DSCAM: UMA PLATAFORMA HARDWARE-SOFTWARE PARA OPERAÇÕES DE VISÃO COMPUTACIONAL

GLAUBER TADEU DE SOUSA CARMO

DSCAM: UMA PLATAFORMA HARDWARE-SOFTWARE PARA OPERAÇÕES DE VISÃO COMPUTACIONAL

Dissertação apresentada ao Programa de Pós-Graduação em Ciência da Computação do Instituto de Ciências Exatas da Universidade Federal de Minas Gerais como requisito parcial para a obtenção do grau de Mestre em Ciência da Computação.

Orientador: Antônio Otávio Fernandes

Belo Horizonte

Junho de 2009

de Sousa Carmo, Glauber Tadeu
C287d DSCam: uma plataforma hardware-software para operações de visão computacional / Glauber Tadeu de Sousa Carmo. — Belo Horizonte, 2009
xxiv, 108 f. : il. ; 29cm

Dissertação (mestrado) — Universidade Federal de Minas Gerais
Orientador: Antônio Otávio Fernandes

1. Visão Computacional. 2. Arquitetura. 3. Smart-cameras. I. Título.

CDU 519.6*82.10

[Folha de Aprovação]

Quando a secretaria do Curso fornecer esta folha,
ela deve ser digitalizada e armazenada no disco em formato gráfico.

Se você estiver usando o `pdflatex`,
armazene o arquivo preferencialmente em formato PNG
(o formato JPEG é pior neste caso).

Se você estiver usando o `latex` (não o `pdflatex`),
terá que converter o arquivo gráfico para o formato EPS.

Em seguida, acrescente a opção `approval={`*nome do arquivo*`}`
ao comando `\ppgccufmg`.

À Saynara Elisa,
pelas vezes que insistiu em me esperar acordada,
me recebendo com um sorriso...

Agradecimentos

Agradeço a Deus, pela oportunidade de ter realizado esse trabalho e por todas as demais experiências que Ele me proporcionou viver nesses mais de dois anos de estudo, incluindo a companhia nas noites solitárias na frente do computador em uma das salas do DCC ou em casa. À intercessão de Maria, sempre se adiantando e providenciando tudo aquilo que precisei nesse periodo, me surpreendendo diversas vezes ao mostrar o quão inúteis eram as minhas preocupações. Quantas portas foram abertas e problemas que pareciam impossíveis se dissolveram com simples toques. Por cada um desses impossíveis, fica meu agradecimento a São Judas Tadeu.

Agradeço ao Professor Antônio Otávio, orientador deste trabalho, sempre disponível e animado, presente em cada etapa passando muito mais que conhecimento e experiência, mas vivência, ensinando uma forma descontraída de encarar as situações. Ao Luiz Fernando, que abriu sua empresa, a *Invent Vision*, oferecendo toda a estrutura para a realização deste projeto, fornecendo o hardware, além de todo o seu conhecimento em visão computacional. Agradeço também ao Professor Diógenes Cecílio, que contribuiu de forma direta na construção do meu conhecimento logo no meu ingresso à universidade e, ao meu egresso, aceitou o convite para participar da banca de avaliação do trabalho. Ao Professor Mário Fernando, que, apesar de não ter acompanhado desde o início do projeto, teve uma importante e decisiva contribuição nas etapas finais, sempre disposto a ajudar, sendo convidado para compor a banca. Por fim, todos os demais professores e funcionários do DCC, em especial o Antônio Loureiro, Claudionor Coelho, Sérgio Campos, José Monteiro, Túlia, Renata e Sheila, que contribuíram de forma direta ou indireta para que eu agregasse cada vez mais conhecimento durante a minha permanência na universidade.

Não poderia deixar de agradecer também aos meus pais, que sempre me deram apoio incondicional, principalmente nos estudos. À minha mãe, que, de tantos textos lidos sobre visão computacional e outras (tentativas de) ajudas nos experimentos, também poderia receber um título. Ao meu pai, que também sempre se mostrou pronto a ajudar e compreendendo que nem sempre eu podia estar disponível (mesmo que fosse

domingo, seis horas da manhã) para auxiliá-lo com o computador. Agradeço à minha irmã que contribuiu desde a compra do computador utilizado durante toda a realização do trabalho e, mesmo estando quase um ano distante, também foi minha companhia de muitas noites. A todos os demais familiares aos quais não pude visitar com frequência, obrigado.

Para que fosse possível eu finalizar uma implementação, um experimento ou o texto final, quantas vezes minha esposa cuidou de todas as tarefas da casa sozinha, deixando até de estudar para a faculdade... sem contar as vezes que ela foi dormir sozinha enquanto eu permanecia namorando o computador. Por todos os momentos que ela esteve do meu lado me apoiando, compreendendo, dando forças para continuar e até me chamando a atenção quando eu enrolava, muito obrigado.

Agradeço ao Antônio Caudeira, com quem trabalhei junto grande parte do tempo para que todo o projeto, desde o ambiente de desenvolvimento, até a execução dos algoritmos na câmera, funcionasse. Ao Bernardo, responsável por toda a configuração do Linux no meu computador (não sei o que seria de mim sem ele, teria eu conseguido fazer o cross-compiler funcionar?) e a todos os demais amigos da iVision, que contribuíram bastante ativa em praticamente todas as etapas do projeto. Merece destaque também o Cadson Alexandre, que teve toda a paciência e dedicação para me ensinar os primeiros passos nas aplicações de visão computacional.

Finalmente, agradeço aos amigos do LECOM e todos os outros que conheci nessa minha curta estadia no DCC. Claro que eu não poderia de deixar de lembrar do Alessandro Justiniano (AJ), uma das primeiras pessoas que conheci na UFMG, me mostrando como as coisas funcionavam por ali.

No mais, isso é tudo o que eu posso dizer: MUITO OBRIGADO A TODOS VOCÊS!

"Eu não consigo ver aonde eu vou chegar
Sinto que estou mais velho, preciso caminhar..."
(Eduardo Faro – Rosa de Saron)

Resumo

O desenvolvimento de arquiteturas de processamento cada vez de menor consumo de energia e maior poder de processamento, aliado ao desenvolvimento de diversas outras tecnologias, possibilitou a criação de uma nova geração de dispositivos com inteligência embarcada, como, por exemplo, as *smart-cameras*. As *smart-cameras*, ou cameras inteligentes, são sistemas embarcados de visão que captura e processa uma imagem para extrair dados importantes a uma determinada aplicação em tempo real. Entretanto, o seu desenvolvimento é desafiador, pois, de um lado, o processamento de video exige uma grande demanda de processamento e, consequentemente, consumo de energia, por outro lado, sistemas embarcados possuem rígidas restrições nesses dois tópicos. Além disso, os algoritmos utilizados no processamento devem ser adaptados à arquitetura, de forma obter o melhor aproveitamento dos recursos disponíveis no processador. Visando tornar o processo de desenvolvimento de aplicações de visão para *smart-cameras* menos árduo e mais acessível, surgiu a idéia de criar a *Digital Smart Camera* (DSCam). A DSCam consiste de uma solução integrada para aplicações de visão computacional, associada a um *framework* de desenvolvimento cujo objetivo é auxiliar na criação de sistemas de visão, eliminando a necessidade do programador conhecer detalhes da arquitetura interna da plataforma utilizada. Neste trabalho é realizado um estudo completo sobre arquiteturas de visão, propondo, implementando e testando a DSCam. Os resultados foram satisfatórios, obtendo uma plataforma bastante flexível, com uma interface de desenvolvimento facilitada e tempos de operação compatíveis aos encontrados em arquiteturas similares.

Palavras-chave: Visão Computacional, Smart-cameras, Arquitetura.

Abstract

The advent of integrated circuits with low power consumption and powerful processor capability, ally to many others news tecnologies, has became possible a new generation of systems with embedded inteligence, as smart-cameras. The smart-cameras are embedded vision systems that capture high-level descriptions of the scene and analyze it to extract important datas to an application in real time. However, the design of smart-camera is a challenging because on one hand video processing has insatiable demand for performance and power, and on other hand embedded systems place considerable constraints on the design. There is also the specific algoritms used in image processing, that will be adapted to processor architecture to obtain a best use of the resources. With the goal of facilitate this development grew the idea of Digital Smart Camera (DSCam). DSCam is an integrated solution for computer vision applications, associated with a development framework to help architects and programmers of the vision applications. Using the DSCam the programmers unaware the architecture details, working in a high-level language. In this work a comprehensive study of vision architectures is done and the DSCam is proposed, implemented, and tested. The results are satisfactory, with a very adaptable platform, an easy development interface and executes times near of similars works.

Keywords: Computer Vision, Smart-cameras, Architecture.

Lista de Figuras

Lista de Tabelas

Sumário

Capítulo 1

Introdução

A cada dia está se tornando mais comum o uso de aplicações de visão computacional no cotidiano das pessoas. Entende-se como aplicações de visão sistemas utilitários que tomam decisões baseadas em características visuais da cena em que se encontram. O objetivo de um sistema de visão computacional é construir automaticamente a descrição de uma imagem ou seqüência de imagens, ou seja, identificar e localizar os objetos baseando-se em suas características físicas, como cor, formato e tamanho, de forma a ser possível extrair informações importantes à aplicação.

1.1 Objetivo

O objetivo deste trabalho consiste em desenvolver uma arquitetura hardware-software que auxilie no desenvolvimento de aplicações de visão computacional, tornando-o de implementação e configuração mais simples, acessível para o uso não apenas em grandes centros de pesquisa, mas também em sistemas comerciais reais.

1.2 Motivação

Tradicionalmente, os sistemas de visão são compostos por uma câmera conectada a um computador, onde é executado um software específico da aplicação, criado geralmente em C++ ou Java. O resultado do processamento pode então ser exibido em um monitor para o operador ou transmitido a um segundo sistema, que pode ser, por exemplo, um atuador, conforme apresentado na Figura 1.1. Entretanto, com o avanço da tecnologia e o desenvolvimento de elementos de processamento cada vez menores, mais potentes e com baixo consumo de energia, vem se tornando realidade a integração de todo o

sistema de visão em um único equipamento. Denominado genericamente de *smart-camera*, esse equipamento é responsável desde a captura, realizando o processamento das imagens e disponibilizando os resultados. A Figura 1.2 apresenta um diagrama simplificado do funcionamento de um sistema utilizando uma *smart-camera*.

Figura 1.1. Organização de um sistema de visão tradicional

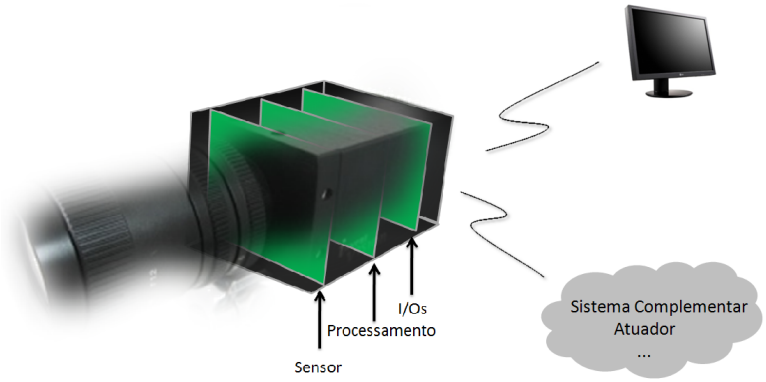

Figura 1.2. Organização de um sistema de visão utilizando uma *smart-camera*

Ao colocar a etapa de processamento junto à captura, além da simplificação do sistema, possibilita-se o uso de visão computacional em aplicações que exigem uma resposta rápida aos eventos. Com o sistema tradicional, o tempo gasto na transmissão da imagem da câmera até o processador pode ser extremamente longo, por se tratar de imagens de alta resolução, em relação às necessidades dessas aplicações. Em uma *smart-camera* o sensor de captura é capaz de se comunicar diretamente com o processador, reduzindo significativamente o tempo entre a captura e o processamento.

Essa integração, apesar de apresentar uma solução bastante interessante para as aplicações de visão, uma vez que condensa todo o sistema, consumindo menos energia

e trabalhando com processadores específicos para atender a essa demanda de processamento, representa um novo desafio ao programador. O desenvolvimento de aplicações para plataformas dedicadas requer do profissional um estudo detalhado de seus recursos, interfaces e diversos outros fatores necessários para uma boa implementação dos algoritmos de visão.

Seguindo a tendência de integração, somado ao objetivo de tornar o processo de desenvolvimento de aplicações de visão para *smart-cameras* menos árduo e mais acessível, surgiu a idéia de criar a *Digital Smart Camera* (DSCam). A DSCam consiste de uma solução integrada para aplicações de visão computacional, dispondo de um *framework* de desenvolvimento cujo objetivo é auxiliar na criação de sistemas de visão, eliminando a necessidade do programador conhecer detalhes da arquitetura interna da plataforma utilizada. Utilizando-se a Figura 1.2 como referência, a DSCam corresponde à parte de processamento indicada na câmera, acrescida de um *framework* que auxilia em sua programação.

Idealmente, uma solução integrada de visão computacional deve ser capaz de capturar imagens, realizar todo o seu processamento e extração de dados, executando operações pré-determinadas pelo usuário em processadores dedicados, e disponibilizar o resultado para que ações possam ser tomadas, quando necessário.

Para aproveitar ao máximo todos os benefícios providos por essa arquitetura ideal, é desejável que exista um *framework* que auxilie na tarefa de desenvolver o fluxo de operações a serem executadas e as mapeie da melhor maneira possível nos recursos de hardware disponíveis. Além de contar com um extenso conjunto de funções de visão computacional comumente utilizado, um *framework* deve fornecer a funcionalidade de adição de recursos, procurando atender a aplicações que necessitem de funções adaptadas ou específicas.

Existem atualmente diversas propostas de arquiteturas dedicadas para o processamento digital de imagens, que exploram principalmente o paralelismo nessas operações. Essas arquiteturas procuram otimizar a execução dos algorítmos de visão computacional desenvolvendo processadores específicos para cada tipo de tarefa. Entretanto, o resultado são arquiteturas de alto custo, inviáveis para grande parte das aplicações comerciais, ficando restritas muitas vezes apenas como elemento de pesquisa de grandes universidades.

Para as aplicações comerciais, o que se utiliza é uma câmera conectada a um computador, que executa softwares de visão computacional. Esses softwares geralmente procuram fornecer recursos que auxiliam em diversas etapas do processo de desenvolvimento, entretanto, uma dificuldade muitas vezes encontrada é a grande curva de aprendizado, que pode chegar a consumir uma parte considerável do tempo de de-

senvolvimento e afetar o prazo de entrega da aplicação. Outra restrição é o fato de dependerem de um computador de propósito geral para realizarem o processamento, utilizando a câmera apenas como dispositivo de captura.

1.3 Sistemas de visão computacional

A visão é um dos sentidos mais fascinantes e complexos do corpo humano, fazendo com que a maior parte do córtex celebral fique dedicada a ela (Pridmore & Hales [1995]). Sua função é a de prover uma descrição detalhada do mundo tridimensional ao redor que se encontra em constantes mudanças. O principal objetivo da visão humana é, dada uma certa imagem, construir uma base de informações que permitam ao corpo tomar decisões e executar ações naquele ambiente. Largura, posicionamento, cores e texturas dos objetos são algumas das características detectadas pela visão. Ao comparar esses dados com características de outras cenas vistas anteriormente, o cérebro é capaz então de reconhecer o objeto.

Visando avançar a interação das máquinas com o mundo real, iniciou-se no final da década de 50 estudos para reproduzir no computador todas estas funcionalidades. Esses estudos receberam o nome de visão computacional, sendo seu principal objetivo entender as representações e processos utilizados pela visão humana com detalhes o suficiente para implementá-los em um computador (Pridmore & Hales [1995]).

A visão computacional ajuda a descrever o mundo real existente em frente a uma câmera, tendo como sinais de entrada as imagens (imagem estática ou seqüência de imagens) e como sinais de saída representações simbólicas, nomes de objetos reconhecidos ou ainda equações matemáticas que descrevem uma superfície.

Enquanto na computação gráfica tem-se como principal motivação a transformação de elementos geométricos definidos por equações matemáticas em imagens de maior realismo, na visão computacional procura-se percorrer o caminho inverso, detectando em imagens reais características que permitam decompô-la em elementos geométricos e que forneçam dados suficientes para se fazer uma completa descrição da cena. A descrição de uma cena, além da identificação de cada objeto, consiste do seu posicionamento, cor e orientação (Coatrieux [2005]; Fung & Mann [2004]).

1.4 Aplicações práticas

O estudo dos sistemas de visão vem atraindo cada vez mais o interesse comercial e, consequentemente, dos centros de pesquisa. Isso se deve principalmente ao fato

de que a cada dia uma área diferente do conhecimento encontra nesses sistemas a solução (ou pelo menos parte) para seus desafios. Alguns exemplos comuns que já podem ser facilmente encontrados nas ruas incluem a leitura de códigos de barras, a identificação de placas veiculares em entradas de estacionamentos e o reconhecimento de faces humanas.

Na biologia, a visão computacional pode ser a esperança para as pessoas que não enxergam, provendo conhecimento para o desenvolvimento de sistemas de visão artificial. Atualmente, um simples exame de fundo de olho pode revelar muito mais informações com o auxílio de um computador, que é capaz de avaliar uma quantidade maior de detalhes que o médico. Outras aplicações médicas incluem a análise de tomografias ou ainda o auxílio em cirurgias e exames como endoscopia.

A identificação de constelações, planetas e radiações capturadas por câmeras especiais são alguns exemplos de aplicações em astronomia que fazem uso em larga escala dos sistemas de visão computacional. Grandes centros de pesquisas espaciais possuem supercomputadores ou *grid* de computadores dedicados a analisarem as imagens enviadas por satélites e identificar planetas, estrelas e outros corpos celestes.

Hoje, uma das áreas de maior expansão no uso de sistemas de visão computacional tem sido a segurança eletrônica. Jamais se viu tamanha quantidade de dispositivos capazes de identificar uma pessoa utilizando as mais diversas características. Essas aplicações vão desde portas que são abertas com a impressão digital, acesso a sistemas por leitura da íris até a identificação do rosto de um suspeito em meio à multidão.

Entretanto, é na engenharia que a visão computacional, até o momento, encontra um maior número de aplicações. Em uma linha de produção moderna chegam a ser incontáveis os dispositivos de visão utilizados principalmente com o objetivo de detectar falhas e manter a qualidade do produto final. Esse tipo de inspeção é bastante interessante uma vez que não há contato físico direto com a peça. Nessa área, pode-se citar como exemplos de verificações nos sistemas de visão:

- cores: detecção de manchas na pintura de um objeto, leitura de um código de cores;
- formatos: cortes inacabados, rebarbas, peças quebradas;
- dimensões: distância entre dois encaixes, espessuras, diâmetro de uma abertura, ângulos.

Um exemplo prático na engenharia onde ocorre um intenso uso de sistemas de visão computacional é na montagem de placas de circuito impresso. Na placa ainda nua

pode-se validar as trilhas desenhadas, visando detectar curtos ou interrupções entre os terminais e conferir os nomes (*labels*) dos componentes, geralmente impressos na placa. Após colocados os componentes, a qualidade dos pontos de solda, a polaridade e até mesmo o valor de cada componente são caracteristicas que podem ser inspecionadas por um sistema de visão.

1.5 Considerações

As aplicações de visão computacional, que até pouco tempo era apenas um objeto de estudo de universidades e utilizadas em casos específicos, hoje vêm sendo absorvidas de forma extremamente rápida no cotidiano das pessoas, tomando espaço em quase todas as suas atividades. Essa explosão de demanda gerou diversas pesquisas na área, desenvolvendo novas técnicas, modelos de desenvolvimento, classe de aplicações e até mesmo processadores dedicados.

Típicamente, aplicações de visão atuais são compostas por câmeras e um computador executando aplicações específicas. Entretanto, assim como está ocorrendo com outras tecnologias, há uma grande tendência de se integrar todo o sistema de visão em um único dispositivo, denominado *smart-camera*, visando maior praticidade e, ao se utilizar uma plataforma específica, melhor desempenho. Uma das primeiras tentativas de levar o processamento para dentro da câmera foi com o uso de FPGAs. Entretanto, apesar de funcional, a codificação de uma arquitetura reprogramável é complexa e pode consumir muito tempo, o que levou ao uso de processadores dedicados para operações com imagens.

Dessa inovação, nasceu a necessidade de *frameworks* que auxiliem no desenvolvimento de aplicações de visão, uma vez que muitos dos modelos e algoritmos eficientes utilizados em um computador não são válidos para arquiteturas dedicadas. Atualmente, o que se espera de um *framework* é que auxilie profissionais a criarem aplicações de visão, entretanto, observando o ocorrido com outras ferramentas, como aplicativos de processamento de imagens, já é possível questionar se num futuro próximo as pessoas não estarão, com o auxílio de ferramentas, criando fluxos de execução de visão computacional no seu próprio celular, tornando uma tarefa hoje extremamente complexa em algo simples, realizada por qualquer um.

O restante deste trabalho está dividido da seguinte forma: o Capítulo 2 procura fazer um estudo completo das principais arquiteturas de visão computacional existentes na literatura; no Capítulo 3 são apresentados e analisados os principais requisitos comuns à maioria dos sistemas de visão computacional; o Capítulo 4 apresenta a arquite-

tura proposta – DSCam, destacando os recursos implementados e a interface com o usuário; no Capítulo 5 são avaliados os desempenhos de diferentes algorítmos de visão em várias plataformas, fazendo um estudo comparativo entre elas e, finalmente, no Capítulo 6 é feita uma análise final do trabalho, sugerindo melhorias e novos recursos que podem ser incorporados futuramente à arquitetura proposta.

Capítulo 2

Arquiteturas existentes

Desde o final da década de 50, vários estudos vêm sendo realizados com o objetivo de aperfeiçoar as técnicas de processamento e análise de imagens digitais, além de auxiliar no desenvolvimento de aplicações de visão. Neste capítulo são apresentados os principais esforços para desenvolver arquiteturas dedicadas a aplicações de visão computacional, avaliando as vantagens, desvantagens e relação custo-benefício para cada uma delas. Não é objetivo criar aqui uma extensa lista ou uma relação completa dos trabalhos já realizados na área, mas fazer um estudo das principais propostas, principalmente das arquiteturas similares ao DSCam.

A experiência em visão computacional mostra que não há uma arquitetura ou software único que forneça todos os recursos necessários à geração de um sistema que atenda inteiramente aos requisitos de uma aplicação de visão. Existem diversos pacotes de softwares bastante completos que oferecem vários recursos interessantes aos desenvolvedores, simplificando consideravelmente a criação de uma aplicação. Entretanto, todos esses pacotes falham em pelo menos um requisito (Thoren [2002]). A construção de uma aplicação de visão computacional completa depende, portanto, da interação entre diferentes sistemas de processamento de imagens e visão. Desta forma, pode-se considerar que a produtividade de um desenvolvedor, ao utilizar-se de uma determinada arquitetura ou plataforma de software, está diretamente relacionada com a sua capacidade de integrar soluções propostas por diferentes sistemas.

Conforme é descrito no Capítulo 3, o processamento realizado em aplicações de visão computacional pode ser dividido em três níveis, onde cada um possui características bem específicas. Diante deste cenário heterogêneo, diversas arquiteturas foram propostas para cada um dos níveis, procurando maximizar o desempenho em cada tarefa realizada. Na busca por um sistema completo, que atenda a todos os requisitos, surgiram também as arquiteturas híbridas, que combinam, de diferentes formas, as

soluções individuais de cada nível, buscando a mais eficiente organização possível.

Além das arquiteturas projetadas especificamente para visão computacional, alguns estudos foram realizados com o objetivo de utilizar arquiteturas paralelas de propósito geral para processamento e análise de imagens. Diversas análises foram realizadas em arquiteturas como *Connection Machine 5* (CM-5) (Prasanna et al. [1993]), da *Thinking Machine Corporation*, SP-2 (Chung et al. [1995]), da IBM, e Paragon (Saini & Simon [1994]), da Intel, entretanto os resultados não foram animadores. Além da inviabilidade econômica dessas arquiteturas para aplicações comerciais, são de complexa programação e não apresentam um bom desempenho em todos os níveis de processamento.

Nas seções seguintes são apresentadas as principais arquiteturas para operações de nível baixo, arquiteturas híbridas e as *smart-cameras*, sistemas embarcados cuja finalidade é capturar e criar uma descrição de alto nível daquilo que é possível visualizar na imagem. As arquiteturas de nível médio e alto não recebem muitos esforços isolados, sendo analisadas apenas nas arquiteturas híbridas.

2.1 Arquiteturas para operações de nível baixo

As arquiteturas para operações de nível baixo são as mais pesquisadas devido à especificidade de suas características e ao grande volume de dados envolvido (Sunwoo & Aggarwal [1990b]).

Um dos modelos topológicos mais conhecidos e elementares de arquitetura para essas operações é o *mesh-connected*. Sua arquitetura consiste em uma matriz quadrada de processadores SIMD, onde cada processador possui uma memória local e se comunica com seus quatro vizinhos, conforme a Figura 2.1. Na maioria dos casos é bastante eficiente em problemas de comunicação local e paralelismo de dados, como nas operações de nível baixo de visão. Alguns exemplos de arquiteturas *mesh-connected* implementadas são *Geometric Arithmetic Parallel Processor* (GAPP) (Dyer [1989]) e MP (Nickolls [1990]).

Contendo mais de dez mil processadores, a arquitetura GAPP foi desenvolvida para atender às necessidades militares, sendo um equipamento portátil, de pouco peso e baixo consumo de energia. Sua arquitetura é escalável, conseguindo obter um alto desempenho mesmo em imagens maiores, mantendo sempre total compatibilidade de rotinas já existentes.

A arquitetura MP, foi desenvolvida pela IBM com o objetivo de aumentar a eficiência no processamento de problemas computacionais com paralelismo de dados.

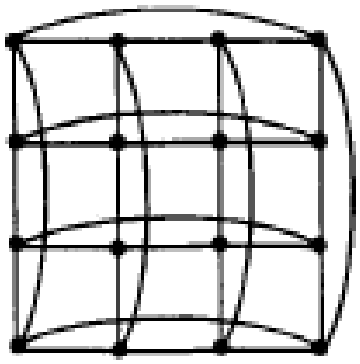

Figura 2.1. Arquitetura mesh-connected.

A MP é composta por um arranjo de 1024 processadores, sendo escalável até 16384, com aumento de performance linear. Os processadores possuem memórias locais não compartilhadas e são organizados em clusters com 16 unidades cada, conectados aos seus oito vizinhos mais próximos. Uma Unidade de Controle (UC) é responsável por buscar uma instrução na memória e replicá-la para todos os *clusters*. Os dados também são carregados da memória e distribuídos aos processadores pela UC.

