Otimização de código fonte C para o processador embarcado Nios II

Rafael de Vasconcellos Peron

Orientador: *Prof. Dr. Eduardo Marques*

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

“VERSÃO REVISADA APÓS A DEFESA”

Data da Defesa:	20/12/2007
Visto do Orientador:	

USP – São Carlos
Fevereiro/2008

AGRADECIMENTOS

Àquele que rege o tempo de todos que na Terra caminham, nos mares sonham, nos céus se inspiram, e ao Universo pertencem.

À minha família, por ser a força que me impulsiona na vida.

À minha namorada, por ser fonte de carinho que sempre me abre um sorriso.

Ao meu orientador, por sempre acreditar, e me fazer crer também.

Ao pesquisador, por me mostrar o caminho para ajudar com um pouco a mais.

Aos meus amigos, por serem o muro no qual subo quando quero enxergar ao longe.

À Deus, Maria Helena, Joel, André, Helena, Renato, Rosana, Cristiana, Eduardo, Vanderlei, PGCompUSP-04, Comp2k, Liceu98, Judy, Genaro, Pingo e Tekinha por, cada um ao seu modo, se encaixarem nas linhas acima descritas.

À CAPES e ao CNPq, pelo apoio financeiro.

A frase mais empolgante para se ouvir em ciência, aquela que proclama novas descobertas, não é “Eureca!” (Descobri!), mas “Engraçado...”.

ISAAC ASIMOV

RESUMO

Este projeto apresenta uma metodologia aplicada à análise da viabilidade de se otimizar código fonte C para o processador embarcado Nios II. Esta metodologia utiliza ferramentas de análise de código que traçam o perfil da aplicação, identificando suas partes críticas em relação ao tempo de execução, as quais são o *gprof* e o *performance counter*. Para otimizar o código para o processador Nios II, são utilizadas tanto instruções customizadas quanto uma ferramenta automática de aceleração de código, o compilador C2H. Como casos de estudo, foram escolhidos três algoritmos devido à sua importância no campo da robótica móvel, sendo eles o *gaxpy*, o EKF e o SIFT. A partir da aplicação da metodologia para se otimizar cada um dos casos, foi comparada a eficiência tanto das ferramentas de análise de código, quanto das ferramentas de otimização, bem como a validade da metodologia proposta.

ABSTRACT

This project presents a methodology applied to analyze the viability of C source code optimization for the Nios II embedded processor. This methodology utilizes the *gprof* and *performance counter* source code analysis tools to profile the source code of an application, and identify its critical time consuming parts. The optimization of C source code for the Nios II processor was performed using custom instructions and an automatic source code acceleration tool, the C2H compiler. Three algorithms were chosen as study cases, based on their importance to mobile robotics. Those were the *gaxpy*, EKF and SIFT algorithms. After applying the presented methodology to optimize each study case, efficiency comparisons were made between the source code analysis tools, as well between the optimization tools, in order to validate the presented methodology.

LISTA DE FIGURAS

Figura 2.1 - Fluxograma de execução da operação Gaxpy.....	7
Figura 3.1 - Evolução do número de transistores através do tempo	14
Figura 3.2 - Diagrama de Blocos de um <i>Core</i> do Processador Nios II.....	17
Figura 3.3 - Exemplo de um sistema do processador Nios II.	19
Figura 4.1 - Diagrama de blocos da ULA do Nios II.	28
Figura 4.2 - Diagrama de blocos de uma instrução customizada do Nios II.	29
Figura 4.3 - Topologia de um sistema com um único acelerador em hardware.	31
Figura 5.1 - Diagrama de blocos do sistema SLAM multi-câmeras.....	50
Figura 5.2 - Um exemplo de localização e mapeamento baseado no algoritmo EKF executando em uma FPGA.	51
Figura 5.3 - Etapas do sistema SLAM utilizado.	60

LISTA DE GRÁFICOS

Gráfico 5.1 - Consumo do Tempo de Execução (desktop), em segundos, utilizando matriz quadrada de inteiros.....	41
Gráfico 5.2 - Consumo do Tempo de Execução (Nios II), em segundos, utilizando matriz quadrada de inteiros.....	42
Gráfico 5.3 - Tempo de execução da aplicação, em segundos, utilizando matriz quadrada de inteiros.....	44
Gráfico 5.4 - Tempo de execução da função <code>gaxpy()</code> , em segundos, utilizando matriz quadrada de inteiros.....	45
Gráfico 5.5 - Consumo de hardware da FPGA para a aplicação <code>gaxpy</code> , em elementos lógicos.....	47
Gráfico 5.6 – Tempo de execução do algoritmo EKF, em segundos.	55
Gráfico 5.7 – Tempo de execução da função <code>MatrixMultiply_ptp</code> , em segundos.....	55
Gráfico 5.8 - Consumo de hardware da FPGA para a aplicação EKF, em elementos lógicos.....	57
Gráfico 5.9 – Tempo de execução do sistema e funções, em segundos (não-otimizado).....	63
Gráfico 5.10 – Tempo de execução do sistema e funções, em segundos (otimizado com FPU).....	65
Gráfico 5.11 – Tempo de execução do algoritmo SIFT, em segundos (não-otimizado vs. otimizado).	67
Gráfico 5.12 – Tempo de execução das partes críticas, em segundos (não-otimizado vs. otimizado).	68
Gráfico 5.13 – Utilização do <i>hardware</i> , em elementos lógicos (não-otimizado vs. otimizado).....	69
Gráfico 6.1 – Comparativo entre Aumento do Custo de <i>Hardware</i> e <i>Speedup</i>	72
Gráfico 6.2 – Custo da Análise de Otimização vs. Complexidade da Aplicação.....	73
Gráfico 6.3 – Comparação do Tempo de Execução das Aplicações (Desktop vs. Embarcado).....	75

LISTA DE TABELAS

Tabela 3.1 - Elementos lógicos nas FPGAs da Altera e número máximo de <i>cores</i> do processador Nios II/s	21
Tabela 5.1 - Tempo de execução em um <i>desktop</i> , em segundos, utilizando matriz quadrada de inteiros.....	40
Tabela 5.2 - Consumo do Tempo de Execução (Nios II), em segundos, utilizando matrizes de inteiros.....	41
Tabela 5.3 - Tempo de execução da aplicação, em segundos, utilizando matriz quadrada de inteiros.....	44
Tabela 5.4 - Tempo de execução da função <code>gaxpy()</code> , em segundos, utilizando matriz quadrada de inteiros.....	45
Tabela 5.5 – Resultados do <i>gprof</i> para a execução do algoritmo EKF, em segundos.	52
Tabela 5.6 – Resultados do <i>gprof</i> para a execução da função <code>MatrixMultiply_ptp</code> , em segundos.	52
Tabela 5.7 – Tempo de execução do algoritmo EKF no Nios II, em segundos.	54
Tabela 5.8 – Tempo de execução da função <code>MatrixMultiply_ptp</code> no Nios II, em segundos.	54
Tabela 5.9 – Tempo de execução do sistema e funções, em segundos (não-otimizado).....	63
Tabela 5.10 – Tempo de execução do sistema e funções, em segundos (otimizado com FPU).....	64
Tabela 5.11 – Tempo de execução do algoritmo SIFT, em segundos (não-otimizado vs. otimizado).	66
Tabela 5.12 – Tempo de execução das partes críticas, em segundos (não-otimizado vs. otimizado).	67

LISTA DE QUADROS

Quadro 2.1 - Algoritmo <i>gaxpy</i>	6
Quadro 5.1 – <i>gaxpy</i> ().	37
Quadro 5.2 - <i>gera_vetores</i> ().	38
Quadro 5.3 - Flat Profile para $n=m=256$	38
Quadro 5.4 - Call Graph para $n=m=256$	39
Quadro 5.5 – Algoritmo da função <i>MatrixMultiply_ptp</i>	53
Quadro 5.6 – Primeiro laço de execução da função <i>cvsmooth</i>	61
Quadro 5.7 – Segundo laço de execução da função <i>cvsmooth</i>	62

LISTA DE ABREVIATURAS

ASIC	<i>Application-Specific Integrated Circuit</i>
C2H	<i>C-to-Hardware</i>
CI	<i>Circuito Integrado</i>
CPU	<i>Central Processor Unit</i>
DSP	<i>Digital Signal Processor</i>
EKF	<i>Extended Kalman Filter</i>
FPGA	<i>Field Programmable Gate Array</i>
FPU	<i>Float-Point Unit</i>
GAXPY	<i>General Ax plus y</i>
GNU	<i>GNU is Not Unix</i>
GPROF	<i>GNU Profiler</i>
HDL	<i>Hardware Description Language</i>
IDE	<i>Integrated Development Environment</i>
ILP	<i>Instruction Level Paralelism</i>
MAC	<i>Multiply and Accumulate</i>
OpenCV	<i>Open Computer Vision</i>
RAD	<i>Rapid Application Design</i>
RISC	<i>Reduced Instruction Set Computer</i>
SDRAM	<i>Synchronous Dynamic Random Access Memory</i>
SIFT	<i>Scale-Invariant Feature Transform</i>
SLAM	<i>Simultaneous Localization and Mapping</i>
SOC	<i>System on a Chip</i>
SOPC	<i>System on a Programmable Chip</i>
VHDL	<i>Very High-Speed HDL</i>

Sumário

1	Introdução.....	1
1.1	Objetivos	4
1.2	Organização do trabalho.....	4
2	Algoritmos de Estudo	5
2.1	GAXPY	6
2.2	EKF	8
2.3	SIFT.....	9
3	Computação Reconfigurável	13
3.1	Processador Soft-Core Nios II.....	16
3.2	FPGAs Altera	20
4	Ferramentas e Metodologia	23
4.1	gprof	23
4.2	Performance Counter.....	26
4.3	Instruções Customizadas	27
4.4	C2H	29
4.5	Otimização de Pontos Críticos	35
5	Desenvolvimento do Trabalho.....	37
5.1	GAXPY	37
5.2	EKF	48
5.3	SIFT.....	58
6	Conclusão	71
6.1	Trabalhos Futuros.....	75
	Referências Bibliográficas	77

1 INTRODUÇÃO

Estudos referentes ao *co-design* de *hardware* e *software* têm evoluído com as necessidades de projetos que demandem alto desempenho, proveniente do *hardware*, com flexibilidade e rápido desenvolvimento, associados às ferramentas RAD (*Rapid Application Design*), inerentes aos atuais projetos de software. Com isto, é possível desenvolver sistemas embarcados em microprocessadores (SOC – *System on a Chip*) ou dispositivos reprogramáveis (SOPC – *System on a Programmable Chip*) que atendam todas as funcionalidades antes providas pelo *software*, mas agora executadas diretamente no *hardware*, aproveitando a alta velocidade de processamento que isto representa.

Neste caminho, duas áreas se destacam na evolução dos *embedded systems*, ou sistemas embarcados, sendo estas as áreas de Compiladores e Arquiteturas de Computadores. À primeira, cabe a função de transformar um algoritmo escrito em linguagem de programação em blocos de *hardware* específicos e funcionais, que possam ser impressos em um circuito integrado (CI) ou inseridos em um sistema que utilize lógica reprogramável.

Na linha de estudo da área de compiladores, dois trabalhos podem ser destacados. O primeiro discorre sobre um compilador de algoritmos em Java para sistemas computacionais reconfiguráveis (Cardoso 2000), e o segundo é o Elcor, desenvolvido pelo antigo CAR (The Compiler and Architecture Research Group s.d.), agora uma empresa denominada Synfora (Synfora Inc. s.d.). Com o título de “Compilação de Algoritmos em Java para Sistemas Computacionais Reconfiguráveis com Exploração do Paralelismo ao Nível das Operações”, o primeiro trabalho propõe uma estratégia para compilar algoritmos descritos em linguagem Java em hardware configurável. Os

resultados obtidos mostraram que as técnicas implementadas realizam a compilação em tempo computacional razoável, com resultados considerados eficientes. Desta forma, este trabalho ampliou a utilização de computação reconfigurável aos programadores de Java que desejam gerar um *hardware* a partir do seu código-fonte, sem a necessidade do desenvolvimento “*from scratch*” do *hardware*. Já o CAR, antigo grupo de pesquisas pertencente aos Laboratórios Hewlett-Packard, desenvolveu no Elcor (The Trimaran Consortium s.d.), uma infra-estrutura de compiladores resultante de pesquisa e engenharia dos diversos aspectos e aplicações de paralelismo ao nível de instrução (ILP), contribuindo para o projeto Trimaran.

Por sua vez, o estudo de Arquiteturas de Computadores tem progredido, em grande parte, devido às novas tecnologias desenvolvidas acerca da computação reconfigurável, como as FPGAs (*Field Programmable Gate Arrays*) (Aditya, Kathail e Rau 1998) (Oldfield 1995), cujo intuito é a rápida prototipação e desenvolvimento de sistemas digitais. Nesta linha, tem-se o processador Nios II (Altera 2007), da Altera (Altera s.d.), um microprocessador baseado na arquitetura RISC, cuja possibilidade de personalização de instruções permite um aproveitamento direto e eficiente dos ganhos de desempenho obtidos pela implementação em *hardware* dos algoritmos em linguagem de programação.

