que será refletida pelo objeto no ponto (x, y). Levando em conta essa afirmação tem-se $f(x, y)$ como mostra a Fórmula 2.1.

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

onde

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

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

Para explicitar melhor os conceitos de luminância e refletância seguem as Tabelas 2.1 e 2.2 respectivamente mostrando alguns valores dos mesmos em algumas situações do cotidiano.

i(x,y)	Situação
900	dia ensolarado
100	dia nublado
10	iluminação média de escritório
0.001	noite de lua cheia

Tabela 2.1: Valores de i(x,y) [em lux ou $^{\text{lúmen}}/_{\text{m}^2}$][14]

r(x,y)	Situação
0.80	parede fosca branca
0.65	aço inoxidável
0.01	veludo preto

Tabela 2.2: Valores para r(x,y)[14]

Como este trabalho lida com o universo digital faz-se necessária a delimitação do valor de intensidade determinado pela função $f(x, y)$, que será chamada de tom ou nível de cinza da imagem monocromática no ponto (x, y) e terá apenas valores inteiros e positivos. Desta maneira temos:

$$Min < f(x, y) < Max \tag{2.4}$$

É comum no intervalo $[Min, Max]$ atribuir-se o valor zero à Min e um valor positivo potência de dois a Max. Desta forma, tem-se frequentemente o zero representando o preto e o valor $Max - 1$ representando o branco. Isso proporciona

ao intervalo $[Min, Max]$ a capacidade representar toda a escala de cinza de uma imagem. Caso a imagem possua intervalos de frequência distintos faz-se necessário a existência de uma função $f(x, y)$ para cada componente de frequência da imagem. Como exemplo desta condição tem-se as imagens coloridas no padrão RGB onde cada ponto (x, y) é formado pelo composição das três cores primárias aditivas o vermelho (*Red*), verde (*Green*) e azul (*Blue*).

Nas proximas seções serão consideradas apenas imagens monocromáticas. Porém, mesmo assim, o conceito da representação RGB ainda é muito importante principalmente quando se trata da manipulação de imagens digitais.

2.1.1 Aquisição

O processo de aquisição de imagens pode ser visto como a transformação de uma cena tridimensional em uma imagem bidimensional. Em visão computacional, considera-se que esse processo é feito por uma câmera de vídeo de forma, automatizada. As duas principais tecnologias para a aquisição de vídeo em visão computacional são os sencores CCD (*Charge Coupled Device*) e CMOS.

O sensor CCD pode ser descrito como um arranjo fixo de fontes de potencial distribuídas por uma superfície retangular ou quadrada onde cada fonte acumula uma carga elétrica proporcional à intensidade de luz incidente [1]. Desta forma, para obter uma imagem expõe-se o sensor à cena criando uma imagem bidimensional através de vários potenciais elétricos no arranjo citado anteriormente. O sinal analógico produzido pelo sensor passa por circuitos eletrônicos para produzir na saída um Sinal Composto de Vídeo analógico e monocromático.

Na Figura 2.2 pode-se acompanhar todo o processo de aquisição em um sensor CCD.

Como sucessores dos sensores CCD, existem os sensores CMOS (*Complementary Metal Oxide Semiconductor*). Este usa uma forma diferente de fabricação e promete uma série de vantagens se comparado ao CCD. A mais importante delas é o custo, uma vez que utiliza uma forma de manufatura equivalente à de *chips* como memórias para computador, enquanto o CCD necessita de linhas de fabricação separadas [3]. Também é possível adicionar circuitos adicionais no *chip* do sensor CMOS, o que possibilita a criação de câmeras com *chip* único. Isso leva a fabricação de sistemas mais comercialmente viáveis, apesar de ter procedimentos de desenvolvimento mais custosos. Ele possibilita também a criação de sistemas mais compactos, robustos e de baixo consumo de energia sendo ideal para dispositivos portáteis.

Outra vantagem é poder ler parte do vetor da imagem ao invés de todo ele; isso se deve ao endereçamento linha-coluna (*row-column addressing*) que possibilita ter

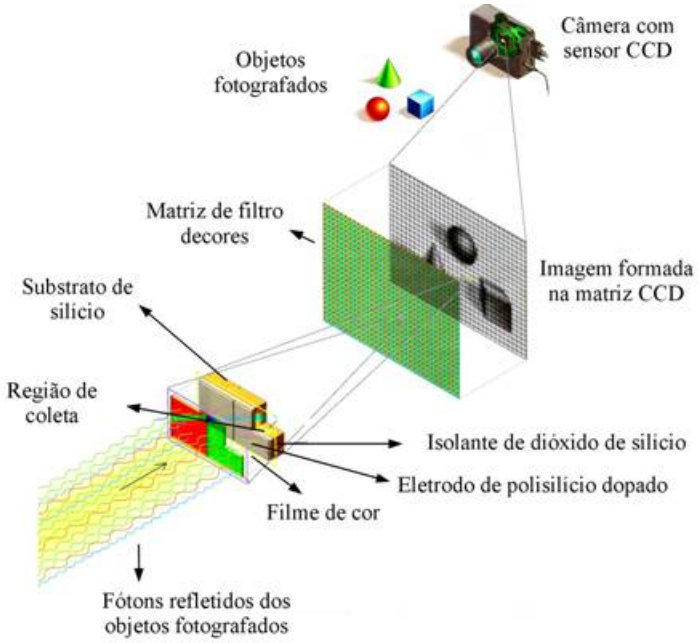


Figura 2.2: Aquisição CCD[1]

acesso direto a qualquer pixel[3].

Por outro lado, sensores CMOS apresentam muito mais ruído que os CCD. Parte por causa do fato de que o sensor CCD ter menos circuitos no *chip* e usar um amplificador de saída comum (*commom output amplifier*) para diminuir a variabilidade. Desta forma, os sensores CCD ainda tem maior sensibilidade, menor ruído, etc [3].

Logo, os sensores CMOS são mais recomendados em aplicações comerciais e o CCD em aplicações mais técnicas e científicas devido sua maior resolução e menor taxa de ruído [3].

2.1.2 Digitalização

Do ponto de vista eletrônico a digitalização de uma imagem consiste, basicamente, em uma conversão analógico-digital. De forma que o valor lido pelo conversor A/D indica a intensidade, em níveis de cinza, correspondente a determinado *pixel*. Para tomar o formato desejável ao processamento computacional o sinal analógico proveniente do sensor precisa ser discretizado espacialmente e em amplitude. Dá-se o nome de "amostragem" ao processo de discretização espacial, e de "quantização" ao processo de discretização [14] . A amostragem converte a imagem analógica, proveniente da aquisição, na matriz A de dimensões M por N, onde cada posição recebe o valor da função de amostragem $f(x, y)$; cada posição dessa matriz recebe o nome de *pixel* ou elemento de imagem. Na Fórmula 2.5 tem-se a representação matemática da matriz A.

$$A = \begin{bmatrix} f(0,0) & f(0,1) & \cdots & f(0,N-1) \\ f(1,0) & f(1,1) & \cdots & f(1,N-1) \\ \vdots & \vdots & \ddots & \vdots \\ f(M-1,0) & f(M-1,1) & \cdots & f(M-1,N-1) \end{bmatrix} \tag{2.5}$$

Já na quantização a matriz A é percorrida, e a cada pixel é atribuído um valor inteiro na faixa que vai de zero até $2n-1$. Perceba que quanto maior o n maiores são os níveis de cinza na escala da imagem digitalizada. Do ponto de vista da matemática formal o processo de amostragem pode ser visto como sendo a divisão do plano xy em uma grade. As coordenadas do centro de cada elemento desta grade são descritas no produto cartesiano de $\mathbb{Z}^2$ (também descrito como o produto cartesiano $\mathbb{Z}\mathtt{x}\mathbb{Z}$), que é o conjunto de todos os pares ordenados (a, b) onde a e b pertencem a $\mathbb{Z}$ (conjunto os números inteiros). Desta forma, tem-se que a função $f(x, y)$ é uma imagem digital se, para qualquer x e y, existe o par ordenado (x, y) pertencente ao produto cartesiano $\mathbb{Z}\mathtt{x}\mathbb{Z}$ e f é uma função que representa o processo de quantização descrito anteriormente. Desta maneira, uma imagem digital passa a ser uma função bidimensional, onde tanto suas coordenadas quanto valores de amplitude são números inteiros [14].

Para especificar-se o processo de digitalização de uma imagem se faz necessário determinar a priore o valor de N, M, e n, visando principalmente a etapa de armazenamento.

Por se tratar de um problema de processamento de imagens o armazenamento é uma questão chave. Logo, se faz necessária. nesse momento, uma discussão mais aprofundada sobre como esta definição de imagem digital afeta o armazenamento da mesma no sistema, principalmente no que diz respeito ao tamanho em *bits* de um *pixel*.

2.1.2.1 O pixel

O *pixel* (abreviação de "*picture element*") é a menor unidade visível de uma imagem digital, possuindo tamanho fixo e formato específico .

A grande problemática envolvida é como definir o tamanho ideal do *pixel* para um dado sistema. O tamanho do mesmo tem relação direta com o armazenamento envolvido no processamento. Mesmo em câmeras comuns a quantidade de *pixels* geradas por imagem capturada é muito grande.

Na classificação de câmeras digitais é comum o uso do termo *megapixel* para definir qual a resolução máxima que tal câmera pode chegar. Cada *megapixel* é correspondente a 1.000.000 de *pixels*.

A medida em que se aumenta o tamanho do *pixel* e a resolução, como pode ser visto na Tabela 2.3, é fácil perceber que o problema do armazenamento das imagens torna se cada vez mais complexo.

Pixel (*bits*) / Resolução	**0.08**	0.8	1.3	2.8	5	8	10
1	0.08	0.8	1.3	2.8	5	8	10
2	0.16	1.6	2.6	5.6	10	16	20
4	0.32	3.2	5.2	11.2	20	32	40
8	0.64	6.4	10.4	22.4	40	64	80
16	1.28	12.8	20.8	44.8	60	128	160
24	**1.92**	19.2	31.2	67.2	120	192	240
32	2.56	25.6	41.6	89.6	120	256	320

Tabela 2.3: Tamanho em *Megabits* de imagens geradas levando em consideração o tamanho do *pixel* e a resolução [14].

Ao imaginar uma aplicação em tempo real, que capta imagens de um meio utilizando uma câmera de 5 *megapixels* , com o tamanho do *pixel* de 2 *Byte* (16 *bits*), tem-se 10 MB a cada imagem gerada. Considerando que a cada segundo capturam-se 30 imagens (30 *fps*, *frames* por segundo), terão que ser processados, no caso de uma aplicação de tempo real, 300 MB por segundo. Se a aplicação tratar de problemas mais complexos como rastreamento (*Tracking*) ou de detecção de movimento (*Motion Detection*) o volume de dados a serem processados aumenta. Logo, para tal aplicação seria necessária a implementação que permita alcançar uma velocidade maior no processamento e uma melhor exploração do paralelismo da aplicação.

Na Tabela 2.3 os valores em negrito são utilizados em dois trabalhos de suma importância para a realização deste, descritos em [24] e [17].

O primeiro trabalho demonstra uma estrutura robusta para processamento de imagens utilizando-se *hardware* reconfigurável, e o segundo descreve uma aplicação para desvio de obstáculos utilizando fluxo ótico. Ambos utilizam 3 *Bytes* para representação do *pixel*,sendo cada componente do RGB representada por um *Byte*. Para a resolução da imagem tem-se duas opções: uma de 640 por 480 (aproximadamente 0.31 *Megapixels*); e a outra de 320 por 240 (aproximadamente 0.08 *Megapixels*). Na tabela a segunda opção está em destaque, pois em [17] toda a implementação é feita em função desta opção.

Do ponto de vista do *hardware*, o tamanho do *pixel* também é um problema a ser considerado uma vez que, além da memória necessária, a manipulação através de barramentos é diretamente afetada. Neste nível de processamento a interpretação do *pixel* está também ligada diretamente ao sensor analógico no que diz respeito principalmente ao processo de quantização e de discretização espacial (número de *pixels*).

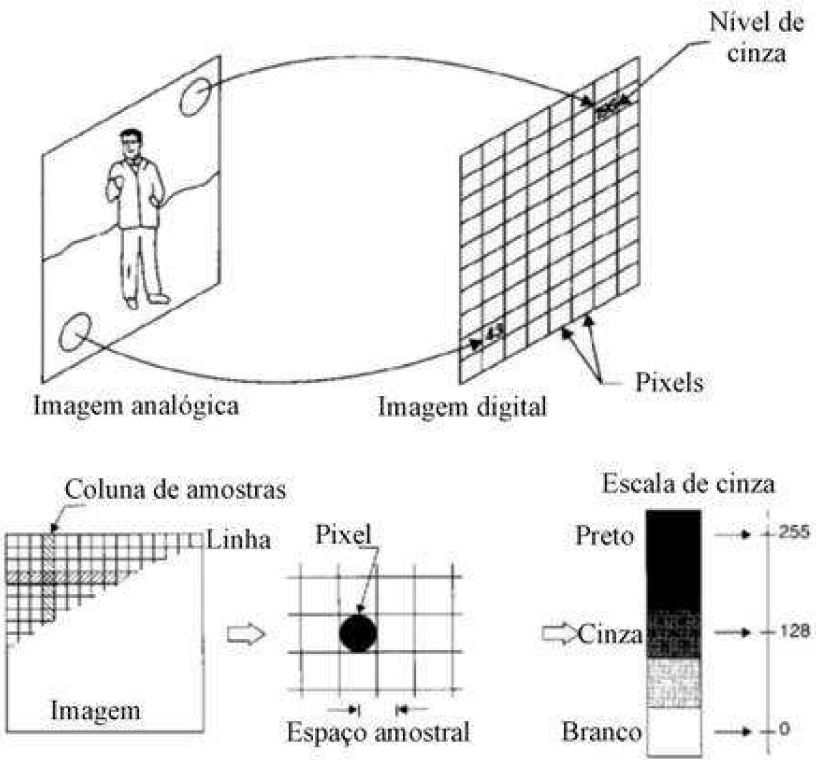


Figura 2.3: Esquema de digitalização[1]

Na Figura 2.3 tem- se representado um esquema que demonstra ambos os processos.

2.2 Estrutura de um Sistema de Visão Artificial

Para esta seção define-se um Sistema de Visão Artificial como sendo basicamente um sistema que interpreta imagens digitais que correspondem a cenas do mundo real. Para melhor entendermos a estrutura de tal sistema usaremos como exemplo uma aplicação que detecta a velocidade que está escrita em uma placa de trânsito. Desta forma, teremos o sistema estruturado como mostra a Figura 2.4. Note que à medida em que descemos na sequência de execução temos tipos de dados cada vez mais abstratos até que se gera um resultado.

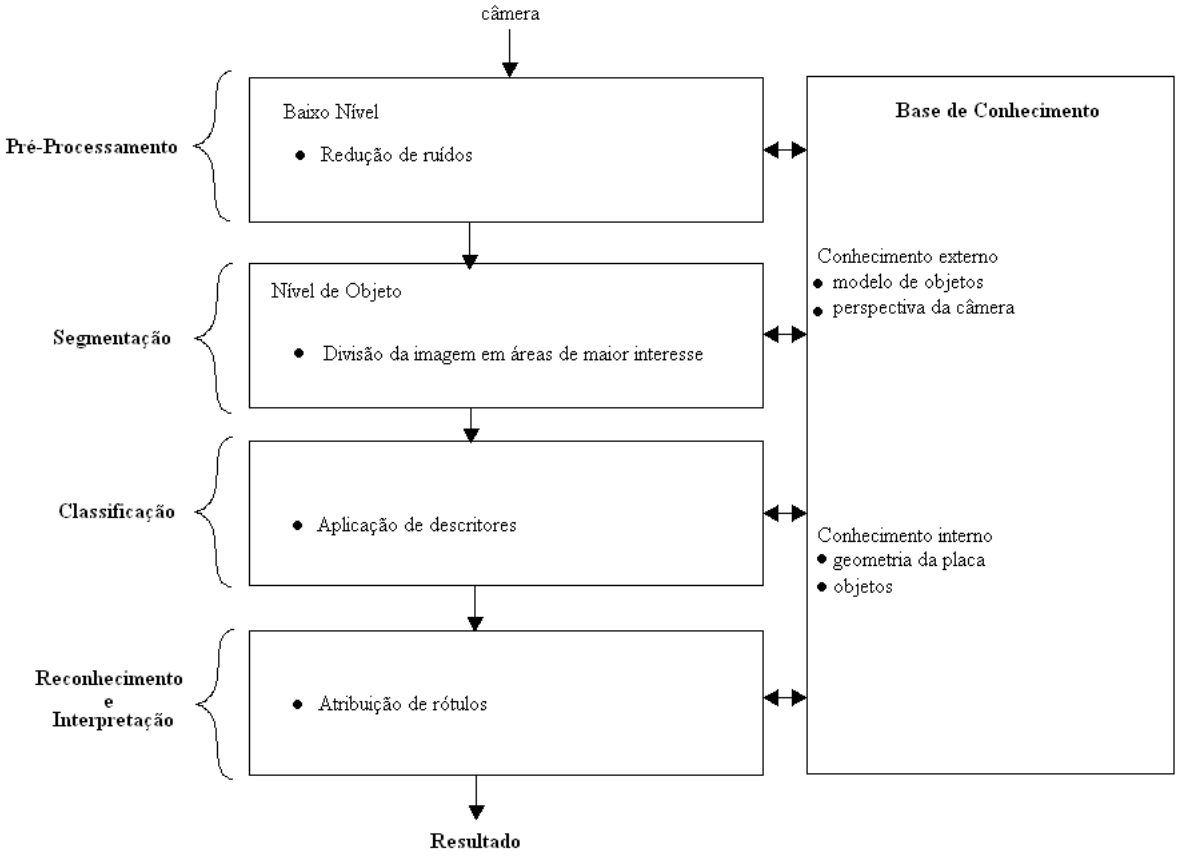


Figura 2.4: Estrutura de um Sistema de Visão Artificial

A título de exemplo pode-se considerar uma aplicação para seguir objetos em movimento em um vídeo identificando-os e guardando sua trajetória. Deste modo, ao fim do processamento é esperado um conjunto dos caminhos feitos pelos objetos.

A primeira etapa, após a captura, é a de Pré-processamento. Nesta fase a aplicação trata de problemas inerentes a captura utilizando redução de ruídos provenientes da captura, estabilização da imagem, ajustes de brilho e contrate dentre outros.

No processamento todas as fases visam fornecer dados mais refinados as próximas. Isso tem como objetivo dividir entre todas a complexidade da implementação e da execução.

Ainda na fase de Pré-processamento algumas informações da imagem podem ser perdidas ou omitidas. O excesso de informações em uma imagem, ou em um conjunto delas, nem sempre acrescenta um melhor processamento. O excesso de informação pode gerar falsos positivos como no caso da detecção de bordas em uma imagem que não foi suavizada adequadamente.

Na fase de Segmentação a imagem passa por um processo onde os valores à aplicação são identificados. No caso da aplicação exemplo é necessário, de antemão, identificar o movimento dos objetos na cena. Para o cálculo do movimento é necessário identificar quais pontos se moveram de uma imagem para a outra.

Porém, se o movimento for calculado para todos os pontos da imagem seriam verificados muitos falsos-verdadeiros. Desta forma, uma vez que a imagem já foi pré-processada, pode-se identificar nesta imagem pontos com a menor probabilidade de gerarem erros, e utilizar apenas estes para o cálculo do movimento na próxima fase.

A detecção de quais pontos se movimentaram e o cálculo do seu movimento, é realizada na fase de Classificação. Nesta fase é gerado um conjunto de dados numéricos que podem ser feitos através do uso de descritores. No caso da aplicação exemplo de rastreamento de objetos pode-se considerar que para a fase de Reconhecimento e Interpretação sejam encaminhados os pontos que se movimentaram na imagem e seus respectivos vetores de movimento.

Na fase de Reconhecimento e Interpretação os dados são transformados em informações pertinentes à aplicação. Para a extração de informações sobre o conjunto de dados proveniente da fase anterior pode-se utilizar técnicas de reconhecimento de padrões para identificar os objetos (conjunto de pontos em movimento) .

A Base de Conhecimento se comunica com todas as fases, pois é necessário entender quais serão os problemas em cada fase do processamento. Os sistemas de visão computacional tendem ser muito sensíveis ao meio no qual ele está inserido.

2.3 Técnicas de processamento de imagens

Existem diversas maneiras de se resolver um problema quando se trata de processamento de imagens. Neste sentido, pode-se pensar em várias opções para resolver os problemas encontrados utilizando uma vasta gama de técnicas existentes para executar ações como a extração de bordas da imagem ou a eliminação de ruídos. Nesta seção serão apresentadas técnicas de processamento de imagem focadas no domínio espacial devido sua maior versatilidade [2], utilizadas para filtragem e extração de características da imagem.