A arquitetura *mesh-connected*, ao interligar apenas processadores vizinhos, gera problemas de *overhead* na comunicação entre eles, além de interrupções para entrada e saída de dados, limitando a capacidade de processamento. Procurando corrigir essas falhas, foi proposta a arquitetura *Sliding Memory Plane* (SliM) (Sunwoo & Aggarwal [1990a]) que implementa um novo sistema de comunicação criando quatro conexões virtuais, além das quatro existentes na *mesh-connected*. As conexões virtuais são criadas por vias secundárias de comunicação, paralelas às propostas originalmente, que não precisam ser lidas por processadores intermediários, entre a origem e o destino, para serem retransmitidas, funcionando como um *by-pass*. Multiplexadores recebem os sinais e reencaminham para o destino, sem interromper o processador, como representado na Figura 2.2. Outra melhoria nessa arquitetura é a introdução de um *buffer* de entrada e saída de dados, que reduz significativamente as interrupções ao processador. Como resultado final, obteve-se uma maior disponibilidade do processador para executar as operações na imagem, sem se ocupar com tarefas secundárias. A arquitetura híbrida *Vision Tri-Architecture* (VisTA), descrita na seção seguinte, utiliza o modelo SliM para processamento de operações de nível baixo.

Um outro tipo de processador que vem sendo estudado para essas operações é o *Digital Signal Processor* (DSP). Os DSPs são processadores de baixo custo especializados no processamento digital de sinais como audio e vídeo. Sua arquitetura é inteiramente projetada para atender aos requisitos de aplicações que envolvem esses

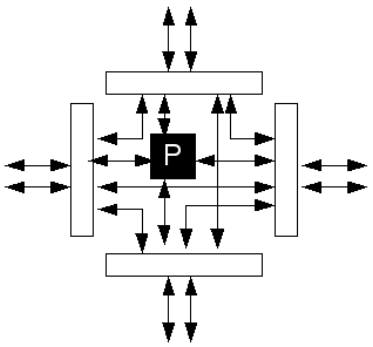

Figura 2.2. Representação das conexões na arquitetura SliM.

sinais, contendo instruções específicas e otimizadas para as operações mais utilizadas, garantindo aos DSPs uma melhor performance frente às demais arquiteturas. Estudos recentes apresentam comparações de execuções de operações de nível baixo entre uma família de DSPs e FPGAs (Baumgartner et al. [2007]). Alguns trabalhos como Delong et al. [2007]; Chattopadhyay & Boult [2007] já utilizam um DSP em aplicações de visão em tempo real com sucesso.

2.2 Arquiteturas híbridas

Dentre as arquiteturas híbridas de visão computacional, destaca-se a NETRA, desenvolvida pela Universidade de Illinois (Choudhary et al. [1993]), a *Image Understanding Architecture* (IUA), projetada pela Universidade de Massachusetts (Weems et al. [1990]) e a VisTA, criada pela Universidade do Texas (Sunwoo & Aggarwal [1990b]). Nas três propostas a solução encontrada para atender a todos os requisitos de processamento foi a definição de uma arquitetura específica para cada nível, sendo classificadas, como arquiteturas híbridas de processamento paralelo dedicado.

A arquitetura NETRA é composta por um grande número de elementos de processamento, distribuídos em *clusters*, gerenciados por um sistema distribuído de escalonamento (SDP). Os *clusters* podem operar nos modos *Single Program, Multiple Data* (SPMD) ou *Multiple Instruction, Multiple Data* (MIMD). Para processamento de nível baixo, onde há um grande paralelismo, utiliza-se clusters SPMD. *Clusters* no modo sistólico são utilizados por processamento de nível médio, enquanto em processamento de nível alto utiliza-se clusters MIMD.

A arquitetura IUA foi criada de forma a incorporar os três níveis de processamento de visão em um estrutura hierárquica, conforme apresentado na Figura 2.3. Cada nível é composto por um processador paralelo, distinto dos outros dois, desenvolvido pela

própria universidade especificamente para aquela classe de operações, e se comunicam com o uso de memórias compartilhadas. Para as operações de nível baixo a IUA utiliza um processador paralelo de arquitetura *mesh-connected* puramente SIMD denominado *Content Addressable Array Parallel Processor* (CAAPP). Os níveis alto e médio de processamento são atendidos por processadores paralelos MIMD.

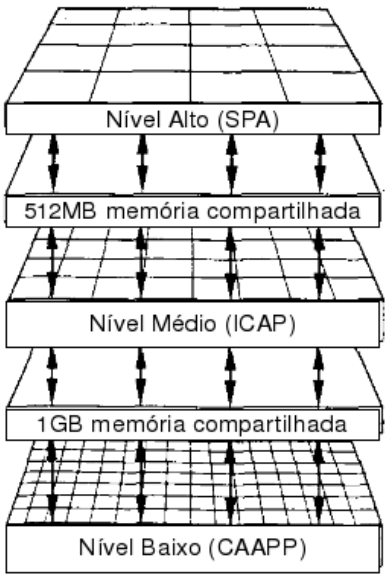

Figura 2.3. Arquitetura IUA.

Seguindo o mesmo modelo do IUA, o VisTA é composto por três níveis de processamento, explorando o paralelismo espacial e temporal dos algorítmos de visão computacional. Sua vantagem encontra-se no modelo de arquitetura utilizado em cada nível. Para o nível baixo, denominado VisTA/1, é implementado o SliM, enquanto para os níveis médio e alto, denominados VisTA/2 e VisTA/3, são propostas as arquiteturas *Flexibly Coupled Multiprocessor* (FCM) e *Flexibly Coupled Hypercube Multiprocessor* (FCHM), respectivamente.

Diversas aplicações científicas foram implementadas para as arquiteturas apresentadas, que se mostraram bastante eficientes. Entretanto, para aplicações comerciais elas não se apresentaram viáveis, o que impossibilitou a expansão dos sistemas de visão (Ratha & Jain [1997]; Baglietto et al. [1996]). Outras desvantagens dessas arquiteturas incluem a sua complexidade e a necessidade de se desenvolver compiladores e sistemas operacionais compatíveis, aumentando ainda mais o custo (Wang et al. [1996]). O desenvolvimento e uso de arquiteturas híbridas só se justificam em casos onde o custo poderá ser dividido em um número maior de aplicações, como ocorre nas universidades.

2.3 *Smart-cameras*

Um sistema de visão tradicional geralmente é composto por uma câmera, com a única função de capturar imagens e transmití-las a um computador de propósito geral, que efetua o processamento. Essa configuração, entretanto, possui diversas desvantagens, como, por exemplo, o tempo de transmissão da imagem da câmera para o computador. Outra desvantagem é o fato do processador não possuir uma arquitetura otimizada para o processamento de imagens (Shi et al. [2006b]), entretanto, a frequência de processamento e dos barramentos, somados a outros fatores como a hierarquia de memória, compensam a diferença, quando comparados com processadores dedicados, que possuem uma arquitetura mais eficiente para essas tarefas, mas trabalham com frequências e memórias menores.

As *smart-cameras* são uma evolução desse modelo câmera-computador, embutindo a etapa de processamento na própria câmera, não necessitando, assim, de um processador externo. A idéia da *smart-camera* é capturar uma imagem e extrair os dados contidos nela, convertendo-os em informação e transmitindo apenas resultados de análises em um nível maior de abstração para outras aplicações (Yu Shi [2005]), ou seja, são sistemas embarcados cuja sua principal função é produzir uma descrição de alto nível de uma imagem capturada de forma que as informações obtidas possam ser utilizadas por um sistema inteligente (Shi et al. [2006a]), como, por exemplo, na tomada de decisão em um sistema de controle automatizado.

A Figura 2.4 ilustra o modelo de funcionamento de uma *smart-camera*. Após a captura da cena pelo sensor, a imagem é passada diretamente ao processador, que extrai dela informações importantes à aplicação. Essas informações são então analisadas e o resultado transmitido por meio de um protocolo estruturado ou simplesmente um sinal elétrico, como, por exemplo, sinalizações de aprovado/reprovado, para um outro sistema.

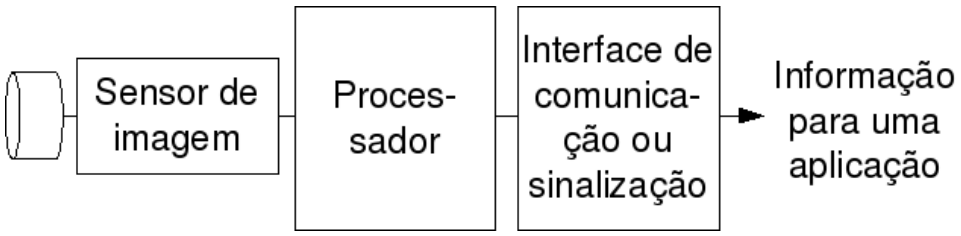

Figura 2.4. Diagrama em blocos de uma smart-camera.

Há quatro famílias de processadores que geralmente são utilizadas em *smart-cameras*, sendo elas microcontroladores, *Application Specific Integration Circuits*

(ASICs), DSPs e *Programmable Logic Devices* (PLDs), como as FPGAs (Yu Shi [2005]). Os microcontroladores são uma opção barata, porém possuem um poder limitado de processamento que geralmente não atende à demanda das aplicações de visão computacional. ASICs são processadores desenvolvidos e produzidos especificamente para uma determinada aplicação, sendo extremamente eficientes e de baixo consumo de energia, entretanto possuem um custo elevado, sendo viável apenas para produção em grandes volumes. Os DSPs são relativamente baratos e possuem uma boa performance no processamento de imagens e vídeos, sendo muito utilizados atualmente. Por sua vez, as FPGAs vem sendo descobertas como uma forte candidata para soluções embarcadas de visão computacional, como as *smart-cameras*, concorrendo com os DSPs. Uma das mais importantes vantagens da FPGA é a capacidade de explorar o paralelismo inerente a muitos algoritmos de visão computacional.

Uma aplicação interessante das *smart-cameras* é o controle de um computador através dos gestos (Shi et al. [2006b]). A GestureCam, como foi denominada, reconhece os gestos, em especial movimentos com as mãos e cabeça, permitindo aos usuários controlar um computador sem ao menos tocá-lo. Para essa aplicação foi utilizada uma FPGA Xilinx Virtex-II Pro 2VP30, carregada com um PowerPC, 2MB de memória RAM, uma interface ethernet e portas RS-232, que são utilizadas para enviar o resultado da análise dos movimentos para o computador, utilizando um protocolo estruturado pré-estabelecido. Não foram publicadas análises de desempenho da aplicação, como o tempo de resposta e a precisão dos movimentos detectados.

Duas outras aplicações das *smart-cameras* que vêm crescendo muito é o reconhecimento de faces (Kleihorst et al. [2004]) e o monitoramento de trânsito (Litzenberger et al. [2007]), substituindo os radares de velocidade convencionais.

Para realizar o reconhecimento de faces, aplicação muito utilizada como forma de segurança, foram utilizados dois elementos de processamento, sendo um processador Xetal trabalhando no modo SIMD para as operações de nível baixo e um DSP Trimedia contendo VLIW (*Very Long Instruction Word*) para as demais. A *smart-camera* INCA, como foi denominada, é capaz de reconhecer até duas faces por segundo.

Finalmente, uma terceira aplicação para *smart-cameras* que vêm sendo bastante estudada é no monitoramento do transito. Em Litzenberger et al. [2007] os autores propoem uma câmera que conta os veículos que passaram por determinado trecho de rua e calcula a velocidade média para cada um deles naquele trecho, enviando para um computador remoto os dados coletados periodicamente via uma interface ethernet. Para essa aplicação foi utilizado um DSP Blackfin 537, com frequência máxima igual a 600MHz, sendo capturada uma imagem a cada 50 milisegundos. O processamento realizado sobre a imagem é extremamente rápido, gastando um tempo próximo a um

milisegundo, uma vez que o veículo pode ser encontrado comparando a imagem corrente com uma na rua sem carros. A velocidade é calculada através da diferença de posição do veículo entre duas imagens capturadas.

Um dos grandes desafios ainda encontrado nas *smart-cameras* é o desenvolvimento dos algoritmos a serem executados. Por conterem processadores de arquiteturas específicas, o desenvolvedor precisa adaptar e otimizar os algoritmos de visão para aquela arquitetura, desencorajando, portanto, o seu uso.

2.4 Considerações

Arquiteturas robustas de visão computacional ainda são realidades distantes das aplicações comerciais, que se limitam a simples processamentos onde podem ser utilizados processadores de propósito geral, como leituras de códigos de barras, ou outras tarefas simples sem grandes impactos no ambiente em que estão inseridas. Isso se deve principalmente, como se pôde observar, à complexidade e custo destas arquiteturas, sendo justificados apenas para aplicações de grande porte ou centros de pesquisas que dividem os esforços em vários projetos.

Entretanto, pesquisas recentes vêm procurando mudar esse cenário, como o desenvolvimento das *smart-cameras*. Diversos estudos nas mais diferentes áreas estão utilizando dessa tecnologia para aperfeiçoar determinada tarefa. Pode-se perceber nos trabalhos já realizados, sendo alguns deles apresentados aqui, que é um consenso o uso de arquiteturas SIMD para o processamento de nível baixo e uma segunda arquitetura para as demais operações. É importante ressaltar que os trabalhos citados são apenas alguns poucos exemplos de aplicações das *smart-cameras* e muitos outros trabalhos, até mesmo procurando resolver os mesmos problemas, podem ser encontrados na literatura.

No estudo das arquiteturas para visão computacional pode-se perceber um maior esforço naquelas destinadas às operações de nível baixo, devido principalmente às suas características específicas. As arquiteturas desenvolvidas para as operações de níveis médio e alto, por sua vez, possuem características similares aos processadores de propósito geral. Desta forma, em muitas aplicações onde as operações são majoritariamente de nível médio ou alto não se justificam o uso de arquiteturas específicas.

Um outro fator importante ao análisar as arquiteturas é a forma com que elas são programadas. Não adianta utilizar hardwares poderosos se não há softwares capazes de explorar seus recursos adequadamente. Para cada nova arquitetura, novos compiladores e, em alguns casos, até extensões de linguagens precisam ser criadas para oferecerem

aos programadores recursos para desenvolverem softwares eficientes e robustos.

Finalmente, o que se pode esperar das próximas aplicações de visão são arquiteturas híbridas compostas por processadores dedicados para as operações de nível baixo e processadores de propósito geral para realizar a extração dos dados e informações nas operações de níveis médio e alto. Os softwares poderão ser escritos em linguagens tradicionais como C ou C++, ou ainda em linguagens gráficas, de forma que o compilador identifique, automáticamente ou através de tags inseridas no código, qual processador executará cada tarefa, visando obter o melhor desempenho final.

Capítulo 3

Especificação teórica

Neste capítulo são descritos e analisados os requisitos de uma arquitetura de visão computacional, tanto em hardware quanto em software, sendo baseados em aplicações reais e citados na literatura. Os detalhes da arquitetura proposta – DSCam – são descritos em capítulos seguintes.

Um sistema de visão computacional deve atender obrigatoriamente a três requisitos básicos: captura, processamento e disponibilização de resultados. Esses três itens são fundamentais para qualquer sistema de visão, em especial para aqueles usados em aplicações de tempo real. Uma plataforma que não oferece qualquer um dos três elementos passa para o usuário a responsabilidade de suprir sua deficiência (Thoren [2002]).

Além desses requisitos essenciais, é desejavel que um sistema de visão inclua outros benefícios como extensibilidade, modularidade, portabilidade e facilidade na sua operação. A extensibilidade, juntamente com a modularidade, oferece ao usuário a possibilidade de adicionar ou alterar facilmente recursos do sistema, sem a necessidade de grandes intervenções. A portabilidade é a propriedade de um sistema que indica a sua capacidade de ser executado em diversos ambientes diferentes. Por fim, a facilidade de operar e interagir com o usuário simplifica o desenvolvimento e a utilização de aplicações de visão, permitindo que pessoas com um mínimo de conhecimento criem e executem suas próprias aplicações.

A seguir, cada um desses requisitos será detalhado e analisado no contexto específico das aplicações de visão computacional.

3.1 Captura

A fase de captura em um sistema de visão computacional é onde todo o processo se inicia. Para que uma cena possa ser analisada por algoritmos computacionais ela deve ser codificada em valores numéricos que possam ser entendidos por processadores. Essa transformação de luz para códigos numéricos é denominada digitalização e é realizada por sensores.

Os sensores são compostos por dois elementos básicos. O primeiro é um dispositivo físico sensível a uma banda do espectro de freqüência (como raios X, ultravioleta, luz visível ou infravermelho) que produza um sinal elétrico de saída proporcional ao nível de energia recebida. Geralmente este dispositivo é uma matriz de estado sólido. O segundo elemento, chamado digitalizador, é um dispositivo que converte a saída elétrica da matriz de estado sólido para a forma digital (Gonzalez & Woods [2000]).

Matrizes de estado sólido são compostas de elementos de imageamento de silício discretos chamados fotodetetores. Esses elementos possuem uma tensão de saída proporcional à intensidade da luz incidente e formam um pixel na imagem final. As matrizes podem ser divididas em varredura de linhas, geralmente usadas em scanners de mesa, e varredura por área, utilizadas em câmeras.

Existem diversos modelos de sensores disponíveis no mercado, permitindo ao desenvolvedor escolher aquele que mais se adequa à sua aplicação. Algumas características importantes para se avaliar ao escolher um sensor são:

- resolução: quantidade de fotossítios da matriz de estado sólido, que resulta na quantidade de pixels que conterá a imagem digitalizada. Geralmente especifica-se a quantidade de pixels que a imagem contém de largura e de altura. Quanto maior a resolução melhor a qualidade da imagem para uma mesma cena. Alguns sensores permitem ser configurados para trabalhar em resoluções menores que as nominais, resultando em imagens menores, mais fáceis de armazenar e de processamento mais rápido;

- quadros por segundo: indica a quantidade de imagens que o sensor consegue captar no intervalo de tempo equivalente a um segundo;

- cor/P&B: o sensor pode ser sensível às cores ou apenas identificar tons de cinza. Em algumas aplicações conhecer a cor dos objetos pode ser fundamental, enquanto em outras não há essa necessidade. Cada pixel do sensor colorido é dividido em três partes e uma máscara RGB (ou outro formato de cor compatível) permite a passagem de apenas uma frequência luminosa para cada uma delas.

Sabendo a intensidade de cada cor no pixel, é possível reproduzir, posteriormente, a cor original.

Além dessas características relacionadas diretamente à imagem a ser adquirida, os sensores podem ainda se diferenciar pela interface e protocolo utilizado para transmitir os dados ao processador.

Como conseqüência de toda essa especificidade, ao desenvolver um sistema de visão, o programador deve entender a forma com que o sensor funciona para que sua aplicação possa capturar a imagem.

Carregar imagens de arquivos é uma outra forma de alimentar sistemas de visão, entretanto essa forma pode ser bastante ineficiente, principalmente para aplicações que contenham requisitos de tempo real.

Diante de tantos parâmetros, alguns sistemas de visão computacional procuram deixar transparentes esses ajustes, solicitando ao usuário o mínimo de dados possível, como a resolução e o modelo do sensor. De posse desses dados e com bibliotecas específicas, o sistema se encarrega de todo o processo de captura, disponibilizando o resultado final na memória para o próximo estágio.

3.2 Processamento

Estando a imagem em formato digital, seja ela capturada com o uso de um sensor ou carregada de um arquivo, é possível aplicar diversos algoritmos de processamento com o objetivo de realçar certas características, tais como bordas, ou extrair alguma informação, como a cor e o posicionamento de um determinado objeto. Entretanto, esses algoritmos consomem grande quantidade de recurso de processamento e algumas arquiteturas desenvolvidas para essas tarefas podem se mostrar mais eficientes.

As tarefas de uma aplicação de visão computacional podem ser classificadas em três níveis (baixo, médio e alto), divididos segundo o tipo de processamento, o compartilhamento/acesso aos dados e a forma de controle (Weems [1991]). O tipo de processamento refere-se aos recursos utilizados e estruturas de dados predominantes nos algorítmos. O compartilhamento e acesso depende dos tipos de dados utilizados, a forma com que são acessados e disponibilizados para o restante da aplicação. Finalmente, o controle baseia-se na estrutura de controle dos algorítimos e nas suas interdependências.

Os níveis de processamento com suas características são descritos a seguir (Olk et al. [1995]; Ratha & Jain [1999]; Weems [1991]; Choudhary et al. [1993]):

- **Operações de nível baixo**: trabalham diretamente com a imagem ou parte dela e costumam ser denominadas como processamento de imagens. Geralmente são caracterizadas pela grande quantidade de dados processados, porém atuando em um único pixel por vez ou em pixels próximos (vizinhos). Os acessos à imagem são geralmente realizados de forma ordenada e previsível. Possuem alta possibilidade de paralelismo. *Smooth* e convolução são alguns exemplos de operações de nível baixo;

- **Operações de nível médio**: possuem o objetivo de extrair dados da imagem e organizá-los em estruturas como listas, árvores, etc. Geralmente atuam apenas sobre uma determinada região da imagem (região de interesse). Na maioria dos casos o acesso à imagem, assim como nas operações de nível baixo, também é ordenado e previsível. O paralelismo nas operações de nível médio nem sempre é evidente. As operações de segmentação pertencem a esse grupo;

- **Operações de nível alto**: são aquelas que trabalham com as estruturas de dados extraídas da imagem, utilizando algoritmos de nível médio, com o objetivo de obter alguma informação que permita tomadas de decisão pela aplicação. São algoritmos, em sua maioria, complexos, sequenciais, cujo acesso à imagem ou estrutura de dados é aleatório e não-determinístico. Um importante exemplo dessa classe de processamento é o reconhecimento de objetos.

Considerando o processamento completo em uma aplicação de visão computacional, tem-se como entrada a imagem adquirida na fase de captura. Algoritmos básicos, compostos em sua maioria por operações aritméticas simples pertencentes às operações de nível baixo, são primeiramente aplicados com o objetivo de eliminar ruídos, melhorar contraste e outros efeitos que destaquem uma determinada característica necessária para a aplicação.

Um segundo estágio, já com a imagem tratada, consiste em extrair dados relevantes à aplicação, utilizando-se operações de nível médio, visando tornar possível, em uma etapa seguinte, a tomada de decisões corretas, com o uso de operações de nível alto, finalizando o processo.

A Figura 3.1 apresenta um exemplo prático de uma aplicação real de visão computacional. O sistema funciona em uma fábrica fictícia de caixas de madeira que deseja verificar se o tamanho de seus produtos estão dentro de um limite especificado. Devem ser avaliadas a tampa e a parte inferior da caixa. Nesse exemplo não há interesse em analisar os métodos utilizados para cálculo do tamanho, mas apenas as tarefas e os dados de entrada e saída de cada nível de operação. A entrada do sistema é a imagem

capturada em uma esteira no final da linha de produção que apresenta a caixa com sua tampa aberta (Figura 3.1a). Uma vez que cores do produto não interferem na análise desejada, é possível usar um sensor que identifique apenas com tons de cinza.

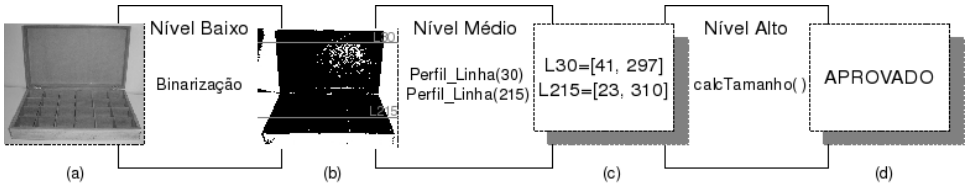

Figura 3.1. Exemplo prático de análise dimensional

A primeira etapa da análise tem o objetivo de realizar um pré-processamento da imagem, visando realçar características importantes. No exemplo apresentado, realiza-se uma operação de binarização da imagem, de forma realçar os limites do objeto que se deseja calcular o tamanho. Basicamente, o processo de binarização consiste em verificar pixel a pixel se seu valor é menor que um determinado limite. Se verdadeiro, o pixel resultante recebe a cor preta, caso contrário, branca. O resultado de um pixel não tem nenhuma influência nos resultados de pixels vizinhos, o que permite uma grande paralelização do algoritmo.

A imagem binária gerada (Figura 3.1b) é então passada ao próximo estágio, que tem o objetivo de extrair dados necessários da imagem, como os limites da caixa. Duas linhas da imagem, sendo uma na altura da tampa e outra na parte inferior da caixa, são estratégicamente escolhidas pelo desenvolvedor para serem analisadas e indicar os limites de cada peça. No exemplo, foram escolhidas as linhas 30 e 215. A análise dessas linhas consiste em criar um perfil, indicando onde ocorre a troca de cor em cada uma delas. Para isso, o acesso à imagem ocorre apenas aos pixels de duas linhas e o resultado são valores gerados com a análise de cada pixel em relação a um outro vizinho. O paralelismo destas operações ainda é bastante claro, mas não tão simples quanto no nível baixo. Criado o perfil de linha, consegue-se então determinar os pixels de inicio e do fim da caixa.

Finalmente, após detectar os pixels limites da caixa (Figura 3.1c), o tamanho pode ser calculado. Para decidir pela aprovação ou reprovação do produto (Figura 3.1d), é necessário configurar o sistema com os limites aceitos. Esses limites podem ser fornecidos em pixels ou qualquer outra medida desejavel. No caso de outras medidas, o sistema deverá possuir uma função de conversão. Pode-se observar que as operações realizadas aqui são puramente matemáticas, sem nenhuma referência direta a imagem. Outra característica importante nesse sistema é a sua completa dependência da aplicação. As técnicas de binarização e perfil de linha utilizadas nos níveis baixo e médio

podem ser aplicadas a outras inspeções visuais, entretanto, o cálculo do tamanho é bastante específico.

Apesar desta visão sequencial do processamento, onde primeiro são executados algoritmos de nível baixo, seguidos pelos algoritmos de nivel médio e alto, ser considerada inadequada atualmente, a classificação segundo a característica de cada algoritmo continua sendo válida. Entretanto, admite-se que o uso dos algoritmos pode-se ocorrer em qualquer sequência, segundo a necessidade da aplicação, independente de sua classificação, ou seja, pode-se, por exemplo, iniciar o processamento com um algoritmo de nível baixo, utilizar um de nível médio, voltar a utilizar um algoritmo de nível baixo, etc.

Ao avaliar todo o processo executado por uma aplicação de visão, pode-se então perceber que as tarefas exigem esforços variados de diferentes recursos computacionais. Desta forma, conclui-se que uma única arquitetura de processador não é capaz de realizar todas estas operações com eficiência, havendo a necessidade de uma configuração híbrida de processamento, com arquiteturas específicas para cada nível (Levialdi [1988]; Wang et al. [1996]).