Instruções customizadas aumentam o conjunto de instruções do Nios II, influenciando tanto na ferramenta de projeto de sistemas quanto na geração de macros em linguagens C e Assembly; tendo também uma influência nos compiladores. Alguns exemplos de aplicações são processamento de fluxo de dados em aplicações de rede; aplicações de processamento específico, como decodificadores de áudio; e otimização de laços (*loops*) inerentes a software. Combinando o *hardware* dedicado com o uso de instruções especificamente desenvolvidas para o projeto em questão, é possível reduzir

seqüências complexas de instruções em uma única instrução, acelerando significativamente algoritmos de *software* (Altera 2004).

Um exemplo disto pode ser visto no Quadro 1.1 (Altera 2004), que define uma multiplicação de ponto-flutuante. Neste quadro, a linha 3 refere-se ao processamento comum que ocorre utilizando as instruções disponíveis, enquanto a linha 4 mostra o desempenho alcançado com o uso de uma instrução customizada para esta tarefa. O desempenho alcançado é melhor que o obtido com o uso de processadores DSP (*Digital Signal Processors*), especializados em operações deste tipo. Adicionalmente, instruções customizadas reduzem a quantidade de código necessária para executar as operações, como neste exemplo:

- Instruções customizadas – 200 Bytes
- Apenas software – 4 000 bytes

Quadro 1.1 - Diminuição do tempo de processamento com uso de instrução customizada.

```
1 float a, b, result_slow, result_fast;  
2  
3 result_slow = a*b; /*Demora 2874 ciclos de clock*/  
4 result_fast = nm_fpmult(a,b); /*Demora 19 ciclos de clock*/
```

Desta forma, este projeto objetiva analisar a viabilidade de se otimizar aplicações escritas em código fonte C para o processador Nios II. Através de três estudos de caso, serão apresentados resultados utilizando uma metodologia de otimização que busca a maior otimização possível do tempo de execução da aplicação, com o menor conhecimento e desenvolvimento necessário de *hardware* por parte do programador. Utilizando-se ferramentas de análise de código, otimização do processador, e aceleração

de funções por *hardware*, buscou-se obter resultados que validassem, ou não, a metodologia aplicada e apresentada neste trabalho.

1.1 OBJETIVOS

O objetivo deste projeto foi validar uma metodologia de otimização de código-fonte C para projetos de *software* que executam sobre um sistema Nios II. Para validar a metodologia, foram utilizados três algoritmos diferentes como casos de estudo, cuja escolha foi embasada em suas importâncias para a área de robótica móvel. Desta forma, o objetivo foi medir o quanto as soluções apresentadas na metodologia proposta poderiam otimizar o algoritmo, implementando suas partes críticas como *hardware* dedicado no sistema Nios II.

1.2 ORGANIZAÇÃO DO TRABALHO

Este trabalho está organizado de modo que no Capítulo 2, são apresentados os algoritmos de estudo utilizados para validar a metodologia apresentada. No Capítulo 3, são apresentados conceitos de Computação Reconfigurável, abordando Reconfiguração Dinâmica, o processador Nios II e as FPGAs da Altera. O Capítulo 4 apresenta a Metodologia proposta neste projeto, bem como as ferramentas utilizadas em cada uma de suas etapas. O Capítulo 5 discorre sobre o desenvolvimento do trabalho, detalhando a aplicação da metodologia proposta em cada caso de estudo e apresentando os resultados obtidos. No Capítulo 6 são apresentados os resultados obtidos, as considerações finais deste trabalho e trabalhos futuros que podem complementá-lo.

2 ALGORITMOS DE ESTUDO

Apesar do objetivo deste trabalho ser a otimização de algoritmos, e do foco estar centrado nesse assunto, foi definido que os algoritmos a serem utilizados deveriam ter algum valor específico para a comunidade em geral. Foram três os algoritmos escolhidos, sendo eles *GAXPY*, *EKF* e *SIFT*. Embora todos sejam descritos a seguir, em seus respectivos sub-capítulos, é importante contextualizar sua utilização principalmente dos dois últimos.

Um dos tópicos que tem recebido atenção da comunidade científica é a execução das tarefas de localização e mapeamento simultâneos, na área da robótica móvel. Tal técnica é conhecida como SLAM (*Simultaneous Localization and Mapping*). Muitos algoritmos de localização partem do princípio de que existe um mapa do ambiente onde o robô deve estimar sua posição, e muitos algoritmos de mapeamento assumem que o mesmo sabe com precisão sua posição no mapa (Smith, Self e Cheeseman 1990).

A tarefa do algoritmo SLAM consiste exatamente em criar o mapa do ambiente e estimar a posição do robô nesse mapa simultaneamente, sem nenhuma informação inicial. Thrun (2002) enfatiza em seu trabalho que para a construção de robôs móveis verdadeiramente autônomos, é necessário que o mapeamento e a localização sejam estimados automaticamente e simultaneamente.

Ambos os algoritmos EKF e SIFT se encaixam no algoritmo SLAM. O EKF é utilizado na tarefa de localização, definindo, por exemplo, o estado atual do robô. Já o SIFT é empregado na tarefa de mapeamento, ao detectar *landmarks* no ambiente, que serão posteriormente utilizados para a navegação (D. Wolf, J. A. Holanda, et al. 2007).

2.1 GAXPY

Uma função *gaxpy* (*General Ax plus y*) é uma equação do tipo $c = Ax + y$, onde c e $y \in \mathbb{R}^n$; $A \in \mathbb{R}^{n \times m}$ e $x \in \mathbb{R}^m$ são dados. Esta multiplicação matriz-vetor tem seu uso amplamente difundido especialmente em sistemas de processamento digital de sinais (DSP), onde é conhecida como a função MAC (*Multiply and Accumulate*). Computadores modernos podem possuir uma unidade MAC dedicada consistindo de um multiplicador, implementado em lógica combinacional, seguido de um somador e um registro acumulador, que salva o resultado a cada ciclo de *clock*. Um modo de realizar seu cálculo é atualizar os componentes um de cada vez (Golub, et al. 1995):

$$y_i = \sum_{j=1}^n a_{ij}x_j + y_i, i = 1:m \quad [1]$$

O algoritmo *gaxpy* é descrito no Quadro 2.1. Por meio deste algoritmo é fácil perceber os dois laços inerentes a esta computação, onde cada um varia de acordo com os valores de n e m . Esta operação de multiplicação se mostra na maioria das vezes o gargalo do sistema (Aragão 1998), visto que geralmente esta operação é executada um número considerável de vezes, o suficiente para interferir de modo significativo no desempenho do sistema. Assim, tem-se que o desempenho do sistema depende da execução eficiente desta operação, tornando-a uma provável candidata à otimização.

Quadro 2.1 - Algoritmo *gaxpy*.

```
1 for (i=0; i<n; i++){
2     c[i] = y[i];
3     for (j = 0; j<m; j++)
4         c[i] = c[i] + A[i][j]*x[j];
5 }
```

Ao separar as operações individualmente, tem-se um fluxograma como o mostrado na Figura 2.1 (Gonçalves e Marques 2002). Com isto, detalhamos cada etapa do processo, dividindo as operações a serem executadas de modo a possibilitar seu mapeamento completo e ordenado, como pode ser visto na quebra da linha 4 do Quadro 2.1 em duas etapas no esquema representado na Figura 2.1.

Figura 2.1 - Fluxograma de execução da operação Gaxpy.

Assim, o algoritmo de uma função *gaxpy* se mostrou um propício caso de estudo neste trabalho. Devido ao fato de ser geralmente utilizada sobre quantidades consideráveis de dados em diversas aplicações, e seu uso ser bem difundido, este algoritmo foi o passo inicial na realização deste trabalho, cujas implementações e resultados são apresentados no Capítulo 5.

2.2 EKF

O Filtro de Kalman (KF) é um conjunto de equações matemáticas que provê um meio computacional eficiente (recursivo) de estimar o estado de um processo (Welch e Bishop 2006). Este filtro é uma variação do filtro de Bayes (Thrun 2002), o qual é um método probabilístico que utiliza evidências amostrais para calcular a probabilidade de eventos. O KF é eficiente principalmente por permitir estimações de estados passados, presentes e futuros, mesmo quando a natureza precisa do sistema modelado é desconhecida.

Esse filtro é utilizado em sistemas dinâmicos normalmente perturbados por ruídos, para estimar variáveis do sistema utilizando a distribuição Gaussiana (Maybeck 1979). Seu uso no mapeamento na robótica foi introduzido por (Smith, Self e Cheeseman 1988) para a construção de mapas contendo a estimativa de relacionamento espacial de objetos e suas incertezas. Além disso, forneceu a base teórica necessária para que outras extensões pudessem ser desenvolvidas.

O Filtro Extendido de Kalman (EKF) é uma variação do KF para quando os dados a serem tratados não se comportam linearmente. Por exemplo, os modelos dos sensores e do controle do robô utilizados no KF são funções lineares com a adição de ruído Gaussiano. No entanto, os dados gerados pelos sensores e o resultado dos comandos executados com os dados de controle do robô às vezes não se comportam linearmente, o que pode ocasionar erros no mapeamento. No EKF, as funções lineares do KF passaram a se comportar de acordo com as condições que identificam o comportamento do robô e dos sensores. Com essa mudança os dados dos sensores e do controle são interpretados como localmente lineares com adição de ruído Gaussiano.

O algoritmo do EKF é dividido em duas fases: predição e atualização, também conhecidas na literatura como *prediction* e *update*, respectivamente (Bonato 2005). A

predição serve para prever qual o valor que as variáveis do vetor estado terão ao sofrer influência de um controle, e a atualização tem a função de, a partir do dado lido de sensores e do novo valor do vetor estado, obtido na fase de predição, fazer a integração dessas informações e novamente atualizar as variáveis do vetor estado, sendo esse, o resultado final obtido após uma iteração do filtro.

A fase de atualização do algoritmo do EKF tem a função básica de integrar distribuições probabilísticas gaussianas de estimativas independentes, a fim de obter como resultado uma nova distribuição probabilística, também gaussiana, porém menos dispersa. Para isso, o filtro possui uma lógica que pondera os dados de entrada de acordo com a dispersão para que os mais precisos recebam maior peso, e conseqüentemente, influenciem com maior intensidade a resposta do filtro.

O algoritmo EKF, por ser um algoritmo recursivo e, na aplicação utilizada, executar uma grande quantidade de processamento computacional, também foi utilizado como caso de estudo e aplicação neste trabalho. Assim como o *gaxpy*, o EKF tem sido utilizado como um dos principais algoritmos de robótica, inclusive nas aplicações desenvolvidas no conjunto de trabalhos ao qual este mestrado fez parte.

2.3 SIFT

O algoritmo SIFT (*Scale-Invariant Feature Transform*) (Lowe 2004) é utilizado na detecção de *landmarks*, associando a eles um código único de identificação. As características utilizadas para detectar os *landmarks* são invariantes a escala, rotação, translação, e parcialmente à variação da iluminação e à visão 3D da câmera. Elas são bem localizadas em ambos os domínios de espaço e frequência, reduzindo a probabilidade de erro por oclusão, variações ou ruído. Diversas características podem ser extraídas de imagens comuns com algoritmos eficientes. Além disso, essas

características são bem distintas, o que permite que uma única característica seja corretamente identificada com alta probabilidade, quando comparada com um grande banco de dados de características, possibilitando uma base para reconhecimento de objetos e cenários.

O custo de extrair essas características é minimizado por meio de uma abordagem de filtragem em cascata, na qual as operações com maior custo operacional são executadas apenas em locais que passaram nos testes iniciais. A seguir estão as etapas mais importantes da computação utilizada para gerar o conjunto de características de uma imagem:

- a. **Deteccção espacial-escalar.** A primeira etapa da computação faz uma busca sobre todos os locais de imagem e escala. Ela é implementada eficientemente por uma função de diferença de Gauss, também conhecida como *blur* Gaussiano, para identificar potenciais pontos de interesse que são invariantes a escala e orientação. Esta é a parte mais custosa do algoritmo.
- b. **Localização de pontos-chave.** A cada local candidato, um modelo detalhado é relacionado para determinar escala e localização. Pontos-chave são selecionados baseados em medidas de sua estabilidade.
- c. **Atribuição de orientação.** Uma ou mais orientações são atribuídas para cada localização de ponto-chave, baseada em direções do gradiente da imagem local. Todas as operações futuras são realizadas nos dados da imagem que foi transformada relativamente à orientação, escala e localização atribuídas para cada característica, fornecendo invariância a essas transformações.
- d. **Descritor de ponto-chave.** Os gradientes da imagem local são medidos na escala selecionada na região ao redor de cada ponto-chave. Estes são

transformados em uma representação que permite níveis significantes de distorção de forma e mudanças na iluminação.

Esta abordagem foi denominada Transformação de Característica Invariante à Escala (do inglês *Scale Invariant Feature Transform*), pois transforma dados de uma imagem em coordenadas invariantes à escala relativas a características locais da imagem. Um aspecto importante desta abordagem é que ela gera um grande número de características que cobrem a imagem densamente, através de toda a variação de escalas e localizações.

Para pareamento e reconhecimento de imagens, as características SIFT são primeiramente extraídas de um conjunto de imagens de referência e armazenadas em um banco de dados. Uma nova imagem é pareada através da comparação individual de cada característica da nova imagem com sua base de dados e encontrando-se características candidatas baseado na distância euclidiana do vetor da característica.