2.3.1 Eliminação de ruído

Muitas vezes, dependendo da forma de aquisição, uma imagem pode possuir algumas alterações. *Pixels* que, na realidade, não são propriamente desta são nomeados de ruído [14]. Para a redução do mesmo existem várias técnicas como o filtro de Gauss, da Mediana dentre outros. O nome "filtro" é proveniente do processamento no domínio de frequência onde se refere à aceitação (*passing*) ou rejeição de certos componentes da frequência na imagem [2]. Filtros que são utilizados como forma de suavizar e reduzir o ruído recebem o nome de filtros de Passa Baixa ou, no caso do domínio do espaço, *Smoothing Spatial Filters* [2]. A escolha dentre diferentes

técnicas depende do domínio da aplicação e do tipo do ruído [2]. Nessa seção serão explicitadas duas dessas técnicas que têm como objetivo a suavização da imagem visando à eliminação de ruído para, posteriormente, auxiliar na aplicação de outros filtros.

Smoothing Linear Filters

Antes de descrever o algoritmo no qual se baseia tal filtro, se faz necessária a explicação do que seria o processo de "convoluir uma máscara em uma imagem". Tal termo é frequentemente usado para denotar o processo de mover a máscara pela imagem executando soma de produtos (multiplicação de matrizes), e não sendo necessariamente uma convolução propriamente dita [2]. Um $SmoothingLinearFilter$ (Também chamado de filtro de Média) retorna a média dos *pixels* que estão na vizinhança da máscara do filtro. A idéia é bem simples, porém a intenção do filtro vai muito mais além [2]. Quando se substitui o valor para cada pixel na imagem pela média dos níveis de intensidade de seus vizinhos definidos pela máscara, gera-se uma imagem com transições de intensidade menos acentuadas.

Como o ruído aleatório normalmente consiste em mudanças bruscas de intensidade, a primeira vista uma aplicação direta do filtro de média é a eliminação de ruídos de uma imagem, mas também pode-se pensar que ele retira detalhes vistos como "insignificantes" para uma aplicação. Porém, as bordas (uma característica importante) também são caracterizadas por mudanças bruscas de intensidade, de tal forma que os filtros de Média têm esse "efeito colateral" indesejável de suavizar também as bordas [2].

Além do filtro de média nessa categoria também existe o filtro gaussiano que utiliza uma máscara baseada na distribuição gaussiana onde se pode variar o desvio padrão que é diretamente proporcional ao "embassamento" (*blur*) gerado .

Na Figura 2.5 tem-se um exemplo de aplicação do filtro de Média em uma imagem com ruído aleatório, pode-se perceber o "embassamento" bem como a redução do ruído presente na imagem anterior.

(a) Imagem com ruído

(b) imagem após a aplicação do filtro de média

Figura 2.5: Aplicação do filtro de Média

Order-Statistic (Nonlinear) Filters

São filtros que se baseiam em ordenar (*ranking*) os pontos que estão contidos na máscara, e depois substituir seu valor central pelo resultado da ordenação (*ranking result*). O melhor filtro conhecido nessa categoria é o filtro de mediana, que, como o próprio nome já diz, substitui o *pixel* central pelo valor da mediana dos da vizinhança, incluindo ele próprio [14]. O filtro de mediana é tão popular porque para certos tipos de ruído ele tem um excelente resultado "embaçando" menos do que o de média de tamanho similar. Ele é particularmente eficiente quando existe a presença de ruídos de impulso, também chamados de ruídos "sal-e-pimenta" por causa da sua semelhança com pontos pretos e brancos na imagem [2]. A mediana de um conjunto é o valor onde metade dos valores é menor ou igual e a outra metade é maior ou igual. Desta forma, dado um conjunto de tamanho n, por exemplo, a mediana se localiza na posição n/2 (n dividido por dois) para n par, e (n+1)/2 para n ímpar. Na Figura 2.6 abaixo se pode observar a mesma imagem com ruído induzido e após a aplicação do filtro de mediana, como na Figura 2.5 tem-se uma redução, porém com menor efeito de "embassamento".

(a) Imagem com ruído

(b) imagem após a aplicação do filtro de mediana

Figura 2.6: Aplicação do filtro de Mediana

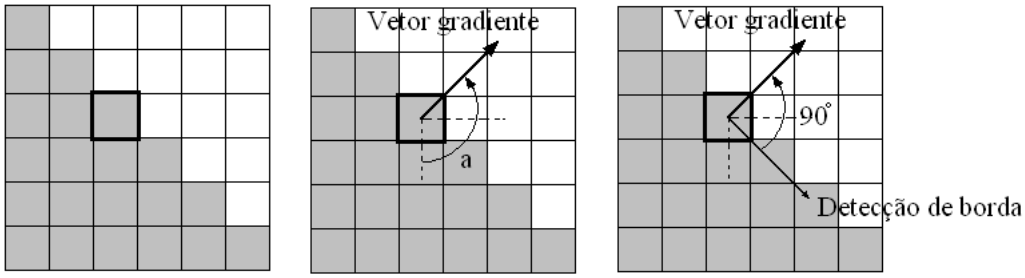


Figura 2.7: Exemplo de secção, com o vetor gradiente [2]

2.3.2 Extração de características

Como próxima etapa depois da suavização (*Smoothing*) da imagem tem-se a extração de características, que é onde se retira alguma informação da mesma. A suavização funciona como uma forma de preparação da imagem para que, posteriormente, possam ser aplicadas técnicas, por exemplo, de segmentação. Mais adiante serão explicitadas técnicas de detecção de bordas e de cantos, que são de suma importância para a proposta deste trabalho.

Bordas (*Edges Detection*)

Na seção anterior foi visto que para "suavizar" a imagem no domínio do espaço pode-se utilizar a média dos *pixels* em uma vizinhança. Como a média é análoga à integração, pode-se pensar nos filtros Passa Alta, ou no caso do domínio do espaço *Sharpening Spatial Filters*, como as derivadas espaciais na imagem [2]. Esta seção tratará apenas da detecção de bordas se utilizando derivadas de primeira ordem, por ser mais interessante ao trabalho. Para a detecção de bordas utilizando derivadas de primeira ordem se faz necessário antes explicar o conceito de gradiente. Para uma função $f(x, y)$ o gradiente de f nas coordenadas (x, y) é definido por um vetor bidimensional, tal como descrito abaixo na Fórmula 2.6 :

$$\nabla f \quad = \quad grad(f) = \begin{bmatrix} \frac{df}{dx} \\ \frac{df}{dy} \end{bmatrix} \tag{2.6}$$

O gradiente, no caso de imagens, tem uma importante propriedade geométrica que é apontar na direção da maior taxa de mudança de f no ponto (x, y). Na Figura 2.7 que representa um exemplo de uma secção ampliada de uma imagem contendo um segmento reto de borda, o ponto destacado é onde o gradiente foi calculado, pode-se perceber que ele aponta na direção da borda bem como é perpendicular a mesma [2].

A magnitude do vetor, denotada como $M(x, y)$, onde

$$M(x,y) \quad = \quad mag(\nabla f) = \sqrt{\left(\frac{df}{dx}\right)^2 + \left(\frac{df}{dy}\right)^2} \tag{2.7}$$

é o valor em (x, y) da taxa de mudança na direção do vetor gradiente. Note que $M(x, y)$ é uma imagem do mesmo tamanho da original, criada quando varia-se x e y por todas as localizações de *pixels* em f. É comum se referir a esta imagem como gradiente. Por se tratar de uma função isotrópica podemos simplificar a função $M(x, y)$ para

$$M(x,y) \quad = \quad \left\|\frac{df}{dx}\right\| + \left\|\frac{df}{dy}\right\| \tag{2.8}$$

com o objetivo de deixar a tarefa menos pesada computacionalmente. Como descrito em [2], podemos calcular as derivadas acima para cada *pixel* passando as seguintes máscaras 3x3:

$$\frac{df}{dx} \rightarrow \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix} \frac{df}{dy} \rightarrow \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} \tag{2.9}$$

Determinando tais derivadas calculamos o valor de $M(x, y)$, e repetindo a operação para todos os pontos da imagem obtemos a imagem gradiente, tal método é conhecido como filtro Sobel. Na Figura 2.8 temos o exemplo do uso do filtro Sobel para detecção de bordas.

(a) Imagem original

(b) imagem após a aplicação do filtro Sobel

Figura 2.8: Aplicação do filtro Sobel

Cantos (*Corner Detection*)

Os cantos são considerados, junto com as bordas, uma importante característica da imagem devido sua robustez no que diz respeito às mudanças de perspectivas e pequenas distorções [3].

Para identificá-los existem vários algoritmos porém, nesta seção será mostrado apenas o de Harris devido ao fato de reagir bem às variações de iluminação e rotação de uma imagem [25].

Tem-se o canto como sendo uma variação brusca em duas direções ortogonais de uma função $f(x, y)$. Pode-se caracterizar a variação de brilho de uma imagem por suas derivadas direcionais (gradiente). Utilizando os vetores de gradiente da imagem é possível determinar bordas e cantos.

Para o cálculo das derivadas direcionais faz-se necessário utilizar um dos filtros de redução de ruído como forma de reduzir o erro [3]. Desta forma, tem-se:

$$\begin{aligned} f_x &= \frac{\partial f}{\partial x} \approx f \circledast M_x \\ f_y &= \frac{\partial f}{\partial y} \approx f \circledast M_y \end{aligned} \tag{2.10}$$

onde $\circledast$ é o operador de convolução, e

$$\begin{aligned} \widehat{f_x^2} &= w \circledast f_x^2 \\ \widehat{f_y^2} &= w \circledast f_y^2 \\ \widehat{f_x f_y} &= w \circledast (f_x f_y) \end{aligned} \tag{2.11}$$

onde $w(x, y)$ é denotado como uma máscara de um dos filtros suavizantes (*smoothing*). Tais derivadas da Equação 2.6 geram a matriz C dada por:

$$C = \begin{bmatrix} \widehat{f_x^2} & \widehat{f_x f_y} \\ \widehat{f_x f_y} & \widehat{f_y^2} \end{bmatrix} \tag{2.12}$$

A matriz C na Equação 2.12 é real e simétrica. Desta forma, ela é positiva definida e possui autovalores positivos e reais, logo

$$M = \begin{bmatrix} a & b \\ b & c \end{bmatrix} \tag{2.13}$$

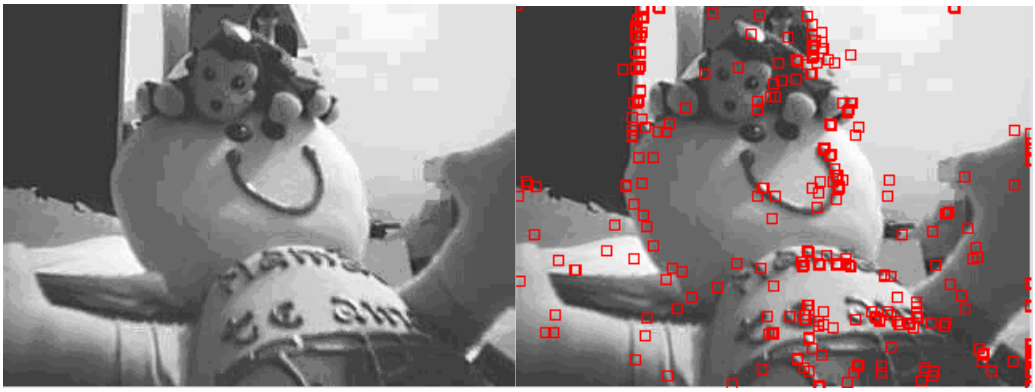
$$\lambda_1 = \frac{a+c}{2} + \frac{1}{2}\sqrt{(a - c^2 + 4b^2)} \tag{2.14}$$

$$\lambda_2 = \frac{a+c}{2} - \frac{1}{2}\sqrt{(a - c^2 + 4b^2)} \tag{2.15}$$

$$v_1 = \left[\frac{1}{2}(a-c) + \frac{1}{2}\sqrt{(a-c)^2 + 4b^2}, 1\right]^T \tag{2.16}$$

$$v_2 = \left[\frac{1}{2}(a-c) - \frac{1}{2}\sqrt{(a-c)^2 + 4b^2}, 1\right]^T \tag{2.17}$$

Nas Equações 2.13, 2.14, 2.15, 2.16 e 2.17 temos o cálculo de autovalores e autovetores para uma matriz exemplo M, na qual λ_1 e λ_2 são os autovalores e v_1 e v_2 são os autovetores. Na Figura 2.9 pode-se ver os cantos pintados em vermelho.



(a) Imagem original (b) Identificação dos cantos em vermelho

Figura 2.9: Detecção de cantos

Para classificar uma região na vizinhança de um *pixel* localizado em (x, y), deve-se levar em consideração os valores dos autovalores. Desta forma, ela pode ser classificada como:

- Praticamente uniforme, se os valores de ambos os autovalores forem pequenos.
- Borda, se λ_1 é grande e λ_2 é pequeno. O autovetor v_1 é ortogonal à borda.
- Canto, se ambos são grandes. Os autovetores são ortogonais às bordas do canto.

Com isso pode-se determinar os cantos em uma imagem. Comparar se um autovalor é grande ou pequeno é arbitrário, sendo possível identificar uma quantidade

maior ou menor de cantos. O detector de cantos de Harris utiliza a idéia de autovalor, porém de maneira implícita, de forma que o cálculo dos autovalores não é necessário. Existe apenas a necessidade de calcular o determinante e a soma da diagonal da matriz C, que são respectivamente iguais à soma e ao produto dos autovalores, como mostram as Equações 2.18 e 2.19.

$$det\, C = \lambda_1 \lambda_2 \tag{2.18}$$

$$sum\, C = \lambda_1 + \lambda_2 \tag{2.19}$$

Em seguida define-se o parâmetro de detecção de Harris como na Equação 2.20.

$$\text{H}=\lambda_1\lambda_2 - \alpha(\lambda_1 + \lambda_2)^2 = det\, C - \alpha(sum\, C)^2 \tag{2.20}$$

Substituindo λ_1 por λ e λ_2 por $k\lambda$, onde $0 \geq k \geq 1$, tem-se:

$$H = \lambda^2(k - \alpha(1+k)^2) \tag{2.21}$$

Sendo que para $H > 0$, o fator que determina um canto temos:

$$\alpha < \frac{k}{(1+k)^2} \leq 0.25 \tag{2.22}$$

Tem-se α como um controle da sensibilidade do detector. Quanto maior o α menor a quantidade de cantos serão detectados. É possível determinar um H_t limitante onde a região é um canto se $H > H_t$. Normalmente H_t é ajustado próximo de zero onde estariam as regiões sem cantos ($flat$) [26].

Existem trabalhos que exploram a possibilidade de paralelização do algoritmo de Harris para detecção de cantos. Tais trabalhos têm se mostrado eficientes na diminuição no tempo de processamento[9][27].

2.3.3 *Template Matching* e Correlação Cruzada (*Cross-correlation*)

Nesta seção será explicitada uma visão sobre como se pode encontrar *pixels* correspondentes em imagens diferentes como forma de identificar uma relação entre os pontos nas imagens.

Provavelmente, a aplicação que se utiliza de imagens bidimensionais formada por *pixels* mais simples é o restrito reconhecimento de caracteres óticos (OCR

- *Optical Character Recognition*) do tipo usado para processar cheques [3]. A variedade de caracteres impressos em cheques de bancos se restringe aos numerais que vão de 0 até 9 e alguns poucos sinais de pontuação. Além disso, os caracteres tem tamanho, localização e orientação fixos e uma fonte fixa especificamente desenvolvida para torná-los facilmente distinguíveis um do outro. Neste ambiente de tão poucas variáveis, uma técnica chamada de *template matching* proporciona um resultado rápido e de baixo custo computacional.

Na Figura 2.10 pode-se observar uma aplicação dessa técnica para as letras A, B e C. Um *template* (modelo) de *pixels* pretos, que cobre a forma de cada caracter alvo, é armazenado na memória. Cada letra a ser reconhecida é combinada a todos os *templates* armazenados utilizando-se uma operação de OR-exclusiva para contar o número de *pixels* que "casam" ou não. O *template* de maior pontuação de rede (número de pontos que bateram, menos os que não) é selecionado como a identificação do caracter. Em algumas implementações, a pontuação pode ser normalizada dividindo-a pelo número de *pixels* pretos no *template*, este número varia substancialmente para os vários caracteres.

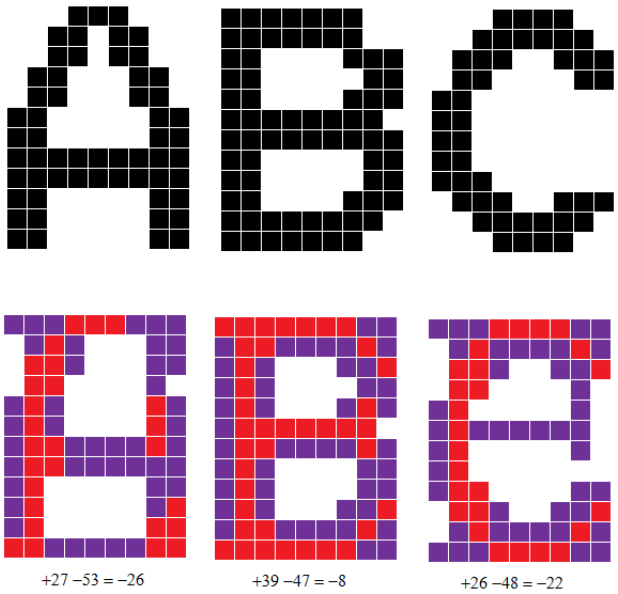


Figura 2.10: O *matching* entre os *templates* e as imagens gerando diferentes pontuações de rede

Dado o ambiente controlado ao qual esta técnica se aplica os resultados são muito satisfatórios, mesmo para uma taxa de erro considerável.

Esse tipo de *template matching* é um tipo específico de correlação cruzada (*cross-correlation*), que pode ser utilizado para imagens em escala de cinza ou coloridas bem como para imagens binárias. O conceito de correlação cruzada nesse trabalho será usado levando em consideração o domínio espacial seguindo o que foi explicitado na Seção 2.3.

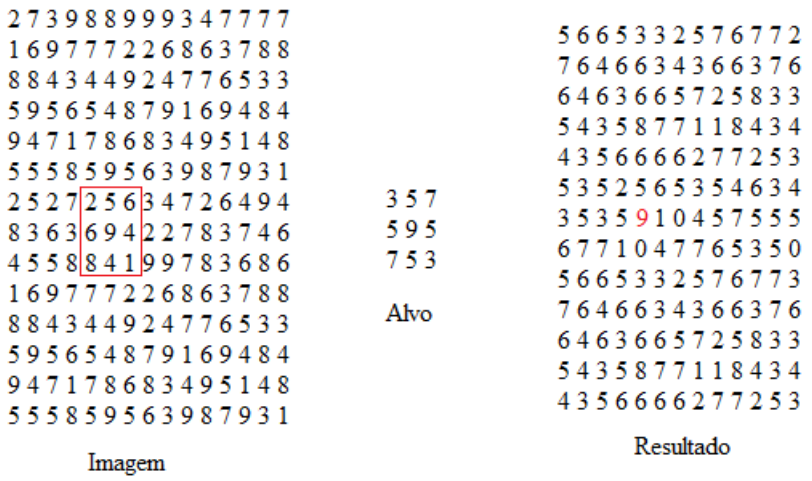


Figura 2.11: Correlação usando *pixels* com valores de um único dígito. O melhor valor é marcado em vermelho, note que ele não é idêntico ao alvo[3], algoritmo proposto em 2.3.3 .

Desta forma, nesse caso o "modelo alvo" (*target pattern*) caminha por toda a imagem e para cada posição (i, j) o valor de correlação é somado de acordo com a Equação 2.18, para adquirir o resultado que se pode ver na Figura 2.11 [3].

$$c(i,j) = \frac{\sum_{x=0}^{s}\sum_{y=0}^{s} B_{x+i,y+j}.T_{x,y}}{\sqrt{\sum_{x=0}^{s}\sum_{y=0}^{s} B^2_{x+i,y+j}.\sum_{x=0}^{s}\sum_{y=0}^{s} T^2_{x,y}}} \tag{2.23}$$

Onde B e T são, respectivamente, valores de brilho da imagem e o alvo (modelo procurado), e o somatório é limitado pelo valor de s.

O valor de $c(i, j)$ é muito grande quando os *pixels* da imagem e do modelo coincidem, e o denominador normaliza o resultado para as variações da imagem e da região. Pode-se usar os resultados para se criar uma imagem em tons de cinza onde o tom indica o quão aquela parte da imagem se assemelha ao alvo procurado, e determinar, por exemplo, um valor de *threshold* (descrito na Seção 2.4.1) para detectar onde se conseguiu achá-lo.