Na arquitetura proposta nesse trabalho, a captura da imagem pode ser incorporada ao próprio sistema, através de um sensor. Dessa forma, o primeiro passo do processamento será feito internamente, antes de disponibilizar qualquer resultado para o usuário. Tendo em vista essa característica, foi utilizada uma arquitetura de processamento que melhor atenda às operações de nível baixo. Operações de nível médio e alto também podem ser executadas internamente, possivelmente com um desempenho menor, ou repassadas a um elemento de processamento externo.

A experiência no desenvolvimento de aplicações mostra que a eficiência da maioria dos sistemas é determinada pela qualidade do mapeamento realizado pelo software da aplicação no hardware (Nickolls & Reusch [1993]), ou seja, quanto mais natural for a descrição do problema dentro da arquitetura, melhor será o resultado obtido. Baseando-se nesse princípio, foram feitos estudos, apresentados nas sub-seções seguintes, de como devem ser as camadas de hardware e de software para melhor atenderem ao problema proposto.

3.2.1 Definições de hardware

Os algoritmos classificados como operações de nível baixo possuem como elemento de entrada uma matriz de inteiros, geralmente de oito bits, contendo os dados da imagem. Cada pixel é acessado através de uma coordenada e muitas das operações possuem resultados binários. A Tabela 3.1 contém alguns exemplos de algorítmos de nível baixo.

Transformação	Bordas	Segmentação
Threshold	Sobel	Crescimento de regiões
Mediana	Canny	Excentricidade
Correção de cinzas	Transformada de Hough	Segmentação por histograma
Convolução Gaussiana	Contraste médio	Conectividade

Tabela 3.1. Algoritmos de Nível Baixo

Algoritmos como *threshold* e correção de níveis de cinza são exemplos de operações pontuais na imagem, sem influência de pixels vizinhos. Convoluções, como Sobel e Gaussiana, envolvem operações com pixels vizinhos e o resultado geralmente é calculado com o uso de uma máscara. Operações como a mediana envolvem o acesso a dados vizinhos e usam como estrutura de dados um vetor ordenado com os valores para definir o valor do pixel na imagem final.

As operações de baixo nível são, em geral, de controle bem simples, com apenas uma *thread* de processamento, sendo predominantemente de números inteiros. Basicamente, os algoritmos são compostos por operações booleanas e aritméticas simples, aplicadas repetidamente em cada pixel da imagem processada. Devido à estrutura matricial (bidimensional) de uma imagem, as arquiteturas voltadas para o processamento de baixo nivel procuram seguir essa organização na memória, agilizando o acesso. (Weems [1991]).

Apesar da grande quantidade de diferentes arquiteturas propostas para o processamento de nível baixo, há um consenso entre os pesquisadores em que as arquiteturas baseadas no modelo *Single Instruction, Multiple Data* (SIMD) são as mais eficientes nesse contexto (Sunwoo & Aggarwal [1990b]; Olk et al. [1995]; Hammerstrom et al. [1996]; Wu et al. [2007]).

Em arquiteturas SIMD uma mesma instrução pode ser executada paralelamente em múltiplos processadores, onde cada um deles possui um dado diferente a ser processado, explorando o paralelismo de dados (Hennessy & Patterson [2007]). A grande aceitação do modelo SIMD para processamento de imagens se deve, inicialmente, a uma boa relação "performance por dolar"em problemas com paralelismo de dados, ou seja, melhor custo-benefício. Sua abordagem é altamente recomendada para aplicações com dependência entre dados, de operações aritméticas simples e alto *throughput* (Parhami [1995]; Sung [2000]). Dentre as principais arquiteturas de visão que se baseiam na arquitetura SIMD para as operações de nível baixo podemos citar *Cellular Logic Image Processor* (CLIP) (Fountain et al. [1988]), *Distributed Array Processor* (DAP) (Reddaway [1973]), *Massively Parallel Processor* (MPP) (Potter [1983]), *Geometric Array Parallel Processor* (GAPP) (Dyer [1989]) e MasPar (MP) (Nickolls & Reusch [1993]).

Estas arquiteturas, apesar de antigas, foram as pioneiras nessa classe de processamento, servindo como base para grande parte dos trabalhos posteriores, até os dias atuais.

3.2.2 Definições de software

A principal importância da camada de software é mapear a aplicação na arquitetura utilizada. De forma simplificada, esse mapeamento pode ser entendido como compreender as tarefas da aplicação, configurar os recursos de hardware e direcionar o fluxo de dados entre eles afim de executá-las. Quando se utiliza um software genérico compatível com a arquitetura, esse mapeamento é realizado de forma bastante simples, não explorando por completo os recursos disponíveis no hardware. Com a utilização de softwares desenvolvidos especificamente para a arquitetura, os recursos são melhor administrados, garantindo melhor performance para a aplicação.

A atividade de compreender as tarefas da aplicação consiste essencialmente em receber dados externos ao sistema e interpretá-los. Esses dados podem ser provenientes de diversas fontes, como, por exemplo, interfaces gráficas acessadas pelo usuário, comandos de voz, botões mecânicos ou ainda de outros sistemas, utilizando-se protocolos de comunicação. Para cada origem do dado é necessário um tipo de tratamento diferente, uma vez que, além da forma da sua leitura, sua interpretação pode variar.

A Figura 3.2 apresenta os diversos níveis de abstração na atividade de trocar uma lâmpada, onde em cada nível se exige uma interpretação diferente. Nesse exemplo, a escada representa um recurso de hardware e o operário o processador que executa a tarefa. No Nível 4 a tarefa é definida de forma bem abstrata, sendo refinada até o Nível 1, onde o operário possui diversas tarefas pequenas e objetivas.

Figura 3.2. Diversos níveis de abstrações na atividade de trocar uma lâmpada

Observe que quanto maior o nível de abstração do comando, menor é a quantidade de dados necessários para descrevê-lo e maior deve ser o conhecimento do operário para executá-lo. Desta forma, tem-se comandos mais próximos da sua linguagem e ambiente cotidiano, permitindo que até aquelas pessoas menos experientes possam

usá-lo. Entretanto, limita-se as opções de configuração e personalização da tarefa, não permitindo, por exemplo, escolher outro local para buscar a lâmpada.

Nos sistemas de visão computacional, assim como na maioria dos sistemas, é desejável que o usuário consiga programá-los e configurá-los de uma forma natural, sem precisar de grandes aprendizados. Cabe, então, às camadas de software estabelecerem os possíveis níveis de abstração, definindo comandos simples e ocultando detalhes de configuração do hardware.

Um risco ao se definir internamente no software detalhes da execução de um comando é a perda da flexibilidade. Aplicações de visão são bastante diversificadas, exigindo da arquitetura uma grande capacidade de adaptação a cada ambiente. Diante desse cenário, surge o desafio de encontrar o ponto de equilíbrio ideal entre comandos simples e a flexibilidade do sistema.

Os comandos oferecidos por um sistema de visão, além de dependerem da aplicação e do grau de instrução do usuário que o utiliza, estão diretamente relacionados aos algoritmos de processamento de imagens implementados. No exemplo apresentado, o operário, mesmo com as operações do Nível 1 bem definidas, ainda precisa saber como ir ao estoque, movimentar a escada, etc. Todas essas funções devem fazer parte de um conjunto de conhecimentos prévios, facilmente acessados. De forma análoga, funções elementares de processamento devem estar acessíveis ao processador na forma de bibliotecas.

Diversos pacotes oferecem um extenso conjunto de funções de processamento e de visão computacional que podem ser usadas por vários kits de desenvolvimento. Dentre os pacotes mais conhecidos encontram-se OpenCv (OpenCV [2009]), DirectX (Microsoft [2009]) e MIMAS (Devices [2009g]). Cada um desses pacotes possui características específicas que os diferenciam e os tornam recomendados para determinados tipos de aplicações. Ao utilizá-los, o sistema de visão incorpora todas as suas funcionalidades, auxiliando no desenvolvimento. Outra vantagem ao permitir a inclusão de bibliotecas é oferecer ao usuário a opção de utilizar pacotes de seu conhecimento por ele, reduzindo, assim, o tempo de aprendizado.

Na utilização de hardwares híbridos, a classificação do tipo de operação da função também deve ser determinada pelo software, que alocará os recursos de forma mais otimizada segundo o nível de processamento requerido. Observa-se aqui que as características básicas indicadas para cada nível de processamento interfere diretamente no mapeamento do algoritmo para uma determinada arquitetura, influenciando o seu desempenho. Um algoritmo de nivel alto, por exemplo, terá um péssimo desempenho em arquiteturas paralelas. Essas características devem ser então detectadas e utilizadas pelo software para definir qual a melhor forma de se utilizar os recursos de hardware

disponíveis, principalmente em arquiteturas híbridas, onde se encontram diferentes paradigmas de programação.

Em hardwares específicos para um determinado nível de processamento, como proposto na DSCam, as bibliotecas devem conter principalmente funções com operações do nível suportado, procurando oferecer algoritmos eficientes naquela arquitetura. Outras funções, entretanto, consideradas essenciais em visão computacional, também devem fazer parte, mesmo que executadas com um baixo desempenho.

Finalmente, é importante ressaltar que o processamento e análise de imagens são caracterizados por soluções específicas. Desse modo, técnicas que funcionam bem em uma aplicação podem se mostrar totalmente inadequadas em outra (Gonzalez & Woods [2000]). Tudo o que é oferecido por plataformas de hardware e software de visão computacional é apenas um ambiente básico que auxilie no desenvolvimento da aplicação em si, sendo de competência do programador selecionar aquela mais adequada para as tarefas a serem realizadas.

3.3 Disponibilização de resultados

Após processar a imagem adquirida, é necessário finalizar o processo, transmitindo o resultado para o usuário, seja ele uma pessoa ou outro sistema. Em um sistema de visão computacional, pode-se obter diferentes tipos de resultados, de acordo com o objetivo da aplicação e o nível de processamento executado.

Um resultado comum, encontrado na maioria das aplicações, é a imagem resultante do processamento de nível baixo ou o resultado do processamento de nível alto plotado na própria imagem original (destaca-se um objeto localizado, por exemplo). Entretanto, essa representação pode ser inadequada em muitos casos onde o resultado não será analisado por seres humanos, gerando incompatibilidade entre sistemas.

Um sistema completo de visão computacional deve ser capaz de fornecer resultados de qualquer nível de processamento, disponibilizando estruturas de dados e outros tipos de informações obtidas. Em muitos casos, por exemplo, pode-se precisar unicamente de um sinal de conforme/não conforme, resultante de um processamento de nível alto.

Para o envio desses resultados, é necessário que a plataforma disponha de interfaces de hardware conhecidas e difundidas, além de protocolos simples, seguros, fáceis de serem implementados e que suportem os tipos de dados que serão transmitidos.

3.4 Modularidade

Entende-se como modularidade a técnica de desenvolvimento onde blocos que executam uma determinada tarefa são implementados e podem ser executados de forma independente, como uma caixa-preta, comunicando-se com o restante da aplicação através de interfaces. Uma vantagem dessa técnica consiste no fato de que, uma vez que uma tarefa está inteiramente encapsulada e com uma interface de comunicação bem definida, o módulo pode ser substituído por outro de mesma interface, alterando a forma que a tarefa é realizada, sem nenhum efeito colateral para o restante da aplicação. Conceitualmente, módulos podem ser compilados juntamente com a aplicação ou de forma independente.

O desenvolvimento de um sistema de visão pode ser considerado, em uma análise superficial, uma atividade que basicamente consiste em ordenar e unir caixas-pretas que realizam algum tipo de processamento de imagens (Thoren [2002]). A modularidade, desta forma, seria uma característica inerente a esses sistemas. Entretanto, características peculiares existentes em cada aplicação não permitem que o processo ocorra desta forma, exigindo que o conteúdo das funções seja adaptado à cada situação. O que ocorre nesses casos é o uso de caixas-brancas, onde o programador tem acesso ao seu conteúdo, mas mantém a independência de sua execução.

Tarefas como captura e processamento de imagens podem utilizar a estrutura de módulos, oferecendo ao usuário a capacidade de escolher como realizar a captura (configurações específicas do sensor ou abrir um arquivo de imagem) e quais implementações de algoritmos serão utilizadas. As demais tarefas que interagem com o usuário e outros sistemas também podem se apresentar interessantes para serem modularizadas.

3.5 Extensibilidade

A extensibilidade é uma importante característica a ser observada no desenvolvimento de sistemas onde considera-se a capacidade de um crescimento futuro (wik [2009]). Para uma plataforma de visão computacional, esse requisito é um grande passo para resolver o problema da especificidade das aplicações, que necessitam de algoritmos ajustados especialmente para cada situação (Hammerstrom et al. [1996]).

Aliada à característica da modularização, a extensibilidade permite ao usuário incorporar ao sistema a capacidade de se realizar determinada tarefa, aumentando o escopo atendido pela plataforma (Thoren [2002]).

Em sistemas de visão computacional a extensibilidade possibilita principalmente a adição de algoritmos de processamento de imagens e visão, muitas vezes adaptados

especificamente para uma determinada aplicação. Bibliotecas inteiras de processamento podem ser incorporadas e utilizadas em uma aplicação, permitindo ao sistema se adequar ao ambiente em que é utilizado.

3.6 Compatibilidade e portabilidade

Diz-se que um sistema é compatível com outro quando eles têm a capacidade de se interagirem. Essa interação pode ser um software comunicando-se com outro, executando sobre um processador ou um elemento de hardware comunicando-se com outros. A portabilidade nasce da compatibilidade de um software com diferentes sistemas operacionais e plataformas de hardware desenvolvidas por diferentes fabricantes. Essa definição é extendida aqui para um elemento de hardware que é compatível com diversos outros.

Um sistema de visão desenvolvido para ter portabilidade pode ser criado para um determinado ambiente e adaptado a outros com um esforço mínimo. Desenvolver sistemas de grande portabilidade pode permitir que esses sejam compatíveis com futuras plataformas, garantindo o seu funcionamento por um tempo prolongado, minimizando a necessidade de atualizações (Thoren [2002]).

Outra importância de se desenvolver uma arquitetura com grande compatibilidade é a variedade de outros sistemas e interfaces que podem compor o ambiente da aplicação de visão computacional, com os quais o sistema poderá interagir.

3.7 Considerações

Ao estudar um sistema de visão computacional completo, pode-se perceber que ele deve ir muito além que o processamento de imagens. Para ser completo, o sistema deve ser capaz de funcionar em diversos tipos de aplicações, com interfaces para interagir com humanos e outros sistemas, oferecendo suporte a vários dispositivos de entrada e saída de dados, além da flexibilidade nos algorítmos de tratamento e análise das imagens.

Um determinado requisito deve estar sempre de acordo com os demais para que todo o sistema funcione conforme esperado. Exigir que o processamento seja feito rapidamente não é o suficiente para obter um sistema eficiente. É necessário também que todos os demais requisitos se adequem dentro das especificações de tempo, capturando imagens e disponibilizando os resultados em uma velocidade compatível, por exemplo.

Os fatores analisados nesse capítulo procuraram apresentar de forma geral as características de um sistema de visão, mas não esgotam os seus possíveis requisitos.

Diversos outros poderiam ser incluídos, muitos deles encontrados na literatura (Thoren [2002]), entretanto foram considerados específicos de determinadas classes de aplicações de visão.

Capítulo 4

DSCam - Digital Smart Camera

A DSCam é uma plataforma desenvolvida para ser utilizada em aplicações de visão computacional comerciais ou até mesmo por amadores, sendo uma plataforma de baixo custo, atendendo de forma satisfatória aos requisitos especificados no Capítulo 3. A plataforma consiste de duas partes distintas, conforme ilustrado na Figura 4.1, sendo um software para configuração e operação e uma câmera com processamento embarcado. A comunicação entre as partes é realizada por uma rede TCP, como a internet.

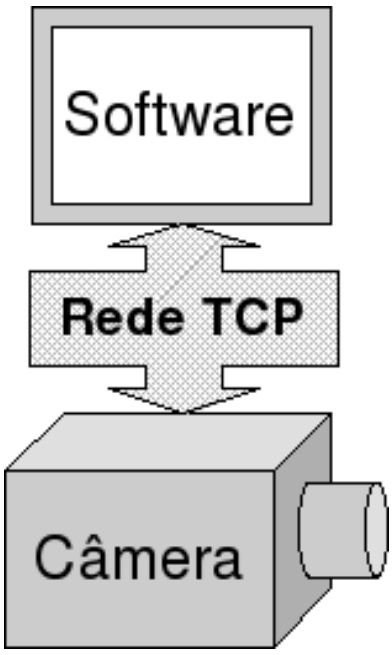

Figura 4.1. Partes que compõem a plataforma DSCam.

A plataforma DSCam possui recursos desenvolvidos especificamente para a manipulação de imagens, permitindo que ela seja facilmente configurada e utilizada com

eficiência em um grande número de aplicações de visão computacional. Através de sua interface gráfica, pessoas sem prática no desenvolvimento de sistemas podem rapidamente criar fluxos de processamento de imagens e executá-los no hardware. Todos os detalhes relativos à implementação das funções de visão, estruturas de imagens, comunicação hardware-software e outras configurações técnicas ficam encapsuladas pelo sistema que, além das operações básicas, oferece alguns serviços adicionais, apresentados neste capítulo.

Uma vez que a plataforma foi projetada para atuar logo no início do processo de um sistema de visão computacional, na etapa seguinte à captura, optou-se por especificá-la de forma atender principalmente aos requisitos do nível baixo de processamento de imagens. Dessa forma, as arquiteturas empregadas e funções de visão incorporadas como exemplos foram escolhidas baseando-se nas características desse nível de processamento, apresentadas no Capítulo 3.

4.1 Especificação geral

A DSCam é uma plataforma projetada e desenvolvida para compor aplicações de visão computacional comerciais, com o objetivo de ser facilmente programável, compatível com diversos ambientes e com flexibilidade para se adaptar aos mais diferentes cenários em que são utilizados sistemas de visão.

O software de configuração e operação oferece ao usuário uma interface gráfica que permite o acesso a todos os recursos da plataforma, simplificando o processo de desenvolvimento e de comunicação com o hardware. Utilizando-se a interface, o usuário pode facilmente definir um fluxo de execução, escolhendo quais funções devem ser aplicadas na imagem, visualizar as imagens processadas, além de enviar comandos de captura e execução do fluxo. Todo o software foi desenvolvido em Java, podendo ser executado na maioria dos sistemas operacionais atuais.

O módulo da câmera consiste de um processador digital de sinais (*digital signal processor* – DSP) executando uma aplicação (*middleware*) que realiza o controle das funções básicas, como a captura, da comunicação com o software e das operações do fluxo de execução definido pelo usuário. Cada câmera possui uma interface ethernet, podendo ser conectada a uma rede local ou até mesmo à internet. A aplicação trabalha como um provedor de serviços, aguardando a conexão de um usuário e a solicitação de uma determinada operação. Os comandos recebidos são tratados por uma máquina de estados, executados e respondidos ao usuário.

Através de um software em seu computador, o usuário define todo o fluxo de

execução a ser aplicado às imagens e o envia para a câmera através da internet. A execução das funções definidas são disparadas por meio de um comando específico, usando a imagem corrente na memória, ou logo após um comando de captura, aplicando as operações sobre a nova imagem. Como será abordado mais adiante, nesse trabalho não foi implementada a função de captura, entretanto, todas as estruturas básicas necessárias para essa operação foram disponibilizadas, incluindo os comandos para acioná-la.

As operações solicitadas no fluxo de execução são sempre funções de bibliotecas, acessadas dinâmicamente com o uso de uma API, conforme é descrito na Sessão 4.3. Desta forma, é possível descarregar para a câmera apenas as bibliotecas que serão utilizadas naquela aplicação, além de permitir o desenvolvimento e o acréscimo de novas funções na plataforma sem grandes esforços, tornando-a bastante flexível e expansível.

Visando facilitar o desenvolvimento de novas bibliotecas ou ainda permitir o uso de bibliotecas já existentes, toda a aplicação executa sobre um sistema operacional, denominado uClinux, que é baseado em uma distribuição Linux. Além desta vantagem, o uso de um sistema operacional estável oferece maior confiabilidade ao projeto, garantindo um correto funcionamento em serviços essenciais como gerência de memória e manipulação de arquivos.

A Figura 4.2 apresenta um diagrama que representa a forma de organização dos módulos da plataforma DSCam.

Como encontra-se representado na figura, o software de configuração e operação é executado em um computador e se comunica com o *middleware* da câmera, executada no DSP, exclusivamente por meio de uma rede TCP/IP. Esta independência entre as partes tem como principal objetivo permitir que outros softwares e/ou outras câmeras sejam desenvolvidos e, uma vez que respeitem o protocolo de comunicação definido (Anexo B), possam ser integrados à plataforma.

Essa característica de independência permitiu também que cada uma das partes se tornasse um trabalho de dissertação distinto, possibilitando a cada aluno aprofundar mais em seu desenvolvimento. O software de configuração e operação é resultado da dissertação do aluno Antônio Celso Caldeira Junior, apresentado ao Programa de Pós-Graduação do Departamento de Ciência da Computação da Universidade Federal de Minas Gerais no mês de março do ano de 2009. A arquitetura do hardware e do *middleware* executado pelo DSP é resultado do presente trabalho, sendo esses apresentados na continuidade deste capítulo e cujos experimentos e análises encontram-se descritos nos capítulos seguintes.

O hardware utilizado no desenvolvimento da câmera foi escolhido baseando-se nas características do processamento de nível baixo de visão apresentadas no Capítulo

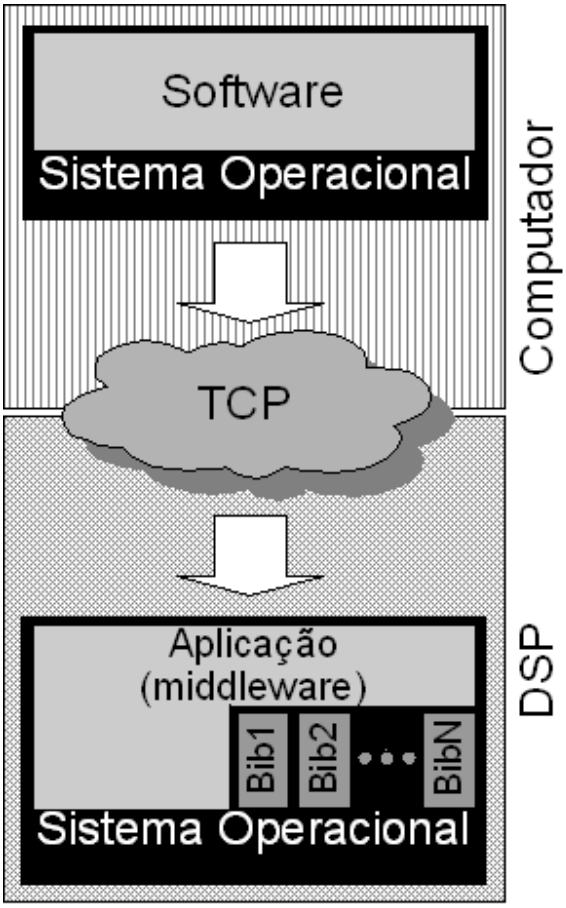

Figura 4.2. Diagrama da forma de organização dos módulos da DSCam.

3 e no custo de aquisição, montagem e desenvolvimento para cada opção existente. O DSP Blackfin (Devices [2009a]), desenvolvido pela Analog Devices Inc. (ADI) (Devices [2009d]), foi o que apresentou a melhor relação custo-benefício para as aplicações a serem atendidas pela plataforma, sendo um produto facilmente encontrado no mercado, com diversas ferramentas de desenvolvimento e extensa documentação. Outra grande vantagem do Blakfin para esse projeto é a implementação da interface PPI, proprietária da própria ADI. PPI é um barramento de comunicação *half-duplex* capaz de transmitir dados de até 16 bits. Possui um pino de clock dedicado e três sinais de sincronização de *frame* que permite uma fácil e precisa interação com sensores, encoders/decoders de vídeo e outros dispositivos analógicos. No seu modo de operação de maior throughput pode transmitir 2 bytes de dados simultaneamente.

O *middleware*, desenvolvido em C, é composto por diversos módulos de forma permitir a adaptação e a inclusão de novos recursos com o menor esforço possível. A Figura 4.3 mostra cada um desses módulos e suas interações.

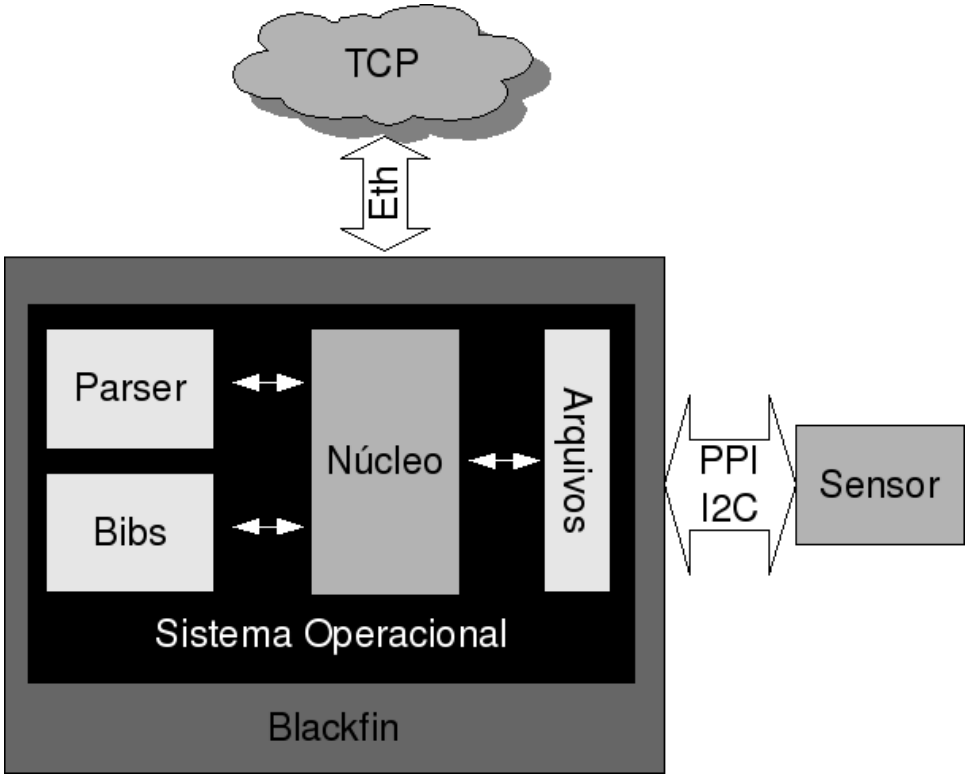

Figura 4.3. Módulos da aplicação DSCam.