Os descritores de pontos-chave são altamente distintos, o que permite que uma única característica encontre seu equivalente correto com grande probabilidade, mesmo em um extenso banco de dados de características. Entretanto, em uma imagem confusa, muitas características do plano de fundo não terão nenhum correspondente correto no banco de dados, dando vazão a muitos pareamentos falsos em conjunto com os corretos. As combinações corretas podem ser filtradas do conjunto completo de combinações através da identificação de subconjuntos dos pontos-chave que estão de acordo com o objeto e sua localização, escala e orientação da nova imagem. A probabilidade de que várias características estarão de acordo com estes parâmetros por acaso é menor que a probabilidade de que uma característica individual seja combinada erroneamente.

Assim como o EKF, o SIFT é um algoritmo de alto custo computacional, o que o tornou um propício caso de aplicação neste trabalho. Os cálculos matemáticos efetuados

por este algoritmo são aplicados a cada ponto de uma imagem, gerando computações repetitivas e extensas, dependendo do tamanho da imagem. Com isto, um alto poder de processamento é necessário para que este algoritmo seja executado de forma eficiente, e meios de otimizá-lo são desejados e muitas vezes necessários, especialmente para aplicações em tempo real.

3 COMPUTAÇÃO RECONFIGURÁVEL

O termo computação reconfigurável refere-se a sistemas que possuam alguma capacidade de re-programação do *hardware*, de modo a adaptar sua utilização para uma aplicação específica sendo executada em determinado momento (Compton e Hauck 2000). Este conceito existe há algum tempo, e tem como uma de suas principais características acelerar aplicações por meio da customização do hardware para necessidades computacionais específicas desta aplicação (Bondalapati e Prasana 2002). Outra característica importante da computação reconfigurável é possuir uma flexibilidade de desenvolvimento próxima às de software, utilizando-se de ferramentas que facilitam o desenvolvimento de SoCs, tornando esta uma das principais alternativas do segmento da microeletrônica contemporânea.

Com a evolução da microeletrônica nos últimos anos, surgiu uma diversidade de dispositivos eletrônicos mais velozes, com maior capacidade para armazenamento de informações, menor consumo de energia e custos cada vez mais baixos, colocando os projetistas de computadores frente a um desafio de balancear velocidade de operação dos sistemas e generalidade de operação desses sistemas. O surgimento dos circuitos integrados permitiu que grandes quantidades de transistores fossem integrados dentro de um único *chip*. O gráfico da Figura 3.1 mostra o crescimento do número de transistores por cm^3 e a evolução esperada para os próximos anos. Atualmente, chips como o Cell, da IBM, contém 234 milhões de transistores, e o Pentium 4 Extreme Edition, da Intel, possui 250 milhões de transistores (Lyons 2006).

A funcionalidade de um chip é determinada pelo seu fabricante, sendo possível projetar *chips* que executem grande quantidade de funções diferentes, que podem ser caracterizados como *chips* de uso geral. Entretanto, a velocidade desses dispositivos

pode ser lenta para certos sistemas, se comparada a dispositivos que executam apenas um conjunto limitado de instruções, destinados a aplicações específicas, tais como processadores para processamento digital de sinais, os chamados DSP (*Digital Signal Processor*). Em função disto, é possível desenvolver *chips* menores, mais velozes e com menor consumo de energia que um processador de uso geral.

Figura 3.1 - Evolução do número de transistores através do tempo

Na maioria dos *chips*, as funções lógicas são fixas e não podem ser modificadas. Assim, atualizações no hardware não são possíveis, sendo que novas funcionalidades adicionadas ao projeto implicam em um novo *chip* a ser prensado para substituir o anterior. No entanto, os FPGAs podem ser alterados, tanto nas funções lógicas como nos blocos lógicos, além das conexões entre esses blocos, simplesmente enviando sinais de configuração para os *chips*. Os blocos lógicos possuem estruturas similares aos *gate arrays* usados em alguns circuitos ASIC, mas são configurados durante a manufatura em uma empresa produtora dos *chips*. Ao contrário, os blocos lógicos em um FPGA

podem ser reescritos e reprogramados repetidamente, muito depois do *chip* ter sido produzido em uma fábrica e testado amplamente.

Recentemente, têm-se desenvolvido novos FPGAs que podem ser configurados com velocidades extremamente altas. Os primeiros FPGAs necessitavam de vários segundos ou mais para alterar seus conteúdos, o que é razoável para verificar novos protótipos em um ciclo de desenvolvimento. Nos dias de hoje, já é possível *hardwares* se adaptarem quase que continuamente em resposta à mudanças de sinais de entrada ou em resposta à sinais do meio externo.

Sistemas computadorizados podem fazer uso dos elementos configuráveis de várias maneiras. Uma das mais comuns seria comutar o hardware para diferentes funções a partir de comandos que seriam semelhantes a fechar um programa e reiniciar outro. O tempo de reconfiguração do hardware seria aceitável para determinadas aplicações. Isso poderia ocorrer também de forma dinâmica, de tal modo que a reconfiguração ocorresse entre tarefas sendo executadas por aquele FPGA.

A computação reconfigurável ainda é uma área em desenvolvimento, diferente de arquiteturas de computadores, cujos conceitos e mecanismos vêm sendo longamente testados e comprovados. Portanto, definir e validar esses tópicos em computação reconfigurável se torna um desafio para os pesquisadores atualmente. Em termos simplificados, a computação reconfigurável combina a velocidade do hardware com a flexibilidade do software. A tecnologia da computação reconfigurável consiste na habilidade de se modificar o hardware da arquitetura para que este se adeque à aplicação. Essa reconfiguração do hardware pode ser realizada de duas maneiras:

Reconfiguração Estática: a utilização do *chip* deve ser suspensa, até que o processo de reprogramação seja concluído.

Reconfiguração Dinâmica: uma parte do *chip* pode continuar sendo utilizado durante a reprogramação. São reprogramadas as regiões do chip que não estiverem sendo usadas naquele determinado momento, e as regiões em uso se mantêm inalteradas.

3.1 PROCESSADOR SOFT-CORE NIOS II

O processador Nios II é um processador *soft-core* configurável, em contrapartida aos microcontroladores, que são elementos de processamento fixos. Neste contexto, “configurável” quer dizer que características podem ser adicionadas ou retiradas para atingir objetivos de desempenho ou custo. Já “*soft-core*” significa que o *core* da CPU é apresentado como um projeto codificado em uma linguagem de descrição de *hardware* como, por exemplo, Verilog ou VHDL, e pode ser implementado em qualquer FPGA da família Altera. Ou seja, a Altera não vende *chips* Nios II, ela vende FPGAs, permitindo aos usuários que configurem o processador Nios II e seus periféricos para alcançar suas especificações, e então programar o sistema em um FPGA da Altera. A Figura 3.2 (Altera 2007) mostra o diagrama de blocos de um *core* de um processador Nios II.

A configurabilidade possibilitada pelo processador Nios II não implica que projetistas precisam criar uma nova configuração do processador Nios II para cada novo projeto. A Altera provê sistemas Nios II prontos para uso, que podem ser incorporados diretamente no projeto. Se estes sistemas pré-projetados atendem aos requisitos, não há necessidade de configurar o processador. Além disto, projetistas de software podem utilizar o simulador de instruções do Nios II para iniciar a escrita e depuração de aplicações Nios II, antes que a configuração final do hardware seja determinada.

Figura 3.2 - Diagrama de Blocos de um *core* do Processador Nios II.

O processador Nios II é um *core* de um processador RISC de propósito geral, que contém:

- Conjunto de instruções, barramento de dados e espaço de endereçamento de 32-bits;
- 32 registradores de propósito geral;
- 32 fontes de interrupção externas;
- Instrução única para multiplicação e divisão 32x32 produzindo um resultado de 32-bits;
- Instruções dedicadas para calcular produtos de multiplicação de 64-bits e 128-bits;
- Frequência do *clock* até 100MHz;
- Acesso a uma variedade de periféricos embarcados no *chip*, e interfaces para periféricos e memórias externas;

- Módulo de depuração auxiliado por hardware que permite controle de início, parada, passo e rastreamento do processador sob um ambiente de desenvolvimento integrado (IDE);
- Ambiente de desenvolvimento de software baseado nas ferramentas GNU C/C++ e Eclipse IDE;
- Arquitetura do conjunto de instruções (ISA) compatível com todos os sistemas com processador Nios II;

Um sistema com processador Nios II é equivalente a um microcontrolador ou um SoC que inclui uma CPU e uma combinação de periféricos e memória em um único *chip*. O termo “sistema do processador Nios II” se refere a um *core* do processador Nios II, um conjunto de periféricos *on-chip*, memória interna e interfaces para memória externa, tudo implementado em um único *chip* da Altera. Do mesmo modo que uma família de microcontroladores, todos os sistemas do processador Nios II usam um conjunto de instruções e um modelo de programação consistentes.

O ambiente de desenvolvimento do Nios II é baseado no compilador GNU C/C++ e na IDE Eclipse, e provê um ambiente familiar e conceituado para o desenvolvimento de software. Usando a IDE do Nios II, é possível projetar e simular aplicações para esse processador. Utilizando-se os projetos de hardware de referência, como o mostrado na Figura 3.3 (Altera 2007), o projetista pode prototipar sua aplicação executando-a em uma placa, antes de construir uma plataforma de hardware personalizado. O projetista pode personalizar o sistema do processador Nios II até que ele atenda aos requisitos de custo ou desempenho.

Figura 3.3 - Exemplo de um sistema do processador Nios II.

Seu conjunto flexível de periféricos é uma das diferenças mais importantes entre o processador Nios II e microcontroladores. Devido à natureza *soft-core* do processador Nios II, os projetistas podem desenvolver sistemas sob demanda com o conjunto exato de periféricos necessários para as aplicações alvo.

Outro ponto importante de uma arquitetura flexível é ter um mapa de endereçamento flexível. Variáveis de software podem ter acesso genérico à memória e aos periféricos, independente da localização do endereço. Portanto, o conjunto flexível de periféricos e o mapa de endereçamento não afetam os desenvolvedores de aplicações.

Os periféricos podem ser aqueles comumente encontrados em microcontroladores, tais como *timers*, interfaces de comunicação serial, E/S de propósito geral, controladores SDRAM e outras interfaces de memória. Além disto, os projetistas podem adicionar seus próprios periféricos personalizados e integrá-los ao processador Nios II.

Para sistemas onde o desempenho é crítico, onde a CPU gasta a maioria dos ciclos de *clock* executando uma parte específica do código, uma técnica comum é criar um periférico personalizado que implementa a mesma função em hardware. Esta abordagem oferece uma melhora dupla no desempenho: a implementação em hardware é mais rápida que a em software, e o processador fica livre para executar outras funções em paralelo enquanto o periférico personalizado opera nos dados.

3.2 FPGAs ALTERA

Atualmente, existem diversos fabricantes de dispositivos FPGA, dentre os quais podemos citar as empresas Altera, Xilinx, Actel, Cypress Semiconductor, QuickLogic, Atmel e Lattice/Vantis Corporation, dentre outras. As empresas Altera e Xilinx são as que representam a maior parte deste mercado, oferecendo uma grande variedade de dispositivos. Os dispositivos FPGAs fornecidos pela empresa Altera são agrupados em famílias conforme suas principais características. As famílias principais são:

- **Cyclone:** FPGAs de primeira geração que têm como características a baixa densidade e o baixo custo;
- **Cyclone II:** FPGAs de segunda geração e de baixo custo para projetos que não tenham grande necessidade de desempenho e densidade;
- **Cyclone III:** FPGAs de terceira geração, para aplicações de alto volume de vendas, baixo custo e baixo consumo de potência.
- **Stratix:** FPGA de propósito geral e com alto desempenho;
- **Stratix GX:** FPGA de alto desempenho com transceptores (*transceivers* integrados);
- **Stratix II:** FPGA de propósito geral com alta densidade e desempenho;

- **Stratix II GX:** FGPA com alta densidade, alto desempenho e com transceptores integrados.
- **Arria GX:** FPGA com protocolo de comunicação otimizado, com suporte a PCI Express, Gigabit Ethernet e Serial RapidIO, e baixo custo para aplicações que demandem tecnologias de alta velocidade.
- **Stratix III:** FPGA de terceira geração de alto desempenho e alta densidade, com um grande número de recursos para processamento digital de sinais (DSP).

Mediante as características de cada família é possível dividi-las em dois grupos. Um grupo dos FPGAs de baixa densidade e baixo desempenho formado pelas famílias Cyclone, Cyclone II e Cyclone III. O segundo grupo compreende os FPGAs de alto desempenho e com um número maior de elementos lógicos (maior densidade), sendo formado pelas famílias Stratix, Stratix GX, Stratix II, Stratix II GX, Arria GX e Stratix III. A Tabela 3.1 mostra a variação no número de elementos lógicos para os FPGAs de cada família da Altera, bem como o número de *cores* do processador Nios II versão *standard* que caberiam em cada FPGA (o tamanho do *core* é de 1400 elementos lógicos para a versão *standard*).