2.4 Detecção de movimento

Nesta seção serão explicitados conceitos referentes a detecção de movimento e sua interpretação como forma de resolver problemas de detecção do mesmo através de câmeras de vídeo.

2.4.1 O movimento

O movimento pode ser percebido por uma sequência obtida de imagens de uma cena em tempos diferentes distribuídos em um determinado intervalo. Note que a pura mudança de posição, apesar de caracterizar o movimento, é uma definição muito vaga. Para este trabalho se faz necessário uma maior explicitação do conceito, principalmente no que diz respeito ao referencial do movimento, ou seja, em relação à que esse movimento ocorre[4]. Para tal criaram-se quatro classes gerais levando em consideração tanto a cena observada, como o próprio observador para facilitar o entendimento do conceito:

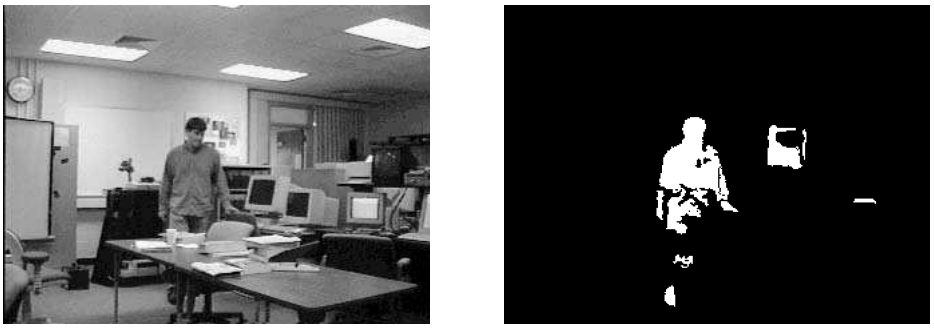
1. um observador parado com apenas um objeto em movimento
2. um observador parado com vários objetos em movimento
3. um observador em movimento em uma cena constante (sem nenhum objeto em movimento)
4. um observador em movimento com vários objetos em movimento

A situação mais simples para a detecção de movimento por um sistema de visão computacional ocorre quando a câmera (observador) está diante de um fundo relativamente estático. Nesta situação o objeto em movimento resulta em mudanças associadas ao próprio objeto nos *pixels* da imagem. Na Figura 2.12 pode-se ver que os objetos em movimentos podem ser detectados através da simples subtração da imagem anterior (imagem de fundo) pela imagem atual.

É necessário fazer duas ressalvas quanto a Figura 2.12:

- a subtração referida aqui é sim a subtração dos valores dos *pixels* da imagem anterior pelos da imagem atual.
- e que para chegar-se na imagem mostrada, além da aplicação de um filtro para eliminação de ruído, é necessária uma binarização da imagem resultante da subtração, ou seja, atribuir a todos os *pixels* os valores 0 (preto) ou 1 (branco) levando em consideração um valor de intensidade específico. A este valor dá-se o nome de limiar[3].

Quando apenas a câmera é movimentada também se cria uma mudança na imagem referente ao próprio movimento da câmera, mesmo que o ambiente permaneça imutável. Esse tipo de situação pode acontecer quando se quer ter uma visão mais abrangente de uma cena, ou até mesmo para classificar o movimento da câmera, por exemplo, se ela está indo para frente ou girando. Dá-se o nome de *egomotion* ao movimento de uma câmera em uma cena estática ou não.



(a) imagem a ser subtraída (b) imagem resultante com o filtro

Figura 2.12: Detecção de movimento em imagem estática [4]

A situação mais complexa de detectar movimento é quando tem-se, além da câmera, vários objetos também em movimento, o que dificulta a determinação de um fundo estático para servir como referencial[28, 29].

Nas seções a seguir foram tratadas algumas técnicas para análise de imagens com o objetivo de detectar mudanças no movimento da cena.

2.4.2 Método de Subtração de Imagem (*Imagem Subtraction*)

Na Seção 2.4.1 foi dada como exemplo uma aplicação que se utiliza deste método para a detecção de movimento em uma cena estática. Ao supor um objeto se movendo em uma cena onde o fundo é todo escuro e uniforme tem-se um objeto de cor diferente de preta atravessando a cena. Se esta ação está sendo capturada por uma câmera com velocidade de captura de 30 *frames* por segundo, e para cada *frame* capturado tem-se na exibição o resultado da subtração deste pelo seu direto antecessor. Percebe-se que a borda da frente e de trás do objeto em movimento permanecem como a resultante do mesmo na imagem final proveniente da subtração. Na Figura 2.12 pode-se ver um exemplo de tal técnica, porém com um fundo que não é uniforme. O algoritmo 2.1 representa uma maneira geral de implementar a detecção de movimento através deste método .

Algoritmo 2.1 Detecção por Subtração[4]

Entradas:

- Duas imagens monocromáticas $I_t[r,c]$ e $I_{t+\partial}[r,c]$ de uma cena tiradas com a diferença ∂ de tempo ;
- valor τ da intensidade de *threshold* ;

Saída:

- Imagem binária I_{out};

1. Para todos os *pixels* [r,c] nas imagens de entrada
   Se $|I_{t+\partial}[r,c] - I_t[r,c]| \geq \tau$, então $I_{out} = 1$;
   senão $I_{out} = 0$;
2. retorna I_{out};

Apesar de muito simples este algoritmo pode ser utilizado em câmeras de segurança para detecção do movimento em uma calçada, monitoramento de pacientes, dentre outras aplicações. Na Figura 2.13 podemos ver o resultado do algoritmo de subtração para o vídeo capturado de uma câmera comum.

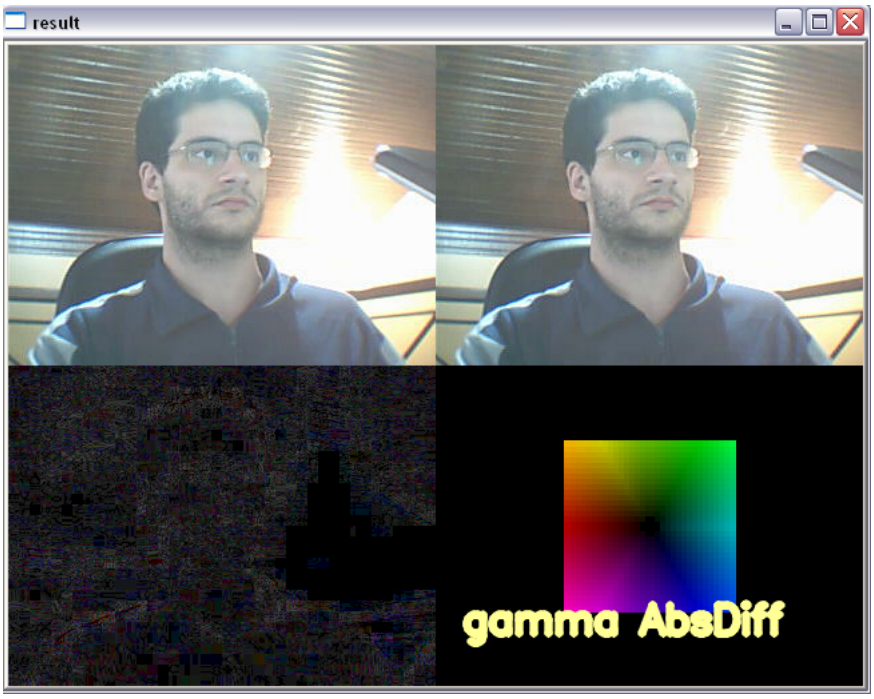
2.4.3 Cálculo de Vetores de Movimento ($Motion\,Vectors$)

Como consta na Seção 2.4.1 uma mudança no movimento no ambiente tridimensional é observado diretamente na sua projeção bidimensional, ou seja, nos *pixels* da imagem capturada. Desta forma, pode-se estimar informações sobre o meio.

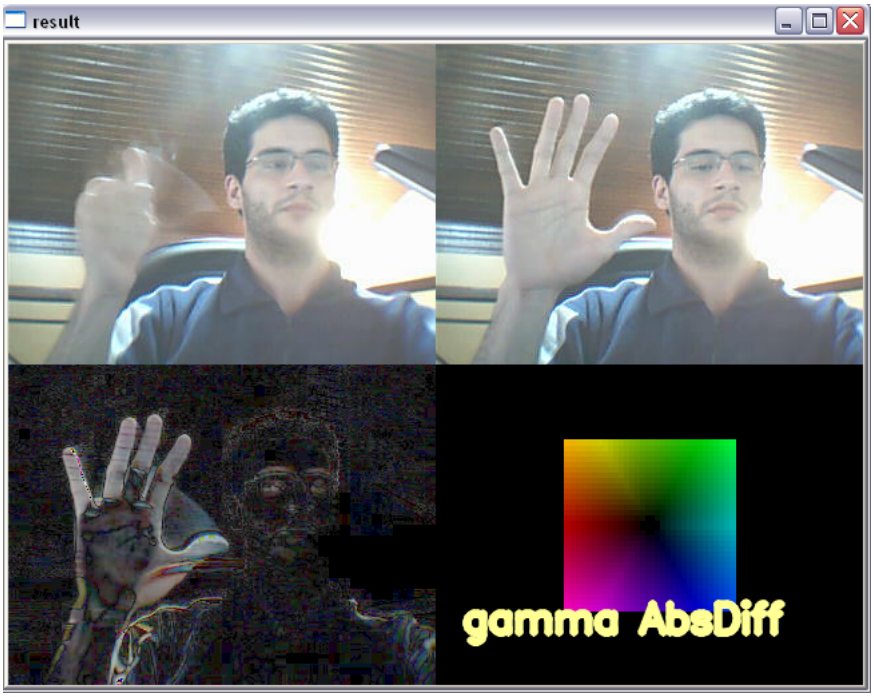
As alterações na imagem podem ser identificadas analisando os efeitos que estas causam nas propriedades (*features*) da imagem. Na Figura 2.14 tem-se um exemplo do efeito causado na imagem de uma câmera se aproximando, se afastando e girando em relação a cena.

Note que é possível representar as imagens da Figura 2.14 através de um array bidimensional (onde cada coordenada representa o pixel equivalente da imagem) de vetores bidimensionais que representam o movimento da cena tridimensional. A esse *array* dá-se o nome de campo de movimento (*motion field*). Os vetores de movimento na imagem representam o deslocamento dos pontos tridimensionais [4].

Percebe-se também que quando a câmera se aproxima e se afasta tem-se um ponto central do qual os vetores de movimento parecem estar, respectivamente, divergindo ou convergindo para ele. Quando todos os vetores do campo de movimento se afastam de um ponto na imagem este recebe o nome de foco de expansão (*focus of expansion*), e quando se percebe os mesmos vetores convergindo para

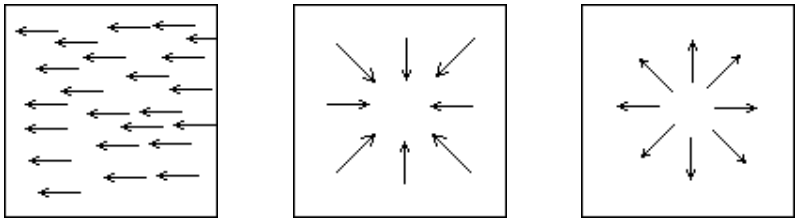


(a) (Objeto parado)



(b) (Objeto em movimento)

Figura 2.13: Aplicação executando o algoritmo de subtração cedido por Jecel Assumpção Jr. Da direta para esquerda, de cima para baixo: *frame* atual, *frame* anterior e subtração resultante



(a) câmera virando para a direita

(b) câmera de afastando da cena

(c) câmera de aproximando

Figura 2.14: Efeitos sobre a imagem com a movimentação da câmera

um ponto na imagem a este dá-se o nome de foco de contração (*focus of contraction*)[29]. Normalmente, os vetores de movimento apontam em uma direção oposta ao movimento da câmera.

Para a detecção do movimento normalmente assume-se que a intensidade dos pontos em deslocamento na cena tridimensional permanece constante no intervalo de tempo que é usado para estimar o movimento do ponto em questão[30, 4]. Alternativamente, pode-se também assumir que a intensidade dos pontos perto das bordas dos objetos em movimento é aproximadamente constantes durante o intervalo avaliado.

Dá-se o nome de fluxo da imagem (*image flow*) ao campo de movimento calculado assumindo que a intensidade perto de pontos correspondentes é relativamente constante.

2.4.4 Correspondências entre Pontos (*Point Correpondences*)

Na seção anterior foi explicitado o conceito de campo de movimento de uma cena. Note que para o cálculo do campo não se pode tomar todos os pontos de uma imagem, isso tornaria o processo muito custoso e ineficiente[4]. Desta forma, ao invés de calcular o campo de movimento por completo, este é calculado de forma esparsa (*sparce motion field*), ou seja, levando em consideração apenas alguns pontos nas imagens. Para isso, os pontos da imagem devem ser distinguidos de tal forma a serem localizados nas duas imagens.

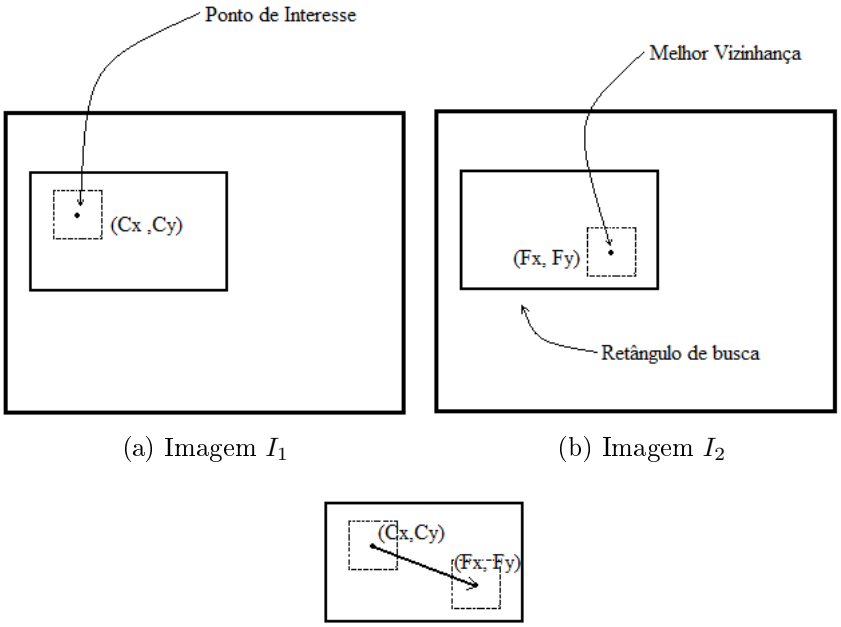
A detecção de pontos de maior interesse (como, por exemplo, os cantos) na imagem é uma opção para a redução da ambiguidade. Esta pode ser implementada utilizando o algoritmo de Harris, que varia muito pouco com relação a fatores como rotação, iluminação e ruído na imagem [25].

Por outro lado, pode-se utilizar um operador de interesse (*interest operator*). Ele computa a variação da intensidade nas direções vertical, horizontal e diagonal da vizinhança de um ponto central, de tal forma que este só é classificado como ponto de interesse caso a menor variação seja maior que um valor de *threshold*.

Note que ambos os métodos retornam "pontos de interesse" . É uma escolha de projeto como tais pontos serão identificados.

Uma vez que o conjunto de pontos de interesse foi identificado na imagem I_1 (capturada no tempo t), é necessário identificá-lo I_2 (capturada no tempo $t + \partial$). Porém, ao invés de primeiro identificar o conjunto de pontos de interesse em I_2 para, posteriormente, tentar fazer as correspondências com os da imagem I_1, pode-se procurar diretamente em I_2 pelos pontos correspondentes aos pontos de interesse da imagem I_1. Pode-se implementar tal busca utilizando o método de correlação cruzada (*cross-correlation*), descrito na Seção 2.3.3. Desta forma, dado um ponto de interesse de I_1, copia-se sua vizinhança e procura-se a vizinhança com a melhor pontuação na imagem I_2.

Perceba que o fato de procurar por toda a imagem I_2 por um ponto que tenha uma vizinhança equivalente a um ponto de interesse em I_1, além de ser muito custoso, pode gerar muita ambiguidade entre pontos que tem as vizinhanças parecidas. Desta forma, levando em consideração que o movimento é limitado, ao invés de percorrer toda a imagem I_2 procurando o ponto correspondente, pode-se considerar apenas uma parte da imagem, limitando a busca, por exemplo, a uma área em forma de retângulo de I_2. Desta forma, a quantidade de ambiguidades e o tempo de busca diminuem. Vale ressaltar que tal redução da análise se deve ao fato de supor que existe um limite imposto às velocidades dos objetos envolvidos na cena. A Figura 2.15 esquematiza a busca em I_2 pelos pontos de interesse de I_1, analisando apenas uma área referente a um retângulo.



(a) Imagem I_1 (b) Imagem I_2

(c) vetor de movimento

Figura 2.15: Método utilizando a correspondência entre pontos

O ponto (C_x, C_y) corresponde à origem do vetor e o ponto (F_x, F_y) ao final. Desta forma, o par ordenado$[(C_x, C_y), (F_x, F_y)]$ é capaz de indicar o sentido do mesmo. Note que o vetor da Figura 2.15c se apresenta no tamanho real de acordo com as representações.

2.5 Fluxo Ótico (*Optical Flow*)

Estimar o campo bidimensional de velocidade em uma sequencia de imagens provenientes de um vídeo é um dos tópicos mais antigos e ativos da pesquisa em visão computacional. Um dos estudos mais importantes nesta área foi o publicado por Horn e Schunk em 1981 [18]. Neste estudo o fluxo ótico é definido como sendo: "o campo de velocidade na imagem, que transforma uma imagem na próxima da sequencia, e como tal não é unicamente determinado. O campo de movimento, por outro lado, é um conceito puramente geométrico, sem nenhuma ambiguidade". O fluxo óptico é considerado uma aproximação do movimento da imagem [16]. O cálculo do fluxo ótico consiste em achar na próxima imagem da sequência os pontos da imagem anterior. Na maioria dos métodos considera-se que os pontos que se moveram de uma imagem para outra mantém o mesmo valor dos *pixels* em escala de cinza [16, 30]. Desta forma, sendo $I(x, y)$ a intensidade do ponto (x,y) tem-se que[31]:

$$I(x,t) \quad = \quad I(x - w.t, 0) \tag{2.24}$$

Onde $w = (u, v)^T$. Expandindo por Taylor tem-se que:

$$\nabla I(x,t).w \quad + \quad I_t(x,t) = 0 \tag{2.25}$$

Onde $\nabla I(x,t) = (I_x, I_y)^T$ é o vetor gradiente e I_t é a derivada temporal, ambos da imagem. Para o cálculo do fluxo ótico é necessário determinar as duas componentes do vetor w. Como existe apenas uma equação para a resolução do sistema linear ele não tem uma solução única[30]. Desta forma, vários métodos assumem mais restrições ao movimento[16].

2.5.1 Técnicas para o cálculo do Fluxo Ótico

Pelo fato de ser necessário assumir algumas restrições ao cálculo do fluxo ótico existem vários métodos para seu cálculo. Apesar das diferenças entre os métodos muitas destas técnicas podem ser vistas em termos de três passos [31] :

1. Suavização ou pré-filtragem da imagem com o objetivo de extrair a estrutura do sinal de interesse e melhorar a taxa sinal-ruído;

2. Extração de características básicas como as derivadas espaço-temporais;

3. Integração destas características para produzir um campo de fluxo bidimensional, o que geralmente envolve mais restrições em relação à suavização e ao subjacente.

Tais métodos podem ser divididos em quatro classes [31] explicitadas nas próximas seções.

Diferenciais

As técnicas diferenciais consistem em computar a velocidade através de derivadas espaço-temporais ou versões filtradas da imagem [31].

Os métodos de primeira ordem utilizam as derivadas de primeira ordem. Porém, mesmo com o cálculo das derivadas é necessária a elaboração de mais restrições para solucionar as componentes da velocidade[31].

Métodos de segunda ordem utilizam derivadas de segunda ordem. Pelo fato de considerarem mais de uma coordenada no movimento, deformações que são encontradas nos métodos de primeira ordem não são encontradas nos de segunda. Porém, devido ao problema da abertura, se naquela região o gradiente variar apenas em uma coordenada não será possível medir o fluxo ótico com precisão[32].

Baseados em Correlação de Região

Uma diferenciação precisa pode ser impraticável, seja devido à grande taxa de ruído, ou devido a um pequeno número de *frames* [31]. Nestes casos pode-se optar por métodos baseados em região. Estes métodos são equivalentes ao mostrado na Seção 2.3.3. Esta classe de métodos calcula o vetor velocidade procurando as regiões mais semelhantes (*matching*). Para este cálculo pode-se utilizar a Equação 2.23 de correlação-cruzada.