Composto por um núcleo principal, o *middleware* é responsável por gerenciar todas as tarefas a serem executadas pela câmera. Através de um *socket* TCP, o núcleo recebe os comandos, os interpreta, executa a ação correspondente e envia uma resposta para o usuário. Todos os comandos são compostos por caracteres ASCII, podendo ser enviados até mesmo de um terminal telnet, caso o usuário conheça bem o protocolo de comunicação.

O fluxo de execução é definido em um arquivo enviado do software para o núcleo, que o mantém armazenado em memória local. Esse arquivo, denominado *Arquivo de Descrição*, deve conter todas as operações a serem aplicadas na imagem, com seus parâmetros e as APIs em que se encontram. O arquivo de descrição é decodificado pelo módulo *Parser* que interpreta os dados nele contidos e os carrega em uma estrutura interna do núcleo, a Lista de Execução. O formato em que os dados estão descritos no arquivo de descrição pode ser personalizado, sendo necessário para isso adaptar o Parser. Neste trabalho utilizou-se no Arquivo de Descrição uma linguagem baseada no XML para a representação do fluxo, cuja sintaxe é apresentada no Anexo A.

As funções de processamento de imagens e visão disponíveis para serem aplicadas na imagem encontram-se implementadas em bibliotecas. Desta forma é possível o usuário adicionar as funções que desejar, implementando novas ou utilizando pacotes

já existentes, como o OpenCv (OpenCV [2009]). Para integrar a biblioteca à aplicação, deve-se criar uma API com um formato pré-defindo, conforme descrito na sessão 4.3. Esta API será responsável por receber as informações do núcleo e convertê-las para um formato compatível com a biblioteca. A Figura 4.4 ilustra todo o caminho de configuração da DS-Cam, desde a criação do Arquivo de Descrição no software até a sua carga para a execução.

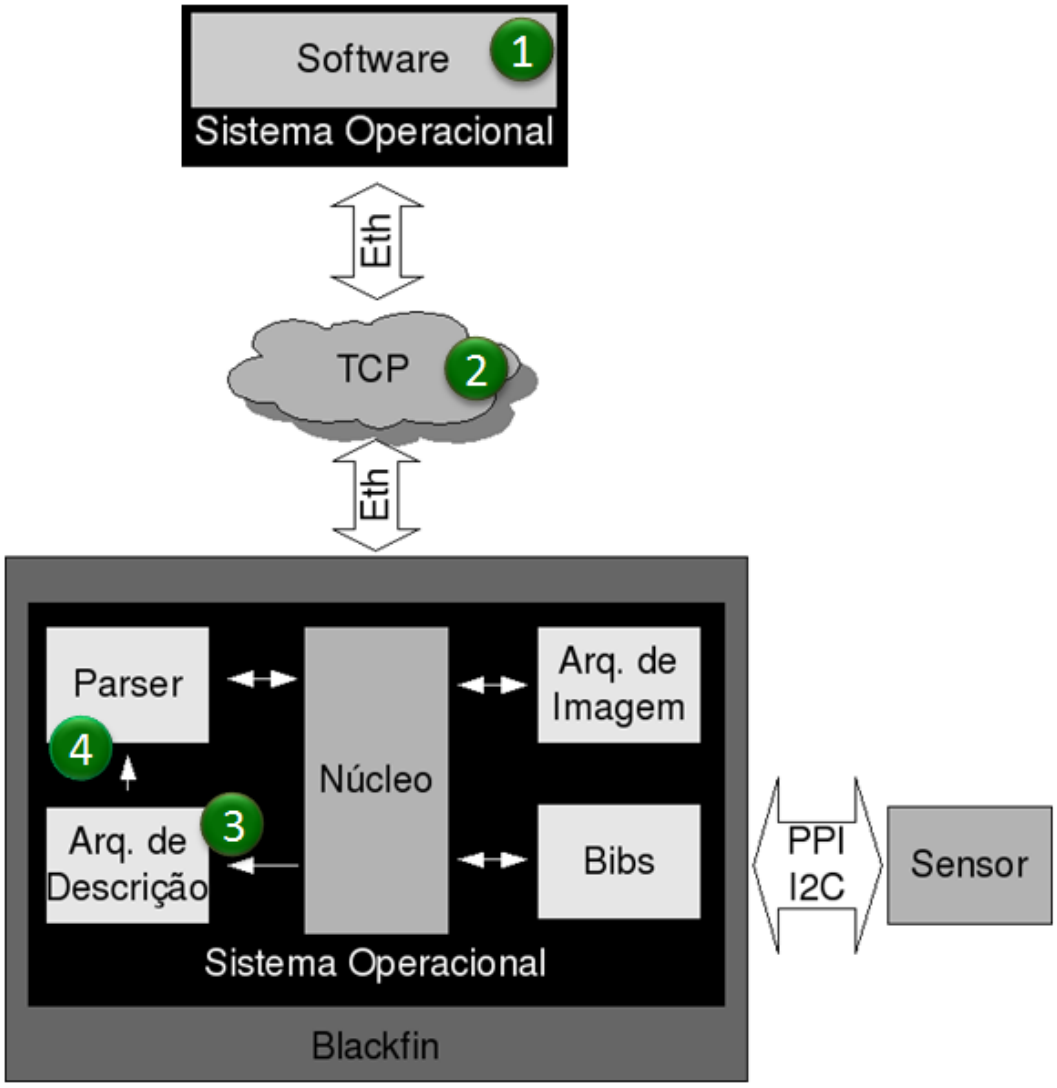

Figura 4.4. Fluxo de configuração da DS-Cam.

1. Usuário define fluxo de execução e gera Arquivo de Descrição;
2. Arquivo de descrição é enviado para a câmera pela rede;
3. Núcleo da aplicação recebe o arquivo e o salva em memória local;

4. Parser lê o arquivo e carrega os dados na Lista de Execução.

Ao receber um comando de execução do fluxo, um ponteiro é direcionado para o primeiro elemento da Lista de execução. Cada elemento dessa lista corresponde a uma função a ser aplicada sobre a imagem, contendo todas as informações necessárias para a sua execução. Ao carregar um elemento, a API a qual pertence a função é identificada e aberta, recebendo todas as demais informações necessárias à execução da tarefa (imagem, parâmetros, tipo de retorno, etc). Após o final da função, o ponteiro avança para o próximo elemento da lista, repetindo o processo de execução, até o fim dos elementos ou a execução do comando *halt*. Funções de desvio e condicionais são as únicas que podem alterar a posição do ponteiro.

Apesar de ter sido projetada para adquirir imagens em tempo real de um sensor, não foi implementado nenhum módulo de captura. Entretanto, a plataforma DSCam permite acessar imagens armazenadas em arquivos e salvar a imagem resultante do processamento. O acesso aos arquivos é realizado pelo módulo *Arquivos*, que possui funções de abrir e salvar imagens. Esse módulo é composto de diversos submódulos, onde cada um é capaz de abrir um tipo de arquivo distinto. Originalmente, foi implementado apenas um submódulo para trabalhar com arquivos de imagens no formato PGM Furlong [2009]. Outros formatos podem ser implementados pelo usuário e facilmente acrescentados à plataforma.

4.2 Arquitetura

Conforme descrito anteriormente, a plataforma DSCam foi proposta e projetada para trabalhar logo após a captura da imagem. Nessa primeira etapa geralmente são utilizados algoritmos de processamento de nível baixo, onde, em sua maioria, recebem a imagem original como entrada, aplicam uma determinada operação pixel a pixel e, como resultado, tem-se uma imagem tratada conforme às necessidades da aplicação, ressaltando algumas características ou eliminando falhas como ruídos. Outra característica marcante no processamento de nível baixo é o acesso de forma previsível e ordenado aos pixels, realizando sempre uma mesma operação sobre todos eles.

Compreendendo estas características do problema abordado e buscando na literatura as soluções adotadas em outros trabalhos similares (Sunwoo & Aggarwal [1990b]; Olk et al. [1995]; Hammerstrom et al. [1996]), foi possível observar que há uma grande aceitação de arquiteturas SIMD para processamento de imagens de nível baixo. No Capítulo 3 há uma discussão mais completa sobre o assunto, sendo utilizado como embasamento teórico para a definição do processador e demais configurações do hardware.

Uma outra característica importante a ser observada em uma arquitetura de processamento de imagens é a existência de um segundo fluxo de processamento, responsável pelo controle das operações do sistema. Além de obter um bom desempenho no processamento da imagem, é necessário que o processador possua recursos para garantir um bom desempenho das operações de controle e auxiliares, como acesso a rede e interação com o usuário.

Em qualquer sistema embarcado, um ítem extremamente importante são as interfaces disponíveis. Para o sistema proposto, há algumas interfaces essenciais que devem ser suportadas pela arquitetura. São elas:

- I2C (também conhecida como TWI): os principais sensores de imagem possuem barramentos diferenciados para dados e controle. Nestes casos, geralmente utiliza-se a interface I2C para controle, transmitindo os dados em um barramento mais veloz;
- Paralela: barramento utilizado na transmissão dos dados de uma imagem nos principais sensores, com velocidade superior à maioria dos barramentos existentes, sendo adequado para transmissão de grandes volumes de dados;
- Ethernet: interface padrão de acesso à redes locais e à internet. Na plataforma proposta, toda a comunicação com o usuário e configuração deverá ser realizada utilizando-se essa interface.

Diante de todas essas características e requisitos citados, optou-se por utilizar o DSP Blackfin 537 (Devices [2009c]), desenvolvido pela ADI, possuindo um núcleo desenvolvido em conjunto com a Intel (*Micro Signal Architecture* – MSA) e suporte a todas as interfaces necessárias descritas. A arquitetura MSA combina as vantagens de um processador RISC tradicional e dois multiplicadores com as vantagens de um microcontrolador de propósito geral e o paralelismo da técnica SIMD, resultando em um processador eficiente tanto em operações de controle quanto em operações de imagem.

Após especificadas as necessidades de hardware e ter escolhido o processador, a etapa seguinte do projeto consistiria no desenvolvimento do circuito elétrico para a arquitetura proposta. Entretanto isso está além do escopo definido para este trabalho e não seria um esforço justificável, uma vez que já existem plataformas de hardware que atendem às necessidades especificadas. Optou-se então por utilizar o kit de desenvolvimento *Ez-Kit BF537 Lite* (Devices [2009b]) que atende aos requisitos do projeto e focar os esforços na aplicação a ser executada sobre o hardware.

O *Ez-Kit BF537 Lite* é uma plataforma de desenvolvimento de baixo custo para o DSP Blackfin 537, desenvolvida pela ADI para demonstrar os recursos do seu pro-

cessador. Além de utilizar o DSP escolhido para o projeto, o *Ez-Kit BF537 Lite* disponibiliza o acesso a todas as interfaces indicadas anteriormente, tornando-se uma plataforma de desenvolvimento ideal para o sistema proposto.

Para um gerenciamento seguro e confiável do hardware, optou-se por utilizar uma distribuição de Linux desenvolvida para processadores que não possuem uma unidade de gerência de memória (MMU), denominada uClinux (lê-se *you see linux*). Além da segurança de se estar utilizando um ambiente estável, outras vantagens são obtidas com o uso do uClinux. Por criar uma camada de abstração do hardware, o desenvolvimento de aplicações torna-se menos suceptível a erros, ocupando um tempo menor de desenvolvimento. A utilização de uma estrutura bastante semelhante à de um Linux tradicional proporciona ao programador um ambiente mais amigável, além de permitir o teste de algumas aplicações até mesmo no computador pessoal. Entretanto, o uClinux não é um sistema operacional de tempo real, podendo não trazer resultados satisfatórios quando o tempo é uma condição rígida. Seus desenvolvedores, por sua vez, garantem um bom desempenho do sistemas em aplicações de tempo real, alegando uma resposta eficiente caso as prioridades forem configuradas corretamente, conforme a necessidade. A ausência de *threads* e processos paralelos são outras limitações do sistema e que, caso fosse possível, poderiam acelerar o processamento das imagens capturadas.

4.2.1 DSP Blackfin

Blackfin é uma família de processadores de 16-32 bits embarcados desenvolvidos especificamente para atender à demanda computacional e o consumo de energia requerido atualmente pelas aplicações de áudio, vídeo e comunicação. Baseado na arquitetura *Micro Signal Architecture* (MSA), desenvolvida em uma parceria da ADI com a Intel, o Blackfin combina as características um processador RISC de 32 bits e dois multiplicadores-acumuladores de 16 bits para processamento de sinais com um conjunto de atributos encontrados em microcontroladores de propósito geral. Essa combinação permite ao Blackfin oferecer uma boa performance tanto para processamento de sinais quanto nas aplicações de controle.

O Blackfin contém dois bancos de memória interna, L1 e L2, além de permitir a adição de memória externa, podendo chegar até a 4GB de espaço. Todos os seus registradores são mapeados em memória, que é endereçada em bytes e possui uma arquitetura Von Neumann. Visando atingir rápidas taxas de transmissão para imagens e audio, além de liberar o processador para outras tarefas, o Blackfin possui um canal DMA dedicado para cada periférico, possibilitando operações como processamento de

vídeo em tempo real.

O núcleo do Blackfin é composto por um processador SIMD de 16-32 bits de alto desempenho, possuindo um *pipeline* de dez estágios e palavras de instrução de tamanhos variados. Visando aumentar ainda mais sua performance para aplicações de audio e vídeo, o Blackfin possui instruções exclusivas para processamento multimídia. A Figura 4.5 apresenta a arquitetura interna do Blackfin.

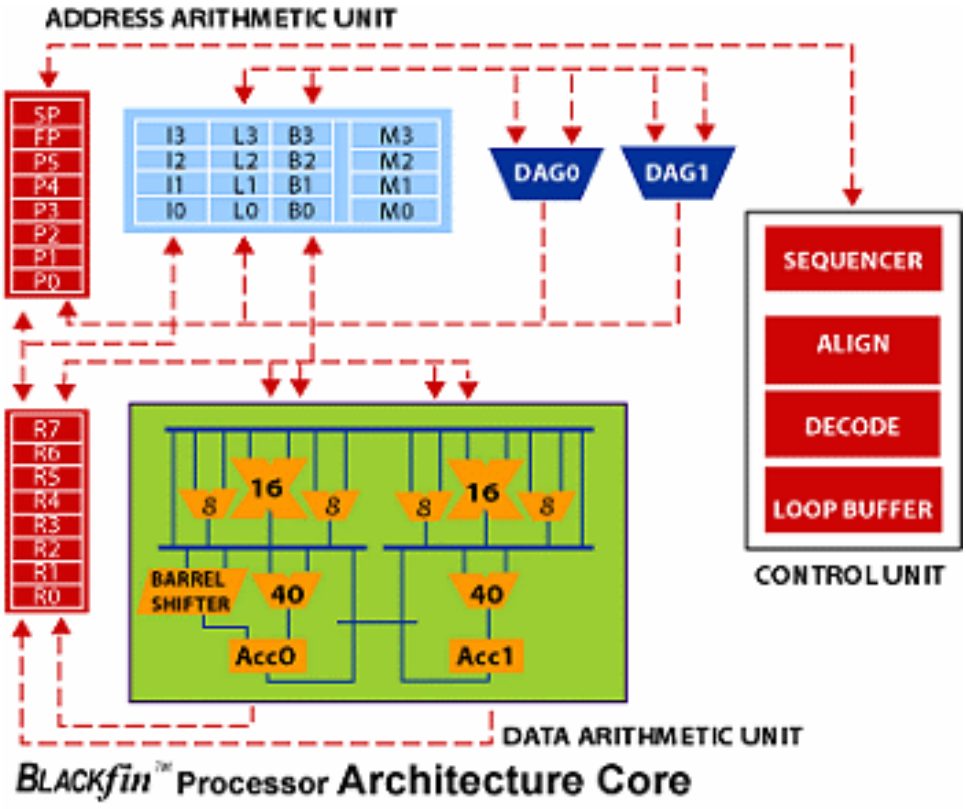

Figura 4.5. Arquitetura interna do Blackfin (Devices [2009f]).

À esquerda na figura da arquitetura do Blackfin encontram-se os oito registradores de 32 bits de propósito geral (R0 a R7) e os registradores de controle como o *stack pointer* e o *frame pointer*. Para cada ciclo de clock, podem ser realizadas até duas escritas e duas leituras nos registradores, que também podem ser acessados em partes de 16 bits.

A unidade aritmética do Blackfin, representada ao centro da figura, é composta por dois multiplicadores-acumuladores (MACs) de 16 bits, duas unidades lógica-aritméticas (ULAs) de 40 bits, quatro ULAs de oito bits específicas para operações com imagens e um shifter. Todas estas unidades realizam operações de 8, 16 ou 32 bits, além das ULAs, que podem trabalhar com 40 bits. Todos esses recursos replicados

permitem a execução paralela de operações lógicas-aritméticas, como é sugerido para o processamento de imagens de nível baixo.

À direita na figura encontra-se o controle de sequência e fluxo de execução. Esse controle é responsável por gerenciar operações de desvio, chamadas de sub-rotinas e garantir o correto funcionamento do *pipeline*, resolvendo os conflitos gerados.

Os geradores de endereço (*data address generators* – DAGs) são capazes de prover dois endereços distintos de memória, onde serão realizadas simultameamente as buscas pelas próximas instruções a serem executadas. O endereçamento de instruções pode ocorrer de forma indireta, auto-incremento, auto-decremento, indexação e bit-reverso.

Para o projeto proposto, é necessário utilizar um processador que, além das características de processamento já discutidas, disponibilize as interfaces descritas anteriormente. O DSP Blackfin 537 é uma extensão da família BF531/BF532/BF533, incluindo as interfaces ethernet, CAN, TWI, entre outras. Outra evolução foi a inclusão de recursos adicionais como a flexibilidade na arquitetura da memória cache, gerenciamento de energia e melhorias no DMA, tornando-o o de maior performance da família. A Figura 4.6 apresenta os principais módulos do DSP BF537.

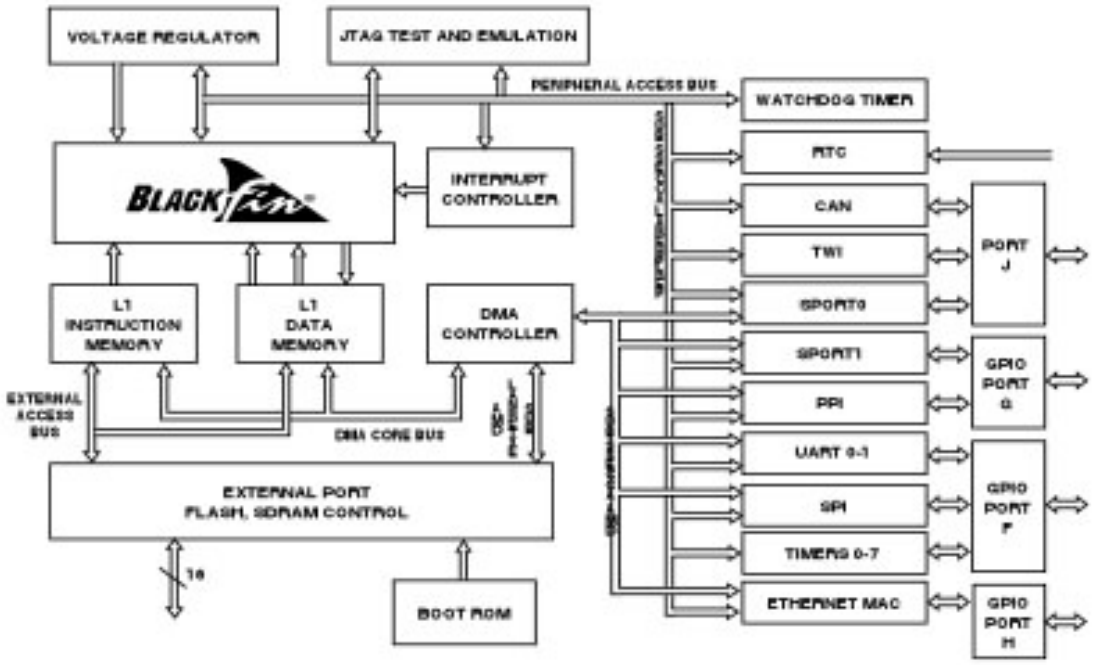

Figura 4.6. *Datapath* do Blackfin 537.

No lado direito da figura é possível observar as interfaces I2C (TWI) e PPI, utilizadas na comunicação com o sensor, e a Ethernet, utilizada para acessar a rede.

Como plataforma de desenvolvimento para o Blackfin 537, a ADI possui um kit

cujo objetivo é demonstrar as funcionalidades e o desempenho de seu processador, permitindo a elaboração de protótipos e eliminando, assim, a necessidade e os custos de criação de um circuito na fase de prototipação. A Figura 4.7 apresenta a arquitetura do *BF537 Ez-Kit Lite*.

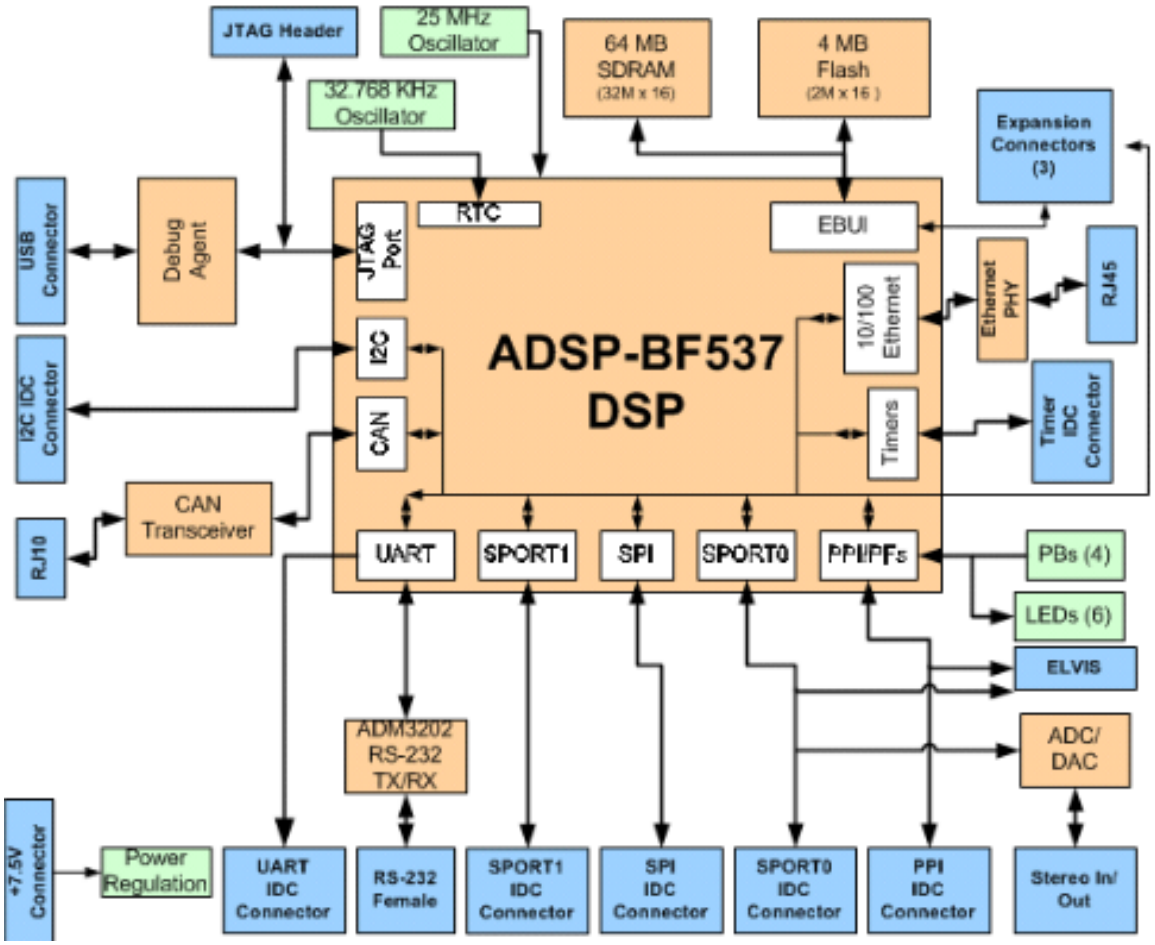

Figura 4.7. Arquitetura Geral do *BF537 Ez-Kit Lite*.

Como é possível observar no diagrama, o *BF537 Ez-Kit Lite* oferece ao desenvolvedor diversos recursos acoplados ao processador. Recursos como o *Debug Agent* com uma interface USB para ser conectada a um computador, tornam o processo de depuração mais eficiente, impactando diretamente no tempo de desenvolvimento e na qualidade da aplicação desenvolvida. Interfaces ADC/DAC e *drivers* RS-232, dentre outros periféricos, complementam o Blackfin, possibilitando o desenvolvimento de um maior número de aplicações. Um periférico importante para a plataforma DSCam é a interface IEEE 802.3 10/100 Ethernet, que permite a conexão do kit a uma rede, como a internet.

O *BF537 Ez-Kit Lite* possui dois módulos externos de memória, sendo 4MB

de memória Flash e 64MB de SDRAM. A memória Flash é utilizada para manter os programas a serem executados no Blackfin, incluindo o boot, enquanto a SDRAM é utilizada durante a execução. Ao ser energizado, o Blackfin busca em um endereço específico da memória Flash a sequência de inicialização (boot), responsável por inicializar os periféricos e carregar a aplicação desejada. A quantidade de memória SDRAM disponível é extremamente importante para a plataforma DSCam. Uma imagem geralmente ocupa grandes quantidades de memória e muitas vezes pode ser necessário carregar duas ou mais simultâneamente. Um limite baixo para o número de imagens carregadas pode limitar o uso para algumas aplicações, o que não seria interessante. Além disso, as bibliotecas de imagem podem utilizar estruturas de dados de tamanho igual ou próximo ao da imagem. Sendo assim, para atender a todas essas características, é necessário uma quantidade elevada de memória SDRAM disponível.

Para auxiliar na gerência dos dispositivos de hardware e fornecer alguns recursos básicos para o desenvolvimento de aplicações, como gerência de memória e arquivos, foi utilizada no kit uma distribuição Linux desenvolvida para o Blackfin. O uClinux é um sistema operacional baseado no Linux adaptado para processadores que não possuam Unidade de Gerenciamento de Memória (MMU). O baixo custo e a existência de uma enorme quantidade de aplicações disponíveis, aliados à facilidade no desenvolvimento de novas aplicações, tornam o uClinux um sistema ainda mais atraente para ser utilizado. Outra vantagem importante é a confiança de trabalhar com recursos já testados por diversas pessoas, eliminando quase por completo as chances de falhas.