Tabela 3.1 - Elementos lógicos nas FPGAs da Altera e número máximo de *cores* do processador Nios II/s

Família	Elementos Lógicos	Nº máximo de cores do Nios II/s	Preço em US\$ (17/02/2008)
Cyclone	2910 a 20060	14	14 a 100
Stratix GX	10570 a 41250	29	300 a 1700
Cyclone II	4608 a 68416	48	14 a 310
Stratix	10570 a 79040	56	190 a 1700
Arria GX	21580 a 90220	64	80 a 490
Cyclone III	5136 a 119088	85	15 a 500
Stratix II GX	33880 a 132540	94	470 a 7800
Stratix II	15600 a 179400	128	210 a 10200
Stratix III	47500 a 338000	241	2180 a 11200

Os elementos lógicos apresentados na Tabela 3.1 correspondem aos blocos básicos que constituem os FPGAs, os quais são utilizados pela Altera para determinar o tamanho dos dispositivos. A medida da capacidade dos dispositivos apresentada através de elementos lógicos é considerada mais tangível do que a medida apresentada por meio de portas lógicas.

4 FERRAMENTAS E METODOLOGIA

A seguir são apresentadas as ferramentas utilizadas para análise do código fonte e otimização das aplicações, seguida por uma apresentação da metodologia proposta e aplicada neste trabalho.

4.1 GPROF

O *gprof* (Fenlason e Stallman 1998) é uma ferramenta de *profiling* que acompanha o compilador GNU Pro e é utilizado para traçar o perfil do código fonte C de um programa. O sistema Nios II utiliza este compilador para gerar os executáveis de um código-fonte escrito na linguagem C. Pode-se verificar onde um programa gasta seu tempo e quais funções chamam outras funções enquanto o programa estava executando analisando-se o *profiling* do código. Com estas informações, é possível descobrir quais partes do programa estão mais lentas que o esperado, e podem ser candidatas à reescrita e otimização para permitir uma execução mais rápida do programa. O *profiling* também pode mostrar quais funções estão sendo mais ou menos chamadas que o esperado, ajudando a encontrar erros que de outra forma não seriam percebidos.

Como o *profiler* utiliza informações coletadas durante a execução real do programa, ele pode ser utilizado em programas que são muito grandes ou complexos para se analisar pelo código-fonte. Entretanto, o modo como o programa é executado afetará a informação exibida nos dados levantados pelo *profiler*. Se algum recurso do programa não é utilizado enquanto seu perfil é levantado, nenhuma informação será gerada para esse recurso.

A maior vantagem do *gprof* é que ele provê uma visão do comportamento de todo o sistema (Altera 2006). Apesar de existir algum *overhead* causado pela adição das macros do *gprof*, esse *overhead* é distribuído igualmente por todo o sistema. As funções identificadas como as que mais consomem tempo de execução continuarão sendo as mesmas quando o programa for executado sem a instrumentação do *profiler*. Como uma desvantagem a ser considerada, problemas sensíveis em relação ao tempo de execução podem ser mascarados devido ao aumento do tamanho do programa causado pelo *gprof*.

Para realizar o *profiling* são necessários vários passos:

- **É preciso compilar e “linkar” o programa com o *profiling* habilitado.** Na IDE do Nios II, isto pode ser habilitado setando-se um parâmetro nas definições de compilação do programa. Com isto, o compilador sabe que, ao gerar o código-objeto, deve inserir as macros do *gprof* para que os tempos de execução sejam calculados.
- **Deve-se executar o programa para gerar um arquivo de dados de *profile*.** O programa deve ser executado normalmente, com seus parâmetros, arquivos e demais opções, como se fosse uma execução normal, inclusive as saídas devem ser as esperadas, visto que o *gprof* não altera a funcionalidade do programa. Entretanto, sua execução será mais lenta que o normal, devido ao tempo gasto na coleta e escrita dos dados no arquivo de *profiling*, denominado *gmon.out*.
- **Executa-se o *gprof* para analisar os dados.** Após a execução e o término do programa, deve-se executar a ferramenta *gprof* para interpretar a informação contida no arquivo *gmon.out*.

Vários modos de saída são disponibilizados para análise. O *flat profile* mostra quanto tempo o programa gastou em cada função, e quantas vezes uma função foi chamada. Este modo facilmente mostra quais funções utilizaram o maior número de ciclos do programa. Suas principais informações são fornecidas pelas colunas *% time*, *self seconds* e *calls*, que mostram, respectivamente, qual a porcentagem do tempo total gasto em uma função, quanto tempo essa função levou para executar, somando-se todas as suas execuções, e quantas vezes essa função foi executada.

O *call graph* mostra, para cada função, qual função a chamou, que outras funções foram chamadas por ela, e quantas vezes isto ocorreu. Também informa quanto tempo foi gasto por uma função e pelas funções chamadas por ela. Com esta informação, é possível descobrir funções que, enquanto elas mesmas podem não ter tomado muito tempo em sua execução, chamaram outras funções que utilizaram quantidades anormais de tempo para executar. Isto pode sugerir lugares onde seria possível tentar eliminar chamadas de funções que usem muito tempo de processamento.

O *line-by-line profiling* mostra o tempo gasto na execução de cada linha de código. Este tipo de saída deve ser utilizada em conjunto com um dos dois modos anteriores, o *flat profile* ou o *call graph*. Enquanto o *flat profile* mostra as informações para cada linha individual de modo análogo ao que mostra para funções, o *call graph* mostra quantas vezes uma determinada linha chamou a execução de uma função e, conseqüentemente, o quanto ela contribuiu para o consumo total dessa função. A listagem *annotated source* é uma cópia do código-fonte do programa, adicionada de um *label* em frente a cada função, com o número de vezes que essa função foi executada.

A imagem que o *gprof* fornece da execução de um programa é baseada em um processo de amostras e, portanto, são passíveis de inexatidão estatística. Se uma função executar apenas algumas vezes, tal que o processo de amostragem consiga pegar essa

função em ação apenas uma vez, existe uma grande chance de que ele classifique essa função como tendo sido executada zero vezes, ou duas vezes. Em contraste, o número de chamadas é calculado por contagem, e não amostragem. Portanto, ele é exato, e não irá variar a cada execução, se o programa for determinístico.

O período de amostragem é impresso no início do *flat profile*, dizendo o quão freqüente as amostras foram tomadas. O atual erro pode ser previsto. Para n amostras, o erro esperado é a raiz quadrada de n . Por exemplo, se o período de amostras é 0,01 segundos, e o tempo de execução é de 1 segundo, n é 100 amostras ($1/0,01$), a raiz quadrada de n é 10, então o erro esperado é 0,1 ($10*0,01$), ou dez por cento do tempo observado. Outro exemplo, com o mesmo período, mas com tempo de execução igual a 100 segundos, levaria a um erro de 1 segundo, ou um por cento do tempo observado. Isto não significa que um período de execução pequeno não tenha utilidade. Se o tempo de execução do programa é grande, um tempo de execução pequeno de uma função diz que ela utilizou uma fração insignificante do tempo total, e que otimizar esta função pode não trazer melhora no desempenho do sistema.

Dois modos de melhorar a eficiência do *profiling* é aumentar a quantidade de dados sobre a qual o programa executa, ou combinar dados de várias execuções, analisando-se com o *grprof*, ao final, os dados acumulados. Para manter o mesmo padrão na outra ferramenta de *profiling* utilizada neste trabalho, o *performance counter*, a ser descrito em seguida, optou-se pela utilização de bases de dados significativas o suficiente para obtenção de resultados satisfatórios.

4.2 PERFORMANCE COUNTER

O *performance counter* (Altera 2006) é uma ferramenta do processador Nios II que utiliza um bloco com contadores em hardware para medições de tempo em seções

definidas do código a ser executado. É possível calcular o tempo de até sete seções do código, utilizando-se dois contadores para fazer as medidas de cada uma delas. Um dos contadores é um contador de tempo de 64 bits que fornece o número de *clock ticks*, enquanto o outro é um contador de eventos de 32 bits (Altera 2007). O número de seções a serem cronometradas pode ser modificado durante a composição do sistema na ferramenta SOPC Builder, no módulo referente ao *performance counter*, bem como nas definições no código do programa.

Os contadores permitem realizar medidas do tempo de execução dos blocos de código em C com precisão. Esta medição é ativada por meio de macros do tipo BEGIN/END inseridas manualmente no código, marcando o início e o fim de cada um dos blocos de interesse no programa. A cada bloco de interesse dá-se o nome de seção.

A vantagem de se utilizar o *performance counter* encontra-se no fato de que ele é muito pouco intrusivo. Para cada macro BEGIN e END são necessárias somente uma ou duas instruções. O único modo de conseguir uma medição menos intrusiva seria utilizar, por exemplo, um analisador lógico, ou um monitor de *hardware* desenvolvido especialmente para tal tarefa, como descrito em Ravagnani (2007). Já uma desvantagem dessa ferramenta está no fato de que não há uma observação do contexto da execução, sendo que uma simples interrupção pode alterar o valor da medição.

4.3 INSTRUÇÕES CUSTOMIZADAS

Uma importante característica do Nios II é a possibilidade de expandir o conjunto de instruções do processador com instruções customizadas para cada projeto. O Nios II permite a inclusão de até 256 novas instruções, cujas possibilidades englobam instruções multi-ciclo com duração variável e a parametrização de instruções customizadas. Com isto, funções críticas de código, como as identificadas por meio de

ferramentas de *profiling* como o *gprof* e o *performance counter*, podem ser implementadas como instruções customizadas, passando a serem executados como parte do *hardware*, e não mais do *software* (Altera 2007).

O desenvolvimento de instruções customizadas é integrado às ferramentas de desenvolvimento do Nios II. A interface de software para instruções customizadas do Nios II abstrai os detalhes da instrução customizada do programador. Para cada instrução customizada, o ambiente de desenvolvimento integrado (IDE) do Nios II produz uma macro que é definida no arquivo de cabeçalhos do sistema. É possível chamar a macro do código C ou C++ de uma função como uma chamada normal de função, e a programação assembly não é necessária, embora as instruções customizadas também possam ser acessadas pelo do código assembly do processador Nios II (Peron, et al. 2007). A Figura 4.1 mostra um diagrama de blocos de uma ULA do Nios II, com um bloco de instrução lógica customizada anexado.

Figura 4.1 - Diagrama de blocos da ULA do Nios II.

A Figura 4.2 mostra um diagrama de blocos de uma instrução customizada para o Nios II. O processador Nios II suporta diferentes tipos de arquiteturas de instruções customizadas. Apenas as portas utilizadas na implementação da instrução customizada específica são necessárias. A Figura 4.2 também mostra uma interface opcional à lógica externa. Esta interface permite aos projetistas incluir uma interface customizada a recursos de sistema externos ao caminho de dados do Nios II.

Figura 4.2 - Diagrama de blocos de uma instrução customizada do Nios II.

4.4 C2H

O compilador de aceleração C-to-Hardware (C2H) do Nios II é uma ferramenta que permite ao desenvolvedor criar aceleradores de *hardware* customizados diretamente do código fonte em ANSI C. Um acelerador de *hardware* é um bloco lógico que implementa uma função C em *hardware*, e que geralmente aumenta o desempenho do sistema em uma ordem de magnitude (Altera 2006). Com o compilador C2H, pode-se

desenvolver e depurar um algoritmo em C desenvolvido para um processador Nios II, e então rapidamente converter parte do código para um acelerador em *hardware* implementado em um FPGA.

O compilador C2H melhora o desempenho de programas a serem executados no Nios II através da implementação de funções em C específicas como aceleradores em *hardware*. Esta ferramenta não é designada para o desenvolvimento de sistemas arbitrários em *hardware* utilizando C como a linguagem de desenvolvimento.

Duas características importantes do C2H são o *parallel scheduling* e o *loop pipelining*. O primeiro significa que o compilador C2H reconhece eventos que podem ocorrer em paralelo, e executa passos independentes simultaneamente em *hardware*. Já o segundo refere-se ao fato do compilador C2H executar em pipeline a lógica implementada por laços, baseado tanto na latência de acesso à memória quanto à quantidade de código executada em paralelo.

O compilador C2H converte apenas seções de código especificadas pelo usuário. Um programa típico contém uma mistura de código de desempenho crítico e outros códigos. Seções críticas geralmente são simples e iterativas, mas consomem a maioria do tempo de execução de um programa. Elas podem ocupar o processador tanto no cálculo de um valor, na transferência de dados, ou ambos. O melhor uso dos recursos do *hardware* é acelerar apenas as funções críticas de um programa, ao invés de converter um programa inteiro em *hardware*.

A Figura 4.3 apresenta a topologia de um sistema com um único acelerador em *hardware*, embora possam ser criados quantos forem necessários em um sistema, desde que existam recursos de *hardware* suficientes. Como pode ser visto, o acelerador se comunica com o sistema, inclusive com o processador Nios II, pelo do barramento Avalon.

Figura 4.3 - Topologia de um sistema com um único acelerador em hardware.

O compilador C2H mapeia cada elemento da sintaxe em C para uma estrutura equivalente usando regras de tradução que instanciam diretamente recursos de *hardware* baseados na entrada em código C. É possível controlar as estruturas de *hardware* geradas com mudanças simples no código fonte. Alguns exemplos de mapeamento são operadores matemáticos, que são convertidos em circuitos equivalentes em hardware (+, -, * seriam add, sub, e multiplicadores); laços se tornam máquinas de estado que iteram sobre as operações até que a condição de parada seja alcançada; ponteiros e vetores se tornam portas mestre do barramento Avalon, que acessam a mesma memória do processador.

Em processadores, o tempo de execução geralmente é o gargalo do sistema. Nos casos dos aceleradores em *hardware*, o gargalo passa a ser a comunicação com a memória, pois o sistema depende da latência e largura da banda. Partes do código cuja execução não possui dependência são executadas em paralelos, e, portanto, o acelerador precisa alimentar com dados cada operação paralela.