Baseados em energia

Também chamados de métodos baseados em freqüência, consideram a energia de saída de filtros de velocidade no domínio da transformada de Fourier. Em alguns trabalhos já foi mostrado que tais filtros têm um desempenho muito parecido com os baseados em correlação e em gradiente [31].

Baseados em Fase

Consideram a velocidade definida em termos do comportamento dos filtros de saída passa-banda.

2.5.2 Comparação entre as Técnicas

Em 1994 foi realizado um trabalho experimental comparando algoritmos de todas as classes citadas [31]. Seus resultados foram referenciados em trabalhos recentes [16, 32, 30]. Os resultados experimentais com os algoritmos concluíram que um dos métodos mais confiáveis é o de Lucas e Kanade[33]. Para este trabalho foi levado em consideração apenas o método Lucas e Kanade uma vez que tem uma boa relação entre implementação e qualidade dos resultados.

Lucas e Kanade

A idéia básica do método de Lucas e Kanade é considerar que para uma região pequena o fluxo ótico é constante. Desta forma, dada uma região nxn na vizinhança de um ponto (x, y) tem-se:

$$\begin{bmatrix} u \\ v \end{bmatrix} = (A^T.A)^{-1}A^T b \tag{2.26}$$

sendo:

$$A = \begin{bmatrix} I_{x1} & I_{y1} \\ I_{x2} & I_{y2} \\ \vdots & \vdots \\ I_{xn} & I_{yn} \end{bmatrix} \quad , \quad b = \begin{bmatrix} I_{t1} \\ I_{t2} \\ \vdots \\ I_{tn} \end{bmatrix} \tag{2.27}$$

O sistema linear que antes tinha apenas uma equação agora tem n^2 equações tornando possível determinar as componentes u e v do vetor que determinam o fluxo óptico[33].

Devido a grande independência de dados deste algoritmo é possível ter um ganho computacional através da paralelização do método. Uma aplicação multiprocessada deste algoritmo pode ser vista em [16].

2.5.3 Problema da Abertura (*Aperture Problem*)

Pelo fato de a maioria dos métodos considerar o movimento em uma região (abertura) da imagem isso faz com que regiões homogêneas tenham movimento ambíguo. Isso acontece principalmente porque em regiões homogêneas, ou que variam apenas em uma direção não é possível determinar o vetor que tangencia o movimento[30].

Este problema recebe o nome de "problema da abertura" [32]. Uma forma de contorná-lo é considerar pontos que variam consideravelmente em duas dimensões [34]. Pontos que tem esta propriedade na imagem são chamados de cantos (*corners*).

2.6 Co-projeto *hardware e software* em Sistemas Embarcados

Sistemas embarcados têm recursos de *software* e *hardware* mais limitados quando comparados à computadores de uso geral[19]. Parâmetros como redução no consumo de energia, custo de produção e tamanho são mais relevantes no projeto de um sistema embarcado do que em um que utiliza um computador convencional como base[19].

Um sistema embarcado pode ter suas funcionalidades somente em *software*, somente em *hardware* ou ainda como uma integração entre as duas anteriores [19]. É uma escolha de projeto como tais funcionalidades serão implementadas. Para tal é levado em consideração a flexibilidade, desempenho e custo do desenvolvimento. Dependendo do tipo de ambiente no qual ele será inserido, outras variáveis como o consumo de energia, por exemplo, podem ser levadas em consideração no projeto.

Quando um sistema é descrito não explicitando quais partes deverão ser implementadas em *hardware* ou em *software*, cabe ao projetista decidir como tal sistema será implementado. No gráfico descrito na Figura 2.16 tem-se uma comparação das três formas de implementar um sistema embarcado.

Percebe-se no gráfico da Figura 2.16 que o co-projeto hardware e software é uma boa solução quando é necessária a flexibilidade do *software*, e o desempenho proveniente do *hardware*, com um baixo impacto no custo de desenvolvimento.

2.7 FPGA´s e Computação reconfigurável

2.7.1 Computação Reconfigurável

No desenvolvimento de *hardware* existem dois métodos tradicionais para o desenvolvimento de sistemas computacionais[35]. Como primeiro método destaca-se o

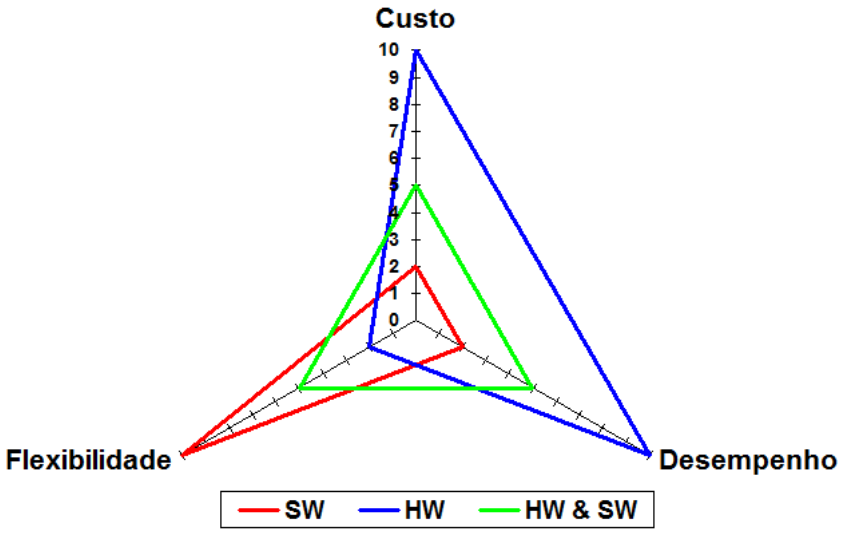


Figura 2.16: Gráfico comparativo entre as três formas de implementação: *Software* (SW), *Hardware* (HW) e *hardware e software*/(HW/SW).[5]

desenvolvimento de circuitos integrados construídos para uma aplicação, conhecidos como ASICs (*Aplication Specific Integrated Circuit*). Porém, nessa primeira opção, uma vez fabricados os *chips* não podem mais ser modificados, tornando o processo de mudança custoso principalmente quando em larga escala.

O segundo método consiste em uma plataforma mais flexível: algoritmos em *software* que rodam em microprocessadores. Desta forma, para mudar a função do sistema basta mudar o *software*. Embora tal opção gere maior flexibilidade existe uma perda em desempenho.

A computação reconfigurável pode ser considerada uma terceira forma de desenvolvimento. Ela possibilita uma flexibilidade relativamente maior do que o *hardware* e pode ter um desempenho maior do que o do *software* [35]. As configurações assumidas pela FPGA podem ser especificadas por:

- diagramas de circuitos digitais
- por programas descritos e por linguagem de *hardware*, como por exemplo, VHDL (*Very High Speed Integrated Circuit* Hardware *Description Language*)
- ou por linguagem de programação.

Na Figura 2.17 pode-se observar no gráfico que, das tecnologias para a implementação de sistemas embarcados, os FPGA´s tem como vantagem flexibilidade e um desempenho considerável.

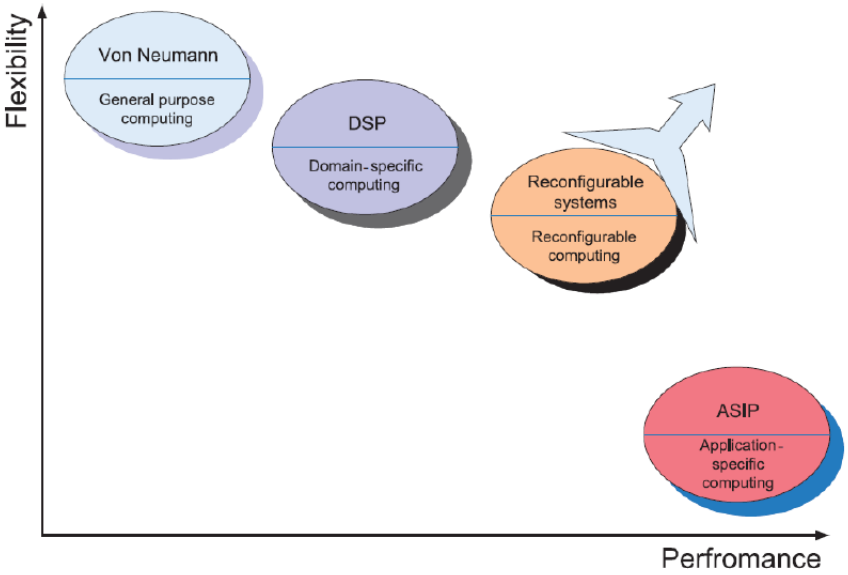


Figura 2.17: Gráfico comparativo de tecnologias de desenvolvimento[6].

2.7.2 Nios II

O processador Nios II da Altera é um processador *soft-core*, isso significa que ele não possui um núcleo fixo em silício como os processadores usuais[21]. Seu núcleo é implementado em uma linguagem de alto nível de descrição de *hardware*. Desta forma, o desenvolvedor pode modificá-lo como forma de melhorar o desempenho do seu sistema, adicionando ou removendo componentes de *hardware*.

A flexibilidade do processador Nios II não implica que para cada aplicação uma nova configuração de núcleo seja criada. São fornecidos núcleos prontos para a integração em um projeto. Além do fornecimento de núcleos prontos, é disponibilizado ao desenvolvedor de software um ambiente de simulação do processador que pode ser usado para depuração do código antes de ser realizada a configuração final do *hardware*[36].

O processador Nios II tem o núcleo de um processador RISC (*Reduced Instruction Set Computer*) de propósito geral. Proporciona também uma camada de abstração de hardware (HAL *Hardware Abstraction Layer*) que é um ambiente que proporciona uma interface entre os *drivers* dos dispositivos em *hardware* com o *software*. A HAL define um conjunto de funções que podem ser usadas para inicializar e acessar cada classe de dispositivos presentes na placa de desenvolvimento[21].

O ambiente de desenvolvimento possibilita a prototipação do sistema em uma placa FPGA antes que seja construído o *hardware* personalizado [36]. Tal característica torna possível o balanceamento com relação ao custo e desempenho da aplicação.

Devido a grande flexibilidade do processador soft-core Nios II é possível de-

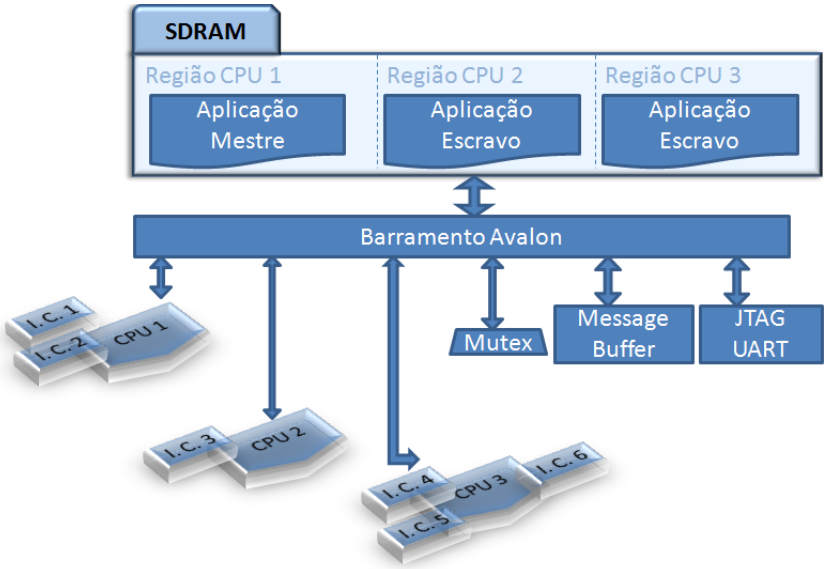


Figura 2.18: Arquitetura heterogênea utilizando o Nios II

senvolver arquiteturas diversas. Uma das possibilidades que será explorada neste trabalho será a de implementação de uma arquitetura paralela para explorar o paralelismo exposto em trabalhos anteriores[9, 27].

A Altera disponibiliza um tutorial de como criar uma arquitetura *multi-core* simétrica. Para este trabalho será explorada a possibilidade de replicar o tutorial modificando-o para uma arquitetura também paralela, porém heterogênia. Com processadores diferentes entre si será possível tanto explorar o paralelismo do código como a possibilidade de customizar instruções em hardware. Na Figura 2.18 tem-se um esquema de uma possibilidade de implementação utilizando processadores heterogêneos.

2.7.3 Instruções customizadas Nios II

Por se tratar de um processador *soft-core* o Nios II permite que sejam customizadas novas instruções e desta forma acelerar um *software* que demanda muito tempo computacional. Utilizando instruções customizadas é possível reduzir uma sequência complexa de instruções padrão para uma implementada em *hardware*. A nova é conectada diretamente na ULA (Unidade Lógica Aritmética) do Nios II como se pode ver na Figura 2.19.

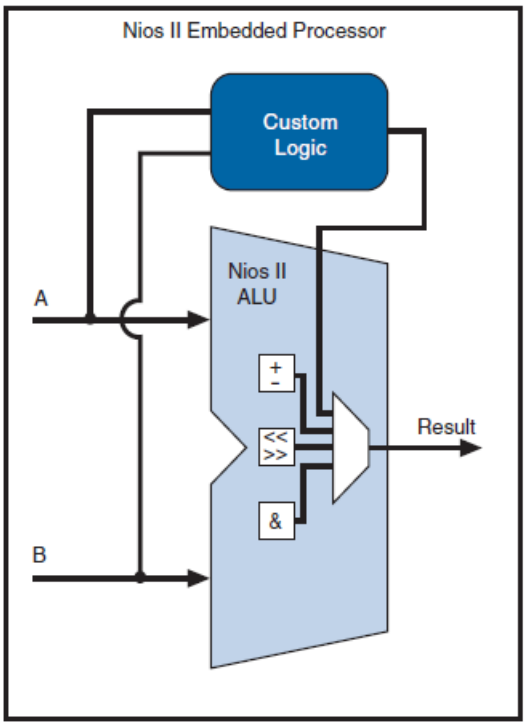


Figura 2.19: Instrução conectada na ULA do Nios II

A customização de instruções abstrai os detalhes da implementação do programador. A nova instrução é declarada como uma macro no código C ou C++ executado no Nios II ou pode ser chamada como uma função[7].

Existem quatro tipos de instrução customizadas aceitas e para cada uma delas os sinais de entrada são diferentes como é possível observar na Figura 2.20.

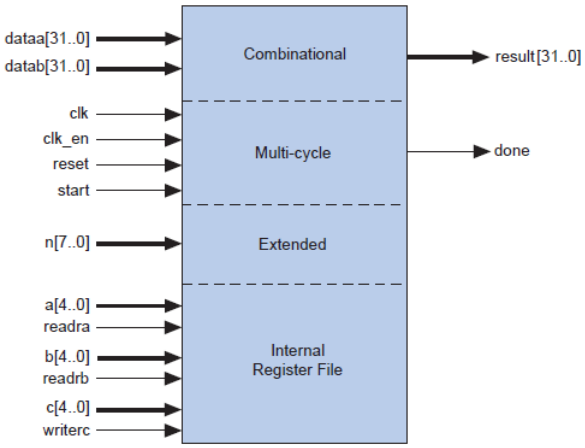


Figura 2.20: Tipos de instruções customizadas[7].

Neste trabalho será utilizada o tipo *Multi-cycle* que terá o formato determinado pelos sinais de entrada e saída desta forma:

- dataa[31..0] e datab[31..0] : são as duas entradas da instrução, cada uma delas tratada como um inteiro de 32 bits.

- clk : clock geral do processador que dita a sincronização de todo o sistema.
- clk_en : sinal que determina o começo da execução da instrução sendo 1 quando a instrução pode começar a executar e 0 depois que termina.
- reset : sinal de reset que reinicia a instrução
- start : este sinal é 1 apenas na primeira borda de subida do clk uma vez que o clk_en já é 1.
- n[7..0] : determina o modo no qual a instrução irá executar, desta forma é possível executá-la de maneiras diferentes dependendo do valor de n.

O diagrama de ondas do comportamento dos sinais pode ser observado na Figura 2.21.

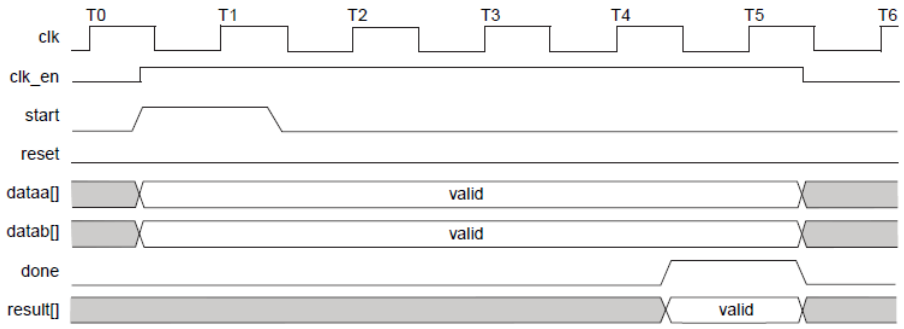


Figura 2.21: Diagrama de ondas da instrução Multi-cycle[7].

2.8 Multiprocessadores

Desde a criação do computador pessoal foi possível acompanhar um crescimento constante da computação. Isso se deve a Lei de Moore que previa que a cada 18 meses a quantidade de transistores que caberiam em uma determinada área dobraria [22, 8]. Embora esta lei ainda se aplique, por volta do ano de 2005 o desempenho dos sistemas computacionais começou não seguir proporcionalmente esta evolução[8]. Atualmente, problemas como o consumo de energia limitaram o desempenho dos *softwares* associados aos *hardwares* atuais. A frequência de um processador sequêncial atingiu barreiras físicas devido a restrições de consumo e de dissipação de calor. Na Figura 2.22 pode-se comparar o que era esperado para o aumento da frequência de um processador com a tendência nos anos atuais.

Desta forma, o crescimento da computação por processadores com apenas um núcleo de processamento está estagnada ou evoluindo de maneira marginal. Na computação sequencial as duas maiores restrições para seu crescimento são [8]:

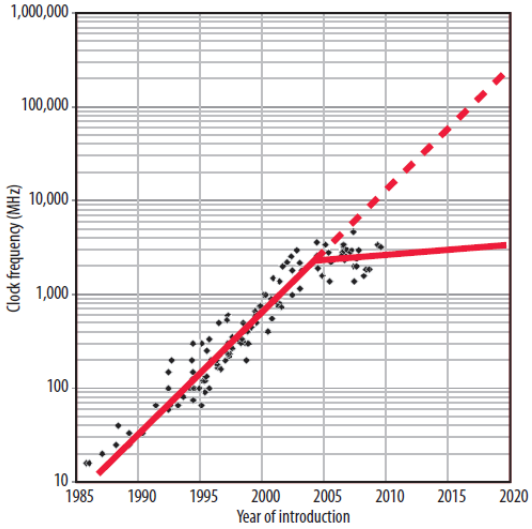


Figura 2.22: Frequencia dos processadores[8]

- A incapacidade de aumentar a freqüência tendo restrições de consumo de energia
- A falta de ganho de desempenho de técnicas como *pipelines* maiores e execução especulativa em arquiteturas sequenciais

Devido a tais problemas a computação paralela tem sido cada vez mais utilizada como forma de ganho de desempenho, tanto para sistemas embarcados como para computadores de uso geral[8, 20]. Desta forma, esta seção tem como objetivo explicitar algumas arquiteturas paralelas que poderão ser utilizadas no trabalho.

Arquiteturas paralelas são formadas por unidades menores chamadas de elementos de processamento (EP)[20]. Os EP´s podem ser desde unidades pequenas e simples como ALU´s até computadores completos. A quantidade de elementos de processamento é determinada pelo tipo de conexão utilizada e pelo propósito da arquitetura[37].

As arquiteturas paralelas podem ser classificadas de acordo com o fluxo de instruções e dados. Desta forma, elas podem ser classificadas como [38]:

- *Single Instruction, Single Data* (SISD): Um único fluxo de instruções emitindo um único fluxo de dados;
- *Single Instruction, Multiple Data* (SIMD): Um único fluxo de instruções sendo executado por mais de um EP gerando assim vários fluxo de dados;
- *Multiple Instruction, Single Data* (MISD): Vários fluxos de instruções executados por apenas um EP;
- *Multiple Instruction, Multiple Data* (MIMD): Vários EP´s executando diferentes fluxos de instruções criando vários fluxos de dados.

Como exemplos da arquitetura MIMD existem os processadores simétricos e os *clusters*. O uso de processadores simétricos é um dos focos deste trabalho. A arquitetura MIMD ainda pode ser dividida levando em consideração o modo de acesso a memória em:

- Arquitetura de memória compartilhada com acesso uniforme (Uniform Memory Access - UMA);
- Arquitetura somente com memória cache (Cache-Only Memory Architecture - COMA);
- Arquitetura de memória distribuída com acesso não uniforme (Nonuniform Memory Access - NUMA).