4.2.2 Hierarquia de memória no *BF537 Ez-Kit Lite*

O blackfin possui capacidade de endereçar 4GB, utilizados para acessar a memória interna e externa, além dos recursos de I/O, que são mapeados em memória. A distribuição dos endereços na hierarquia de memória no BF537, utilizado nesse trabalho, é representada na Figura 4.8.

Desse total de 4GB de endereçamento, 8MB pertencem à primeira parte da hierarquia de memória, localizando-se internamente no blackfin e alcançando as maiores performances. No primeiro nível da hierarquia (L1), com 48KB de capacidade, estão as memórias de maior velocidade de acesso, sendo SRAMs posicionadas próximas ao núcleo do processador. Devido à existência de múltiplos barramentos, diferentes blocos podem ser acessados simultaneamente. A arquitetura utilizada é uma Harvard adaptada, onde os dados e instruções são armazenados separadamente, porém acessados por um único esquema de endereçamento. Em algumas famílias de blackfins, como a BF537, porções da memória de Nível 1 podem ser configuradas como cache e trabalhar

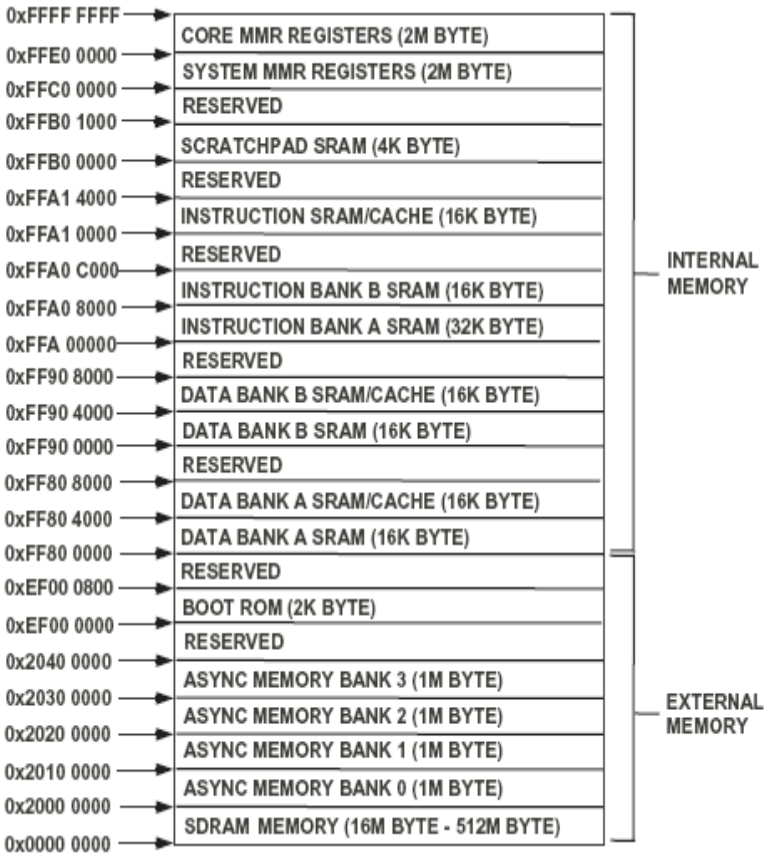

Figura 4.8. Mapa de Memória do BF537.

na frequência do processador. Além da cache, é possível utilizar uma área de 4KB como memória *scratchpad* para dados, reduzindo o tempo de acesso.

Ainda internamente, alguns blackfins contam com um segundo nível na hierarquia de memória, composto de memórias SRAMs com capacidade de 80KB que também trabalham na frequência do processador e são baseadas na arquitetura Von-Neumann. Entretanto esse espaço não pode ser configurado como cache e não está presente no BF537.

Registros e demais configurações e acesso a I/Os (MMR) também são mapeados em memória interna, ocupando um espaço de 4MB. Os demais endereços são de uso reservado do processador. A Figura 4.9 apresenta a disposição e a interconexão das principais estruturas de memória dentro do processador. Nela é possível observar que é possível utilizar o recurso de DMA para transferir dados diretamente de um recurso, como I/Os, para qualquer ponto da hierarquia de memória, com excessão da memória *scratchpad*.

Os 3,2GB restantes de endereçamento são destinados à memória externa, sendo essa distribuída em boot, Flash e SDRAMs. O espaço de boot consiste em uma memória

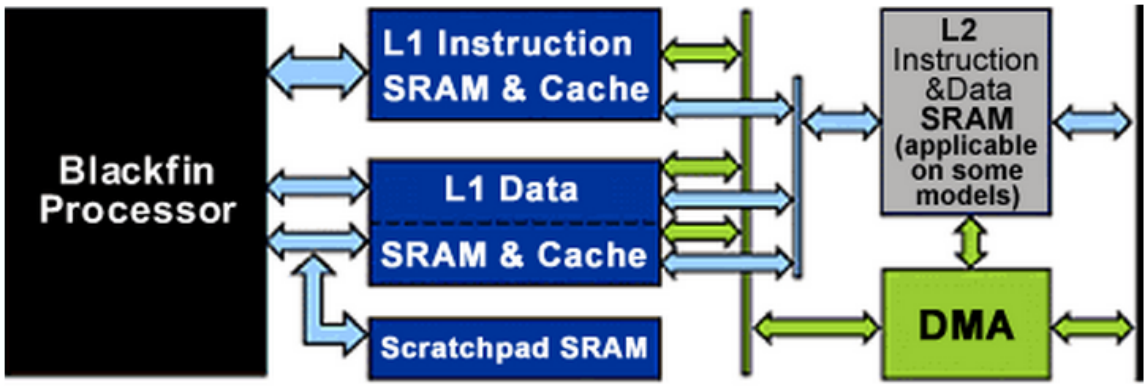

Figura 4.9. Hierarqia de Memória no Blackfin (Devices [2009e]).

ROM destinada à rotina de inicialização do sistema, localizado em um endereço padrão, possuindo uma capacidade de 2KB.

Memórias Flash e SDRAMs compõem o terceiro nível da hierarquia, podendo ser utilizadas conforme a necessidade da aplicação. Seu acesso chega a ser até 20 vezes mais lento que a memória interna.

4.3 Bibliotecas

Um dos principais recursos da plataforma DSCam é a sua possibilidade de extensão através do uso de bibliotecas. Funções de processamento de imagens podem ser facilmente acrescentadas à plataforma na forma de bibliotecas, podendo ser referenciadas no arquivo de descrição e acessadas por meio de uma API. Apesar de serem definidas aqui para processamento de imagens, teoricamente as funções de uma biblioteca podem realizar qualquer tipo de processamento que não interfiram nas estruturas internas da plataforma DSCam. Por estar sendo executada sobre um sistema operacional difundido, é possível também compilar bibliotecas de processamento de imagens já conhecidas, como OpenCV e MIMAS, acrescentando apenas a API.

Para o processo de validação e testes da plataforma, foram utilizadas duas bibliotecas, sendo uma criada baseando-se na biblioteca *Image Analysis*, fornecida pela ADI, e a outra, denominada uOpenCV, com algumas poucas funções existentes no OpenCV.

As APIs utilizadas para acessar as bibliotecas possuem a função de padronizar a interface entre as partes, efetuando a conversão entre as estruturas da plataforma e as estruturas das funções de processamento de imagem utilizadas. Toda API deve conter a função *run*, que recebe como parâmetro a imagem corrente e o nome da função a ser executada com os seus parâmetros. Caso a função passada não pertença àquela

biblioteca ou algum dos parâmetros estejam incorretos, a execução é interrompida com uma mensagem de erro. As APIs também são responsáveis por converter o retorno da função, quando existir, para um formato compatível com a plataforma.

4.3.1 Biblioteca System

Conforme foi descrito anteriormente, as bibliotecas de processamento tradicionais não possuem acesso às estruturas internas da plataforma DSCam. Entretanto, algumas funções especiais disponibilizadas na plataforma necessitam deste acesso, como operações de desvios, acesso a arquivos e a lista de imagens. Visando possibilitar a execução destas operações, uma biblioteca especial, denominada *System*, foi desenvolvida.

A implementação da biblioteca *System* encontra-se de forma distribuída entre o núcleo do DSCam e a biblioteca em si. Desta forma, parte das suas funções são fixas e de difícil acesso, enquanto uma outra parte pode ser ajustada com maior facilidade. Além da flexibilidade, a implementação da biblioteca *System* tem o objetivo de manter o mesmo padrão de execução em todos os comandos.

Muitos dos recursos oferecidos como comandos no arquivo de descrição, como desvios e manipulação de arquivos, são definições simplificadas de funções da biblioteca *System*, passando para o Parser a responsabilidade de colocar todas as informações detalhadas, como nome da API, função e parâmetros.

4.4 Operações adicionais

Além das operações básicas de processamento de imagens, foram incorporados à plataforma alguns recursos extras que fornecem maior potencial, como a lista de imagens, que possibilita manter diversas imagens em memória, e o serviço de autenticação, limitando a operação da DSCam apenas a pessoas autorizadas. A seguir, esses recursos serão explicados detalhadamente.

4.4.1 Autenticação do usuário

Uma das maiores aplicações das *smart-cameras* é em uma linha de produção, procurando não-conformidades no produto de cada etapa, visando uma rápida detecção de erros e evitando perdas maiores, ou ainda no controle de qualidade de produtos acabados, visando detectar peças defeituosas ou fora da especificação antes que ela chegue ao cliente. Ao utilizar a DSCam para essa aplicação, poderia surgir uma insegurança em relação ao vazamento de dados (imagens) sigilosos, uma vez que todo o controle

é realizado via rede e, teoricamente, qualquer um que conheça o protocolo e o I.P. da câmera poderia acessá-la ou até mesmo alterar sua configuração, causando enorme prejuízo.

Para evitar essa situação, foi adicionado à DSCam um recurso de autenticação do usuário, onde, ao abrir uma conexão com a câmera, é necessário digitar uma senha para executar qualquer comando. Uma vez que a senha foi digitada corretamente, o usuário poderá realizar qualquer operação disponível, até que a conexão se encerre ou seja enviado um comando de encerramento de sessão (logoff).

A senha utilizada na autenticação pode ser configurada pelo usuário, podendo conter até 50 símbolos e ser formada por qualquer caractere ASCII. Durante a transmissão a senha não é criptografada, trafegando aberta pela rede. Entretanto isso pode ser facilmente alterado criando uma nova implementação para a biblioteca *criptSenha*. Essa biblioteca é composta por duas funções, *code* e *decode*, que possuem a tarefa de criptografar e descriptografar a senha, respectivamente. Atualmente essas funções apenas retornam a senha passada, sem efetuar nenhuma operação.

4.4.2 Lista de imagens

Muitas vezes o processamento de uma única imagem não é o suficiente para obter alguma informação desejada ou, ainda, pode ser necessário utilizar uma mesma imagem em diferentes fluxos de execução. Originalmente, a DSCam trabalha com apenas uma imagem em memória, executando todas as operações sobre ela, funcionando como um acumulador. Para permitir a permanência de mais que uma imagem na memória, foi implementada a Lista de imagens.

A Lista de imagens é uma estrutura interna da plataforma DSCam que permite manter mais de uma imagem em memória, sem armazená-las em arquivo. Através de comandos é possível guardar e resgatar imagens da lista, a qualquer momento, mantendo cópias da imagem original ou de estágios intermediários do processamento.

Os comandos para guardar e retirar uma imagem da lista podem ser executados diretamente pelo usuário ou inseridos no arquivo de descrição, conforme apresentado nos Anexos A e B.

4.5 Considerações

A implementação da plataforma DSCam atingiu quase por completo o seu objetivo proposto, incorporando praticamente todas as características definidas para um sistema de processamento de imagens, ficando a desejar apenas no recurso de captura.

Possuindo uma interface bastante simples com o usuário, a DSCam cumpriu com sua proposta de tornar fácil o processo de desenvolvimento de sistemas de imagem, mantendo as estruturas de dados transparentes ao desenvolvedor.

Uma característica que superou as espectativas é a capacidade de definição de fluxos da linguagem utilizada no Arquivo de Descrição. Com a linguagem implementada, é possível definir qualquer fluxo de execução que não acesse diretamente um pixel da imagem ou que não utilize mais de uma imagem simultaneamente. Para esses casos, torna-se necessário a implementação de uma API. Esse limite de capacidade de implementação foi considerado ideal, uma vez que o objetivo da plataforma não é possibilitar a implementação de algoritmos de processamento de imagens, mas aplicações que utilizem esses algoritmos.

Apesar de não tem sido implementada a captura, toda a plataforma foi desenvolvida considerando-se a existência deste recurso. Desta forma, não haverá grandes alterações no *middleware* para incorporar esta funcionalidade. Entretanto, o maior desafio, no momento, é criar um *driver* para o uClinux se comunicar com o sensor.

Capítulo 5

Análises e Resultados

Visando avaliar e validar o desempenho da plataforma proposta foram realizados dois experimentos, sendo os tempos de execução de diversos algoritmos analisados. O primeiro experimento consistiu na execução de parte do *benchmark* proposto pelo *Defense Advanced Research Projects Agency* – Agência de Pesquisas em Projetos Avançados dos Estados Unidos – (DARPA) para avaliar as arquiteturas de visão computacional. Para o segundo experimento foram selecionados alguns algoritmos de visão de nível baixo com diferentes características, com o objetivo de medir o desempenho do hardware proposto para cada um deles.

Para os testes foram montados cinco ambientes computacionais diferentes, com o objetivo de estudar detalhadamente cada camada da plataforma proposta e ainda possibilitar a avaliação do seu impacto no sistema final. Os algoritmos utilizados nos testes foram escritos em C e compilados em compiladores adequados a cada ambiente. A seguir encontram-se descritos cada um dos ambientes utilizados nos experimentos.

- AMBIENTE 1: PC – computador pessoal, com processador Celeron de 1.8GHz, 1MB de memória cache e 1GB de memória principal, executando o sistema operacional SuSE Linux 9.2. Os algoritmos foram compilados utilizando o GCC (GNU [2009]), com otimização e os tempos de execução calculados utilizando-se o relógio do sistema operacional;

- AMBIENTE 2: PC+DSCam – computador pessoal, com processador Celeron de 1.8GHz, 1MB de memória cache e 1GB de memória principal, executando o sistema operacional SuSE Linux 9.2 e o *middleware* DSCam. Os algoritmos foram compilados em forma de bibliotecas para a DSCam utilizando o GCC, com otimização e os tempos de execução calculados utilizando-se o relógio do sistema operacional;

- AMBIENTE 3: Bfin – *BF537 Ez-Kit Lite*. Os algoritmos foram compilados utilizando o VisualDSP++ 4.0 (Devices [2009h]), com otimização.

- AMBIENTE 4: Bfin+uClinux – *BF537 Ez-Kit Lite*, executando o sistema operacional uClinux. Os algoritmos foram compilados utilizando o *GCC-Cross Compiling* (uClinux.org [2009]), com otimização e os tempos de execução calculados utilizando-se o relógio do sistema operacional e os tempos de execução calculados utilizando-se recursos de contagem de ciclos de execução do VisualDSP++ 4.0;

- AMBIENTE 5: Bfin+uClinux+DSCam – *BF537 Ez-Kit Lite*, executando o sistema operacional uClinux e o *middleware* DSCam. Os algoritmos foram compilados na forma de bibliotecas para a DSCam utilizando o *GCC-Cross Compiling*, com otimização e os tempos de execução calculados utilizando-se o relógio do sistema operacional.

Para evitar a interferência de fatores externos, como tráfego na rede e acesso ao disco, somente foi considerado para os testes sobre a aplicação DSCam o tempo após a identificação do comando EXECUTAR, até o resultado estar pronto para ser disponibilizado, incluindo a leitura da lista de execução, acesso às bibliotecas e a execução dos algoritmos. Nos demais testes (sobre o linux e diretamente sobre o blackfin), o tempo foi contado do momento da chamada do algoritmo analisado até o seu término completo, retornando à função principal. Nenhum dos testes acessam arquivos em disco, sendo restrito às operações de processamento e controle, no caso do *middleware* DSCam. Todas as imagens utilizadas, quando possível, foram previamente carregadas de um arquivo para a memória da aplicação, procurando minimizar o impacto do desempenho de periféricos na análise final.

5.1 Experimento 1: *Benchmark* DARPA

Como pode ser observado durante todo este trabalho, as arquiteturas utilizadas em visão computacional possuem um conjunto de características específicas que influenciam no desempenho e na capacidade de se executar uma aplicação. Considerando essa especificidade, foram propostos diversos *benchmarks* que possibilitam avaliar o desempenho de uma arquitetura para estas operações. Um *benchmark* geralmente especifica uma série de algoritmos a serem executados, em uma ordem pré-determinada, em imagens específicas. Entretanto, o maior desafio ao se projetar um *benchmark* é garantir que estão sendo utilizados algoritmos relevantes, que trabalhem em todos os aspéctos da arquitetura. O resultado de um *benchmark* em uma determinada arquitetura só

pode ser analisado quando comparado com o mesmo processo aplicado a uma outra arquitetura.

Para avaliar a plataforma proposta neste trabalho, utilizou-se um dos *benchmarks* mais conhecido e utilizado na comunidade científica. Sendo proposto pelo DARPA em 1988, o *DARPA Image Understanding Benchmark* especifica algoritmos que envolvem os três níveis de processamento de imagens, sendo utilizado para validar grande parte das arquiteturas de processamento, em especial, arquiteturas paralelas. Seus algoritmos propõem uma aplicação próxima à prática, envolvendo a localização de objetos retangulares em uma imagem.

5.1.1 Imagens utilizadas

O DARPA define duas imagens sobre as quais devem ser aplicados os algoritmos. Ambas as imagens consistem de caixas retangulares, com uma de suas superfícies orientadas de forma paralela ao plano de fundo. Cada uma das caixas encontra-se suspensas perpendicularmente por fios invisíveis, gerando um efeito de profundidade entre os retângulos. Três características podem ser utilizadas para caracterizar uma determinada caixa: tamanho dos lados, profundidade e orientação (rotação sobre o fio de sustentação).

A Figura 5.1, denominada de Imagem de Intensidade pelo DARPA, possui o tamanho de 512 x 512 pixels, sendo cada pixel representado por oito bits. Os retângulos nessa imagem aparecem sobre um fundo preto, possuindo um tom de cinza constante e poucas variações de tonalidade entre eles (oito tons apenas). Os retângulos podem assumir qualquer posição na imagem, inclusive se sobreporem.

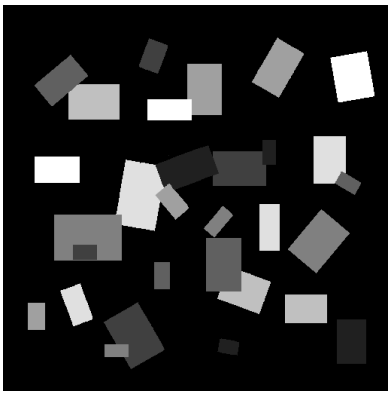

Figura 5.1. Imagem de Intensidade utilizada pelo *benchmark* DARPA

Entretanto, com a Imagem de Intensidade pode não ser possível localizar todos os objetos da cena, devido à oclusão. Assim, para completar o processo, é utilizada

a Imagem de Profundidade, apresentada na Figura 5.2. A imagem possui também o tamanho de 512 x 512 pixels, sendo cada pixel indicado por um ponto-flutuante de 32 bits (representação IEEE) representando a profundidade daquela caixa na cena.

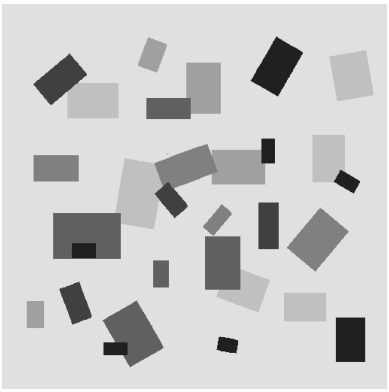

Figura 5.2. Imagem de Profundidade utilizada pelo *benchmark* DARPA

O plano de fundo da Imagem de Profundidade possui uma tonalidade constante e quanto menor a profundidade em que uma caixa se encontra (mais afastada do fundo), mais escuro é o tom de cinza atribuído a ela. Esta imagem, entretanto, não pôde ser utilizada com fidelidade na DSCam, uma vez que o formato adotado na plataforma permite apenas oito bits por pixel.

5.1.2 Algoritmos utilizados

O DARPA propõe a aplicação de cinco algoritmos sobre as imagens, envolvendo operações de baixo, médio e alto nível. Todos os algoritmos são bastante utilizados em aplicações de visão e podem ser implementados de diversas formas. Não há uma implementação padrão para os algoritmos sugeridos, entretanto, para evitar divergências nos resultados por fatores relacionados à codificação, são definidas algumas decisões de programação que devem ser seguidas. As implementações utilizadas no trabalho encontram-se agrupadas no Anexo C.

Para cada imagem é proposto um conjunto diferente de algoritmos. Para a Imagem de Intensidade são sugeridos os algoritmos *Label Connected Components* e *K-curvature* enquanto para a Imagem de Profundidade são os algoritmos *Smoothing*, *Gradient Magnitude* e *Threshold*. A seguir são descritos cada um dos algoritmos.

- *Label Connected Components*: cada componente conexo da imagem recebe um identificador inteiro único. Um componente conexo é qualquer coleção contígua de pixels que possuem o mesmo tom de cinza. Define-se pixels contíguos como

adjacentes em qualquer direção (norte, sul, leste, oeste). O resultado desta operação é uma matriz de inteiros com as mesmas dimensões da imagem, onde cada posição contém o rótulo (*label*) correspondente ao pixel de mesma coordenada;

- *K-curvatue*: para cada componente conexo deve-se calcular a *k-curvature* em suas bordas. A *k-curvature* é o ângulo formado por um pixel de borda como origem (O) e dois outros pixels de borda a uma distância quatro da origem (A e B), ou seja, sendo O um pixel de borda qualquer, A e B dois pixels de borda que possuem uma distância de quatro pixels em relação a O, o ângulo procurado é formado por AÔB, conforme apresenta a Figura 5.3. Calculando o ângulo para todos os pixels de borda de cada componente, destacam-se aqueles próximos a 90 graus. Em um passo futuro, a análise desses ângulos indicarão a presença de um retângulo na imagem;

- *Smoothing*: utilizado para atenuar as variações de intensidade luminosas entre pixels vizinhos, o *smoothing*, em sua versão mais simples, consiste em aplicar um filtro de média, utilizando uma máscara 3x3 onde todas as posições possuem o mesmo peso;

- *Gradient Magnitude*: efetua a operação *Sobel* sobre a imagem, utilizando-se uma máscara de dimensão 3 x 3, onde os pixels além das bordas da imagem são considerados nulos (zero). O gradiente consiste na raiz quadrada da soma dos quadrados das magnitudes de X e Y resultantes da operação *Sobel*;

- *Threshold*: consiste na binarização de uma imagem, onde os valores de pixels superiores a um determinado limite recebem a cor branca e os demais valores recebem a cor preta.

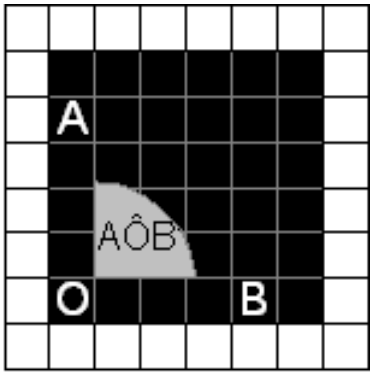

Figura 5.3. Representação de um ângulo identificado pelo K-curvature.

Cinco outras tarefas completam o *benchmark* proposto pelo DARPA, entretanto são destinadas a arquiteturas de nível médio e alto, extrapolando o escopo deste trabalho. Informações sobre o *benchmark* DARPA completo podem ser encontradas em Weems et al. [1988].

5.1.3 Metodologia

A execução do Experimento 1 foi dividida em três partes. Inicialmente avaliou-se o desempenho dos algoritmos e do *middleware* DSCam sobre um computador de propósito geral, como os utilizados na grande maioria das aplicações de visão computacional atualmente e, posteriormente, essa avaliação é feita sobre o hardware proposto no trabalho. Finalmente, realizou-se uma primeira avaliação do hardware proposto.

A primeira parte do experimento foi realizada executando os algoritmos indicados sobre os ambientes computacionais descritos nos Ambientes 1 (PC) e 2 (PC+DSCam). Cada um dos algoritmos foi implementado separadamente e independentemente em arquivos. Para a sua execução no Ambiente 1, criou-se um arquivo principal cuja tarefa consiste em carregar a imagem para a memória e chamar cada uma das funções, conforme especifica o DARPA, calculando também o tempo decorrido para a execução de cada uma delas. Para a execução sobre o Ambiente 2, foi criada uma API e compilada, juntamente com os algoritmos, em uma biblioteca compatível com o DSCam. Dois arquivos de descrição também foram escritos para orientar a execução dos algoritmos, conforme operação normal da plataforma, sendo um para cada imagem. Para a tomada de tempo na execução dos algoritmos sobre o DSCam foi necessário habilitar, em tempo de compilação, uma função que realizasse essa funcionalidade.

A segunda parte do experimento 1 foi realizada sobre os ambientes computacionais descritos nos Ambientes 4 (Bfin+uClinux) e 5 (Bfin+uClinux+DSCam), utilizando os mesmos arquivos e estruturas de teste da primeira parte, apenas recompilando-os segundo o ambiente.

Para a terceira parte do experimento, as imagens propostas pelo DARPA foram reduzidas à dimensão 25x25 pixels, com o objetivo de serem inteiramente carregadas na memória cache do processador utilizado na arquitetura. Os algoritmos indicados foram então aplicados às imagens novas nos Ambientes 1 e 4.

Em todos os casos, cada execução foi repetida 50 vezes, reiniciando o ambiente a cada cinco execuções, calculando a média dos tempos obtidos. O desvio padrão foi muito baixo em todos os casos, ficando abaixo de 2% da média.

5.1.4 Resultados

A Tabela 5.1 apresenta os dados gerados nas execuções dos algoritmos na primeira parte do experimento.