Algumas limitações não permitem que o C2H seja usado amplamente à vontade do usuário. Uma dessas limitações é o fato do compilador C2H não suportar tipos de dados *float*, *double*, *_complex*, *_boolean* e *_imaginary*. Os dois primeiros, principalmente, são tipos comumente usados, e que limitam consideravelmente a utilização do C2H como uma ferramenta completa para otimizações de um sistema. Além disso, atribuições tais como inicialização de vetores, cujo tamanho deve ser estabelecido na declaração, e declaração posterior de variáveis, constituem regras a serem seguidas. Outros fatores, tais como muitas operações sequenciais que não podem ser agrupadas em um laço, e chamadas de sistema, tornam uma função ou um programa inapropriado para tal ferramenta.

Em (Altera 2006) são apresentados alguns fatores que afetam o desempenho do compilador C2H, tais como acesso à memória e variáveis; operações lógicas e aritméticas; declarações; fluxo de controle; chamadas à sub-funções; compartilhamento de recursos; e dependência de dados. Estes fatores são vistos com mais detalhes a seguir, onde são comparadas as diferenças entre um compilador C e o compilador C2H.

Existem diferenças entre um compilador C e o compilador C2H na questão de mapeamento de acesso à memória. Um compilador C geralmente aloca muitos tipos de variáveis em sua memória de dados, o que as torna relativamente vantajosas devido ao baixo custo de memória por bit, e relativamente mais lenta devido ao *overhead* de instruções *load* e *store* utilizadas para acessá-las. Em algumas situações, registradores

do processador são utilizados, tornando os acessos relativamente rápidos e caros. O compilador C2H aloca variáveis escalares globais em registradores implementados com elementos lógicos, que possuem custo moderado e são rápidos.

Operações aritméticas e lógicas são mapeadas em uma ou mais instruções por um compilador C. Em alguns casos pode ser necessário chamar uma função para implementar a operação, como quando o Nios II não possui multiplicadores ou divisores em hardware, por exemplo. O compilador C2H implementa algumas operações lógicas como *wire*, sem consumir lógica alguma, tornando-as rápidas e gratuitas. Como exemplo temos *shifts* por constantes, multiplicar e dividir por uma potência de 2 constante, e operações lógicas binárias por constantes. Um compilador C mapeia expressões longas (com muitos operadores) em instruções. O compilador C2H mapeia longas expressões em lógica, o que pode levar a longos caminhos de tempo.

Quanto ao fluxo de controle, temos os casos de declarações *if* e laços. No caso de um *if*, um compilador C mapeia em um *branch* condicional e instruções associadas de avaliação de expressões. O compilador C2H mapeia um *if* em lógica, para avaliar as condições e controlar se o *hardware* implementado como corpo do *if* é executado. No caso de laços, ambos os compiladores tratam a avaliação da expressão parte de um laço da mesma forma que num *if*. Entretanto, o compilador C2H tenta colocar em um *pipeline* cada iteração do laço para alcançar um *throughput* de uma iteração por ciclo.

No caso de chamadas a sub-funções, um compilador C mapeia chamadas à sub-funções em algumas instruções para passar argumentos para ou da sub-função, e algumas instruções para chamar a sub-função. O compilador C2H mapeia uma chamada à sub-função, feita em sua função acelerada no nível topo, em um novo acelerador. Isto é custoso, e prende o fluxo do *pipeline* na função *top-level* que foi otimizada pelo C2H, o que pode resultar em severa perda de desempenho.

O compilador C2H cria instâncias únicas de recursos de *hardware* para cada operação encontrada em seu código C. Se isto exceder os requisitos de recursos, deve-se alterar o código C para compartilhar recursos, tais como utilizar sub-funções aninhadas. Desta forma, o compilador C2H cria uma única instância da função em *hardware*, compartilhada por todas as funções que chamam essa sub-função.

Dependência de dados geralmente não permite a um compilador C realizar possíveis otimizações, o que tipicamente resulta em consideráveis perdas de desempenho. Quando o C2H mapeia código em *hardware*, uma dependência de dados faz com que ele escalone operações sequencialmente, ao invés de concorrentemente, o que pode causar perdas de desempenho cruciais à aplicação.

O fluxo de projeto para o compilador C2H inicia com um ou mais arquivos C cuja compilação para o processador Nios II conclui com sucesso. Antes de se acelerar uma função com o C2H, é necessário identificar as funções que necessitam ser aceleradas e depurar as funções, inicialmente visando o Nios II. Isto porque, uma vez aceleradas, definições dentro das funções não mais podem ser depuradas.

As funções a serem aceleradas podem tanto ser funções existentes no código C inicial ou funções desenvolvidas com o propósito específico de convertê-las em *hardware*. Em ambos os casos, os resultados obtidos serão melhores se as funções forem estruturadas para o compilador C2H. Inicialmente, pode-se acelerar uma função do modo em que ela se encontra, e verificar se o resultado atende aos requisitos do projeto.

Um típico fluxo de projeto para acelerar uma função utilizando o compilador C2H envolve os seguintes passos:

1. Desenvolver e depurar a aplicação ou algoritmo em C visando o processador Nios II.

2. Traçar o perfil do código para identificar as áreas que poderiam se beneficiar da aceleração por *hardware*.
3. Isolar em uma função C individual o código que se deseja acelerar e re-compilar o projeto com a opção de acelerar essa função.
4. Traçar o perfil dos resultados.
5. Se os resultados obtidos não forem os desejados, modificar o código-fonte em C e a arquitetura do sistema.
6. Voltar ao passo 3 e iterar.

O fluxo típico de projeto do compilador C2H é um processo iterativo de acelerar uma função, comparar o desempenho com os requisitos do projeto e modificar o código-fonte C para melhorar os resultados. A primeira iteração com código em C que não foi otimizado para o compilador C2H pode não melhorar substancialmente o desempenho do sistema. Iterações seguintes, modificando-se o código para uma estrutura de hardware otimizada, geralmente melhoram os resultados obtidos.

4.5 OTIMIZAÇÃO DE PONTOS CRÍTICOS

A metodologia apresentada neste trabalho divide o processo em etapas. Desta forma, tem-se uma estrutura consistente que serve de guia durante a personalização de um sistema. Para o desenvolvimento de uma instrução customizada, a seguinte seqüência de passos é apresentada (Altera 2004):

- **Traçar o perfil do código** – com as ferramentas *GNU Profiler* e *Performance Counter* é possível determinar a quantidade de tempo que partes específicas do código necessitam para executar e quais são as funções utilizadas.

- **Identificar laços internos críticos** – estabelecendo quais são as partes do código mais propícias a serem implementadas em hardware.
- **Criar uma instrução lógica customizada ou acelerar a função em hardware** – substituindo uma ou todas as instruções das partes críticas do código encontradas na etapa anterior.
- **Importar a instrução lógica customizada no projeto** – adicionando o bloco de *hardware* customizado ao conjunto de instruções do processador, caso seja uma instrução customizada, ou ao barramento, caso seja um *hardware* acelerador.
- **Chamar a instrução implementada no código C ou Assembly** – no caso do sistema Nios II esta etapa é feita automaticamente quando executado o C2H, ou quando a instrução customizada sobrepõe uma instrução padrão do processador (multiplicação utilizando o sinal *, por exemplo). Caso contrário, deve ser utilizada a referência criada pelo próprio sistema Nios II após o *hardware* ser compilado (uma instrução especial que se chame *cus_inst*, por exemplo).

Estes cinco passos podem ser divididos em dois tópicos distintos, ou etapas de maior grandeza, os quais seriam o *profiling* do código, com o uso de ferramentas de *profiling* e a conseguinte identificação das suas partes críticas, englobando assim os dois primeiros passos do processo; e a utilização de hardware dedicado para otimizar as partes críticas, desenvolvendo-se instruções customizadas ou um acelerador em hardware, constituída dos três passos restantes.

5 DESENVOLVIMENTO DO TRABALHO

Este capítulo apresenta a aplicação da metodologia em cada um dos estudos de caso apresentados no Capítulo 2, bem como os resultados obtidos em cada caso.

5.1 GAXPY

A função `gaxpy()`, listada no Quadro 5.1, mostra a implementação da operação *gaxpy*. Esta função é chamada pela função principal 10000 vezes. Este número de chamadas a esta função foi utilizado para que o tempo de processamento fosse suficiente para que pudesse ser relatado pelo *gprof*.

Quadro 5.1 – `gaxpy()`.

```
int gaxpy(int n, int m, int *c, int *y, int *x, int **A)
{
    int i=0, j=0;

    for (i=0; i<n; i++)
    {
        c[i] = y[i];
        for (j = 0; j<m; j++)
            c[i] = c[i] + A[i][j]*x[j];
    }
    return(1);
}
```

No Quadro 5.2, tem-se o código da função `gera_vetores()`, função esta que tem a tarefa de inserir valores aleatórios nos vetores `x` e `y`, bem como na matriz `A`, para que se possa realizar a computação dos valores por meio da função `gaxpy()`. Também esta função é executada 10000 vezes. Isto foi realizado para que fosse possível comparar o tempo utilizado por uma função de atribuição de valores e outra que realiza cálculos matemáticos. Entretanto, devido a utilização da função `rand()` para gerar os valores

aleatórios, deve-se perceber a influência que esta tem no tempo total de execução da função `gera_vetores()`.

Quadro 5.2 - `gera_vetores()`.

```
int gera_vetores(int n, int m, int *c, int *y, int *x, int **A)
{
    int i=0, j=0;

    for (i=0; i<n; i++)
    {
        y[i] = rand();
    }
    for (j=0; j<m; j++)
    {
        x[j] = rand();
    }
    for (i=0; i<n; i++)
    {
        for (j = 0; j<m; j++)
            A[i][j] = rand();
    }
    return(1);
}
```

Além das funções já citadas e da função `main()`, uma outra função, chamada `string2int()` também aparece nos relatórios. Esta função foi utilizada apenas para passar parâmetros na execução do programa, e não tem utilidade prática neste experimento. Seu tempo de execução é tão pequeno que a função só é listada porque é chamada duas vezes. O Quadro 5.3 e o Quadro 5.4 mostram, respectivamente, um exemplo do gráfico *Flat Profile* e do *Call Graph*, para ($n = m = 256$).

Quadro 5.3 - Flat Profile para $n=m=256$.

```
Flat profile:

Each sample counts as 0.01 seconds.
%   cumulative   self           self         total
time  seconds  seconds   calls   ms/call  ms/call  name
64.76    34.28    34.28    10000     3.43     3.43  gaxpy
30.68    50.52    16.24    10000     1.62     1.62  gera_vetores
 4.52    52.91     2.39             0.00     0.00     0.00  rand
 0.04    52.93     0.02             0.00     0.00     0.00  main
 0.00    52.93     0.00              2     0.00     0.00  string2int
```

No Quadro 5.3, a principal informação é a denominada *self seconds*, que nos diz quanto tempo uma função levou para executar todas as suas chamadas. Outros dados importantes são a porcentagem do tempo total de execução, bem como o número de chamadas que uma função teve, e a média do tempo de execução por chamada. O *cumulative seconds* indica o tempo de execução de uma função somados os tempos de execução das funções anteriores.

Quadro 5.4 - Call Graph para n=m=256.

```

Call graph (explanation follows)

granularity:  each sample hit covers 4 byte(s) for 0.02% of 52.93
seconds

index % time      self  children      called      name
                                <spontaneous>
[1]      95.5      0.02   50.52
                                main [1]
                                34.28   0.00   10000/10000   gaxpy [2]
                                16.24   0.00   10000/10000   gera_vetores [3]
                                0.00   0.00         2/2       string2int [8]
-----
[2]      64.8      34.28   0.00   10000/10000   main [1]
                                34.28   0.00   10000         gaxpy [2]
-----
[3]      30.7      16.24   0.00   10000/10000   main [1]
                                16.24   0.00   10000         gera_vetores [3]
-----
[4]      4.5       2.39   0.00
                                <spontaneous>
                                rand [4]
-----
[8]      0.0       0.00   0.00         2/2         main [1]
                                0.00   0.00         2         string2int [8]
-----

```

Já no Quadro 5.4 é possível ver claramente quais funções uma determinada função chamou, quanto tempo de execução foi tomado pelos seus filhos e qual função requisitou sua execução. Um dado interessante de se observar é sobre a função `rand()`, que por não ser uma função do programa em questão, mas sim de uma biblioteca, é classificada como *spontaneous*.

A Tabela 5.1 contém os dados de todas as execuções, para as quais $n=m=2^k$, $k=4..9$. Quando um campo de uma função está marcado com um traço (-) na tabela, significa que o tempo de execução foi insignificante, e não gerou dados para o relatório do *gprof*. Estes dados foram coletados mediante a execução em um computador equipado com processador Pentium III 700MHz.

Tabela 5.1 - Tempo de execução em um *desktop*, em segundos, utilizando matriz quadrada de inteiros.