Em arquiteturas NUMA, cada processador possui uma área da memória, porém com um único espaço para endereçamento. Desta forma, qualquer processador acessa qualquer parte da memória. Na arquitetura COMA, a memória fisicamente remota é acessada apenas pela cache[20].

2.8.1 *Symmetric Multiprocessor* (SMP)

Com a queda dos preços e com a crescente demanda por desempenho o multiprocessamento simétrico tem se tornado um dos modelos mais comuns ultimamente [20]. O termo SMP é relacionado ao comportamento do sistema operacional e da arquitetura em questão. A arquitetura envolve mais de um processador utilizando a mesma memória. Todos os processadores estão ligados através de um barramento à memória e a outros dispositivos. Os processadores nesta arquitetura também possuem uma capacidade de computação equivalente. A tarefa do sistema operacional é controlar para que não ocorram anomalias referentes ao acesso simultâneo a dispositivos e a memória [38].

Atualmente, a maioria dos sistemas operacionais tem suporte ao modelo SMP [20]. Embora o sistema operacional possua este recurso ainda existem muitos programas que não exploram o paralelismo como forma de melhorar o desempenho [8].

2.9 Trabalhos Relacionados

O desenvolvimento de sistemas para cálculo de fluxo ótico embarcado data seu início do começo da década de 80. Porém, aplicações embarcadas para tal são muito mais recentes. Devido o fato de a visão computacional demandar uma carga computacional elevada, o seu uso em sistemas embarcados gera uma problemática

ainda maior. Neste capítulo são explicitados alguns trabalhos que serviram como base a este. Embora alguns trabalhos não se referenciem ao cálculo do fluxo ótico propriamente dito, questões como aplicações do fluxo e de métodos de paralelização foram inspiradas nestes artigos. Mesmo sendo citados trabalhos que calculam o fluxo ótico embarcado, estes se utilizam de técnicas diferentes das utilizadas neste projeto, o qual utilizou o algoritmo de Lucas e Kanade para sua implementação.

2.9.1 Uma estimação confiável do fluxo ótico em tempo real baseado em FPGA´s

Este artigo trata de uma técnica para o cálculo do fluxo ótico combinado a detecção de movimento. Esta combinação traz um ganho no sentido de reduzir o custo computacional do cálculo dos vetores do fluxo se comparado às técnicas tradicionais. Este ganho torna possível a implementação do cálculo e aplicações de tempo real. Neste artigo também é implementado um pipeline para a estimação de vetores do fluxo ótico. Ao fim no artigo a implementação atingiu a taxa de 156 fps para imagens de 785x576 utilizando apenas uma FPGA [39]. Vale ressaltar que o artigo utilizou técnicas diferentes deste trabalho para o cálculo do fluxo ótico.

2.9.2 Implementação de um Algoritmo para cálculo em tempo real do fluxo ótico e sua utilização em estabilização de video

Neste artigo é apresentado um procedimento simplificado para o cálculo do fluxo ótico e sua implementação em hardware utilizando FPGA´s. Modificando o algoritmo de pareamento de blocos e se baseando na função de correlação normalizada sem multiplicação e divisão foi reduzida taxa de processamento do algoritmo. As taxa de erro foram verificadas e o sistema foi prototipado em uma FPGA para a estabilização de imagens provenientes de uma câmera CMOS. Os resultados foram satisfatórios para implementações de tempo real como navegação autônoma[40].

2.9.3 Evolução do Método Não Parametrizável de Harris para Detecção de Cantos: Uma Abordagem Distribuída

Muitos métodos para detecção de *features* são descritos na literatura. Um dos primeiros detectores foi o de Harris detectando cantos na imagem.

Desta forma, este trabalho apresentou uma versão paralela do algoritmo de Harris. Os softwares foram modelados como *loops* independentes[9].

Quando o tempo de execução entre os *loops* é igual o escalonador estático atinge melhor desempenho. Para este trabalho foi desenvolvido um escalonador para distribuir dinamicamente e de forma heterogênea uma grande carga computacional.

Para testar o escalonador associado ao algoritmo de Harris em paralelo. Foi utilizado um conjunto de 24 processadores Xeon dual-core[9]. Neste trabalho percebe-se o potencial paralelo que este algoritmo tem devido sua baixa dependência de dados. No gráfico da Figura 2.23 pode-se perceber o *speedup* do sistema crescendo proporcionalmente ao número de processadores.

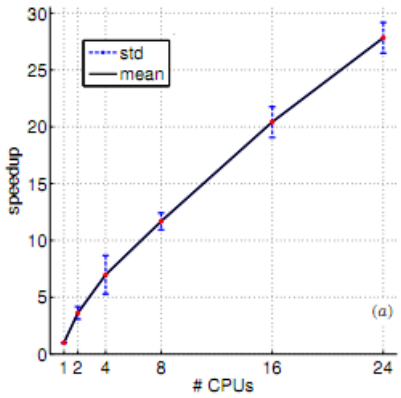


Figura 2.23: Desempenho do escalonador executando o algoritmo de Harris em paralelo[9].

2.9.4 Processamento de Imagem Multi-camada para Detector de Cantos de Harris Multiescalar em uma Implementação Digital

Neste trabalho foi desenvolvido um co-projeto hardware e software que explora o paralelismo do algoritmo de Harris [26, 9] tanto utilizando uma arquitetura paralela como *pipelines*.

Focado em processos multi-camadas (*multilayer*) para processamento de imagem, este trabalho explora o fluxo de dados dos algoritmos como forma de evitar a espera de todos os passos anteriores. Um novo esquema de *pipeline* é proposto. Como forma de verificação o algoritmo de Harris multi-camada em conjunto com uma aplicação de reconhecimento de padrões completa o experimento de tempo-real do co-projeto hardware e software.

Ao fim do trabalho pode-se concluir que para o processamento em tempo-real de imagens a capacidade computacional do *hardware* é notoriamente melhor do que a de um software com a mesma aplicação sendo executado em um PC. Após a paralelização do algoritmo foi possível atingir uma taxa de processamento de 46 *frames* por segundo em execução utilizando uma FPGA Cyclone 2S35 da Altera.

Na Tabela 2.4 pode-se perceber o ganho de desempenho que o hardware embarcado na FPGA tem em relação ao computador de uso geral.

	PC	FPGA
Operação de abrir e fechar	4.9 fps	60 fps
Operação de suavização	5.3 fps	60 fps
Detector de cantos de Moravec	6.2 fps	60 fps
Detector de cantos de Harris	4.6 fps	55 fps
Detector de cantos multi-escala de Harris	2.1 fps	46 fps

Tabela 2.4: Tabela de comparação do desempenho do sistema gerado [9].

2.9.5 Projeto de um Sistema de desvio de Obstáculos para Robôs Móveis baseado em Computação Reconfigurável

Visando o crescimento cada vez maior da robótica móvel é necessário que exista o desenvolvimento de robôs pessoais de baixo custo. Tendo isso em vista, este trabalho propõe que invés de sensores caros como lasers, o robô utilize sensores mais baratos como câmeras[10]. Porém, frequentemente o uso de sensores mais baratos acarreta em um maior esforço computacional. Como os robôs pessoais interagirão com as pessoas dentro de casa e em ambientes abertos eles não podem ser muito grandes nem consumir muita energia.

Desta forma, este problema se encaixa no contexto de sistemas embarcados. Na Figura 2.24 pode-se observar um robô Pioneer acoplado a uma FPGA que o controla. Como forma de contornar o problema de consumo e tamanho este trabalho utiliza a computação reconfigurável.

Figura 2.24: Robô Pioneer 3 DX com sistema embarcado baseado em FPGA do Laboratório de Computação Reconfigurável ICMC-USP[10].

Utilizando o paralelismo inerente à aplicações em hardware, é possível operar um computador configurável a uma frequência muito menor que um computador

de uso geral. Desta forma, com o objetivo de contornar as restrições inerentes aos algoritmos para o cálculo do fluxo ótico, neste trabalho foi desenvolvido um projeto em hardware que utiliza o fluxo ótico como forma de desviar de obstáculos[10]. O fluxo ótico foi utilizado como forma de identificar obstáculos através de uma câmera monocular conectada a uma FPGA. Por fim foi desenvolvido um sistema em hardware que no futuro pode ser utilizado como um sensor para robôs controlados por FPGA´s.

2.9.6 Proposta de uma Arquitetura Multi-core para Detecção de Pedestres Baseada em Câmeras

Uma das aplicações embarcadas que têm ganhado cada vez mais importância é o ADAS (*Advanced Driver Assistance Systems*). Tais sistemas têm como objetivo auxiliar o motorista em diversas atividades no trânsito, diminuindo a incidência de acidentes e sinistros. Em outras palavras, o ADAS é um sistema que tem o objetivo de fornecer informações relevantes ao motorista sobre condições operacionais e ambientais ao redor do veículo. Um dos tipos de aplicação de ADAS é a detecção de pedestres.

Uma das formas de implementar um sistema de detecção de pedestres é utilizando uma câmera. Embora o processamento de imagens tenha um alto custo computacional, um ADAS necessita de um baixo consumo de energia. Uma das formas de contornar este problema é através de arquiteturas *multi-core* [13].

Este projeto leva em consideração o uso de sistemas multi-processados como forma de diminuir o consumo de energia, uma vez que o projeto está inserido no contexto embarcado. O trabalho desenvolvido nesta dissertação tem como objetivo contribuir com o sistema desenvolvido em [13] no que diz respeito à estabilização de vídeo utilizando fluxo ótico. Na Figura 4.9 pode-se ver o esquema de blocos do sistema desenvolvido em [13].

Capítulo 3

Metodologia de Desenvolvimento

3.1 Material e Métodos

3.1.1 OpenCV

A biblioteca OpenCV foi desenvolvida pela empresa Intel com o objetivo de tratar, à princípio, aplicações de alto custo computacional. Depois se tornou uma ferramenta que proporciona uma estrutura de visão computacional mais acessível tendo como principais objetivos:

- A pesquisa avançada em visão, provendo uma infraestrutura básica e aberta para tal.
- Disseminar conhecimentos de visão para que desenvolvedores pudessem construir em cima da biblioteca utilizando-a como base.
- A criação de aplicações comerciais baseadas em visão, devido sua licença que, apesar de ser aberta, não obriga que as aplicações criadas com a biblioteca também o sejam.

Para efeito de comparação dos resultados será utilizada a biblioteca OpenCV que já implementa vários algoritmos de visão computacional, dentre eles o Lucas e Kanade otimizados[41].Na Figura 3.1 tem-se um exemplo de um fluxo ótico calculado apenas para pontos de interesse. O "Frame 1" representa a imagem anterior com os pontos de interesse destacados em verde e o "Frame 2" é a imagem atual com os vetores de movimento destacados em vermelho.

Figura 3.1: Imagem do fluxo ótico sendo calculado utilizando a biblioteca OpenCV

3.1.2 Active-HDL Aldec

O Active-HDL é uma solução para criação e simulação de sistemas baseados em FPGA´s. Nesta ferramenta é possível desenvolver todo o sistema desde a parte de implementação até a fase de documentação, passando pelas fases de síntese e *place&route* (programação da FPGA). Tal característica tem como vantagem o uso de uma mesma plataforma em todas as fases do projeto[42].

Para este projeto ele foi utilizado para simular o bloco de *hardware* responsável pelo cálculo do fluxo ótico.

3.1.3 ModelSim

A ferramenta de simulação ModelSim possibilita a simulação de blocos de *hardware* em linguagens diferentes. Ele é um dos simuladores suportados pelo Quartus II (ferramenta de síntese) e nele existem bibliotecas prontas para a simulação em cada placa da Altera.

Com esta ferramenta é possível simular o processador Nios II executando um software por inteiro e observar seu comportamento [43]. Desta forma, é possível perceber problemas de integração e de sincronismo como, por exemplo, em uma instrução customizada.

3.1.4 Bluespec

A linguagem Bluespec SystemVerilog [11] foi criada no MIT (*Massachusetts Institute of Technology*) pelo professor Arvind, visando acelerar a implementação de sistemas digitais altamente confiáveis. Devido o fato da ferramenta de produzir sistemas confiáveis em tempo muito mais hábil do que as linguagens RTL, novas implementações de problemas, antes visto como inviáveis em hardware, vem sendo

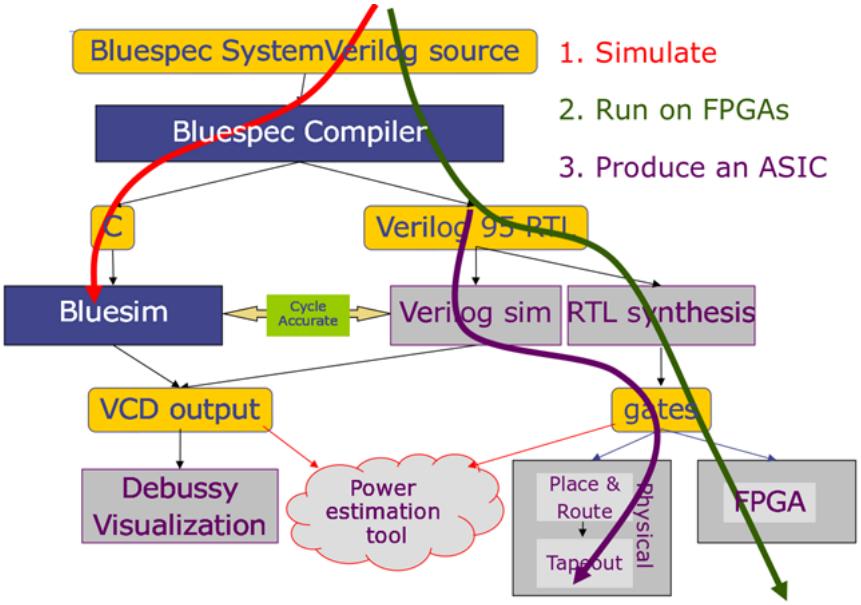


Figura 3.2: Fluxo de desenvolvimento em Bluespec [11].

produzidas nesta nova linguagem. A principal característica desta linguagem, além de ser uma HLS (*High-Level-Synthesis*), é ter o seu código desenvolvido 100% sintetizável em hardware.

O sistema de desenvolvimento Bluespec inclui um compilador, um simulador (Bluesim) e um ambiente de desenvolvimento integrado, como pode ser observado na Figura 3.2. O compilador recebe o código-fonte em Bluespec System Verilog (BSV) podendo a partir deste gerar um código C para simulação ou Verilog para síntese/simulação posterior.

Uma das maiores vantagens do Bluespec é a velocidade na simulação do código BSV através do Bluesim. Outra vantagem inerente ao Bluespec é a existência de transações atômicas (*rules*) o que evita conflitos ao acessar interface de diferentes módulos diminuindo erros de programação.

O Bluespec "encapsula" toda a lógica de controle de recursos compartilhados o que torna uma opção muito viável para diminuir o tempo de desenvolvimento de um sistema. Na Figura 3.2 pode-se observar o fluxo de desenvolvimento em Bluespec. Neste fluxo é possível perceber a versatilidade no que diz respeito a simulação e sínteze do *hardware* através da linguagem.

3.1.5 Arquitetura Sequencial

O desenvolvimento de uma arquitetura não paralela também será feita, pois se faz necessária a implementação de um código sequencial para comparação com o algoritmo paralelo. Para o desenvolvimento de todas as arquiteturas será utilizada a ferramenta SOPCBuilder da empresa Altera que possibilita a elaboração de

arquiteturas sequenciais e paralelas utilizando o processador Nios II.

3.1.6 *gprof* e *Profiling*

O *profiling* de um programa permite identificar quanto tempo é gasto e quais funções são processadas durante sua execução. Desta forma, pode-se determinar quais partes do programa levam mais tempo para serem executadas do que o esperado, sendo estas candidatas a uma nova implementação visando melhor desempenho.

Pode-se identificar também quais funções são executadas com mais frequência, ajudando detectar problemas que poderiam não ser percebidos [44]. Aos programas que realizam essa tarefa dá-se o nome de *profilers*, como exemplo desse tipo de aplicação tem-se o *gprof*.

Como o *profiler* utiliza informações coletadas durante a execução do programa é necessário seguir alguns passos:

- Compilar e "linkar" o programa com o *profiling* ativado;
- Executar o programa para gerar um arquivo com os dados de *profiling*;
- Executar o *gprof* para analisar o arquivo de *profiling*.

Existem várias formas de executar o *gprof*; como forma de avaliar o tempo gasto em cada função (*flat profile*); identificar quais funções chamam outras funções e quais são elas (*call graph*); quantas vezes executou cada linha do programa (*annotated source*), etc. Para este trabalho será utilizada a ferramenta *gprof* como forma de identificar "gargalos" do sistema e funções candidatas à reescrita.

3.1.7 Kit de Desenvolvimento DE2-70

Como ambiente de desenvolvimento este trabalho leva em consideração a placa DE2-70 da Terasic a qual contém uma FPGA do modelo Cyclone® II EP2C70F896C6, saída de vídeo (VGA 10-*bit* DAC), entrada de vídeo (NTSC/PAL/Multi-format) e USB 2.0[12]. Um esquema dos componentes e do *layout* da placa pode ser observado na Figura 3.3.

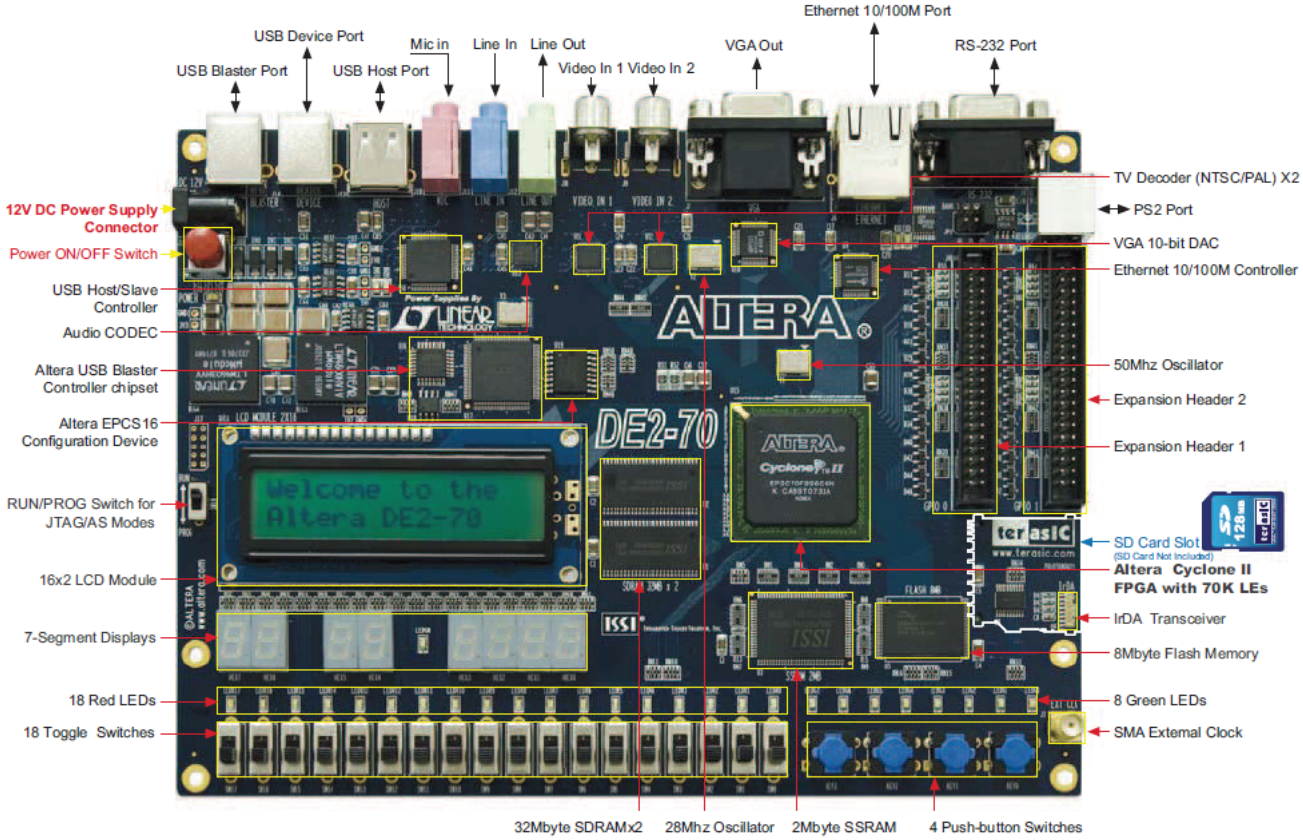


Figura 3.3: Placa DE2-70[12].