Algoritmo	**Ambiente 1**	**Ambiente 2**	***Overhead***
Label Connected Components	7,94 seg	7,95 seg	0,12%
K-curvatue	0,37 seg	0,38 seg	2,70%
Smoothing	0,03 seg	0,03 seg	0%
Gradient Magnitude	0,09 seg	0,09 seg	0%
Threshold	0,01 seg	0,01 seg	0%
Tempo total	8,44 seg	8,46 seg	0,24%

Tabela 5.1. Resultados obtidos nos Ambientes 1 (PC) e 2 (PC+uClinux) com os algoritmos do DARPA

Conforme é possível observar, o DSCam não gerou praticamente nenhum atraso na execução dos algoritmos, obtendo um *overhead* muito próximo a zero. Com esse resultado pode-se concluir que o *middleware* DSCam pode ser utilizado como ferramenta na criação de aplicações de visão computacional, simplificando o trabalho de desenvolvimento sem nenhum custo adicional de tempo na sua execução, mesmo quando não se utiliza a plataforma de hardware proposta nestetrabalho.

Na segunda parte do experimento, onde os algoritmos foram executados sobre a plataforma de hardware proposta, foram obtidos os resultados apresentados na Tabela 5.2.

Algoritmo	**Ambiente 4**	**Ambiente 5**	***Overhead***
Label Connected Components	84,67 seg	84,74 seg	0,08%
K-curvatue	69,02 seg	69,09 seg	0,10%
Smoothing	0,32 seg	0,38 seg	15,79%
Gradient Magnitude	19,98 seg	20,05 seg	0,35%
Threshold	0,17 seg	0,23 seg	26,08%
Tempo total	174,16 seg	174,49 seg	0,19%

Tabela 5.2. Resultados obtidos nos Ambientes 4 (Bfin+uClinux) e 5 (Bfin+uClinux+DSCam) com os algoritmos do DARPA

Assim como ocorreu nos Ambientes 1 e 2, o *overhead* gerado pelo *middleware* DSCam nos Ambientes 4 e 5 foi estatísticamente desprezível, com um atraso de 0,19% no tempo de execução. Entretanto, ao analisar separadamente cada algoritmo, pode ser observado *overheads* superiores a 25%. Apesar de corresponderem a tempos absolutos extremamente baixos (0,06 segundos de atraso), devem ser bem estudados e avaliados

o uso do DSCam para aplicações críticas, como em uma linha de produção que chega a montar várias peças por segundo.

Finalizadas as análises no computador e no hardware proposto, fica evidente a diferença nos tempos de execução entre eles. Todavia, essa diferença pode ser facilmente compreendida com o estudo de parâmetros que vão além da arquitetura do processamento, como, por exemplo, a frequência de operação e o tamanho da memória cache. Para confirmar essa hipótese, foi realizada a terceira parte do experimento que trabalha com imagens de tamanho inferior à cache da arquitetura proposta. Uma vez já validado o *middleware* DSCam, apenas os Ambientes 1 e 4 foram utilizados nesta terceira parte. As Tabelas 5.3 e 5.4 apresentam os resultados obtidos nesses ambientes com a imagem grande (512x512 pixels) e a imagem pequena (25x25 pixels), respectivamente. A coluna Ganho nas tabelas indica o quanto os resultados obtidos no Ambiente 1 foram melhores que os obtidos no Ambiente 4.

Algoritmo	**Ambiente 1**	**Ambiente 4**	**Ganho**
Label Connected Components	7,9434 seg	84,6700 seg	966,37%
K-curvatue	0,3728 seg	69,0200 seg	18554,05%
Smoothing	0,0260 seg	0,3168 seg	966,67%
Gradient Magnitude	0,0902 seg	19,9834 seg	22100,00%
Threshold	0,0124 seg	0,1736 seg	1600,00%

Tabela 5.3. Resultados obtidos nos Ambientes 1 (PC) e 4 (Bfin+uClinux) com os algoritmos do DARPA sobre imagens grandes (512x512)

Algoritmo	**Ambiente 1**	**Ambiente 4**	**Ganho**
Label Connected Components	0,0033 seg	0,0107 seg	224,24%
K-curvatue	0,0007 seg	0,1031 seg	14628,57%
Smoothing	0,0003 seg	0,0004 seg	33,33%
Gradient Magnitude	0,0003 seg	0,0422 seg	13966,67%
Threshold	n.d.	n.d.	n.d.

Tabela 5.4. Resultados obtidos nos Ambientes 1 (PC) e 4 (PC+uClinux) com os algoritmos do DARPA sobre imagens pequenas (25x25)

Devido à extrema velocidade de execução do algoritmo *Threshold* para a imagem pequena (Tabela 5.4), não foi possível determinar o seu tempo de execução para as arquiteturas e, consequentemente, o seu ganho. Com esses resultados é possível observar que o ganho do computador (Ambiente 1) foi menor com a imagem pequena. Esses resultados sugerem que um dos motivos para o baixo desempenho da arquitetura proposta em relação a um processador de propósito geral pode ser atribuído ao

tamanho da memória cache, onde no Ambiente 1 a imagem maior pode ficar inteiramente, sem necessidade de acesso à memória principal, o que não ocorre no Ambiente 2. Outros fatores, como frequência do processador e barramentos de memória, poderiam ser testados para averiguar qual a participação de cada um deles no desempenho da arquitetura, entretanto, essa análise se afasta da proposta central do trabalho.

Ainda, analisando esses resultados, observa-se que dois algoritmos que se destacam pela diferença de performance nos ambientes apresentados, com ambas as imagens. Os algoritmos *K-curvatue* e *Gradient Magnitude* obtiveram um desempenho pelo menos treze vezes inferior para a imagem pequena e quinze vezes na imagem grande com a arquitetura proposta, desviando da média de aproximadamente duas e dez vezes dos demais algoritmos, respectivamente. Com a realização do Experimento 2, descrito na seção 5.2, observou-se que o uClinux possui uma maior influência no desempenho dos algoritmos que contém grande carga matemática, como o *K-curvatue* e *Gradient Magnitude*, justificando, portanto, o ganho que obtiveram na análise.

Esses resultados confirmam a especificidade da arquitetura para operações de nível baixo, uma vez que ambos os algoritmos de maior ganho podem ser classificados de nível médio ou alto. Os algoritmos que apresentaram menor ganho (melhor resultado) são *Smoothing* e *Threshold*, ambos classificados de nível baixo.

5.2 Experimento 2: Algoritmos de Nível Baixo

Conforme demonstrado no Experimento 1, a arquitetura proposta apresenta um melhor desempenho para os algoritmos de nível baixo. Esses algoritmos ainda podem ser sub-dividídos em diferentes classes, segundo a forma e a operação realizada na imagem. Há alguns, por exemplo, que avaliam um pixel por vez enquanto outros avaliam pequenas regiões, alguns fazem operações básicas para determinar o pixel resultante, como comparações de valores ou cálculos simples, outros, por sua vez, realizam complexas operações matemáticas, muitas vezes dependentes de tabelas pré-definidas. Para cada uma dessas características a arquitetura pode apresentar um desempenho diferente, mesmo todos eles sendo classificados como nivel baixo, segundo os critérios citados no Capítulo 3.

Visando modelar o comportamento e mapear os pontos críticos de cada uma dessas sub-classes de algoritmos na arquitetura proposta, foi realizado o Experimento 2, contendo alguns algoritmos de nível baixo com diferentes características. Esses algoritmos foram executados sobre os Ambientes 3 (Bfin) e 4 (Bfin+uClinux), com o objetivo de determinar com maior precisão qual a camada da arquitetura (hardware

ou sistema operacional) gera maior impacto sobre o tempo final de execução.

5.2.1 Imagem utilizada

Para a execução das operações de nível baixo, optou-se por utilizar uma imagem bastante simples que pudesse até mesmo ser gerada automaticamente com o uso de um algoritmo. Essa característica é importante para o Ambiente 3, onde não há o conceito de arquivos, uma vez que a programação é realizada diretamente no processador, sem a intermediação de um sistema operacional. Para atender a esse requisito, utilizou-se a imagem apresentada na Figura 5.4.

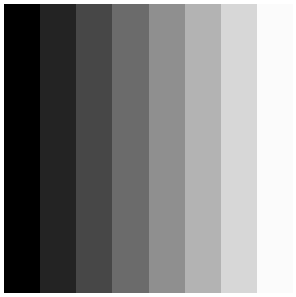

Figura 5.4. Imagem gerada para execução do Experimento 2

A imagem é composta de sete barras verticais em tons de cinza que variam do preto, mais a esquerda, aumentando 36 unidades na tonalidade do pixel a cada barra, até atingir o branco, mais a direita.

Essa imagem foi utilizada em dois tamanhos no experimento, sendo uma 255x255 pixels, excedendo o tamanho da memória cache da arquitetura e obrigando o acesso à memória principal, e a outra 25x25 pixels, podendo ser mantida inteiramente na memória cache.

5.2.2 Algoritmos utilizados

Para uma completa análise da arquitetura proposta para algoritmos de nível baixo, foram selecionados seis algoritmos comumente utilizados em aplicações de visão computacional com diferentes características de processamento. Cada algoritmo foi implementado em um arquivo diferente, em linguagem C, possibilitando compilá-los para cada um dos ambientes utilizados no experimento. No Anexo C encontram-se as implementações utilizadas. A seguir encontra-se uma breve descrição de cada um dos algoritmos selecionados (Klinger [2003]).

- Erosão: procura reduzir o brilho de pixels cujos vizinhos possuem uma intensidade luminosa menor. Sua execução consiste em grande parte de comparação de pixels vizinhos, sem muitos cálculos matemáticos;

- Gaussiano: utilizado para atenuar as variações de intensidade luminosa próximas à um pixel. Sua execução consiste na convolução de uma matriz 7x7 pixels sobre a imagem, gerando uma grande quantidade de cálculos matemáticos;

- Mediana: realiza a operação de mediana com os pixels vizinhos, procurando remover pixels isolados (ruídos) e detalhes da imagem;

- Perímetro: detecta todas as bordas dos objetos da imagem, destacando-as. Sua execução consiste na comparação de pixels com seus vizinhos, sem nenhuma operação matemática;

- Sobel: utilizado para extrair o contorno dos objetos de uma imagem, detectando variações bruscas de intensidade luminosa. Sua execução consiste na convolução de uma matriz 3x3 pixels sobre a imagem, gerando uma quantidade razoável de cálculos matemáticos;

- Soma: consiste na soma pixel-a-pixel de duas imagens, podendo ser utilizada para a união delas, correção de plano de fundo, entre outros.

5.2.3 Metodologia

A execução desse experimento ocorreu de forma bastante simples, principalmente no Ambiente 4 (Bfin+uClinux), onde foi criado um arquivo principal responsável por carregar a imagem, realizar a chamada das funções e calcular o tempo gasto em cada uma delas.

Para a execução do experimento no Ambiente 3 (Bfin), foi criado um arquivo que, além de fazer a chamada às funções, gerava as imagens a serem processadas, sendo a pequena criada diretamente na área de memória da aplicação e a grande gravada em memória principal. Para trabalhar com a imagem maior, os algoritmos foram adaptados para acessarem a memória e não mais buscarem a imagem internamente. O cálculo do tempo foi realizado com o uso de uma rotina fornecida pela própria ADI, integrada ao arquivo principal.

Cada algoritmo foi executado 500 vezes, sendos o ambientes reiniciados a cada 50 execuções, calculando a média dos tempos obtidos. O desvio padrão foi muito baixo em todos os casos, ficando abaixo de 2% da média.

5.2.4 Resultados

A Tabela 5.5 apresenta os dados gerados nas execuções dos algoritmos nos dois ambientes utilizando a imagem grande, de 255x255 pixels.

Algoritmo	**Ambiente 3**	**Ambiente 4**	***Overhead***
Erosão	0,00345 seg	0,00510 seg	50,00%
Gaussiano	0,34978 seg	29,58600 seg	8357,98%
Mediana	0,11404 seg	0,18550 seg	62,72%
Perímetro	0,00624 seg	0,02970 seg	379,03%
Sobel	0,05467 seg	4,61640 seg	8339,49%
Soma	0,00260 seg	0,02678 seg	93,08%

Tabela 5.5. Resultados obtidos nos Ambientes 3 (Bfin) e 4 (Bfin+uClinux) com os algoritmos especificados para o experimento 2 utilizando a imagem grande (255x255 pixels)

Como pode ser observado, o uClinux gerou um grande *overhead* na execução dos algoritmos, atingindo tempos três vezes superiores em relação à execução direta no processador. Dois algoritmos, Gaussiano e Sobel, destacam-se por ultrapassar demasiadamente esse limite.

Pode-se constatar que os algoritmos que apresentaram menor *overhead* são aqueles que trabalham com pixels vizinhos e possuem nenhuma ou poucas operações matemáticas. O algoritmo Perimetro, entretanto, apesar de possuir essas características, apresentou um resultado ruim. Isso se deve ao fato do número de operações lógicas realizadas ser quase duas vezes maior que nos outros algoritmos.

Os algoritmos Gaussiano e Sobel, que apresentaram um grande *overhead* com a execução sobre o uClinux, são caracterizados por realizarem várias operações matemáticas. Pode-se observar, entretanto, que ao serem executados no Ambiente 3, não apresentaram tempos demasiadamente fora da média dos demais algoritmos. Esse fato, associado ao tempo de execução do Perímetro, sugere que o sistema operacional utilizado também faz grande uso das estruturas lógicas-aritméticas do processador que, mesmo com recursos replicados, não é capaz de atender à demanda das operações.

Para avaliar a influência do sistema operacional em relação à utilização da memória cache, os algoritmos foram executados na imagem menor, de dimensão 25x25 pixels, cujos resultados são apresentados na Tabela 5.6.

Os resultados obtidos demonstram um enorme *overhead* causado pelo sistema operacional, chegando a gastar de sete a até mais de cem vezes o tempo de execução. Esse fato pode ser atribuído ao compartilhamento da cache entre o algoritmo e o uClinux, gerando um maior número de acessos à memória principal.

Algoritmo	Ambiente 3	Ambiente 4	*Overhead*
Erosão	0,000003 seg	0,000046 seg	1433,33%
Gaussiano	0,001630 seg	0,197000 seg	11985,89%
Mediana	0,000169 seg	0,001400 seg	728,40%
Perímetro	0,000008 seg	0,000112 seg	1300,00%
Sobel	0,000282 seg	0,036400 seg	12807,80%
Soma	0,000001 seg	0,000012 seg	11900,00%

Tabela 5.6. Resultados obtidos nos Ambientes 3 (Bfin) e 4 (Bfin+uClinux) com os algoritmos especificados para o experimento 2 utilizando a imagem pequena (25x25 pixels)

Pode-se novamente observar que o atraso gerado nos algoritmos Gaussiano e Sobel foram bastante superiores aos demais, acompanhando os resultados anteriores. O algoritmo de soma, devido ao compartilhamento da cache com duas imagens, além do sistema operacional, gerou também um *overhead* grande. Na execução direta sobre o processador, ambas as imagens podiam ser mantidas simultâneamente na memória cache, obtendo um resultado extremamente rápido.

Apesar do grande *overhead* causado pelo compartilhamento da memória cache com o sistema operacional, grande parte das aplicações reais utilizam imagens de tamanho superior à capacidade da cache da maioria dos processadores utilizados em sistemas embarcados. Dessa forma, o ambiente encontrado na prática é próximo ao encontrado na primeira parte do experimento, com a imagem grande. Para esse caso, o *overhead* gerado pelo sistema operacional foi satisfatório, para a maioria dos algoritmos.

5.3 Considerações

Após fazer uma completa análise da plataforma proposta, pode-se concluir que a DSCam se apresentou como uma solução viável de ser utilizada em aplicações de visão computacional, em especial, para realizar a primeira etapa do processamento, onde se caracteriza o uso de algoritmos de nível baixo.

Uma aplicação interessante em que a DSCam pode ser bastante eficiente é na prototipação e validação de aplicações de visão, atividades que podem se tornar bastante simples com o uso das bibliotecas, desenvolvendo através de uma interface gráfica, e usufruindo de todos os recursos de hardware especificados para essas operações.

Diversas outras análises ainda poderiam ser realizadas para identificar melhor os pontos positivos e negativos desta plataforma, entretanto extrapolam o objetivo

proposto de avaliar e validar a sua viabilidade.

Capítulo 6

Conclusão e Perspectivas

Este trabalho procurou abordar de forma detalhada todos os requisitos básicos e desejáveis comuns para a maioria dos sistemas de visão computacional. Todo o processo, desde à captura até a disponibilização do resultado foi caracterizado, com atenção especial à etapa de processamento. Requisitos e características básicas para um bom desempenho na execução de algoritmos de visão foram indicados, classificando-os em três níveis, segundo o tipo de processamento, o compartilhamento/acesso aos dados e a forma de controle, propondo modelos de arquiteturas que melhor se adequariam. Concluíu-se que, devido à diversidade dos algoritmos aplicados em um sistema completo, torna-se inviável a definição de uma arquitetura que atenda com eficiência a todos os casos. Foi observado, também, que os algoritmos de nivel baixo são aqueles que possuem características mais específicas,

Com o conhecimento dos requisitos e características de um sistema de visão, foi realizado um extenso estudo sobre as arquiteturas existentes que trabalham com esses sistemas. Algumas arquiteturas, como o CM-5 e o SP-2 se propuseram a resolver os problemas de visão com um único processador, o que resultou em uma plataforma complexa, de difícil operação, sem apresentar um bom desempenho na execução de um sistema completo. Arquiteturas propostas para os algoritmos de nível baixo, como o GAAP, o MP e suas derivações, procuram formas de aumentar o paralelismo no processamento, sem perder velocidade de comunicação entre os núcleos. Essas arquiteturas, entretanto, possuem um alto custo, se tornando inviáveis para aplicações comerciais. Uma opção de processadores que está sendo utilizada em muitos projetos de visão para executar os algoritmos de nível baixo é o DSP. Para as operações de nível médio e alto, processadores de propósito geral geralmente atendem com sucesso e estão compondo as arquiteturas híbridas. Essas arquiteturas propõem a integração de diferentes processadores em um único sistema, de forma atender de forma satisfatória os requisitos

da maioria dos algoritmos.

Com o desenvolvimento de processadores cada vez de menor consumo de energia e maior poder de processamento, uma nova geração de dispositivos com inteligência embarcada vem sendo utilizadas, como as *smart-cameras*. As *smart-cameras* são sistemas embarcados de visão que realizam todas as tarefas de um sistema de visão, desde a captura até a disponibilização do resultado. Esses sistemas, entretanto, são de difícil programação, devido às restrições naturais dos sistemas embarcados e a arquitetura dedicada do processador utilizado. Visando facilitar esse desenvolvimento, foi proposta a *Digital Smart Camera* (DSCam).

A DSCam consiste de uma solução integrada para aplicações de visão computacional, associada a um *framework* de desenvolvimento cujo objetivo é auxiliar na criação de sistemas de visão, eliminando a necessidade do programador conhecer detalhes da arquitetura interna da plataforma utilizada, que é acessada com o uso de um *middleware*. Seu desenvolvimento foi realizado de forma modular, permitindo facilmente sua expansão e adaptação para soluções específicas. Para tornar compatível com a grande maioria dos sistemas, a DSCam mantém todas as funções de processamento de imagens e visão em bibliotecas, permitindo fácil adição e alteração. Outra facilitade consiste no fato de ser inteiramente controlada por uma interface ethernet, podendo ser conectada a qualquer rede TCP/IP. Sua programação é realizada com o uso de um *framework*, onde o usuário pode definir um fluxo de execução, selecionando quais funções devem ser aplicadas na imagem, visualizar as imagens processadas, entre outras funcionalidades.

Finalmente, para avaliar e validar o desempenho da plataforma proposta, foram realizados dois experimentos. Concluiu-se que o *middleware* desenvolvido não gerou um impacto significativo sobre a execução dos algoritmos de visão, podendo ser utilizado para auxiliar no desenvolvimento desses sistemas. O sistema operacional utilizado, entretanto, foi um fator de preocupação, principalmente para as operações que fazem um maior uso das estruturas lógicas-aritméticas.

Dentre os diversos tipos de algoritmos testados, a DSCam apresentou melhor desempenho naqueles de são classificados como nível baixo e que não fazem grande uso de operações aritméticas. Esse resultado está coerente com aquilo que era esperado da plataforma, onde é realizado um primeiro processamento da imagem, deixando que as operações de nível médio e alto sejam realizadas por outro dispositivo, como um computador com processador de propósito geral.

Como trabalhos futuros, propõem-se a integração do sensor de captura imagem à plataforma DSCam e a realização de testes em aplicações reais, com diferentes características. Um avanço importante a ser integrado à arquitetura é a capacidade de se trabalhar com processamento distribuído, onde mais de uma câmera, conectadas à

mesma rede, poderiam avaliar diferentes partes da imagem e trocarem informações de forma obter um resultado unificado.

O uso de processadores *multi-core* é um outro recurso interessante a ser testado, sendo uma provável solução para o *overhead* causado pelo sistema operacional. Arquiteturas *multi-core* também poderiam ser úteis na paralelização do fluxo de execução, permitindo que duas ou mais operações possam ser realizadas simultaneamente.

Outro avanço pode ser a inclusão de diferentes processadores, com características de processamento diversificadas, aproximando-se das arquiteturas híbridas. Desta forma, o *framework* de desenvolvimento fica com a responsabilidade de gerenciar a distribuição de cargas entre eles, mantendo toda a arquitetura transparente para o programador.

Inúmeros outros trabalhos, além dos propostos aqui, ainda poderiam ser desenvolvidos sobre a arquitetura proposta, tornando a plataforma cada vez mais potente e robusta. O objetivo do presente trabalho foi integralmente cumprido, ao oferecer um sistema fácil de operar e viável para ser utilizado na maioria das aplicações. Apesar disso, ainda ficou distante o esgotamento das possibilidades de recursos que podem ser oferecidos ao desenvolvedor de um sistema de visão.

Referências Bibliográficas

(2009). Extensibility. DisponÌvel em: `http://en.wikipedia.org/wiki/Extensible`, ultimo acesso: 15/07/2009.

Baglietto, P.; Maresca, M.; Migliardi, M. & Zingirian, N. (1996). Image processing on high-performance risc systems. *Proceedings of the IEEE*, 84(7):917–930.

Baumgartner, D.; Rossler, P. & Kubinger, W. (2007). Performance benchmark of dsp and fpga implementations of low-level vision algorithms. *Computer Vision and Pattern Recognition, 2007. CVPR '07. IEEE Conference on*, pp. 1–8.

Chattopadhyay, A. & Boult, T. (2007). Privacycam: a privacy preserving camera using uclinux on the blackfin dsp. *Computer Vision and Pattern Recognition, 2007. CVPR '07. IEEE Conference on*, pp. 1–8.

Choudhary, A.; Patel, J. & Ahuja, N. (1993). Netra: a hierarchical and partitionable architecture for computer vision systems. *Parallel and Distributed Systems, IEEE Transactions on*, 4(10):1092–1104.

Chung, Y.; Prasanna, V. K.; Wang, C.-L.; Cantoni, V.; Lombardi, L.; Mosconi, M.; Savini, M. & Setti, A. (1995). A fast asynchronous algorithm for linear feature extraction on ibm sp-2. In *CAMP '95: Proceedings of the Computer Architectures for Machine Perception*, p. 294, Washington, DC, USA. IEEE Computer Society.

Coatrieux, J. (2005). Computer vision and graphics: frontiers, interfaces, crossovers, and overlaps in science. *Engineering in Medicine and Biology Magazine, IEEE*, 24(1):16–19.

Delong, P.; Polikarpov, B. & Krumnikl, M. (2007). Face detection by dsp using directly connected camera. *Radioelektronika, 2007. 17th International Conference*, pp. 1–3.

Devices, A. (2009a). Adsp blackfin. DisponÌvel em: `http://www.analog.com/en/embedded-processing-dsp/blackfin/content/index.html`, ultimo acesso: 16/07/2009.

Devices, A. (2009b). Adsp blackfin. DisponÌvel em: http://www.analog.com/en/embedded-processing-dsp/blackfin/bf537-hardware/products/product.html, ultimo acesso: 16/07/2009.

Devices, A. (2009c). Adsp blackfin 537. DisponÌvel em: http://www.analog.com/en/embedded-processing-dsp/blackfin/adsp-bf537/processors/product.html, ultimo acesso: 16/07/2009.

Devices, A. (2009d). Analog devices inc. DisponÌvel em: http://www.analog.com/, ultimo acesso: 16/07/2009.

Devices, A. (2009e). Arquitetura do blackfin. DisponÌvel em: http://www.analog.com/en/embedded-processing-dsp/blackfin/content/blackfin_architecture/fca.html, ultimo acesso: 20/07/2009.

Devices, A. (2009f). Blackfin processor core basics. DisponÌvel em: http://www.analog.com/en/embedded-processing-dsp/blackfin/content/blackfincorebasics/fca.html, ultimo acesso: 16/07/2009.

Devices, A. (2009g). Mimas. DisponÌvel em: http://vision.eng.shu.ac.uk/mediawiki/index.php/Mimas, ultimo acesso: 15/07/2009.

Devices, A. (2009h). Visualdsp++. DisponÌvel em: http://www.analog.com/en/embedded-processing-dsp/software-and-reference-designs/content/visualdsp_software_test_drive/fca.html, ultimo acesso: 18/07/2009.

Dyer, C. (1989). Parallel algorithms and architectures for image analysis and computer vision. pp. 7–.

Fountain, T.; Matthews, K. & Duff, M. (1988). The clip7a image processor. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 10(3):310–319.

Fung, J. & Mann, S. (2004). Computer vision signal processing on graphics processing units. In *Proceedings of the IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP 2004*, pp. 93--96.

Furlong, C. (2009). The pgm image file format. DisponÌvel em: http://users.wpi.edu/~cfurlong/me-593n/pgmimage.html, ultimo acesso: 17/07/2009.

GNU (2009). Gcc, the gnu compiler collection. DisponÌvel em: http://gcc.gnu.org/, ultimo acesso: 22/07/2009.

Gonzalez, R. C. & Woods, R. E. (2000). *Processamento de Imagens Digitais*. Edgard Blücher.

Hammerstrom, D. W.; Daniel & Lulich, P. (1996). Image processing using one-dimensional processor arrays. In *Proceedings of IEEE*, pp. 1005--1018.

Hennessy, J. L. & Patterson, D. A. (2007). *Computer Architecture: A Quantitative Approach*. Elsevier, 4 edição.

Kleihorst, R.; Reuvers, M.; Krose, B. & Broers, H. (2004). A smart camera for face recognition. volume 5, pp. 2849–2852 Vol. 5.

Klinger, T. (2003). *Image Processing with LabVIEW and IMAQ Vision*. Pearson Education.

Levialdi, S. (1988). Computer architectures for image analysis. *Pattern Recognition, 1988., 9th International Conference on*, pp. 1148–1158 vol.2.

Litzenberger, M.; Belbachir, A.; Schon, P. & Posch, C. (2007). Embedded smart camera for high speed vision. pp. 81–86.