Matriz nxn	16	32	64	128	256	512
gaxpy()	0,1	0,45	1,51	6,28	34,28	167,73
gera_vetores()	0,09	0,24	1,08	3,91	16,24	67,18
rand()	-	0,03	0,15	0,59	2,39	10,25
main()	-	0,02	-	0,01	0,02	-
string2int()	0	0	0	0	0	0

O Gráfico 5.1 traduz a Tabela 5.1 visualmente, excetuando as duas últimas funções da tabela, por motivo de praticidade gráfica. Este gráfico demonstra como o crescimento do tempo de execução da função *gaxpy()* é mais acentuado que o das demais funções, indicando que esta função está sendo o fator mais crítico na execução do programa. Com estes dados, tem-se a conclusão de que esta seria uma função cuja otimização, implementando-a como uma instrução customizada, poderia melhorar o desempenho do sistema.

A partir disto, passou-se à implementação e execução da função sobre um sistema Nios II *full featured* operando à 50MHz em uma placa Stratix II EPS2S60 (Holanda, Peron e Marques 2006). Os testes foram os mesmos utilizados no ambiente Linux. A Tabela 5.2 mostra os resultados dessas execuções.

Gráfico 5.1 - Consumo do Tempo de Execução (desktop), em segundos, utilizando matriz quadrada de inteiros.

Tabela 5.2 - Consumo do Tempo de Execução (Nios II), em segundos, utilizando matrizes de inteiros.

Matriz nxn	16	32	64	128	256	512
gaxpy()	1,98	8,73	35,85	142,64	569,03	961,11
gera_vetores()	0,6	2,47	10,4	36,48	145,29	617,32
rand()	4,03	15,39	59,46	242,67	971,2	1203,89

O Gráfico 5.2 mostra visualmente os resultados obtidos das execuções sobre o Nios II. É possível ver que as funções *gaxpy* e *gera_vetores* praticamente mantiveram seus comportamentos, apesar dos tempos de execução terem sido superiores aos testes no Linux, como esperado, visto que a frequência do processador Nios II é consideravelmente inferior à frequência do Pentium III.

Gráfico 5.2 - Consumo do Tempo de Execução (Nios II), em segundos, utilizando matriz quadrada de inteiros.

O ponto principal que se notou na execução no Nios II foi a função `rand()`, que é uma função do sistema e teve um custo de processamento maior que a função `gaxpy()`, sendo que no ambiente Linux seu custo tinha sido muito inferior. Isto mostra que nem sempre um comportamento obtido em um tipo de sistema será parecido em outro sistema. Analisando os resultados, a conclusão a que se chega é que em uma implementação no Nios II, a função `rand()` também seria uma candidata à otimização, desenvolvendo-se uma instrução customizada ou um hardware dedicado para ela. Entretanto, neste trabalho, a função `rand()` foi ignorada, devido a finalidade puramente experimental do algoritmo, e do objetivo real ser a otimização do algoritmo *gaxpy*, cujas importantes aplicações já foram expostas previamente. A função `rand` está sendo implementada totalmente em *hardware* em outro trabalho do grupo, uma vez que ela exige um considerável esforço de pesquisa e desenvolvimento para a geração de eficientes números aleatórios (Mazzotti 2007).

Terminada a primeira etapa, que foi traçar o perfil do código com o *gprof*, tem-se em mãos os dados de sua execução no sistema embarcado Nios II. Devido ao *overhead* que o *gprof* gera no código no qual é inserido, executamos o código agora sem o *gprof*, mas com os contadores do *performance counter* inclusos, para confirmar os dados levantados pelo *gprof* de que as funções identificadas como mais custosas em relação ao tempo de execução são, de fato, as mesmas. No caso do algoritmo *gaxpy*, a função *gaxpy()* mostrou-se a parte mais crítica do código, e foi a escolhida para otimização.

Definida a função, a próxima etapa foi analisar a mesma e verificar qual seria o meio de otimizá-la com uma instrução customizada. Visto que o principal processamento efetuado por ela é matemático, mais especificamente operações de adição e multiplicação, optou-se por utilizar uma FPU (*Floating Point Unit*, ou Unidade de Ponto Flutuante) em *hardware* para otimizar seu desempenho. Esta FPU foi inserida no sistema Nios II como uma instrução customizada. Para esta etapa, foi necessário modificar manualmente o *hardware* do sistema para incluir a nova instrução no conjunto de instruções da arquitetura do Nios II. Além disso, poderia ser somado a necessidade de se desenvolver uma FPU, incluindo os conhecimentos de linguagens de programação de *hardware*, tais como Verilog e VHDL. Como o Nios II provê uma FPU pronta para uso, esta foi utilizada para diminuir o ciclo de desenvolvimento, em comparação ao ciclo da ferramenta C2H.

Como uma segunda alternativa de otimização, utilizou-se a ferramenta C2H para otimizar toda a função *gaxpy()*, executando esta operação diretamente na IDE de desenvolvimento do Nios II, sem necessidade, por parte desenvolvedor, de realizar quaisquer modificações manuais no *hardware*. Além destes dois casos, uma terceira alternativa de otimização foi realizada, onde a função *gaxpy()* foi otimizada com a

ferramenta C2H, e o programa foi executado sobre um hardware que possuía uma FPU em *hardware*.

Os resultados podem ser observados na Tabela 5.3 e no Gráfico 5.3 com relação à medição do tempo de execução total do algoritmo, enquanto na Tabela 5.4 e no Gráfico 5.4 podemos ver os resultados relacionados ao tempo de execução apenas da função `gaxpy()`. Os tempos relacionados ao *desktop* em questão foram medidos com o *gprof*, com o algoritmo executando sobre um processador Pentium III 700MHz. Por motivos práticos, os testes foram realizados com matrizes quadradas $n \times n$.

Tabela 5.3 - Tempo de execução da aplicação, em segundos, utilizando matriz quadrada de inteiros.

Matriz $n \times n$	16	32	64	128	256
Desktop	0,19	0,74	2,74	10,79	52,93
Perf. Counter	7,19	28,61	125,96	469,70	1882,30
FPU	3,16	13,53	56,52	214,64	862,22
C2H	5,27	20,47	83,61	338,62	1350,04
C2H + FPU	5,33	20,09	88,05	341,76	1350,00

Gráfico 5.3 - Tempo de execução da aplicação, em segundos, utilizando matriz quadrada de inteiros.

Tabela 5.4 - Tempo de execução da função `gaxpy()`, em segundos, utilizando matriz quadrada de inteiros.

Matriz nxn	16	32	64	128	256
Desktop	0,09	0,45	1,51	6,28	34,28
Perf. Counter	3,49	14,53	66,74	245,69	974,32
FPU	0,90	4,51	17,05	68,84	302,54
C2H	2,87	11,44	44,14	192,84	790,29
C2H + FPU	2,93	11,14	48,64	196,00	790,23

Gráfico 5.4 - Tempo de execução da função `gaxpy()`, em segundos, utilizando matriz quadrada de inteiros.

A primeira informação que se destaca é o quanto a aplicação é mais lenta executando-a no sistema embarcado, quando comparado com os tempos do *desktop*. Segundo a medição realizada com o *performance counter*, o menos intrusivo dos contadores, o tempo de execução da aplicação no Nios II é de 35 a 45 vezes maior que o tempo de execução no *desktop*, enquanto o tempo de execução da função `gaxpy()` é de 28 a 44 vezes maior. Utilizando a FPU em *hardware* como instrução customizada do processador Nios II, conseguimos uma diminuição de 52% a 54% no tempo de

execução da aplicação no Nios II, sendo a diminuição de 69% a 74% quando comparamos apenas a função `gaxpy()`. Com isto, o tempo de execução da aplicação ficou 16 a 20 vezes maior que o tempo de execução no *desktop*, sendo que a função `gaxpy()`, especificamente, teve uma variação de 8 a 11 vezes para mais no tempo de execução.

Quando utilizado o acelerador C2H, o ganho no tempo de execução foi de 27% a 34%, considerando-se a aplicação como um todo, ou de 18% a 34% quando considerada apenas a função `gaxpy()`. Com isto, os resultados finais se mostraram de 25 a 31 vezes mais lento para a aplicação, quando comparados com os resultados do *desktop*, e de 23 a 31 vezes mais lento quando comparada apenas a função `gaxpy()`. Já a combinação da FPU com instrução customizada, mais a utilização do C2H para otimizar a função `gaxpy()`, gerou resultados de 26% a 30% mais otimizados para a aplicação, quando executados no Nios II, e de 16% a 27% quando considerados apenas os tempos da função `gaxpy()`. Estes resultados se mostraram de 25 a 30 vezes mais lentos para a aplicação, e de 16 a 27 vezes mais lento para a função `gaxpy()`, quando comparados ao tempo do *desktop*.

O Gráfico 5.5 mostra o consumo de *hardware* da FPGA para cada um dos casos. Por este gráfico, pode-se ver que com uma adição de 11% no *hardware* consumido no caso da FPU, foi possível obter uma diminuição de até 54% no tempo de execução da aplicação. Já no caso do C2H, um acréscimo de 4% de *hardware* resultou em um ganho de até 34% no desempenho final da aplicação. E no caso da utilização conjunta das duas ferramentas de otimização, 14% no aumento do *hardware* resultou em até 30% de otimização. Deste modo, adotando-se a relação $\frac{speedup}{\acute{a}rea}$, por exemplo, temos que o melhor modo de otimização foi utilizando instrução customizada.

Os resultados refletem que a FPU é uma melhor solução quando o objetivo é a otimização máxima do tempo de execução da aplicação, enquanto o C2H se mostra mais efetivo no baixo consumo de *hardware*, mas com menor eficiência ao otimizar a aplicação. A primeira diferença entre ambas as aplicações que vem à tona é o fato de a FPU, por ser uma instrução customizada, e por ter sobrescrito os operadores “+”, “-”, “/” e “*” por suas versões em *hardware*, otimizou não apenas a função escolhida, mas todas as operações de adição, subtração, divisão e multiplicação da aplicação. O C2H, por outro lado, foi aplicado especificamente na função, e apenas ela se beneficiou de sua utilização. Assim, as diferenças nos resultados e na utilização de *hardware* se mostram mais claras, pois se a FPU utilizou mais *hardware*, sua abrangência também foi maior.

Gráfico 5.5 - Consumo de hardware da FPGA para a aplicação gaxpy, em elementos lógicos.

O caso da utilização conjunta das duas otimizações é interessante de ser analisado, pois foi o que mais consumiu recursos da FPGA, e o que obteve resultados piores na otimização do tempo de execução do sistema. O que aconteceu neste caso foi que ambos os *hardwares* da FPU e da função otimizada pelo C2H estavam presentes no

sistema final. Pelos resultados anteriores, vimos que a otimização da FPU foi mais eficiente que a do C2H. Neste caso, como o C2H foi utilizado para otimizar a função `gaxpy()`, a FPU não teve influência nas operações da função mais custosa da aplicação, visto que agora existia um outro sistema para executar a função, e que não utilizava as funções de multiplicação da FPU. Isto explica também os resultados praticamente idênticos com relação à otimização da função `gaxpy()` utilizando o C2H, visto que em ambos os casos ela foi otimizada de modo idêntico. Assim, o papel da FPU ficou relegado a otimizar os cálculos existentes nas outras partes da aplicação, o que neste caso eram mínimas, e não surtiram efeito sensível no resultado final.

É importante considerar que todas as medições realizadas no sistema Nios II foram realizadas utilizando-se o *performance counter*. Assim, a diferença entre as versões é apenas do método escolhido para a otimização realizada em cada caso, visto que o *overhead* causado pelo *performance counter*, independente de quanto tenha sido, foi o mesmo em todos os casos. Isto porque os contadores do *performance counter* foram colocados estrategicamente nas chamadas das funções, e não dentro das funções, evitando-se qualquer interferência quando da otimização delas, principalmente no caso do C2H.

5.2 EKF

O algoritmo EKF (Extended Kalman Filter) foi o segundo algoritmo utilizado como foco de aplicação deste projeto. Sua utilização em projetos de robótica móvel foi o incentivo principal para realizar experimentos quanto a capacidade de se otimizar a execução do algoritmo em aplicações embarcadas (Bonato, Peron, et al. 2007). Seguindo a mesma metodologia aplicada nos experimentos com o algoritmo `gaxpy`, foi testada uma aplicação que utilizava o algoritmo EKF para realizar cálculos de

localização e mapeamento no controle de um robô móvel (Bonato 2005) (D. Wolf, J. A. Holanda, et al. 2007).

Em (Bonato, Holanda e Marques 2006) é explicado que, no cálculo do EKF, a localização e movimentação de um robô que utiliza o sistema SLAM é realizada através do cálculo de *features* que compõem o ambiente que o cerca. Estas *features*, que podem ser obtidas por meio de sensores ou de um mapa previamente fornecido ao robô, representam objetos tais como obstáculos (móveis ou estáticos), paredes e esquinas, e influenciam diretamente na navegação do robô. Identificadas estas *features*, o robô decide qual o melhor caminho para atingir seu objetivo. Estes cálculos são realizados considerando-se todas as *features* conhecidas pelo robô em determinado momento, e quanto maior seu número, maior o poder computacional necessário para calcular a rota a ser seguida.

O algoritmo EKF foi codificado em linguagem C com o objetivo de compilá-lo para o processador Nios II *fast core* 50MHz, implementado em uma FPGA Altera Stratix II EPS2S60. Este algoritmo foi implementado como parte de um sistema SLAM multi-câmeras, apresentado na Figura 5.1 e discutido em (Bonato, Holanda e Marques 2006).