Com a DE2-70 é possível desenvolver aplicações como televisões, câmeras digitais, gravadores de áudio dentre outras aplicações áudio-visuais todas embarcadas na FPGA Cyclone II [12]. Um esquema de como a FPGA está conectada à placa pode ser observada na Figura 3.4.

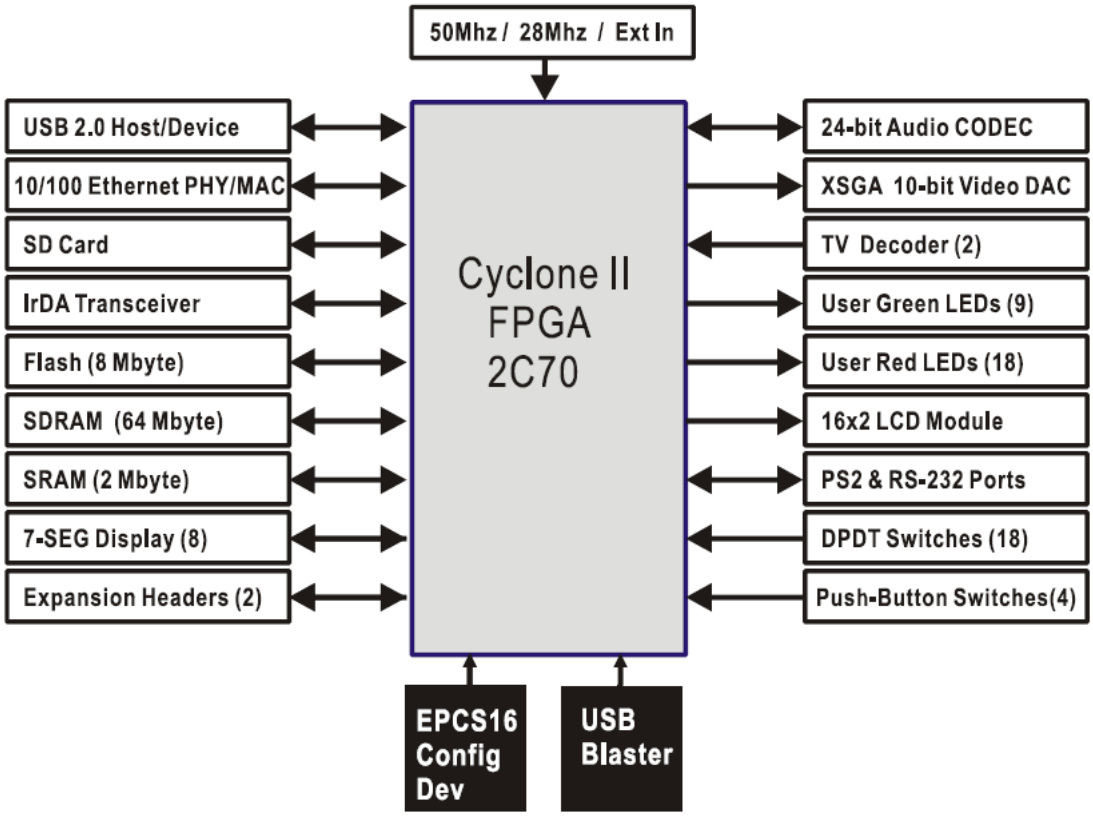


Figura 3.4: Esquema da Cyclone II conectada à DE2-70[12]

A programação da FPGA é feita através do bloco "USB *Blaster*" que se comunica com o *driver* de mesmo nome instalado em computador com o ambiente de desenvolvimento Quartus II da Altera.

3.2 Considerações finais

As ferramentas de simulação tiveram grande importância na validação e elaboração das máquinas de estado que serviram de controle do bloco desenvolvido em Bluespec. A biblioteca OpenCV foi utilizada para prototipação do algoritmo e serviu para experimentos teste quanto a estabilização do vídeo. A arquitetura sequencial em conjunto com as ferramentas de profilling foi utilizada para identificar o gargalo de processamento no algoritmo de inversão de matrizes. Embora a arquitetura paralela tenha sido descartada, seu desenvolvimento foi necessário para a percepção de que a implementação do algoritmo de fluxo ótico não poderia ser feita no Nios II atendendo oa requisitos de tempo real.

Capítulo 4

Implementação do Hardware do Fluxo Ótico e Resultados

No desenvolvimento inicial desse projeto, adotou-se a filosofia de implementação do fluxo ótico através do co-projeto de hardware e software. Entretanto, no decorrer do mesmo, após diversas tentativas de implementação, chegou-se a conclusão de que para atender os requisitos de tempo real da aplicação robótica descrita, o sistema deveria ser implementado em 100% de hardware. Deste modo, a solução final para a implementação do fluxo ótico foi baseada somente em hardware. As diversas tentativas de implementação são descritas a seguir.

No algoritmo de Lucas e Kanade é necessária a resolução de um sistema linear para determinar as componentes de cada vetor velocidade. Para a determinação dos sistemas lineares será necessária uma forma otimizada para inverter matrizes.

Desta forma, foi desenvolvida uma arquitetura paralela utilizando três Nios II com uma memória *on-chip* compartilhada. Esta arquitetura foi desenvolvida para o cálculo de matrizes inversas e mostrou resultados satisfatórios com relação ao ganho em velocidade de processamento. A inversão de matrizes é um problema crítico para bom funcionamento do sistema. A proposta inicial foi elaborar uma arquitetura para explorar o paralelismo do algoritmo Lucas-Kanade e utilizar o cálculo da inversão de matrizes para avaliação da mesma. Para a comparação dos resultados várias arquiteturas para inversão de matrizes foram desenvolvidas. Os dados de cada arquitetura podem ser observados na Tabela 4.1.

Versão	Modelo NiosII	% E.L.	FPU	Cache
p.1	NiosII/e	12	-	-
p.2	NiosII/f	49	X	instrução

Tabela 4.1: Dados das arquiteturas paralelas

Na Figura 4.1 é possível observar uma ilustração de como foi organizado o

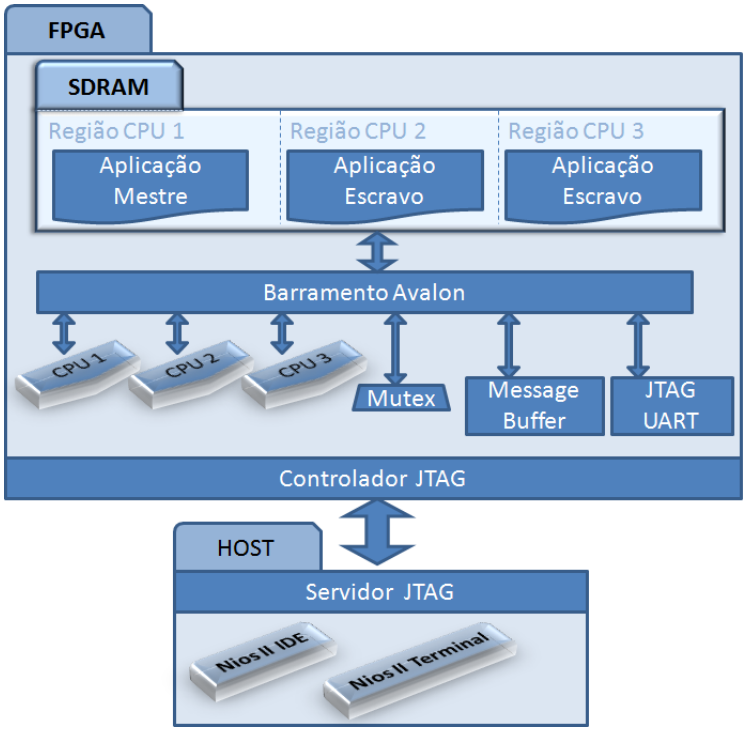


Figura 4.1: Arquitetura paralela utilizando três Nios II

sistema. O CPU 1 no desenho esquemático representa o processador *master* o qual é responsável por acionar os outros dois processadores em pontos críticos do algoritmo.

Analisando o problema observou-se que a memória on-chip compartilhada era de apenas 1024 bytes e não era suficiente para todas as variáveis utilizadas neste projeto. Então foi estudada uma forma de utilizar a memória SDRAM com 32 MBytes. O kit DE2-70 vem com dois módulos desta memória. Como a SDRAM1 estava sendo utilizada para o *software*, optou-se por utilizar o segundo módulo, SDRAM2 como memória compartilhada. Para efeito de determinar o gargalo do sistema foram elaboradas várias configurações de Nios II que executam o código de inversão em sequencial. Na Tabela 4.2 pode-se observar as configurações de cada sistema utilizado.

Versão	Modelo NiosII	%E.L.	FPU	Cache
s.1	NiosII/e	7	-	-
s.2	NiosII/e	16	X	-
s.3	NiosII/s	18	X	instrução
s.4	NiosII/f	18	X	instrução
s.5	NiosII/f	19	X	instrução e dados

Tabela 4.2: Dados das arquiteturas sequenciais

Foram realizados vários *profilings* dos sistemas levando em consideração matrizes quadradas de ordem 3, 6, 9, 12, 15 e 18. Um gráfico comparativo entre as

versões pode ser observado na Figura 4.2.

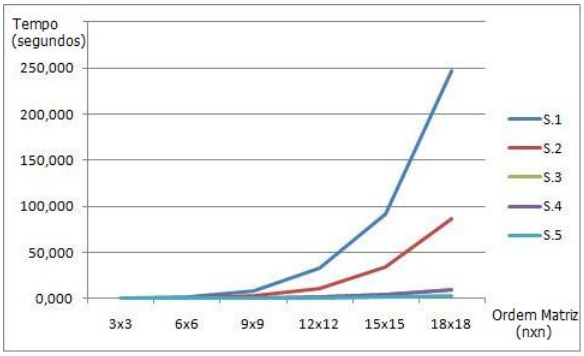


Figura 4.2: Gráfico comparativo de todas versões sequenciais

Após a extração do tempo percebeu-se que a função que tomava mais tempo em todas as versões do sistema era a *multiplica_ matriz*. Esta função é responsável pela multiplicação de duas matrizes *nxn*, e foi possível perceber que à medida que se aumenta a ordem das matrizes o tempo de execução cresce de forma exponencial. Na Figura 4.3 pode-se perceber pelo gráfico a porcentagem do tempo de execução da função em relação ao tempo total do sistema.

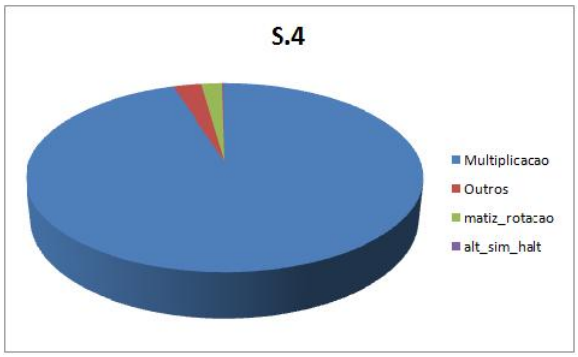


Figura 4.3: Porcentagem do tempo de execução utilizado pela multiplicação na versão s.4

A multiplicação de matrizes é um problema conhecido por ter uma baixa dependência de dados. Um grafo de dependência de uma multiplicação de duas matrizes 3x3 pode ser observado na Figura 4.4. Como é possível perceber pelo grafo, no caso hipotético da existência de infinitos processadores, todo o problema, independente da ordem da matriz, pode ser resolvido em dois passos.

No fluxograma da Figura 4.5 pode-se observar como foi implementado o algoritmo de multiplicação de matrizes em paralelo.

Na versão p.2 do sistema não foi utilizada a *cache* de dados devido a problemas técnicos encontrados no seu uso em sistemas multiprocessados. Nos gráficos das Figuras 4.6 pode-se ver uma comparação entre as duas versões paralelas.

Como não foi utilizada uma cache de dados na versão p.2 seu resultado foi comparado com a versão equivalente com apenas um processador, a s.4. Um

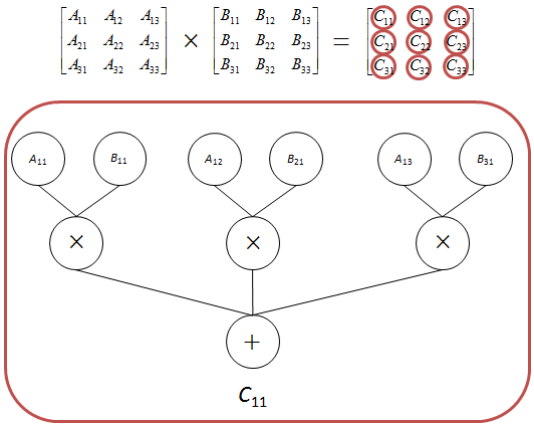


Figura 4.4: Grafo de dependência de dados da multiplicação de matrizes para C_{11}

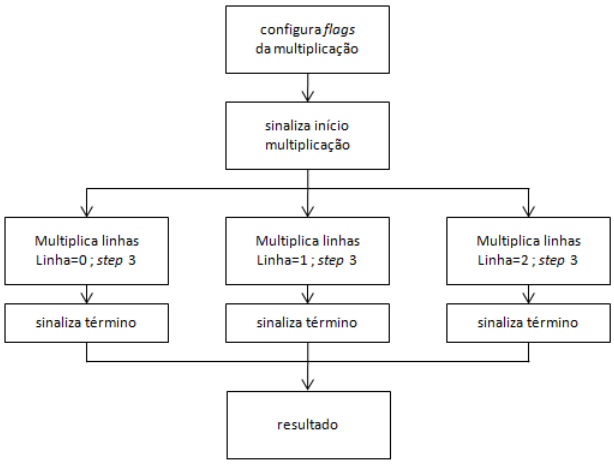


Figura 4.5: Fluxograma do algoritmo de multiplicação paralelo

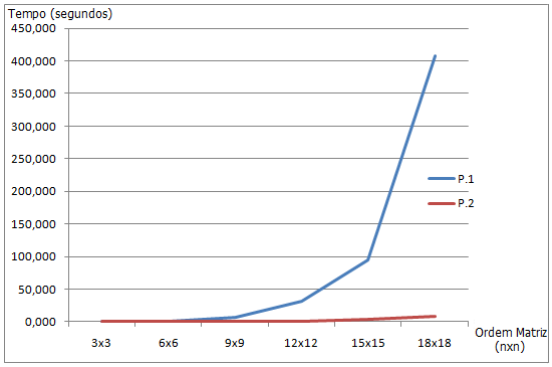


Figura 4.6: Gráfico comparativo entre as versões paralelas

gráfico comparativo das duas versões pode ser visto na Figura 4.7. Na Figura 4.8 um gráfico mostrando o *speedup* e a eficiência de cada processador utilizado na arquitetura paralela.

É possível perceber que no gráfico da Figura 4.8 o *speedup* diminui conforme se aumenta a ordem da matriz. Desta forma, com a tendência de queda do *speedup* é possível prever que em algum momento a versão paralela e a sequencial terão o

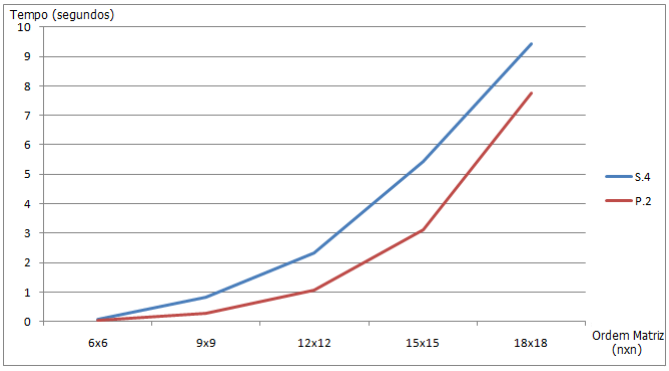


Figura 4.7: Gráfico comparativo entre a versão paralela e sequencial

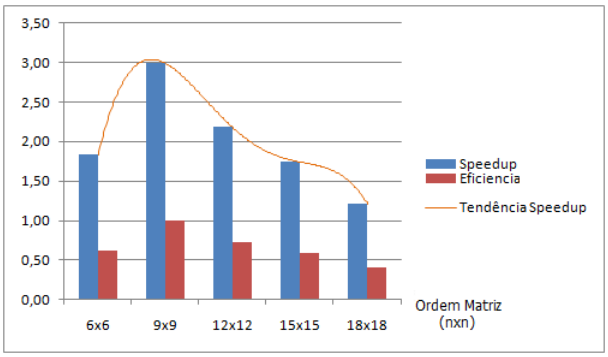


Figura 4.8: Speedup da versão paralela em relação à sequencial

mesmo tempo de execução.

No final do ano de 2011 o aluno de mestrado Leandro Martinez defendeu o projeto intitulado "Projeto de um sistema embarcado de predição de colisão a pedestres baseado em computação reconfigurável". Este projeto tinha como objetivo a construção de um sistema embarcado para detectar pedestres utilizando computação reconfigurável com captura de imagens através de uma única câmera acoplada a um veículo que trafega em ambiente urbano[13]. Um esquema do projeto como um todo pode ser visto na Figura 4.9.

O sistema desenvolvido funciona com alguns problemas devido ao estabilizador

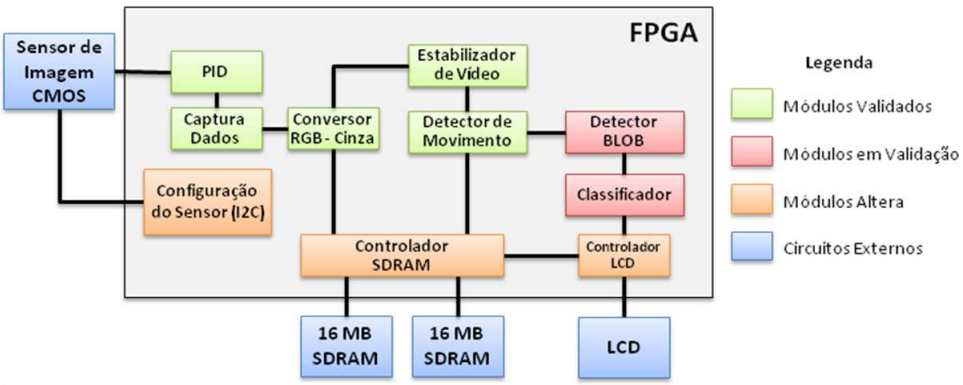


Figura 4.9: Estrutura do sistema de detecção de pedestres[13]

de vídeo. Em decorrencia da trepidação constante do asfalto é necessária uma estabilização da imagem como forma de que a detecção de movimento e o resto do sistema funcione de forma correta. No seu doutoramento o projeto será continuado como forma de aprimorá-lo.

Durante a elaboração da revisão bibliográfica foi encontrado o *site* de uma disciplina do MIT (*Massachusetts Institute of Technology*) ministrada pelo criador do Bluespec o professor Arvind [45][11]. Neste curso foi ressaltada a prática no uso do Bluespec para que, desta forma, os alunos tivessem a possibilidade de desenvolver projetos completos em um prazo relativamente curto. Em um destes trabalhos um dos grupos desenvolveu um bloco de hardware para o cálculo do fluxo ótico em uma imagem 64x64 em escala de cinza utilizando 8 bits para representação de um *pixel* [46]. Desta forma, foi pedido atravéz de e-mails o código deste bloco para a utilização como referência. Foi decidido integrar o cálculo do fluxo ótico com o sistema de [13] utilizando como base o bloco de hardware cedido pelo MIT.

Por se tratar de uma linguagem nova e de relativa complexidade o Bluespec a avaliação e adaptação do bloco apresentou algumas dificuldades. O código Bluespec desenvolvido é extenso (aproximadamente 1500 linhas de código) e para a sua análise por completo foi necessário um mês. Com o fim da análise percebeu-se que o sistema não visava uma aplicação real. Como não foi possível integrar o código Bluespec no sistema de [13] sua utilização como bloco de *hardware* do mesmo foi descartada. A maior parte do problema de integração se deu por conta da interface de controle criada pelo Bluespec. Com o objetivo de verificar e garantir a funcionalidade do *hardware* testes e simulações foram realizadas, sendo possível também descrever o comportamento de como o cálculo era realizado. Embora tenha sido gerado um *hardware* completamente funcional, para a sua utilização ainda era necessário um controle externo ao bloco de *hardware*. Uma vez gerado o código Verilog pelo o Bluespec é possível gerar um bloco que pode ser inserido em qualquer projeto do Quartus II da Altera. Na Figura 4.10 pode-se observar a representação do código Verilog como uma "caixa-preta" com apenas entradas e saídas.