Microsoft (2009). Microsoft directx. DisponÌvel em: `http://www.microsoft.com/directx`, ultimo acesso: 15/07/2009.

Nickolls, J. (1990). The design of the maspar mp-1: a cost effective massively parallel computer. *Compcon Spring '90. Intellectual Leverage. Digest of Papers. Thirty-Fifth IEEE Computer Society International Conference.*, pp. 25–28.

Nickolls, J. R. & Reusch, J. (1993). Autonomous simd flexibility in the mp-1 and mp-2. In *SPAA '93: Proceedings of the fifth annual ACM symposium on Parallel algorithms and architectures*, pp. 98--99, New York, NY, USA. ACM.

Olk, J. G. E.; Jonker, P. P.; Cantoni, V.; Lombardi, L.; Mosconi, M.; Savini, M. & Setti, A. (1995). A programming and simulation model of a simd-mimd architecture for image processing. In *CAMP '95: Proceedings of the Computer Architectures for Machine Perception*, p. 98, Washington, DC, USA. IEEE Computer Society.

OpenCV (2009). Opencv wiki. DisponÌvel em: `http://opencv.willowgarage.com/wiki/`, ultimo acesso: 15/07/2009.

Parhami, B. (1995). Simd machines: do they have a significant future? *SIGARCH Comput. Archit. News*, 23(4):19--22.

Potter, J. (1983). Image processing on the massively parallel processor. *Computer*, 16(1):62–67.

Prasanna, V.; Wang, C.-L. & Khokhar, A. (1993). Low level vision processing on connection machine cm-5. *Computer Architectures for Machine Perception, 1993. Proceedings*, pp. 117–126.

Pridmore, T. & Hales, W. (1995). Understanding images: an approach to the university teaching of computer vision. *Engineering Science and Education Journal*, 4(4):161–166.

Ratha, N. & Jain, A. (1997). Fpga-based computing in computer vision. *Computer Architecture for Machine Perception, 1997. CAMP '97. Proceedings Fourth IEEE International Workshop on*, pp. 128–137.

Ratha, N. K. & Jain, A. K. (1999). Computer vision algorithms on reconfigurable logic arrays. *IEEE Trans. Parallel Distrib. Syst.*, 10(1):29--43.

Reddaway, S. F. (1973). Dap—a distributed array processor. In *ISCA '73: Proceedings of the 1st annual symposium on Computer architecture*, pp. 61--65, New York, NY, USA. ACM.

Saini, S. & Simon, H. D. (1994). Applications performance on intel paragon xp/s-15. In *HPCN Europe 1994: Proceedings of the nternational Conference and Exhibition on High-Performance Computing and Networking Volume II*, pp. 493--498, London, UK. Springer-Verlag.

Shi, Y.; Raniga, P. & Mohamed, I. (2006a). A smart camera for multimodal human computer interaction. pp. 1–6.

Shi, Y.; Taib, R. & Lichman, S. (2006b). Gesturecam: A smart camera for gesture recognition and gesture-controlled web navigation. pp. 1–6.

Sung, M. (2000). Simd parallel processing. Technical report.

Sunwoo, M. & Aggarwal, J. (1990a). A sliding memory array processor for low level vision. *Pattern Recognition, 1990. Proceedings., 10th International Conference on*, ii:312–317 vol.2.

Sunwoo, M. & Aggarwal, J. (1990b). Vista for a general purpose computer vision system. *Pattern Recognition, 1990. Proceedings., 10th International Conference on*, ii:635–641 vol.2.

Thoren, P. D. S. (2002). Phission: A concurrent vision programming system software development kit for mobile robots. Master's thesis, B. S. University of Massachusetts at Lowell.

uClinux.org (2009). Gcc cross compiling for blackfin. DisponÌvel em: `http://docs.blackfin.uclinux.org/doku.php?id=cross_compiling`, ultimo acesso: 17/07/2009.

Wang, C.-L.; Bhat, P. & Prasanna, V. (1996). High-performance computing for vision. *Proceedings of the IEEE*, 84(7):931–946.

Weems, C. (1991). Architectural requirements of image understanding with respect to parallel processing. *Proceedings of the IEEE*, 79(4):537–547.

Weems, C.; Rana, D.; Hanson, A.; Riseman, E.; Shu, D. & Nash, J. (1990). An overview of architecture research for image understanding at the university of massachusetts. *Pattern Recognition, 1990. Proceedings., 10th International Conference on*, ii:379–384 vol.2.

Weems, C.; Riseman, E.; Hanson, A. & Rosenfeld, A. (1988). Iu parallel processing benchmark. pp. 673–688.

Wu, C.; Aghajan, H. & Kleihorst, R. (2007). Mapping vision algorithms on simd architecture smart cameras. pp. 27–34.

Yu Shi, S. L. (2005). Smart cameras: A review. Technical report.

Apêndice A

Arquivo de Descrição

O Arquivo de Descrição é onde fica armazenado o fluxo de execução a ser aplicado sobre uma imagem na câmera. Seu objetivo é indicar para o servidor a sequência de operações que deverão ser aplicadas na imagem e fornecer as informações necessárias para sua execução, como a localização e os seus parâmetros. Este arquivo é gerado automáticamente no software de configuração e operação da plataforma, onde o usuário pode definir o fluxo de execução facilmente, com o uso de uma interface gráfica.

A linguagem utilizada no Arquivo de Descrição pode ser definida pelo usuário, desde que se implemente um Parser compatível. O Parser é um módulo da plataforma DSCam criado para ler Arquivos de Descrição e carregar as suas informações em uma estrutura interna, denominada Fila de Execução. Visando oferecer uma maior flexibilidade à plataforma, esse módulo foi projetado como uma biblioteca, podendo ser facilmente substituído por um outro sem gerar impactos ao restante da aplicação.

Neste trabalho optou-se por implementar um Parser para a linguagem XML, sendo esta também a linguagem adotada pela aplicação de configuração e operação.

Um Arquivo de Descrição XML possui basicamente três partes: cabeçalho, declarações e operações. O cabeçalho possui informações que permitem a identificação do fluxo de execução. As variáveis utilizadas durante o processamento são definidas nas declarações, indicando seu nome, tipo e tamanho. Por fim, o fluxo de execução com os comandos e operações a serem aplicadas na imagem compoem a última parte do Arquivo de Descrição.

A.1 Cabeçalho

O cabeçalho do arquivo de descrição tem o objetivo de identificar o fluxo, além de manter informações importantes sobre o autor e a sua versão. A sintaxe a ser utilizada

no cabeçalho é apresentada a seguir.

```
1. <head>
2.     <name>nome-do-arquivo</name>
3.     <author>nome-do-autor</author>
4.     <version>versao-do-arquivo</version>
5.     <date>data-de-criacao</date>
6. </head>
```

O cabeçalho é delimitado por duas tags e possui quatro propriedades que, obrigatóriamente, devem ser preenchidas. A primeira propriedade, $< name >$, apresentada na segunda linha, indica o nome daquele fluxo de execução. Este nome tem a dimensão de um a cinquenta caracteres e pode ser formado por qualquer letra, número ou símbolo diferente de $<$.

A segunda propriedade do cabeçalho refere-se ao nome do autor do Arquivo de Descrição. Seu valor também pode ir de um a cinquênta caracteres, sendo formado por qualquer letra, número ou símbolo diferente de $<$. A versão do Arquivo de Descrição, exibida na linha 4, indica a versão em que o arquivo se encontra e deve ser formada apenas por números e pontos. O último campo do cabeçalho é a data em que o arquivo foi criado. Esses dois últimos campos procuram fornecer ao usuário recursos para um melhor controle e rastreamento de alterações realizadas no fluxo de execução.

A.2 Declarações

A segunda parte do Arquivo de Descrição XML consiste na declaração das variáveis utilizadas no fluxo de execução. As declarações de variáveis indicam para o servidor o tamanho do espaço de memória a ser alocado e cria alias para cada uma delas. As variáveis possuem tamanhos definidos pelo usuário, possibilitando a criação de vetores, e podem assumir todos os tipos básicos da plataforma, sendo eles *int*, *double*, *char*, *float* e *short*. Essas informações são gerenciadas por uma estrutura específica e não fazem parte da Lista de Execução. Não há uma necessidade real para manter as declarações juntas no arquivo após o cabeçalho, entretanto, isso facilita na definição dos comandos de desvio, conforme será explicado posteriormente. A seguir é apresentada a sintaxe utilizada para declaração de variáveis no Arquivo de Descrição XML.

```
1. <decl>
2.     <var>nome-da-variável</var>
3.     <type>tipo-da-variável</type>
```

```
4.      <size>tamanho-da-variável</size>
5. </decl>
```

O primeiro campo de uma declaração de variável consiste em um nome (alias) único que será utilizado para referenciar esta variável no restante do arquivo. O tamanho do valor pode variar de um a cinquênta caracteres, contendo letras, números e símbolos diferentes de <.

O tipo da variável é definido no parâmetro $< type >$ e seu valor pode ser qualquer um dos tipos básicos da plataforma. Todas as variáveis no DSCam são consideradas vetores. Por esse motivo, todas as declarações possuem o campo de tamanho $< size >$. O valor deste parâmetro é um número inteiro que indica quantas posições a variável terá. Para variáveis simples (não vetores) deve-se colocar o tamanho com valor igual a um.

A.3 Operações

A terceira parte do Arquivo de Descrição XML consiste na definição das funções a serem executadas na imagem. A execução ocorre de forma sequêncial, iniciando-se do primeiro até o último bloco de função definido no XML. Entretanto, operações de desvio e término da execução podem alterar a ordem de execução das funções.

Há dois tipos de funções que podem ser executadas pela câmera. O primeiro tipo de função consiste de operações internas da plataforma, implementadas pela biblioteca *System.so* e, muitas vezes, definidas como comandos. O segundo tipo de função é composto por operações implementadas em bibliotecas de processamento de imagem e visão diversas, acessadas pela plataforma com o uso de APIs. As funções definidas como comandos são abordadas na sessão A.3.1.

Para o segundo tipo de função descrito, é necessário informar à câmera a API em que a função será encontrada, além do nome da função, parâmetros e tipo de retorno, caso exista. A sintaxe utilizada para referenciar uma função no Arquivo de Descrição XML é apresentada a seguir.

```
1. <function>
2.      <api>nome-da-api</api>
3.      <name>nome-da-função</name>
4.      <param>
5.          <value>parametro-1</value>
6.          <type>tipo-do-param1</type>
```

```
 7.      </param>
 8.      <param>
 9.          <value>parametro-N</value>
10.          <type>tipo-do-paramN</type>
11.      </param>
12.      <return>
14.          <type>tipo-do-retorno</type>
15.          <var>nome-da-variável</var>
16.          <position>posição</position>
17.      </return>
18. </function>
```

O primeiro campo de uma função consiste na identificação da API em que ela está implementada. Esse nome deve ser exatamente igual ao do arquivo da biblioteca da API, sem a extensão. No campo $< name >$ deve ser digitado o nome pelo qual a API reconhece a função. Os parâmetros a serem passados para a função são descritos na tag $< param >$, onde define-se o valor e o tipo. Todos os tipos básicos da plataforma podem ser utilizados como parâmetros, além do tipo *var*, utilizado para indicar que o valor inserido refere-se à uma variável que deve ser acessada. Uma função pode ter quantos parâmetros forem necessários. A imagem corrente é sempre passada para a API, não sendo necessária inserí-la na lista de parâmetros.

A maioria das funções de processamento de imagem de nível baixo geram como resultado uma nova imagem, derivada da inicial. Para estes casos a própria plataforma já provê um recurso de receber a imagem de retorno sem a interferência do usuário. Entretanto, outras funções podem retornar valores. Para esses casos, deve-se utilizar a tag $< return >$, como a linha 12 do bloco $< function >$.

A.3.1 Comandos

Dentre as operações descritas no Arquivo de Descrição, algumas delas consistem de funções internas da plataforma que, em sua maioria, não podem ser implementadas apenas como uma biblioteca, como ocorre com as funções de processamento de imagem. Estas funções internas recebem o nome de Comandos e possuem sintaxes específicas, como apresentadas a seguir.

A.3.1.1 Comando CAPTURAR

Descrição: realiza nova captura de imagem durante o fluxo de execução.

```
1. <capture>
2.     <height>altura-da-imagem</height>
3.     <width>largura-da-imagem</width>
4. </capture>
```

Os sensores de imagem geralmente possuem uma resolução máxima nominal e também trabalham com resoluções inferiores. Utilizando-se os parâmetros $< height >$ e $< width >$ essa flexibilidade é mantida para o usuário, podendo escrever fluxos que melhor se adequem à sua aplicação.

A.3.1.2 Comando ABRIR ARQUIVO

Descrição: carrega a imagem de um arquivo como imagem corrente.

```
1. <load>
2.     <name>nome-do-arquivo</name>
3. </load>
```

A tag $< name >$ deve conter o nome do arquivo a ser aberto e deve conter extensão pois esta indicará o formato em que a imagem será lido.

A.3.1.3 Comando SALVAR ARQUIVO

Descrição: grava a imagem corrente em um arquivo.

```
1. <save>
2.     <name>nome-do-arquivo</name>
3. </save>
```

A tag $< name >$ deve conter a ser colocado no arquivo gerado com a imagem e deve conter a extensão, pois esta indicará o formato em que a imagem será gravada.

A.3.1.4 Comando ATRIBUIÇÃO

Descrição: atribui um valor ou o resultado de uma expressão a uma posição de uma variável.

```
1. <assign>
2.     <var>nome-da-variável</var>
3.     <position>posição</position>
4.     <value>valor-atribuído</value>
5. </assign>
```

A tag $< var >$ indica o nome da variável que receberá a atribuição. Em $< position >$ é indicada a posição da variável em que o valor deverá ser atribuído. As posições válidas são de 0 (zero) à $tamanho - 1$. Nas variáveis de tamanho um, a tag $< position >$ deverá assumir o valor zero. Finalmente, o valor a ser atribuído é indicado na tag $< value >$. Esse valor poderá ser um número (double, int, short, etc) ou uma expressão lógica-aritmética.

As expressões lógicas-aritméticas podem ser formadas por números, variáveis e as principais operações lógicas e aritméticas, permitindo a definição de prioridades com o uso de parênteses. Entre cada elemento da expressão sempre deverá existir um espaço. A Tabela A.1 contém todas as operações suportadas nas expressões e sua sintaxe no XML.

Operação	**Sintaxe**	**Descrição**
+	+	Calcula o total de dois operadores
−	−	Calcula a diferença de dois operadores
*	*	Calcula o produto de dois operadores
/	/	Calcula o quociente de dois operadores
mod	%	Calcula o resto entre dois operadores
&&	&*amp*	Efetua a operação lógica AND
||	|	Efetua a operação lógica OR
neg	neg	Efetua a operação lógica NOT
==	=	Efetua comparação de *iguala* entre dois operadores
!=	!=	Efetua comparação de $diferentede$ entre dois operadores
>	&*gt*;	Efetua comparação de *maiorque* entre dois operadores
>=	&*gt*;=	Efetua comparação de *maiorouigualque* entre dois operadores
<	&*lt*;	Efetua comparação de *menorque* entre dois operadores
<=	&*lt*;=	Efetua comparação de *menorouigualque* entre dois operadores

Tabela A.1. Operações suportadas nas expressões lógicas-aritméticas

A.3.1.5 Comando ENVIAR PARA A LISTA DE IMAGENS

Descrição: copia a imagem corrente para uma nova entrada na lista de imagens. São suportadas duas sintaxes para o comando.

Sintaxe 1:

```
1. <push/>
```

Sintaxe 2:

```
1. <push>
2.     <position>posição-a-ser-inserida</position>
```

```
3. </push>
```

Na Sintaxe 1, a imagem corrente é colocada na primeira posição da lista, funcionando como uma pilha. A Sintaxe 2 permite escolher em qual posição a imagem será inserida, sendo a primeira posição a zero. Caso seja indicada uma posição maior que o tamanho da lista a execução será interrompida com uma mensagem de erro.

A.3.1.6 Comando LER DA LISTA DE IMAGENS

Descrição: extrai uma imagem da lista de imagens e a coloca como imagem corrente. São suportadas duas sintaxes para o comando.

Sintaxe 1:

```
1. <pop/>
```

Sintaxe 2:

```
1. <pop>
2.     <position>posição-a-ser-extraída</position>
3. </pop>
```

A Sintaxe 1, assim como no comando ENVIAR PARA LISTA DE IMAGENS, lê a primeira posição da lista. A Sintaxe 2 permite que se escolha a posição a ser lida, sendo a primeira posição a zero. Caso seja indicada uma posição inválida a execução será interrompida com uma mensagem de erro.

A.3.1.7 Comando IR PARA

Descrição: desvia o fluxo de processamento para uma determinada função.

```
1. <goto>id-função-destino</goto>
```

Define-se como o id da função a posição absoluta que ela ocupa no fluxo de execução, ou seja, considerando apenas a parte de operações do arquivo de descrição, a posição em que a função é escrita, sendo que a primeira função possui id igual a zero. O comando IR PARA permite o desvio para qualquer função dentro do fluxo de execução. Caso a função destino seja o próprio comando IR PARA, a execução entrará em um ciclo infinito. O uso de um id de função inválido é entendido como término da execução, gerando o mesmo efeito do comando SAIR.

A.3.1.8 Comando DESVIO CONDICIONAL

Descrição: desvia o fluxo de execução para uma determinada função caso uma determinada condição seja verdadeira.

```
1. <if>
2.     <expression>expressão</expression>
3.     <goto>id-função-destino</goto>
4. </if>
```

A tag < *expression* >, apresentada na linha 2, contém uma expressão lógica-aritmética que deve ser avaliada na tomada de decisão do desvio. A sintaxe da expressão deve seguir a sintaxe indicada na tag < *value* >, na linha 4 do comando ATRIBUIÇÃO. A tag < *goto* > deve seguir as mesmas orientações do comando IR PARA, respeitando a mesma definição para o id da função. O uso de um id de função inválido é entendido como término da execução, gerando o mesmo efeito do comando SAIR quando a condição tem resultado verdadeiro.

A.3.1.9 Comando SAIR

Descrição: encerra execução do fluxo.

```
1. <halt/>
```

Apêndice B

Protocolo de Comunicação

Entende-se como protocolo de comunicação um conjunto de regras que definem a ordem e a sintaxe de mensagens trocadas entre dois agentes, sejam eles humanos ou aplicações computacionais. Atualmente, cada vez mais dispositivos vem ganhando a capacidade de se comunicarem uns com os outros, principalmente por meio da internet, e, para isso utilizam-se de protocolos. A arquitetura DSCam foi projetada para seguir essa tendência, podendo ser conectada à internet ou a uma rede local e ser operada por qualquer outro equipamento conectado à mesma rede que implemente o protocolo de comunicação utilizado.

A aplicação executada na câmera funciona como um servidor de serviços, sempre aguardando uma conexão da aplicação de configuração ou qualquer outro cliente. Uma vez estabelecida a conexão, o cliente pode então solicitar a execução de comandos da plataforma. Uma máquina de estados é utilizada para orientar a interpretação e execução dos comandos. Todos os comandos são compostos de códigos ASCII e sempre geram algum tipo de resposta, conforme detalhado na sessão B.1 a seguir.

B.1 Descrição dos Comandos

Esta sessão tem o objetivo de descrever detalhadamente cada comando, incluindo as mensagens do protocolo envolvidas.

Para qualquer comando, a resposta LE sinaliza que o sistema encontra-se bloqueado e deve ser realizado o login antes de qualquer operação. Quando um determinado comando não é reconhecido, o servidor retorna a mensagem E e volta o interpretador de comandos para seu estado inicial. Todas as mensagens de resposta do servidor terminam com uma quebra de linha (n).

As operações realizadas ao executar um comando não são persistentes em caso

de reinicialização da aplicação, com excessão do comando SALVAR IMAGEM e as configurações que são ajustadas no comando OPÇÕES.

B.1.1 Comando: RESET

Descrição: retorna o interpretador de comandos para o seu estado inicial. Pode ser utilizado a qualquer momento, ignorando os ajustes/parâmetros passados para a execução de um comando. O comando RESET não interrompe nenhuma operação já iniciada.

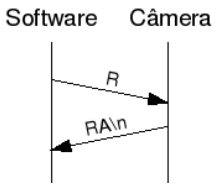

Figura B.1. Mensagens trocadas no comando RESET.

B.1.2 Comando: LOGIN

Descrição: efetua login na câmera. A senha pode ser composta por qualquer caractere ASCII e não é criptografada durante a transmissão.

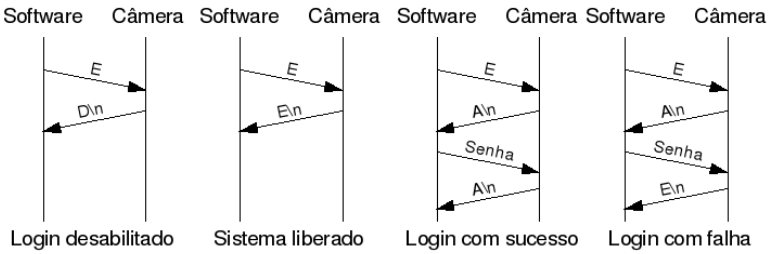

Figura B.2. Mensagens trocadas no comando LOGIN.

B.1.3 Comando: LOGOFF

Descrição: efetua logoff no sistema, sendo necessário efetuar login novamente para efetuar qualquer operação.

B.1.4 Comando: VERSÃO

Descrição: envia a data da compilação e a versão do servidor em execução.

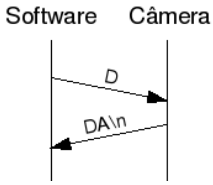

Figura B.3. Mensagens trocadas no comando LOGOFF.

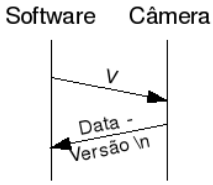

Figura B.4. Mensagens trocadas no comando VERSÃO.

A data é formada por três campos (dia, mês e ano), separados por uma barra: dd/mm/aaaa

A versão é uma sequência de números separados por ponto, onde cada campo indica a versão dos principais módulos do sistema: A.B.C.D.E.F.G.H.I sendo: A : release do sistema B : núcleo do sistema C : módulo de rede D : interpretador de comandos E : módulo de execução F : estrutura de imagem G : lista de imagens H : lista de execução I : gerenciador de variáveis

B.1.5 Comando: RECEBE ARQUIVO

Descrição: copia um arquivo para a câmera. Pode ser utilizado para transmitir o arquivo de descrição ou, ainda, imagens a serem utilizadas no processamento. O nome do arquivo deve seguir o padrão utilizado pelo MS-DOS (máximo de 8 caracteres com três de extenção separados por um ponto).

B.1.6 Comando: ABRE IMAGEM

Descrição: carrega um arquivo de imagem. O nome do arquivo deve conter extensão pois indicará o formato da imagem que será lida.

B.1.7 Comando: SALVA IMAGEM

Descrição: salva imagem corrente em um arquivo. O nome do arquivo a ser gerado deve conter a extensão pois indica o formato em que a imagem deve ser gravada. O arquivo gerado persiste em caso de reinicialização da aplicação, entretanto, devido à

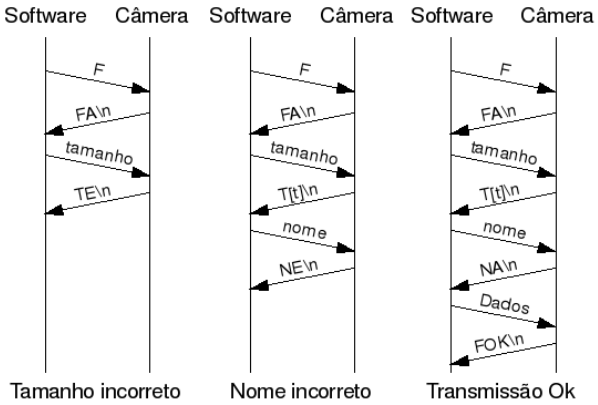

Figura B.5. Mensagens trocadas no comando RECEBE ARQUIVO.

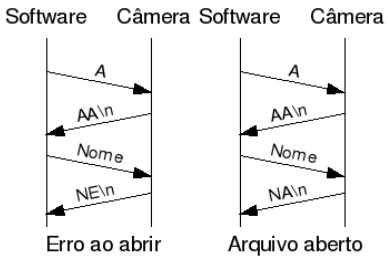

Figura B.6. Mensagens trocadas no comando ABRE IMAGEM.

arquitetura de hardware e o sistema operacional adotado, não é mantido em caso de reinicialização do sistema operacional.

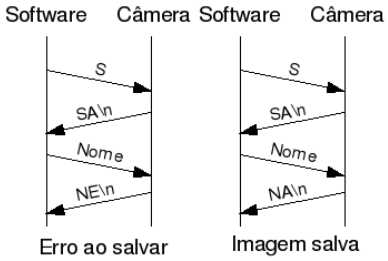

Figura B.7. Mensagens trocadas no comando SALVA IMAGEM.

B.1.8 Comando: CAPTURA

Descrição: captura uma imagem do sensor de imagem.

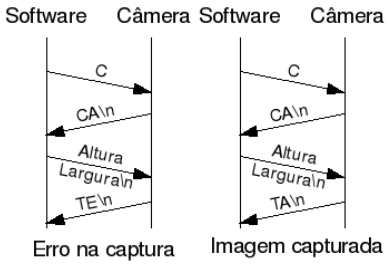

Figura B.8. Mensagens trocadas no comando CAPTURA.

B.1.9 Comando: ENVIA IMAGEM

Descrição: envia a imagem corrente. Caso não tenha nenhuma imagem carregada, é gerado um *degradee* de 255x255 para realização de testes.

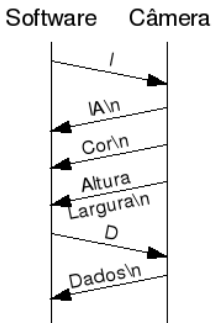

Figura B.9. Mensagens trocadas no comando ENVIA IMAGEM.

B.1.10 Comando: CARREGA EXECUÇÃO

Descrição: carrega a sequência de funções e comandos descritas no arquivo de descrição, gerando a Lista de Execução. O nome do arquivo de descrição a ser carregado pode ser configurado utilizando-se o comando OPÇÕES. O nome padrão para a plataforma é default.xml.

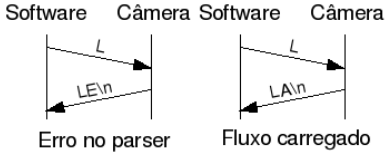

Figura B.10. Mensagens trocadas no comando CARREGA EXECUÇÃO.

B.1.11 Comando: EXECUTA

Descrição: executa as funções e comandos carregados na Lista de Execução.