Para realizar as tarefas de localização e mapeamento, o EKF recebe como entrada informações de odometria e a posição das *features* nas proximidades do robô. A Figura 5.2 mostra um ambiente de $20m^2$ com 27 *features* posicionadas aleatoriamente. Cada “x” representa a verdadeira posição de uma *feature*; os pontos, a posição estimada; e as elipses, a incerteza da posição calculada pelo algoritmo. A elipse ao redor de R significa a incerteza com relação à posição do robô, e as linhas contínua e tracejada representam as trajetórias real e estimada, respectivamente.

Figura 5.1 - Diagrama de blocos do sistema SLAM multi-câmeras.

Nestes experimentos, os dados provenientes dos sensores foram gerados artificialmente, porque a extração de *features* de imagens capturadas com câmeras ainda estava em desenvolvimento. Com a finalidade de tornar a avaliação do sistema mais realística, foi inserido ruído Gaussiano em ambos os modelos de movimentação e de medição. Para analisar o desempenho do sistema, consideramos conjuntos de 15, 30 e 45 *features*, posicionadas aleatoriamente no ambiente simulado. Para cada conjunto de *features*, a predição do EKF e as fases de atualização foram avaliadas 1800 vezes, com o propósito de estabilizar os resultados obtidos. Assim, os tempos de execução

apresentados nas tabelas e gráficos a seguir correspondem a 1800 iterações do algoritmo EKF. A metodologia aplicada foi a mesma apresentada na seção 4.5 e já aplicada na seção 5.1.

a - epoch 450

b - epoch 900

c - epoch 1350

d - epoch 1800

Figura 5.2 - Um exemplo de localização e mapeamento baseado no algoritmo EKF executando em uma FPGA.

Inicialmente, foi medido o tempo de execução do EKF em um computador equipado com um processador Intel Pentium 4 de 3GHz. Esta medição incluiu o uso do *gprof*

para traçar o perfil do algoritmo no *desktop*, para uma posterior comparação com as execuções no Nios II. A Tabela 5.5 mostra os resultados obtidos nas execuções do algoritmo tanto no *desktop* quanto na FPGA. Como pode ser visto, o tempo de execução na FPGA é entre 48 e 88 vezes maior que no *desktop*. Entretanto, é válido frisar que se está comparando uma implementação embarcada executando em uma FPGA com um sistema de propósito geral com frequência do *clock* 60 vezes maior.

Tabela 5.5 – Resultados do *gprof* para a execução do algoritmo EKF, em segundos.

Features	15	30	45
Desktop (Pentium 4 3GHz)	0,84	3,72	9,25
FPGA (Nios-II Fast Core 50 MHz)	73,79	251,35	445,36

Segundo os dados da execução do *gprof*, em ambos os casos a mesma função se mostrou a parte crítica do código. A função *MatrixMultiply_ptp*, que executa multiplicação de matrizes, foi a mais custosa em relação ao tempo de processamento. Como pode ser visto na Tabela 5.6, os resultados, quando comparados com a Tabela 5.5, mostram que a função *MatrixMultiply_ptp* consome até 88% do tempo de processamento no desktop, e até 72% do tempo de processamento na FPGA, o que a tornou a principal parte crítica do código, e foco de otimização neste trabalho.

Tabela 5.6 – Resultados do *gprof* para a execução da função *MatrixMultiply_ptp*, em segundos.

Features	15	30	45
Desktop (Pentium 4 3GHz)	0,63	3,16	8,14
FPGA (Nios-II Fast Core 50 MHz)	34,46	156,85	322,53

A função *MatrixMultiply_ptp* tem como foco a multiplicação de matrizes e, portanto, tem a multiplicação e a soma de elementos como as operações a serem realizadas. Como é possível ver no Quadro 5.5, a função *MatrixMultiply_ptp* é semelhante à

função *gaxpy*, exceto pelo fato de que o segundo membro da multiplicação também é uma matriz de duas dimensões, e não apenas um vetor, como na função *gaxpy*. Outra diferença é que neste caso as multiplicações são entre números em ponto flutuante, diferentemente dos números inteiros utilizados na função *gaxpy*.

Quadro 5.5 – Algoritmo da função *MatrixMultiply_ptp*.

```
void MatrixMultiply_ptp(float **A, float **B, int m, int p, int n,
float **C)
{
// A = input matrix (m x p)
// B = input matrix (p x n)
// m = number of rows in A
// p = number of columns in A = number of rows in B
// n = number of columns in B
// C = output matrix = A*B (m x n)
int i, j, k;

for (i=0;i<m;i++)
for (j=0;j<n;j++)
{
C[i][j]=0;
for (k=0;k<p;k++)
C[i][j]=C[i][j]+A[i][k]*B[k][j];
}
}
```

Aplicando a metodologia desenvolvida neste trabalho (Holanda, Peron e Marques 2006) (Peron, et al. 2007) (Bonato, Peron, et al. 2007), foi escolhida a FPU em *hardware* para ser aplicada como instrução customizada do processador Nios II. Na próxima etapa, foi utilizado o compilador C2H para otimizar a função *MatrixMultiply_ptp*. Por fim, foi aplicada uma abordagem combinada, com o compilador C2H otimizando a função *MatrixMultiply_ptp*, enquanto o *hardware* do sistema contava com a FPU em *hardware* como instrução customizada do processador Nios II. A Tabela 5.7 e a Tabela 5.8 mostram os resultados destas execuções para o algoritmo EKF e para a função *MatrixMultiply_ptp*, respectivamente. Todas as medições de ambas as tabelas foram realizadas utilizando-se o *performance counter* e, portanto, diferenciam-se apenas nas otimizações, ou falta delas, aplicadas em cada caso.

Tabela 5.7 – Tempo de execução do algoritmo EKF no Nios II, em segundos.

Features	15	30	45
Perf. Counter	51,39	142,66	249,57
FPU	23,54	41,72	61,86
C2H	47,51	77,20	104,18
C2H + FPU	43,10	69,64	96,33

Tabela 5.8 – Tempo de execução da função MatrixMultiply_ptp no Nios II, em segundos.

Features	15	30	45
Perf. Counter	23,54	98,72	191,59
FPU	2,84	13,78	28,23
C2H	4,44	19,84	39,19
C2H + FPU	4,48	20,00	39,15

Comparando-se os dados da execução no Nios II obtidos utilizando-se o *gprof*, como vistos na Tabela 5.5 e na Tabela 5.6, com os resultados obtidos utilizando o *performance counter* na Tabela 5.7 e na Tabela 5.8, pode-se destacar que o *gprof* gerou um *overhead* até 78% maior que o *performance counter*. Considerando-se o fato do *gprof* inserir macros para cálculo no código todo, enquanto o *performance counter* o faz por meio de contadores em *hardware*, esta diferença se mostra natural. Assim, o tempo no Nios II ficou entre 27 e 61 vezes maior que o medido no *desktop*.

O Gráfico 5.6 mostra os resultados combinados apresentados na Tabela 5.5 e na Tabela 5.7, enquanto o Gráfico 5.7 mostra os resultados combinados apresentados na Tabela 5.6 e na Tabela 5.8.

Gráfico 5.6 – Tempo de execução do algoritmo EKF, em segundos.

Gráfico 5.7 – Tempo de execução da função MatrixMultiply_ptp, em segundos.

Conforme os dados apresentados, tomando-se por base as medidas efetuadas com o *performance* counter, é possível ver que a solução onde foi utilizado a FPU como instrução customizada apresentou resultados 54% a 75% mais eficientes, em relação ao algoritmo EKF, e 85% a 88% mais eficientes quando comparamos apenas a função `MatrixMultiply_ptp`. Com isto, esta otimização resultou em uma solução embarcada com tempo de execução de 7 a 28 vezes maior que sua execução no *desktop*, considerando-se o algoritmo EKF, e entre 3,5 a 4,5 vezes maior quando considerada apenas a função `MatrixMultiply_ptp`.

Quando são considerados os resultados obtidos com otimização através do compilador C2H, os resultados se apresentam de 7% a 58% mais eficientes em relação ao algoritmo EKF, e 79% a 81% mais otimizados quando é considerada apenas a função `MatrixMultiply_ptp`. Assim, esta otimização se mostrou de 11 a 56 vezes mais lenta que a execução em *desktop*, considerando-se o algoritmo como um todo, e com tempo de execução de 5 a 7 vezes maior quando é considerada apenas a função `MatrixMultiply_ptp`. É importante notar o fato de que, apesar das otimizações com o C2H terem sido parelhas às otimizações com a FPU, no que diz respeito à função `MatrixMultiply_ptp`, o mesmo não ocorreu quando se avaliou o algoritmo EKF. Isto porque a otimização do compilador C2H agiu localmente na função, enquanto a otimização pela FPU influenciou todo o algoritmo, como ocorrido no caso do algoritmo *gaxpy*.

Por fim, a combinação entre o compilador C2H e a instrução customizada FPU gerou resultados intermediários, embora mais próximos aos resultados obtidos separadamente com o C2H. No caso do algoritmo, foram obtidos resultados 16% a 61% mais eficientes, considerando-se o algoritmo todo, e 79% a 81% quando é considerada apenas a função `MatrixMultiply_ptp`. Isto corresponde a execuções de 16 a 61 vezes mais lentas

que as execuções no *desktop*, e entre 5 a 7 vezes mais lentas quando considerados apenas os resultados da função `MatrixMultiply_ptp`.

O Gráfico 5.8 mostra o consumo de *hardware* da FPGA para cada um dos casos. Por este gráfico, pode-se ver que com uma adição de 4% no *hardware* consumido no caso da FPU, foi possível obter resultados até 75% mais otimizados na execução do algoritmo EKF. Neste caso, o uso de *hardware* foi menor que no caso da aplicação *gaxpy* porque não foi utilizada a divisão em *hardware*, o que resultou em uma economia de 7% no consumo de elementos lógicos. Já no caso do C2H, um acréscimo também de 4% de *hardware* resultou em um ganho de até 58% no desempenho final da aplicação. E no caso da utilização conjunta das duas ferramentas de otimização, 7% no aumento do *hardware* resultou em até 61% de otimização. Adotando-se mais uma vez a relação $\frac{\text{speedup}}{\text{área}}$, temos que o melhor modo de otimização foi utilizando instrução customizada.

Gráfico 5.8 - Consumo de hardware da FPGA para a aplicação EKF, em elementos lógicos.

Os resultados mostram que mais uma vez a FPU foi a melhor solução quando o objetivo é a otimização máxima do tempo de execução da aplicação. Na questão do

consumo de *hardware*, ambas as opções se mostraram idênticas. Assim, mesmo utilizando quantidades semelhantes de elementos lógicos, a solução utilizando FPU se mostrou entre 40% e 50% mais eficiente, o que a tornou a solução ótima neste caso.

5.3 SIFT

O algoritmo SIFT, como explicado na seção 2.3, é utilizado para encontrar pontos de referência, ou *landmarks*, em imagens, de modo que estes *landmarks* sejam invariantes a escala, rotação, translação, e parcialmente à variação da iluminação e à visão 3D da câmera. O SIFT é um algoritmo complexo, devido às várias operações que ele executa para chegar a um resultado. Nesta implementação utilizada, ele utiliza a biblioteca *Open Computer Vision* (OpenCV), da Intel (Intel Corporation 2007). Esta biblioteca é encarregada de executar as operações nos pontos da imagem, tal como o filtro de *blur* Gaussiano.

Primeiro, as imagens são importadas da câmera. Em um sistema com câmeras estéreo, uma sequência de imagens intercaladas corresponderia a uma visão de 360° em relação ao ponto central da câmera. A seguir, o algoritmo percorre cada pixel da imagem, encontrando pontos que, depois de aplicados diversos filtros, sejam invariantes às condições mencionadas anteriormente. Com isto, pontos comuns em imagens adjacentes seriam utilizados para unificar estas imagens, quando da composição de uma visão periférica, ou para conectar imagens de um robô, quando este se encontra em movimento, com seu campo visual sendo alterado continuamente. Neste segundo caso, baseando-se nos *landmarks* já conhecidos, o algoritmo de navegação procura por eles na próxima imagem, mapeando o ambiente e estabelecendo seu posicionamento no espaço, bem como orientando sua navegação.

A partir disto, é visível o relacionamento direto entre os algoritmos EKF e SIFT, visto que o primeiro se encarrega da localização e navegação do robô móvel, enquanto o segundo é utilizado para mapear continuamente o ambiente, fornecendo informações para o primeiro. Com a principal função matemática do EKF é a multiplicação ponto-a-ponto de matrizes, o que se configura no algoritmo *gaxpy* adaptado, com *y* sendo também uma matriz em duas dimensões, a correlação entre todos os algoritmos fica estabelecida, bem como a necessidade deles para se formar um sistema eficaz para a navegação de robôs móveis (Karlsson, et al. 2005) (Se, Lowe e Little 2002) (Se, Ng, et al. 2004). A Figura 5.3 apresenta uma representação das principais etapas deste sistema (Bonato 2005).

Para realizar a otimização do algoritmo SIFT, foi seguida a metodologia apresentada, excluindo-se a execução no *desktop*, sendo testado diretamente na FPGA. Assim, a primeira etapa foi traçar o perfil do código com o *gprof*. Entretanto, devido à alta densidade do código, o *gprof* não foi capaz de mapear de forma eficiente o algoritmo. Devido às inúmeras chamadas a funções executadas, principalmente pelas funções da biblioteca *opencv*, os resultados do *gprof* ficaram demasiado fragmentados em chamadas de baixo nível do sistema, não possibilitando o rastreamento reverso para identificar suas partes críticas. Desta forma, a alternativa escolhida foi utilizar o *performance counter* para encontrar as partes críticas, sem ter como guia os resultados prévios do *gprof*.