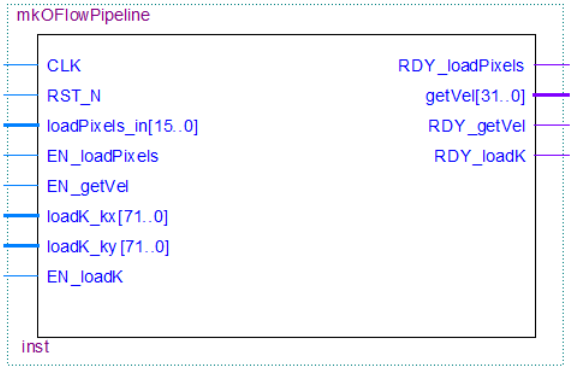


Figura 4.10: Representação do bloco gerado pelo Bluespec.

Por se tratar de um *hardware* completamente funcional é possível fazer a sua síntese para a placa de FPGA.

No trabalho de [13] as imagens utilizadas para a detecção de pedestres tem a resolução de 640x480. O bloco de *hardware* cedido apresenta apenas a resolução de 64x64. Após uma discussão com o realizador do trabalho concluiu-se que não faria sentido calcular o fluxo óptico para apenas uma pequena região da imagem. Desta forma, seria necessário alterar o código para a resolução de 640x480. Na Figura 4.11 é possível observar como ocorre o cálculo dentro do bloco.

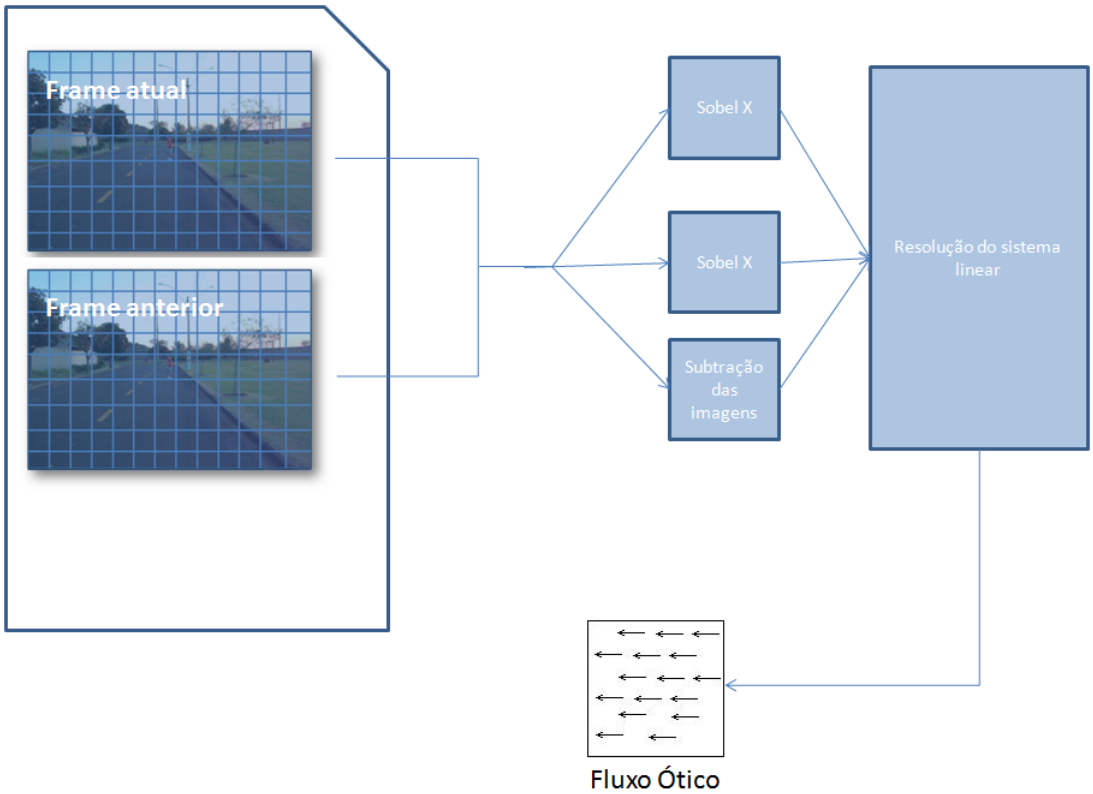


Figura 4.11: Representação do cálculo realizado pelo bloco.

O bloco Bluespec foi contruído como um *pipeline*. Desta forma, entre os estágios

são necessárias estruturas para sincronização, no caso deste foram utilizadas filas FIFO (*First In, First Out*). Com o aumento da resolução o tamanho destas filas crescem esponencialmente. Tal aumento na lógica traria a necessidade do uso de memórias externas, que, além de mais lentas, necessitam de uma lógica adicional para lidar com a memória pelo barramento Avalon (barramento de periféricos do Nios II). Por estes motivos esta solução foi descartada.

Inserindo o bloco de *hardware* dentro de um ambiente de *software* é possível manipulá-lo mais facilmente. O controle de memória é facilitado devido a interface com a memória já ser implementada. Desta forma, foi decidido que o bloco de *hardware* seria inserido como uma instrução customizada do Nios II.

Como o *hardware* será tratado como uma função em C o bloco pode ser controlado pelo próprio *software* no caso de uma instrução *Multi*-cycle explanada anteriormente na Seção 2.7.3. Por ser transparente ao *software* pode-se pensar em um sistema com vários núcleos do Nios II, cada um tratando em blocos de 64x64 duas imagens de tamanho MxN qualquer, com a única condição que tais imagens não ocupem um espaço maior do que o existente nas memórias oferecidas. Na Figura 4.12 é possível observar uma representação de como seria executado o processamento de duas imagens por uma arquitetura multiprocessada com dois núcleos customizados do Nios II.

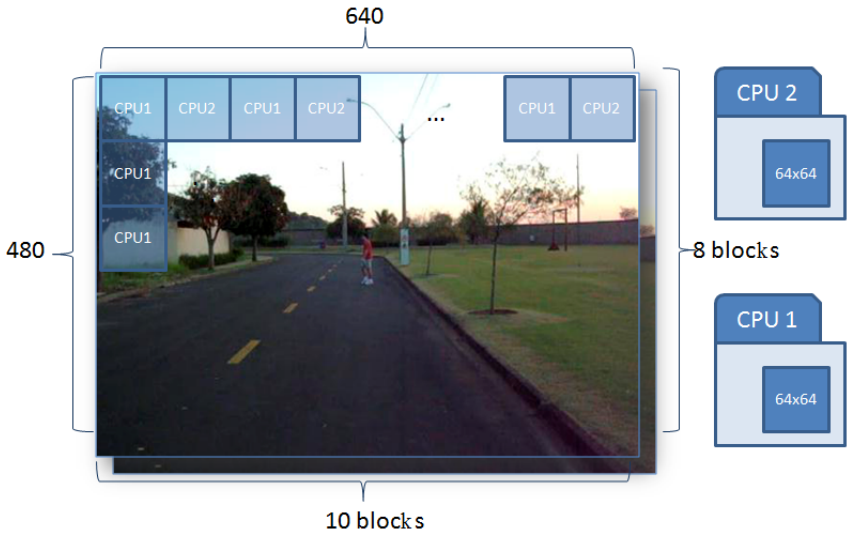
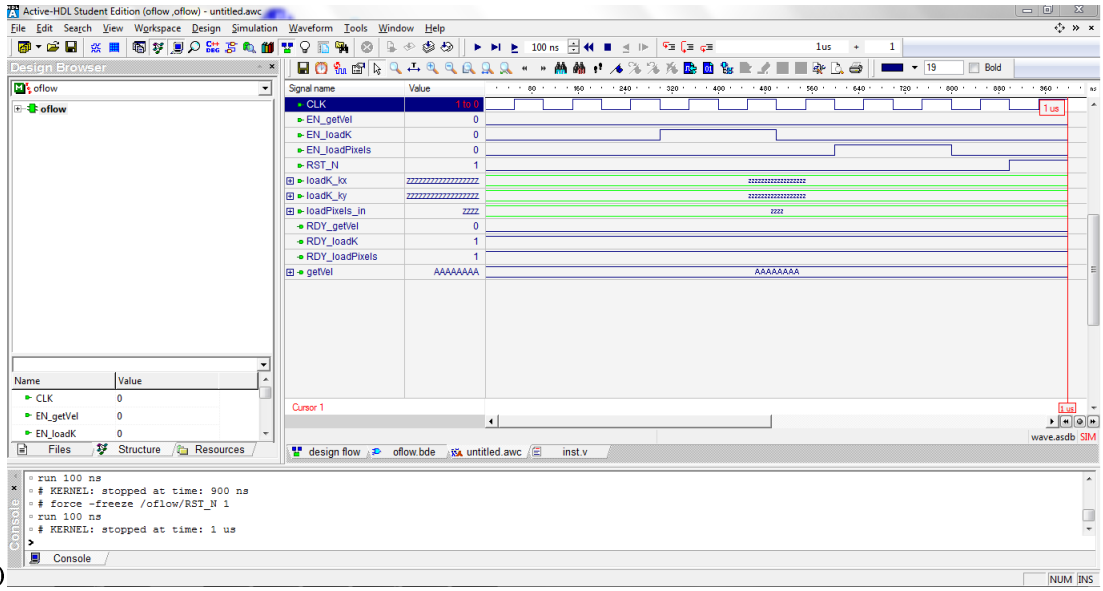


Figura 4.12: Exemplo do processamento por uma arquitetura multiprocessada.

Tendo isso em vista, foi proposto o desenvolvimento de um Nios II customizado como forma de tornar o cálculo do fluxo de uma imagem de 64x64 possível. Uma vez elaborado este núcleo sua integração com outros núcleos se tornaria viável, podendo calcular o fluxo com uma resolução de 640x480. Outras vantagens do uso da instrução customizada são a comparação entre o co-projetohardware e software com uma implementação apenas do *software* e o carregamento de arquivos, por

exemplo, imagens.

Para a implementação da instrução customizada foi necessário simular o bloco gerado representado na Figura 4.10 como forma de entender como os sinais se comportam. Com este objetivo, foi utilizada a ferramenta de simulação do ActiveHDL. Esta ferramenta possibilita, a cada ciclo de *clock,* alterar os sinais de entrada. Também é possível visualizar as mensagens retornadas por funções do Verilog simulado como, por exemplo, a função *$display()* equivalente ao *printf()* da linguagem C porém só usada na simulação. Na Figura 4.13 é possível observar uma parte da simulação do *hardware.*



D

Figura 4.13: Simulação utilizando o ActiveHDL.

Com o comportamento do código mapeado pelos sinais de entrada e saída, foi possível elaborar uma máquina de estados que controlaria o bloco de *hardware.* Foi decidido que esta máquina de controle possuiria 4 estados para controlar o pipeline do bloco:

- *Load Kernel*: é um estado de inicialização da máquina onde os *kernels* das derivadas espaciais (Sobel) nas coordenadas X e Y são carregados para o bloco de *hardware.*

- *Load Pixel*: os *pixels* são carregados no bloco em pares, sendo o primeiro da imagem atual e o outro da imagem anterior como forma de calcular seu deslocamento no espaço. A imagem é considerada como um vetor e não como matriz neste caso.

- *get Vel*: conforme o *pipeline* vai sendo populado é possível obter os primeiros vetores do fluxo ótico antes de ter carregado todo o par de imagens. Desta forma ele existe para "desempilhar" o vetor que já está pronto na fila.

- *NOP*: nele o bloco é chamado sem nenhuma função. Por se tratar de *pipeline* de tempos em tempos é necessário esperar o processamento dos *pixels* que já foram inseridos, e caso ainda não existam vetores do fluxo ótico a serem lidos o bloco é chamado como "nenhuma operação" (*No Operation*).

Na Figura 4.14 pode-se observar a representação gráfica da máquina.

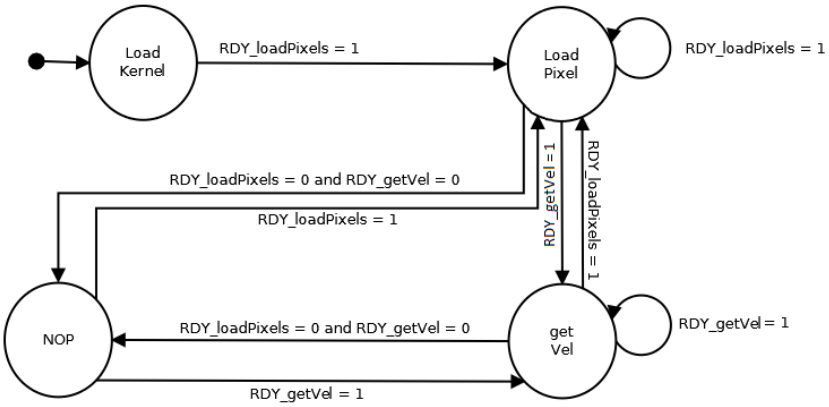


Figura 4.14: Representação gráfica da máquina de estados.

Uma vez elaborada a máquina que dita o comportamento da unidade de controle do bloco, é possível a integração com o NiosII como uma instrução customizada. Porém, a primeira implementação da instrução customizada apresentou erros de execução ao ser integrada. Embora fosse executada, a instrução não retornava um valor válido. Após várias novas implementações e novas execuções não foi possível corrigir o erro. Cogitou-se que o erro era proviniente da comunicação do Nios com a instrução. Foi necessário simular tanto o *software* quanto o *hardware* para que fosse possível entender como estava ocorrendo esta comunicação e qual seria o erro. Com este propósito foi necessário a utilização de um outro simulador. O ModelSim é o simulador que faz parte do pacote de ferramentas da Altera. Nele é possível simular o Nios II executando o *software* que é carregado para a memória. Mesmo assim não foi possível ainda encontrar o erro na sincronização. Acreditou-se que o erro encontra-se na sincronização dos sinais da instrução. Na Figura 4.15 é possível observar a tela ModelSim simulando o Nios II.

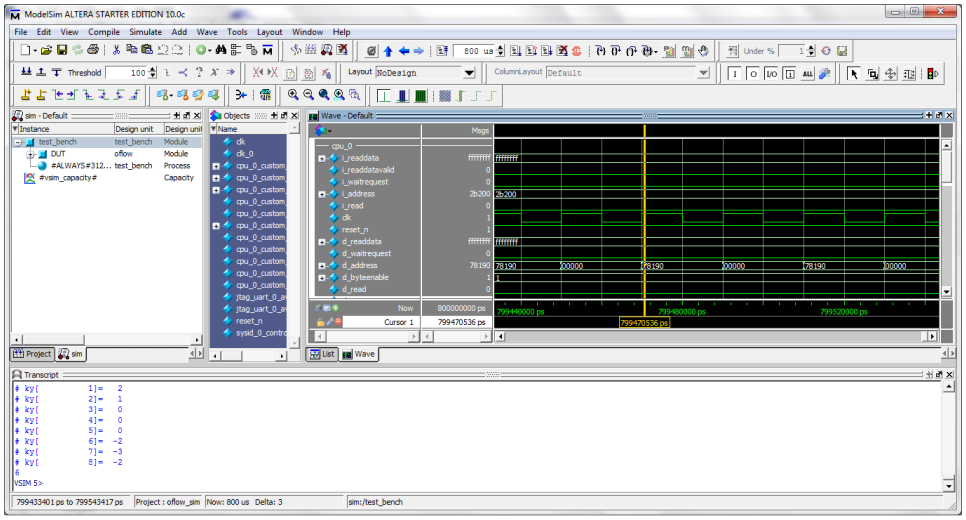


Figura 4.15: Simulação utilizando o ModelSim.

Um método interessante para a sincronização de sinais de um sistema é o uso de máquinas de estado. O Active-HDL proporciona um ambiente de modelagem que as utiliza para posterior implementação em hardware. Desta forma, modelou-se uma máquina de estado utilizando tal ferramenta com o objetivo de controlar o bloco de hardware. Na Figura 4.16 pode-se observar a máquina de estado modelada pela ferramenta.

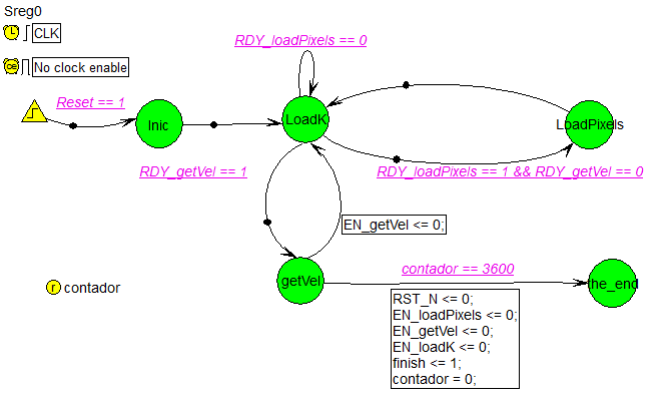


Figura 4.16: Máquina de estado modelada pelo Active-HDL.

É possível modelar ações tanto nos estados quanto nas transições de forma a tornar mais prática a utilização. As ações são descritas em Verilog ou VHDL, neste caso elas estão em Verilog. Para a Figura 4.16 foram omitidas algumas ações.

Após elaborada a MEF (Máquina de Estados Finitos) foi simulada considerando o estudo anterior feito do bloco para o cálculo de fluxo ótico.

Outro ambiente proporcionado tanto pelo Active-HDL quanto pelo Quartus da Altera é o de modelagem através de blocos. Uma vez importado o código de descrição de hardware para a ferramena é possível criar uma “caixa preta” e

interligá-la a outros blocos utilizando ou não lógica combinacional. Isto proporciona uma melhor separação entre os dois blocos podendo simular e acompanhar os valores de cada barramento de interconexão, sendo possível perceber melhor como ocorre a troca de dados de controle. Na Figura 4.17 observa-se uma ilustração de como o bloco de cálculo de fluxo ótico se interliga com a máquina de controle, denominada de MEF1.

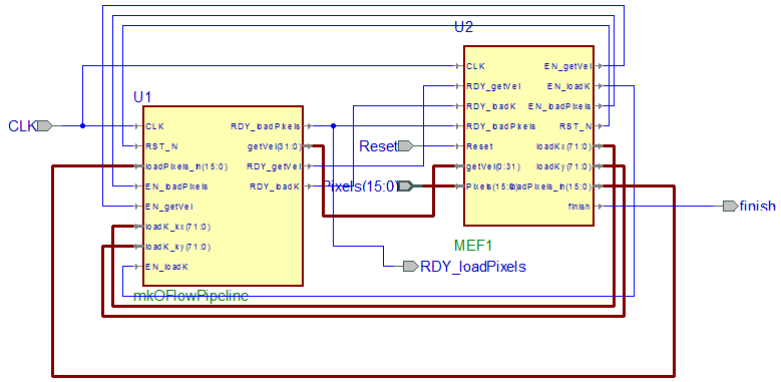


Figura 4.17: Diagrama de blocos do sistema para o cálculo do fluxo ótico de uma imagem de 64x64.

Com os dois blocos interligados agora é possível simular o sistema. Uma simulação do sistema mostrou que a ligação funcionava muito bem com duas imagens aleatórias.

Foi constatado através de simulações que a memória gerada pelo Quartus necessita de dois ciclos de *clock* para leitura. Desta forma, é necessário que para todas as vezes que o bloco que calcula o fluxo óptico necessite de um novo par pixels aguarde este tempo para ler o *pixel* da memória. A máquina de estados que controla foi elaborada levando isso em consideração. Um esquema de tal máquina pode ser observado na Figura 4.18.

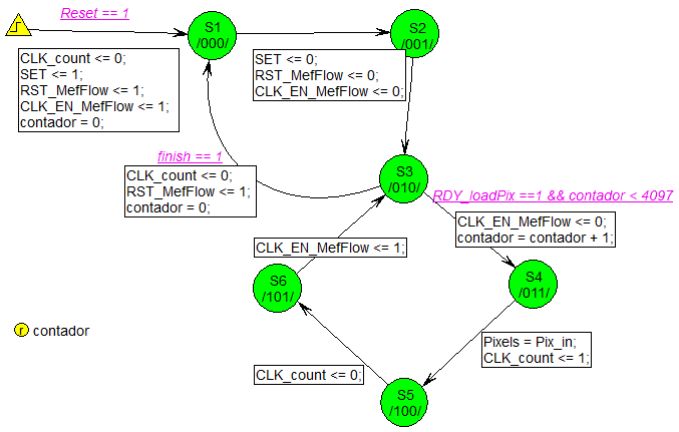


Figura 4.18: Máquina de estado que controla o acesso a memória.

Esta máquina se comunica com o bloco basicamente por dois sinais o 'finish' e

o 'RDY_loadPixels'. O sinal de 'finish' sinaliza o término do cálculo para o bloco de 64 bits. Já o 'RDY_loadPixels' sinaliza quando que o bloco necessita de um novo par de pixels oque gera uma interrupção no bloco.

O sinal 'CLK_EN_MefFlow' é responsável por gerar a interrupção. Para todas as vezes que é necessária uma leitura da memória o sinal vai para 0 o que inibe o *clock* do bloco. Isto faz com que ele pare até que o novo par válido de *pixels* esteja pronto para ser lido. Na Figura 4.19 pode ser observado o controlador da memória conectado no bloco que calcula o bloco de 64x64. O bloco 'stratixiv_and2' é uma porta lógica *and* que torna possível a interrupção do *clock* do bloco que calcula o fluxo óptico.