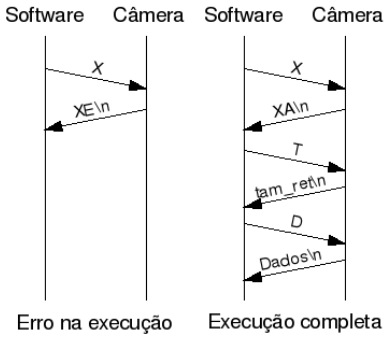

Figura B.11. Mensagens trocadas no comando EXECUTA.

B.1.12 Comando: OPÇÕES

Descrição: permite alterar as configurações da plataforma DSCam. Todas as configurações alteradas com este comando são persistentes em arquivos. Entetanto, devido à arquitetura de hardware e o sistema operacional adotado, os arquivos não são persistentes em caso de reinicialização do sistema operacional.

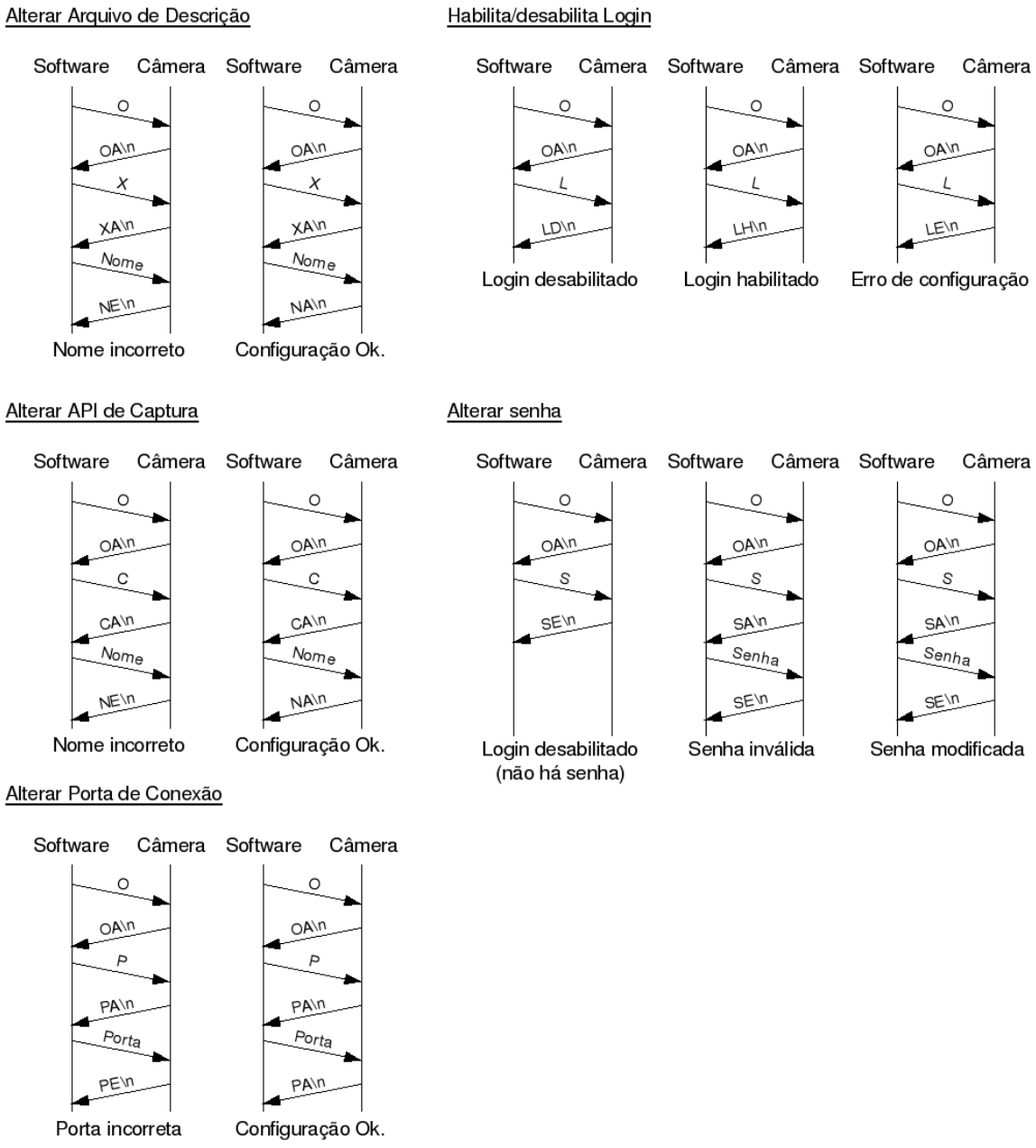

Figura B.12. Mensagens trocadas no comando OPÇÔES.

Apêndice C

Algoritmos Implementados

C.1 *Benchmark* DARPA

C.1.1 *Label Connected Components*

```
// DARPA Benchmark - Label Connected Components
#include ''Label.h''

void label_connected(unsigned char* image, unsigned short* labels,
                     int x_size, int y_size)
{
       unsigned short lb=1;    // label atual
       int p_an=0;             // conta pixels analisados
       int x, y;               // pixel atual
       int m=1;                // há novo pixel marcado
       unsigned short visit[262144];   // marca pixels já analisados

       // limpa labels e matriz de visitados
       for(x=0; x<x_size; x++)
              for(y=0; y<y_size; y++)
              {
                     labels[x+(y*x_size)] = 0;
                     visit[x+(y*x_size)]  = 0;
              }

       // pixel inicial
```

```
        labels[0] = 1;
        visit[0]  = 1;

        // enquanto não avalia todos os pixels
        while(p_an<x_size*y_size)
                if(m==0)
                {
                        // percorre toda a matriz procurando pixel não analisado
                        for(x=0; x<x_size && m==0; x++)
                                for(y=0; y<y_size && m==0; y++)
                                        if(visit[x+(y*x_size)]==0)
                                        {
                                                visit[x+(y*x_size)] = 1;
                                                lb++;
                                                labels[x+(y*x_size)] = lb;
                                                m = 1;
                                        }
                }
                // enquanto há novos pixels marcados
                else while(m>0)
                {
                        m = 0;
                        // percorre toda a matriz procurando definir label do
                        // pixel marcado
                        for(x=0; x<x_size; x++)
                                for(y=0; y<y_size; y++)
                                        if(visit[x+(y*x_size)]==1)
                                        {
                                                p_an++;
                                                visit[x+(y*x_size)] = 2;
                                                m += marca(image, labels, visit,
                                                     x, y, x_size, y_size);
                                        }
                }
}

int marca(unsigned char* image, unsigned short* labels, unsigned short* visit,
```

```
          int x, int y, int x_size, int y_size)
{
        int m=0;
        // verifica se pertence a algum objeto adjacente já identificado
        if(x>0 && labels[x-1+(y*x_size)]==0 &&
           image[x-1+(y*x_size)]==image[x+(y*x_size)] && visit[x-1+(y*x_size)]!=2)
        {
                m = 1;
                labels[x-1+(y*x_size)] = labels[x+(y*x_size)];
                visit[x-1+(y*x_size)] = 1;
        }
        if(x<(x_size-1) && labels[x+1+(y*x_size)]==0 &&
           image[x+1+(y*x_size)]==image[x+(y*x_size)] && visit[x+1+(y*x_size)]!=2)
        {
                m = 1;
                labels[x+1+(y*x_size)] = labels[x+(y*x_size)];
                visit[x+1+(y*x_size)] = 1;
        }
        if(y>0 && labels[x+((y-1)*x_size)]==0 &&
           image[x+((y-1)*x_size)]==image[x+(y*x_size)] &&
           visit[x+((y-1)*x_size)]!=2)
        {
                m = 1;
                labels[x+((y-1)*x_size)] = labels[x+(y*x_size)];
                visit[x+((y-1)*x_size)] = 1;
        }
        if(y<(y_size-1) && labels[x+((y+1)*x_size)]==0 &&
           image[x+((y+1)*x_size)]==image[x+(y*x_size)] &&
           visit[x+((y+1)*x_size)]!=2)
        {
                m = 1;
                labels[x+((y+1)*x_size)] = labels[x+(y*x_size)];
                visit[x+((y+1)*x_size)] = 1;
        }

        return m;
}
```

C.1.2 *K-Curvature*

```
// DARPA Benchmark - K-Curvature
#include ''K_curvature.h''
#include <stdio.h>

void k_curvature(unsigned short* lb, double* ang, int x_size, int y_size)
{
	int x, y, i;
	double gauss[7];

	/* calcula o ângulo para cada pixel de borda */
	for(x=0; x<x_size; x++)
		for(y=0; y<y_size; y++)
			if(lb[x+(y*x_size)]!=1) // verifica se é um componente
			{
				// verifica se é pixel de aresta (para isso deve ter
				// um pixel adjacente que não pertença ao componente)
				if(aresta(lb, x, y, x_size, y_size)==1)
					encontraAngulo(lb, ang, x, y, x_size, y_size);
			}

	/* aplica filtro de Gauss */
	// inicializa máscara
	gauss[0]=gauss[6]=3;	gauss[1]=gauss[5]=27;
	gauss[2]=gauss[4]=90;	gauss[3]=130;
	// passa máscara
	for(x=0; x<x_size; x++)
		for(y=0; y<y_size; y++)
		{
			ang[x+(y*x_size)] = 0;
			for(i=-3; i<=3; i++)
				ang[x+(y*x_size)]+=((x+i)>0 &&
				    (x+i)<x_size)?(lb[x+i+(y*x_size)]*gauss[i+3]):0;
		}
}
```

```
void encontraAngulo(unsigned short* lb, double* ang,
                  int x, int y, int x_size, int y_size)
{
       int i, j, t, w, p;
       int x1, x2, y1, y2;
       double a[262144];

       // limpa angulos
       for(i=0; i<30; i++)
              a[i] = 180;
       ang[x+(y*x_size)] = 0;
       p = 0;

       // percorre área com raio 4 em relação ao pixel origem
       for(i=-4; i<=4 && ang[x+(y*x_size)]!=180; i++)
           for(j=-4; j<=4 && ang[x+(y*x_size)]!=180; j++)
               // verifica se está dentro dos limites da imagem,
               // se não é o próprio pixel analisado, se pertence
               // ao componente e é aresta
               if((x+i)>=0 && (x+i)<x_size && (y+j)>=0
                 && (y+j)<y_size && ((i!=0) || (j!=0)) &&
                 lb[x+(y*x_size)]==lb[x+i+((y+j)*x_size)] &&
                 aresta(lb, (x+i), (y+j), x_size, y_size)==1)
               {
                   x1 = x+i;       y1 = y+j;
                   for(t=-4; t<=4; t++)
                       for(w=-4; w<=4; w++)
                           // verifica condições do pixel
                           if((x+t)>=0 && (x+t)<x_size && (y+w)>=0 &&
                             (y+w)<y_size && ((t!=0) || (w!=0)) &&
                             ((t!=i) || (w!=j)) &&
                             lb[x+(y*x_size)]==lb[x+t+((y+w)*x_size)]
                             && aresta(lb, (x+t), (y+w), x_size,
                             y_size)==1)
                             {
                                 x2 = x+t;       y2 = y+w;
                                 a[p] = calculaAngulo(x1,y1,x,y,x2,y2);
```

```
                                if(a[p] == 180) ang[x+(y*x_size)] = 180;
                                p++;
                            }
                }

        if(ang[x+(y*x_size)]!=180)
                ang[x+(y*x_size)] = moda(a);
}

double calculaAngulo(int xA, int yA, int xO, int yO, int xB, int yB)
{
        double a1, a2, a, xAd, yAd, xOd, yOd, xBd, yBd;
        short qA, qB;

        xAd = (double) xA;
        yAd = (double) yA;
        xOd = (double) xO;
        yOd = (double) yO;
        xBd = (double) xB;
        yBd = (double) yB;

        // calcula inclinação das retas AO e OB
        a1 = (xAd==xOd)?90:(atan((yAd-yOd)/(xAd-xOd)) * 57.295791433);
        a2 = (xBd==xOd)?90:(atan((yBd-yOd)/(xBd-xOd)) * 57.295791433);

        // reduz ao primeiro quadrante
        if(a1<0) a1 *= -1;      if(a1>180) a1 -= 180;   if(a1>90) a1 -= 90;
        if(a2<0) a2 *= -1;      if(a2>180) a2 -= 180;   if(a2>90) a2 -= 90;

        // localiza quadrantes dos pontos em relação ao ponto de origem
        //  1 | 3
        // ---O---
        //  2 | 4
        qA = ((yAd>yOd)?1:2) + ((xAd>xOd)?2:0); // localiza ponto A
        qB = ((yBd>yOd)?1:2) + ((xBd>xOd)?2:0); // localiza ponto B

        // calcula ângulo entre as retas
```

```
        // paridades iguais dos quadrantes subtrai-se os ângulos
        if((qA\%2) == (qB\%2)) a = a1-a2;
        else a = a1+a2;
        // caso específico onde os pontos estão em quadrantes vizinhos
        // e os ângulos são zero
        if( ( (qA==1 && qB==2) || (qA==2 && qB==4) || (qA==3 && qB==4) ||
              (qA==1 && qB==3) || (qB==1 && qA==2) || (qB==2 && qA==4) ||
              (qB==3 && qA==4) || (qB==1 && qA==3) ) && (a1==0 && a2==0) )
                a = 180;

        // se ângulo negativo
        if(a<0) a*=-1;

        return a;
}

int aresta(unsigned short* lb, int x, int y, int x_size, int y_size)
{
        if((x>0        && lb[x+(y*x_size)]!=lb[x-1+(y*x_size)])   ||
           (x<x_size-1 && lb[x+(y*x_size)]!=lb[x+1+(y*x_size)])   ||
           (y>0        && lb[x+(y*x_size)]!=lb[x+((y-1)*x_size)]) ||
           (y<y_size-1 && lb[x+(y*x_size)]!=lb[x+((y+1)*x_size)]))
                return 1;
        return 0;
}

double moda(double* a)
{
        int i, j, max;
        double am[30];
        int med[30];

        // limpa vetores
        for(i=0; i<30; i++)
        {
                am[i] = 0;
                med[i] = 0;
```

```
        }

        // calcula moda
        // verifica incidência de cada valor
        for(i=0; i<30 && a[i]!=180; i++)
            for(j=0; j<30 && a[j]!=180; j++)
                // o valor de tolerância 2 é definido na especificação do algoritmo
                if(i!=j && (a[i]+2)>=a[j] && (a[i]-2)<=a[j])
                {
                    am[i] = ((am[i]*med[i]) + a[j])/(med[i]+1);
                    med[i]++;
                }
        // procura maior incidência
        max = med[0];
        j = 0;
        for(i=1; i<30 && a[i]!=180; i++)
            if(max<med[i])
            {
                max = med[i];
                j = i;
            }

        return am[j];
}
```

C.1.3 *Smoothing*

```
// DARPA Benchmark - Smoothing
#include ''Smoothing.h''

void smoothing(unsigned char* image1, unsigned char* image2, int x_size, int y_size)
{
 int x, y;
 int min, med, max;

 for(x=0; x<x_size; x++)
   for(y=0; y<y_size; y++)
```

```
      if(x==0 || y==0 || x==x_size-1 || y==y_size-1) // image border
        image2[x+(y*x_size)]=image1[x+(y*x_size)];
      else
      {
        min = least(image1[x-1+((y-1)*x_size)], image1[x+((y-1)*x_size)],
                    image1[x+1+((y-1)*x_size)]);
        med = middle(image1[x-1+(y*x_size)], image1[x+(y*x_size)],
                     image1[x+1+(y*x_size)]);
        max = maximum(image1[x-1+((y+1)*x_size)], image1[x+((y+1)*x_size)],
                      image1[x+1+((y+1)*x_size)]);
        image2[x+(y*x_size)] = middle(min, med, max);
      }
}

int least(int n1, int n2, int n3)
{
 if(n1<n2 && n1<n3)
   return n1;
 if(n2<n3)
   return n2;
 return n3;
}

int middle(int n1, int n2, int n3)
{
 if((n1<n2 && n1>n3) || (n1>n2 && n1<n3))
   return n1;
 if((n2<n1 && n2>n3) || (n2>n1 && n2<n3))
   return n2;
 return n3;
}

int maximum(int n1, int n2, int n3)
{
 if(n1>n2 && n1>n3)
   return n1;
 if(n2>n3)
```

```
   return n2;
 return n3;
}
```

C.1.4 *Gradient Magnitude*

```
// DARPA Benchmark - Gradient Magnitude Computation
#include ''Gradient.h''

void gradient(unsigned char* image1, unsigned char* image2, int x_size, int y_size)
{
 int weight[3][3];
 double pixel_value;
 double min, max;
 int x, y, i, j;

 // Inicializando matriz
 weight[0][0] = -1; weight[0][1] = -2; weight[0][2] = -1;
 weight[1][0] =  0; weight[1][1] =  0; weight[1][2] =  0;
 weight[2][0] =  1; weight[2][1] =  2; weight[2][2] =  1;

 // Definindo valores limites
 min = DBL_MAX;
 max = -DBL_MAX;
 for (x = 1; x < x_size - 1; x++)
   for (y = 1; y < y_size - 1; y++) {
     pixel_value = 0.0;
     for (j = -1; j <= 1; j++)
           for (i = -1; i <= 1; i++)
             pixel_value += weight[j + 1][i + 1] * image1[x+i+((y+j)*x_size)];
     if (pixel_value < min) min = pixel_value;
     if (pixel_value > max) max = pixel_value;
   }
 if ((int)(max - min) == 0) {
   printf(''ERRO!\n\n'');
   return;
 }
```

```
 // Limpando imagem
 for (y = 0; y < y_size; y++)
   for (x = 0; x < x_size; x++)
     image2[x+(y*x_size)] = 0;

 // Aplicando filtro
 for (x = 1; x < x_size - 1; x++)
   for (y = 1; y < y_size - 1; y++) {
     pixel_value = 0.0;
     for (i = -1; i <= 1; i++)
       for (j = -1; j <= 1; j++)
         pixel_value += weight[j + 1][i + 1] * image1[x+i+((y+j)*x_size)];
     image2[x+(y*x_size)] = (unsigned char) 255 * (pixel_value - min)
/ (max - min);
   }
}
```

C.1.5 *Threshold*

```
// DARPA Benchmark - Threshold

#include ''Threshold.h''

void threshold(unsigned char* image1, unsigned char* image2,
               int x_size, int y_size, int lim)
{
 int i, j;

 for(i=0; i<x_size; i++)
   for(j=0; j<y_size; j++)
     image2[i+(j*x_size)] = (image1[i+(j*x_size)]>lim)?255:0;
}
```

C.2 Algoritmos de Nível Baixo

C.2.1 Erosão

```
/* erode.c */
#include<stdio.h>
#include<stdlib.h>

#define WHITE 255
#define BLACK 0

void erode(unsigned char image1[262144], unsigned char image2[262144],
int x_size, int y_size, int neighbor)
    /* 4/8-neighbor erode of binary image           */
    /* input image1[y][x] ===> output image2[y][x]   */
{
 int x, y;  /* loop variable */

 for (y = 0; y < y_size; y++)
   for (x = 0; x < x_size; x++)
     image2[x+(y*x_size)] = image1[x+(y*x_size)];

 for (y = 0; y < y_size; y++)
   for (x = 0; x < x_size; x++)
     if(image1[x+(y*x_size)]==BLACK)
     {
       if(y>0)         image2[x+((y-1)*x_size)] = BLACK;
       if(y<y_size-1) image2[x+((y+1)*x_size)] = BLACK;
       if(x>0)         image2[x-1+(y*x_size)]   = BLACK;
       if(x<x_size-1) image2[x+1+(y*x_size)]   = BLACK;
       if(neighbor>4) // neighbor=8
       {
         if(y>0         && x>0)         image2[x-1+((y-1)*x_size)] = BLACK;
         if(y>0         && x<x_size-1) image2[x+1+((y-1)*x_size)] = BLACK;
         if(y<y_size-1 && x>0)         image2[x-1+((y+1)*x_size)] = BLACK;
         if(y<y_size-1 && x<x_size-1) image2[x+1+((y+1)*x_size)] = BLACK;
       }
```

```
    }
}
```

C.2.2 Gaussiano

```
/* gauss.c */
#include ''Gauss.h''

void gauss(unsigned char* image1, unsigned char* image2, int x_size, int y_size)
{
 double pixel_value;
 double min, max;
 int x, y, i, j;
 int weight[7][7] = {
 { 1,  4,  8, 10,  8,  4,  1},
 { 4, 12, 25, 29, 25, 12,  4},
 { 8, 25, 49, 58, 49, 25,  8},
 {10, 29, 58, 67, 58, 29, 10},
 { 8, 25, 49, 58, 49, 25,  8},
 { 4, 12, 25, 29, 25, 12,  4},
 { 1,  4,  8, 10,  8,  4,  1},
};

 // Definindo valores limites
 min = DBL_MAX;
 max = -DBL_MAX;
 for (x = 3; x < x_size - 3; x++)
   for (y = 3; y < y_size - 3; y++) {
     pixel_value = 0.0;
     for (j = -3; j <= 3; j++)
           for (i = -3; i <= 3; i++)
             pixel_value += weight[j + 3][i + 3] * image1[x+i+((y+j)*x_size)];
     if (pixel_value < min) min = pixel_value;
     if (pixel_value > max) max = pixel_value;
   }
 if ((int)(max - min) == 0) {
   printf(''ERRO!\n\n'');
```

```
   return;
 }

 // Limpando imagem
 for (y = 0; y < y_size; y++)
   for (x = 0; x < x_size; x++)
     image2[x+(y*x_size)] = 0;

 // Aplicando filtro
 for (x = 1; x < x_size - 1; x++)
   for (y = 1; y < y_size - 1; y++) {
     pixel_value = 0.0;
     for (i = -3; i <= 3; i++)
       for (j = -3; j <= 3; j++)
         pixel_value += weight[j + 3][i + 3] * image1[x+i+((y+j)*x_size)];
     image2[x+(y*x_size)] = (unsigned char) 255 * (pixel_value - min)
/ (max - min);
   }
}
```

C.2.3 Mediana

```
/* mediana.c */
#include ''Mediana.h''

void mediana(unsigned char* image1, unsigned char* image2, int x_size, int y_size)
{
       int x, y, i, j, n, xp, yp;
       unsigned char num[9];
       char visit[9];
       unsigned char menor = 255;

       for(x=1; x<x_size-1; x++)
               for(y=1; y<y_size-1; y++)
               {
                   for(n=0; n<9; n++)
                       visit[n] = 0;
```

```
                    for(n=0; n<9; n++)
                    {
                        for(i=-1; i<2; i++)
                            for(j=-1; j<2; j++)
                                if((image1[x+i+((y+j)*x_size)] < menor) &&
                                    (visit[i+1+((j+1)*3)]==0))
                                {
                                    xp = x+i;
                                    yp = y+j;
                                    menor = image1[x+i+((y+j)*x_size)];
                                }
                            num[n] = menor;
                            visit[i+1+((j+1)*3)] = 1;
                            menor = 255;
                    }

                    image2[x+(y*x_size)] = num[4];
                }
}
```

C.2.4 Perímetro

```
/* perimeter.c */
void perimeter(unsigned char image1[262144], unsigned char image2[262144],
               int x_size, int y_size)
{
 int x, y;
 int min, med, max;

 for(x=0; x<x_size; x++)
   for(y=0; y<y_size; y++)
     if(x==0 || y==0 || x==x_size-1 || y==y_size-1) // image border
       image2[x+(y*x_size)]=image1[x+(y*x_size)];
     else
     {
       if (((image1[x+((y-1)*x_size)]==0)|| (image1[x-1+(y*x_size)]==0) ||
            (image1[x+1+(y*x_size)]==0)  || (image1[x+((y+1)*x_size)]==0) ) &&
```

```
            (image1[x+(y*x_size)]==255))
          image2[x+(y*x_size)] = 255;
        else
          image2[x+(y*x_size)] = 0;
      }
}
```

C.2.5 Sobel

```
/* sobel.c */
#include ''sobel.h''

void sobel(unsigned char image1[262144], unsigned char image2[262144],
           int x_size, int y_size, int type)
{
 int weight[3][3];
 double pixel_value;
 double min, max;
 int x, y, i, j;  /* Loop variable */

 if(type==0) // Definition of Sobel filter in horizontal direction
 {
   weight[0][0] = -1; weight[0][1] = -2; weight[0][2] = -1;
   weight[1][0] =  0; weight[1][1] =  0; weight[1][2] =  0;
   weight[2][0] =  1; weight[2][1] =  2; weight[2][2] =  1;
 }
 else //Definition of Sobel filter in vertical direction
 {
   weight[0][0] = -1; weight[0][1] =  0; weight[0][2] =  1;
   weight[1][0] = -2; weight[1][1] =  0; weight[1][2] =  2;
   weight[2][0] = -1; weight[2][1] =  0; weight[2][2] =  1;
 }

 /* Maximum values calculation after filtering*/
 min = DBL_MAX;
 max = -DBL_MAX;
 for (y = 1; y < y_size - 1; y++)
```

```
  for (x = 1; x < x_size - 1; x++) {
    pixel_value = 0.0;
    for (j = -1; j <= 1; j++)
          for (i = -1; i <= 1; i++)
            pixel_value += weight[j + 1][i + 1] * image1[x+i+((y+j)*x_size)];
    if (pixel_value < min) min = pixel_value;
    if (pixel_value > max) max = pixel_value;
  }
 if ((int)(max - min) == 0) {
   printf(''Nothing exists!!!\n\n'');
   exit(1);
 }

 /* Initialization of image2[y][x] */
 for (y = 0; y < y_size; y++)
   for (x = 0; x < x_size; x++)
     image2[x+(y*x_size)] = 0;

 /* Generation of image2 after linear transformtion */
 for (y = 1; y < y_size - 1; y++)
   for (x = 1; x < x_size - 1; x++) {
     pixel_value = 0.0;
     for (j = -1; j <= 1; j++)
           for (i = -1; i <= 1; i++)
             pixel_value += weight[j + 1][i + 1] * image1[x+i+((y+j)*x_size)];
     pixel_value = MAX_BRIGHTNESS * (pixel_value - min) / (max - min);
     image2[x+(y*x_size)] = (unsigned char)pixel_value;
   }
}
```

C.2.6 Soma

```
/* soma.c */
#include ''soma.h''

void soma(unsigned char* image1, unsigned char* image2, int x_size, int y_size)
{
```

```
 int i, j;

 for(i=0; i<x_size; i++)
   for(j=0; j<y_size; j++)
     image2[i+(j*x_size)] = (image1[i+(j*x_size)] + image2[i+(j*x_size)])>=255 ?
                            255 : (image1[i+(j*x_size)] + image2[i+(j*x_size)]);
}
```