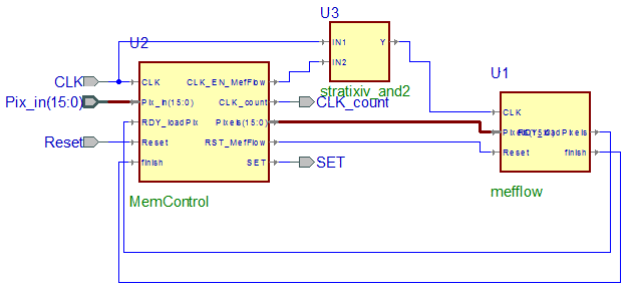


Figura 4.19: Controlador de memória ligado ao bloco de cálculo de fluxo ótico.

Como os pixels são lidos em sequência não é necessária uma busca mais aprimorada na memória. Sendo assim, o endereço da memória é constantemente incrementado em uma unidade. Com tal objetivo foi inserido na porta de endereço da memória um contador (bloco 'contador0') que começa em zero e termina no valor de 32768 (quantidade de pixels em uma imagem de 420 x 680). Os sinais de 'SET' e 'CLK_count' são responsáveis por controlar o contador. O sinal de 'SET' inicializa o contador em 0 e o 'CLK_count' é o clock do mesmo. de O esquema do sistema completo conectado na memória pode ser observado na Figura 4.20.

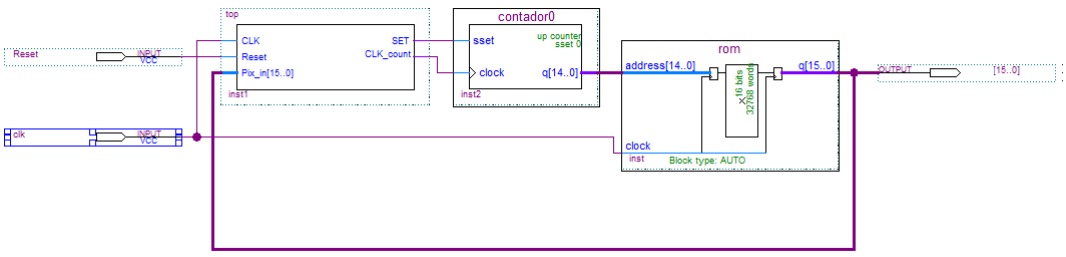


Figura 4.20: Sistema ligado à memória.

Logo após a elaboração de cada bloco que compõe o sistema foi realizada uma simulação do mesmo. O bloco mais importante no sistema final é o bloco denominado 'top'. Neste bloco é tratada tanto a comunicação com o bloco que

calcula a secção de 64x64 da imagem, como o controle da memória. Esta estrutura pode ser observada na Figura 4.19.

Na simulação é possível perceber que o bloco se comunica sem erros. Durante a simulação é possível ressaltar quatro fases importantes do sistema

Na inicialização o bloco é preparado para o cálculo. O carregamento dos kernels, o sinal de 'reset' são manipulados nesta fase com o objetivo de iniciar a execução do cálculo para a secção de 64x64 da imagem. O sinal de 'SET' inicializa o contador da memória em zero. Na Figura 4.21 estão marcados os sinais manipulados nesta fase.

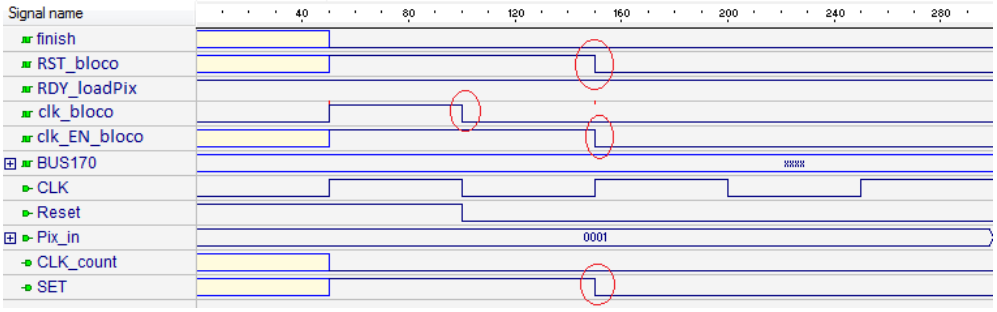


Figura 4.21: Simulação da inicialização do sistema (escala em nanosegundos).

O *clock* do bloco de cálculo do fluxo ótico ('clk_bloco') é controlado pelo sinal 'clk_EN_bloco'. Isto sincroniza do cálculo para que ele não seja executado utilizando pixels de valor inválido.

Cada acesso a memória carrega um par de *pixels*, um de cada frame. Como a memória tem um atraso de dois ciclos de clock é necessário parar o cálculo para carga dos *pixels*. O primeiro carregamento é feito junto com a carga dos *kernels*. Desta forma, todos os carregamentos posteriores tratam apenas dos *pixels*. Na Figura 4.22 é exemplificado, consecutivamente, a primeira e as posteriores cargas de *pixels* no sistema.

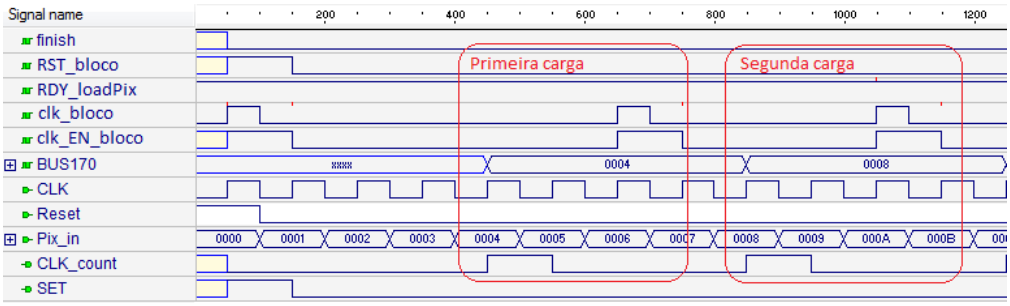


Figura 4.22: Carga dos pixels (escala em nanosegundos).

O sinal 'CLK_count' incrementa o contador da memória todas as vezes que ela é acessada. Uma vez incrementado o contador da memória o par de pixels é lido

e "repassado" para o cálculo. Nesta simulação o barramento 'Pix_in' é a porta que será conectada à memória. Ele se comporta como um contador para efeito de simulação. Todas as vezes que a memória precisa ser acessada o cálculo para, incrementa-se o endereço e lê o novo dado da memória. Para este processo são necessários três ciclos de *clock*.

Cada vez que o cálculo feito para uma secção de 64x64 da imagem é necessário reiniciar o bloco. O sinal 'finish' sinaliza este término sendo igual a um. Desta forma, o bloco de controle da memória identifica isto e atribui o valor um para o sinal 'RST_bloco' reinicializando a máquina de estado responsável pelo cálculo. Na Figura 4.23 é possível perceber isto.

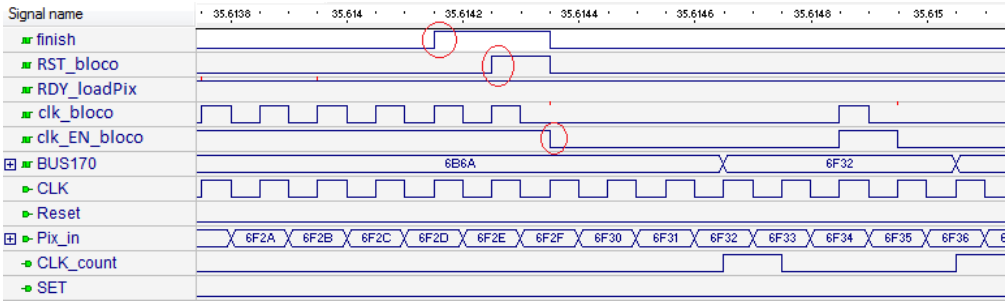


Figura 4.23: Reinicialização do bloco que calcula para a secção de 64x64 (escala em nanosegundos).

Na simulação, cada secção da imagem é calculada em 35.614.150 *ns* utilizando uma frequência de 10 MHz. Como na imagem considerada (resolução de 420x680) são necessárias 80 secções o tempo aproximado para o cálculo de do par de imagens é de aproximadamente 3 segundos utilizando apenas um bloco.

Os dados da síntese do sistema podem ser observados na Tabela 4.3.

	ALUTs	Registradores	DSPs	Bits de Memória
Quantidade	31630	49765	82	526,368
Porcetagem da FPGA	17%	27%	6%	4%

Tabela 4.3: Dados da síntese.

Outro dado gerado pela síntese é a frequência máxima que o sistema pode operar. Neste caso a frequência máxima foi de 78.44 *Mhz*. Utilizando este dado na simulação, o bloco de 64x64 foi executado em 4.540.283 *nanos*egundos. Desta forma, para toda a imagem de 420x680, o cálculo é realizado em, aproximadamente, 0.32 segundos, tendo assim uma média de processamento de 2.75 *fps* (frames por segundo). No trabalho desenvolvido em [23], realizou-se o cálculo do fluxo ótico de uma imagem de 240x320. Para este cálculo foi utilizado apenas um processador NiosII no kit de desenvolvimento DE2. Após algumas versões do sistema, o tempo atingido para o cálculo de dois frames foi de aproximadamente 24 minutos. Este

tempo faz com que o robô Pioneer 3dx colida e ultrapasse o objeto em 2 *km.* No sistema desenvolvido nesta pesquisa para o cálculo de fluxo ótico, o robô Pionner 3dx não colide com o objeto, pois o mesmo percorre a distância de apenas 50 *cm* durante o processamento dos dois frames de imagens com o dobro da resolução. Desta forma, é possível destacar o ganho de desempenho desta implementação em hardware em comparação com a aplicação embarcada [23] que utilizou apenas software.

Capítulo 5

Conclusões e Trabalhos Futuros

A princípio foi proposta uma arquitetura paralela para a execução do cálculo. Observou-se, porém, que com o processador Nios II as ferramentas de *debugging* e *profiling* de uma arquitetura paralela são precárias ou inexistentes (no caso do *profiling*). Desta forma, seria muito complexa a elaboração de uma arquitetura paralela com instruções customizadas. Embora o implementação de uma instrução customizada neste trabalho tenha sido descartada ela ainda é uma ferramenta poderosa para a elaboração de co-projeto hardware e software.

A proposta de uma implementação de um bloco de *hardware* controlado pelo Nios II também se mostrou inviável. Mesmo que não tenha sido realizada a integração com sucesso do bloco de *hardware* como uma instrução customizada, a comunicação do processador com o bloco poderia comprometer o tempo de execução devido o volume de informações trocadas. Se as duas imagens estivessem na memória do processador ainda seria necessário um tempo de carga para tal criando assim mais *overhead*.

A implementação de tanto o bloco para o cálculo do fluxo ótico como sua máquina de controle ambos em *hardware* pareceu a melhor opção. Uma vez que o controlador de memória a torna transparente para o cálculo este pode ser realizado em imagens de qualquer tamanho sem muitas adaptações. Outra vantagem da implementação em hardware é a escalabilidade. Uma vez que, as secções da imagem são calculadas sem dependência de dados, pode-se gerar uma arquitetura composta por vários sistemas propostos neste trabalho funcionando como elementos de processamento de uma arquitetura paralela. A possibilidade de um ganho no desempenho existe embora a comunicação com a memória ainda continue sendo um gargalo.

5.1 Trabalhos Futuros

Uma possibilidade de expansão do sistema sem muitas alterações é a duplicação do bloco 'top' da Figura 4.20. Com o uso de uma memória dual-port seria possível calcular o mesmo par de imagens duas vezes mais rápido, uma vez que, cada bloco seria responsável por apenas metade do par de imagens. Desta forma, a estrutura do sistema seria a representada na Figura 5.1.

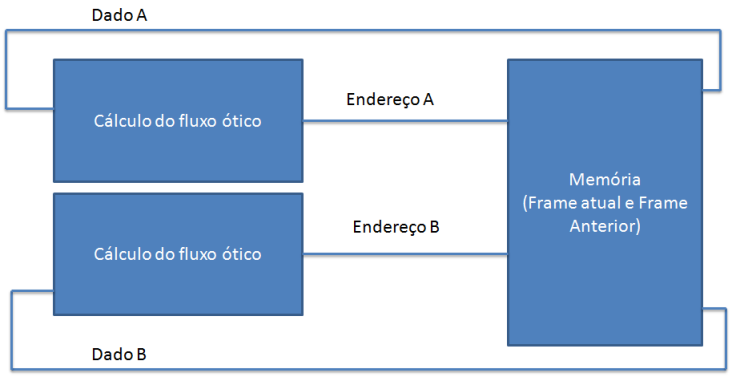


Figura 5.1: Reinicialização do bloco que calcula para a secção de 64x64 (escala em nanosegundos).

Como outras possibilidades de trabalhos futuros existem as opções a seguir:

- Elaborar uma arquitetura integrada ao sistema de cálculo como forma de conectá-lo a câmeras e dispositivos de vídeo.
- Expandir o sistema como forma de um melhor cálculo do fluxo ótico utilizando por exemplo algoritmos piramidais que possibilitam o cálculo de vetores de deslocamento maiores.
- Elaboração de uma ferramenta para a elaboração de um *profiling* em arquiteturas paralelas utilizando o NiosII, possibilitando a identificação de gargalos nos diferentes processadores da arquitetura e uma melhor elaboração de arquiteturas heterogêneas.

Referências Bibliográficas

[1] J. M. A. R. Antonio A. de Carvalho, Romeu R. da Silva, "O mundo das imagens digitais," *Matéria*, vol. 8, no. 2, pp. 167–186, 2003.

[2] R. C. Gonzalez and R. E. Woods, *Digital Image Processing (3rd Edition)*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 2006.

[3] J. C. Russ, *Image Processing Handbook, Fourth Edition*. Boca Raton, FL, USA: CRC Press, Inc., 2002.

[4] L. Shapiro and G. Stockman, *Computer Vision*. USA: Prentice Hall, 2001.

[5] V. Bonato, B. Mazzotti, and E. Marques, "Introdução: Técnicas de co-projeto de hw/sw baseadas em sistemas embarcados," 2009.

[6] C. Bobda, *Introduction to Reconfigurable Computing: Architectures, algorithms and applications*. Springer Verlag, 2007.

[7] Altera, "Nios ii custom instruction user guide," 2008.

[8] S. H. Fuller and L. I. Millett, "Computing performance: Game over or next level?," *Computer*, vol. 44, pp. 31–38, 2011.

[9] P. Hsiao, C. Lu, and L. Fu, "Multilayered Image Processing for Multiscale Harris Corner Detection in Digital Realization," *Industrial Electronics, IEEE Transactions on*, vol. 57, no. 5, pp. 1799–1805, 2010.

[10] J. Assumpção, "Projeto de um sistema de desvio de obstáculos para robôs móveis baseado em computação reconfigurável," 2009. Tese de mestrado.

[11] B. inc., "http://www.bluespec.com/index.php," 2009.

[12] T. Technologies, "De2-70 development and education board user manual," 2008.

[13] L. Martinez, "Projeto de um sistema embarcado de predição de colisão a pedestres baseado em computação reconfigurável," 2011. Tese de Mestrado - USP.

[14] O. Marques Filho and H. Vieira Neto, *Processamento Digital de Imagens.* Rio de Janeiro, Brazil: Editora Brasport, 1999.

[15] D. Bailey, *Design for Embedded Image Processing on FPGAs.* Wiley and Sons, Incorporated, John, 2011.

[16] J. Marzat, Y. Dumortier, and A. Ducrot, "Real-time dense and accurate parallel optical flow using cuda," in *Symposium on Advances in Image and Video Technology*, 2009.

[17] J. Assumpcao, D. F. Wolf, and E. Marques, "Towards a hardware accelerated obstacle avoidance system for mobile robots using monocular vision," in *IEEE Second International Symposium on Industrial Embedded Systems - SIES*, pp. 365–368, 2007.

[18] B. Horn and B. Schunck, "Determining optical flow," *Artificial intelligence*, vol. 17, no. 1-3, pp. 185–203, 1981.

[19] W. Wolf, *High-Performance Embedded Computing.* San Francisco, CA, USA: MORGAN KAUFMAN PUBLISHERS, 2007.

[20] M. Bueno, E. Marques, C. Moron, and J. e Silva, "Análise e implementação de suporte a SMP (multiprocessamento simétrico) para o sistema operacional eCos com aplicação em robótica móvel," 2007.

[21] Altera, "Nios ii software developers handbook," 2008.

[22] G. Moore *et al.*, "Cramming more components onto integrated circuits," *Proceedings of the IEEE*, vol. 86, no. 1, pp. 82–85, 1998.

[23] T. M. Lobo, "Estudo de técnicas de processamento de imagens envolvidas no cálculo de egomotion," 2009. Monografia final de conclusão de curso.

[24] V. Bonato, "Projeto de um módulo de aquisição e pré-processamento de imagem colorida baseado em computação reconfigurável e aplicado a robôs móveis," 2004. Tese de mestrado.

[25] C. Schmid, R. Mohr, and C. Bauckhage, "Evaluation of interest point detectors," *International Journal of Computer Vision*, vol. 37, no. 2, pp. 151–172, 2000.

[26] C. Harris and M. Stephens, "A combined corner and edge detection," in *Proceedings of The Fourth Alvey Vision Conference*, pp. 147–151, 1988.

[27] F. Bellavia, M. Cipolla, D. Tegolo, and C. Valenti, "An Evolution of the Non-Parameter Harris Affine Corner Detector: A Distributed Approach," in *Parallel and Distributed Computing, Applications and Technologies, 2009 International Conference on*, pp. 18–25, IEEE, 2010.

[28] M. Irani, B. Rousso, and S. Peleg, "Recovery of ego-motion using image stabilization," 1994.

[29] W. Burger and B. Bhanu, "Estimating 3-D Egomotion from Perspective Image Sequences," *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, vol. 12, no. 11, pp. 1040–1058, 1990.

[30] N. Paragios, Y. Chen, and O. Faugeras, *Handbook of Mathematical Models in Computer Vision*. Springer, 2006.

[31] J. Barron, D. Fleet, and S. Beauchemin, "Performance of optical flow techniques," *International journal of computer vision*, vol. 12, no. 1, pp. 43–77, 1994.

[32] M. Bleyer, M. Gelautz, and C. Rhemann, "Region-based optical flow estimation with treatment of occlusions," in *Joint Hungarian-Austrian Conference on Image Processing and Pattern Recognition (HACIPPR)*, pp. 235–242, Citeseer, 2005.

[33] B. Lucas and T. Kanade, "An iterative image registration technique with an application to stereo vision," in *International joint conference on artificial intelligence*, vol. 3, pp. 674–679, Citeseer, 1981.

[34] J. Shi and C. Tomasi, "Good features to track," in *Computer Vision and Pattern Recognition, 1994. Proceedings CVPR '94., 1994 IEEE Computer Society Conference on*, pp. 593 –600, June 1994.

[35] K. Compton and S. Hauck, "Reconfigurable computing: A survey of systems and software," 2002. ACM Computing Surveys, n. 2, p. 171-210.

[36] R. de Vasconcellos Peron, "Otimização de código fonte c para o processador embarcado niosii," 2008. Disertação de mestrado.

[37] M. Abd-El-Barr and H. El-Rewini, *Fundamentals of computer organization and architecture*. Wiley-Interscience, 2005.

[38] W. Stallings, *Computer Organization and Architecture*. Prentice Hall, 2003.

[39] M. E. A. M. M. Abutaleb, A. Hamdy and E. M. Saad, "A reliable fpga-based real-time optical-flow estimation," *International Journal of Electrical and Electronics Engineering*, vol. 4, no. 4, pp. 291–296, 2010.

[40] S. K. Robert Piotrowski, Stanislaw Szczepanski, "Fpga-based implementation of real time optical flow algorithm and its applications for digital image stabilization," *INTERNATIONAL JOURNAL ON SMART SENSING AND INTELLIGENT SYSTEMS*, vol. 3, no. 2, pp. 253–272, 2010.

[41] D. G. R. Bradski and A. Kaehler, *Learning opencv, 1st edition.* O'Reilly Media, Inc., 2008.

[42] Aldec, "http://www.aldec.com/files/products/," 2012.

[43] Altera, "Simulating nios ii embedded processor designs," 2011.

[44] J. Fenlason and R. Stallman, "Gnu gprof," 2003.

[45] MIT, ""http://csg.csail.mit.edu/6.375"," 2011.

[46] A. W. Jud Porter, Mike Thomson, "Lucas-kanade optical flow accelerator," 2011.