A abordagem na utilização do *performance counter* teve de ser elaborada de forma a otimizar a busca pelas partes críticas, visto que devido às limitações com relação ao número de contadores do *performance counter*, não é possível espalhá-los por todo o código. O fato do código ter sido bem estruturado em funções de mais alto nível facilitou esta avaliação. Partiu-se da função principal, calculando-se o tempo gasto por

cada uma das funções chamadas por ela. A partir disto, foi elucidado que a função `_sift_features` era a mais custosa, tomando quase todo o tempo de execução do programa em alguns casos.

Detectar *Landmarks* naturais usando o SIFT

Estimar posição 3D dos landmarks
usando o Filtro de Partículas

Estimar posição 3D da câmera (*ego-motion*)

Gerar o mapa do ambiente com *landmarks*
usando o EKF

Figura 5.3 - Etapas do sistema SLAM utilizado.

Passou-se então a medir internamente esta função, mediante o tempo gasto pelas funções que ela chamava, e pelo tempo gasto pelas codificações internas a ela. Esta avaliação mostrou que a função *_sift_features* chama uma outra função, *build_gauss_pyr*, sendo que esta última tomava a maior parte do tempo de execução da função *_sift_features*. Avaliando internamente a função *build_gauss_pyr*, foram obtidos resultados que mostravam ser a função *cvsmooth* a mais custosa, sendo executada em praticamente todo o tempo de execução da função *build_gauss_pyr*, que a chamou. A função *cvsmooth* pertence à biblioteca *opencv*, o que indicou que a maior parte do tempo gasto era efetivamente na manipulação da imagem, neste caso, uma função responsável pelo filtro de Gauss. Não se preocupa este trabalho em descrever quais os objetivos de cada função. Mais informações podem ser encontradas em (Intel Corporation 2007).

Por fim, foi analisada a função *cvsmooth*, e descobriu-se internamente duas partes de seu código que tomavam praticamente todo seu tempo de execução. Ambas essas partes eram constituídas de um laço de execução (*for*) cada uma. O código do primeiro laço de execução pode ser visto no Quadro 5.6, enquanto o código do segundo laço pode ser visto no Quadro 5.7. Em ambos os quadros é possível ver novamente o padrão de uma multiplicação de valores seguida de uma soma, característica da função *gaxpy*.

Quadro 5.6 – Primeiro laço de execução da função *cvsmooth*.

```
int ker_x;  
int t0;  
float *tsrc;  
int ker_x_n;  
float* fmaskX;  
  
for( j = 1; j <= ker_x; j++ )  
    t0 += ((tsrc[i+ker_x_n+j])+(tsrc[i+ker_x_n-j]))*(double)fmaskX[j];
```

Quadro 5.7 – Segundo laço de execução da função *cvsmooth*.

```
double val0, val1, val2, val3;

for( i = 1; i <= ker_y; i++ )
{
    float *trow1, *trow2;
    double m = fmaskY[i];
    trow1 = rows[ker_y - i];
    trow2 = rows[ker_y + i];
    val0 += (trow1[x] + trow2[x]) * m;
    val1 += (trow1[x+1] + trow2[x+1]) * m;
    val2 += (trow1[x+2] + trow2[x+2]) * m;
    val3 += (trow1[x+3] + trow2[x+3]) * m;
}
```

Para realizar os testes do tempo de execução, foi utilizada uma mesma imagem com 320x240 *pixels* de tamanho fixo. Em uma configuração típica, um *desktop* com um processador Intel Core 2 Duo de 1.66GHz demora aproximadamente 1 segundo para processar um frame de 320x240 *pixels*. O algoritmo SIFT foi codificado em linguagem C, com o objetivo de ser compilado para o processador Nios II *fast core* 50MHz, implementado em uma FPGA Altera Stratix II EPS2S60. Com o objetivo de analisar um padrão no aumento do tempo de execução conforme o tamanho da imagem, os testes foram realizados utilizando frações da imagem de tamanhos $1/n$, $n=10..2$; fracionamento este feito no tamanho vertical da imagem, ou seja, no valor de 240 *pixels*. É importante notar que o tamanho máximo da imagem com que trabalhamos foi igual à metade de seu tamanho original. Isto foi resultado de limitações do sistema de *hardware* utilizado, que não suportou trabalhos superiores a esse valor, com relação ao tamanho da imagem.

A Tabela 5.9 mostra os resultados das execuções para cada uma das funções. Estes resultados podem ser visualizados no Gráfico 5.9. Para fins de facilitar a identificação, ambos os laços internos à função *cvsmooth* foram agrupados sob a denominação *for_parts*, e o cálculo do tempo de execução de cada um foi somado para ser apresentado como um único resultado.

Tabela 5.9 – Tempo de execução do sistema e funções, em segundos (não-otimizado).

Fração da Imagem	1/10	1/9	1/8	1/7	1/6	1/5	1/4	1/3	1/2
SIFT	124,89	140,22	170,56	193,86	264,07	294,45	460,38	640,56	1099,75
_sift_features	124,28	135,31	164,62	190,65	222,83	272,06	342,25	448,11	677,74
build_gauss_pyr	97,05	105,80	122,06	139,90	164,69	197,51	246,50	329,26	493,47
cvsmooth	97,02	105,77	122,03	139,87	164,65	197,47	246,45	329,20	493,38
for_parts	98,85	107,73	124,33	142,34	167,60	201,04	250,97	335,20	502,48

Gráfico 5.9 – Tempo de execução do sistema e funções, em segundos (não-otimizado).

Pelos valores da Tabela 5.9, pode-se perceber que o tempo gasto pelas *for_parts* varia de 50% a 80% do tempo total de execução do algoritmo, destacando estes laços como as partes críticas do código. Além disto, pode-se ver que conforme aumenta o tamanho da imagem, a influência destes laços no tempo total de execução diminui, embora ainda tome mais de 50% do tempo de execução para si. Outro ponto a ser analisado é que a soma do tempo de execução dos laços é maior que o tempo de execução da função *cvsmooth*, da qual fazem parte. Isto se deve ao considerável número de ocorrências de cada laço. Quando considerado o erro incluído nas medições

realizadas pelo *performance counter*, este erro pode se tornar mais visível. Como exemplo, quando utilizada metade do tamanho da imagem, no caso o máximo utilizado neste trabalho, a soma de ambos os laços foi de 1.470.750 ocorrências, que trouxe à vista o erro considerado pelo *performance counter* em suas medições.

Tendo definido as partes críticas, e o modo como otimizá-las, partiu-se para a implementação da FPU como instrução customizada do processador Nios II. Também neste caso foram executadas variações no tamanho da imagem utilizando frações da imagem de tamanhos $1/n$, $n=10..2$; fracionamento este feito no tamanho vertical da imagem, ou seja, no valor de 240 *pixels*. A Tabela 5.10 mostra os resultados dessas execuções. Pode-se notar que também neste caso ocorreu o fato das partes críticas, cujo tempo corresponde o medido sob o nome de *for_parts*, terem ultrapassado o tempo de execução da função à qual pertencem, *cvsmooth*. Visto que o número de ocorrências é o mesmo que a versão não otimizada, a explicação anterior também se aplica neste caso. O Gráfico 5.10 mostra visualmente os resultados da Tabela 5.10.

Tabela 5.10 – Tempo de execução do sistema e funções, em segundos (otimizado com FPU).

Fração da Imagem	1/10	1/9	1/8	1/7	1/6	1/5	1/4	1/3	1/2
SIFT	101,59	109,00	135,78	155,29	198,80	223,20	279,96	371,46	545,41
_sift_features	98,91	107,09	131,80	152,83	178,38	218,32	275,00	358,70	543,14
build_gauss_pyr	76,60	83,01	95,77	109,79	129,22	154,92	193,31	258,17	387,00
cvsmooth	76,59	83,00	95,75	109,77	129,20	154,89	193,27	258,13	386,93
for_parts	77,01	83,46	96,31	110,31	129,85	155,70	194,32	259,50	389,04

A próxima etapa da metodologia foi aplicar o compilador C2H para otimizar as partes críticas encontradas, a fim de se comparar seus resultados com a otimização por meio da FPU. Porém, no caso do algoritmo SIFT, o C2H não foi capaz de otimizar o código. O compilador interrompeu a execução no meio da compilação em todas as tentativas, alegando o erro C2H ERROR: copysign already declared. Este

erro não foi possível de ser sanado, visto que as informações de erro geradas pelo C2H se mostraram escassas, sem nem ao menos indicar em que parte do código o erro havia sido encontrado. Analisando-se o comportamento do programa, chegou-se à conclusão que foi um erro durante a geração do *hardware*, dentro da compilação do C2H, e não do código-fonte em si. Como esta ferramenta é fechada, não foi possível tentar alterações na mesma para contornar o erro.

Gráfico 5.10 – Tempo de execução do sistema e funções, em segundos (otimizado com FPU).

Especulando-se sobre uma possível causa disto, revisando as limitações do compilador C2H e as características das partes críticas a serem otimizadas, algumas suposições podem ser feitas. Primeiro, seria o fato da densidade do algoritmo, e das partes críticas deste estarem intercaladas com outras funções dentro de um mesmo arquivo. Entretanto, a alternativa de se retirar os laços para fora, em um arquivo separado, resultou no mesmo erro. No caso, cada loop da função *cvsmooth* é recursivo,

seno executada ao menos 18 vezes, dependendo do número de *features* a serem analisadas, além de um complexo controle de estabilidade. Assim, é de se esperar que uma ferramenta automática não tenha facilidade em resolver tais dependências, como no caso do C2H. Uma possível solução seria explorar paralelismo em *hardware*, algo que a ferramenta C2H não é capaz de fazer, mas que foi implementado em (Bonato 2008).

Outro fator que poderia ter gerado tal erro seria o fato do algoritmo trabalhar com ponteiros para números em ponto flutuante, bem como com o tamanho *double* (32 bits). Neste caso, modificações destas variáveis não foram possíveis, visto que interfeririam em todo o código da biblioteca, podendo afetar o resultado final, até mesmo influenciando-o de alguma maneira que prejudicasse o algoritmo. Assim, no caso do algoritmo SIFT, optou-se apenas pela otimização com a FPU.

A Tabela 5.11 mostra a comparação entre os resultados de execução do sistema não-otimizado, com o sistema otimizado com a FPU, considerando-se o tempo total de execução do algoritmo SIFT. Esta mesma comparação é ilustrada no Gráfico 5.11. Por estes resultados, é possível ver que a otimização utilizando uma FPU em *hardware* implementada como instrução customizada do processador Nios II obteve resultados entre 18% e 50% melhores, com relação ao tempo de execução.

Tabela 5.11 – Tempo de execução do algoritmo SIFT, em segundos (não-otimizado vs. otimizado).

Fração da Imagem	1/10	1/9	1/8	1/7	1/6	1/5	1/4	1/3	1/2
SIFT	124,89	140,22	170,56	193,86	264,07	294,45	460,38	640,56	1099,75
SIFT (com FPU)	101,59	109,00	135,78	155,29	198,80	223,20	279,96	371,46	545,41

Gráfico 5.11 – Tempo de execução do algoritmo SIFT, em segundos (não-otimizado vs. otimizado).

Com relação apenas às partes críticas, a comparação entre as execuções não-otimizadas e otimizadas com FPU é mostrada tanto na Tabela 5.12 quanto no Gráfico 5.12. Neste caso, pode-se ver que a otimização com a FPU gerou resultados aproximadamente 22% mais rápidos com relação ao tempo de execução, em todos os casos. Estes resultados mostram que no caso do SIFT, o benefício da FPU foi bem mais aproveitado pelas outras funções do sistema, visto que o algoritmo como um todo apresentou resultados até 50% melhores.

Tabela 5.12 – Tempo de execução das partes críticas, em segundos (não-otimizado vs. otimizado).

Fração da Imagem	1/10	1/9	1/8	1/7	1/6	1/5	1/4	1/3	1/2
for_parts	98,85	107,73	124,33	142,34	167,60	201,04	250,97	335,20	502,48
for_parts (com FPU)	77,01	83,46	96,31	110,31	129,85	155,70	194,32	259,50	389,04

Gráfico 5.12 – Tempo de execução das partes críticas, em segundos (não-otimizado vs. otimizado).

O Gráfico 5.13 mostra o consumo de *hardware* de ambas as implementações. Neste caso, como no EKF, a utilização da FPU sem a função de divisão implicou em um acréscimo de 4% no custo total do *hardware* utilizado. Ou seja, com um acréscimo de 4% no custo do *hardware*, foi possível obter otimizações de até 50% no tempo de execução do algoritmo.

Comparando o resultado do tempo de execução do algoritmo SIFT em uma FPGA com o Nios II otimizado, com a execução do SIFT em um *desktop*, vemos que a execução embarcada é 500 vezes mais lenta. É preciso considerar que a velocidade do processador do *desktop* em questão é 33 vezes maior que a velocidade do processador da FPGA. Além disso, há que se considerar que, para um sistema de robótica móvel ideal utilizando este algoritmo, os resultados devem ser apresentados em tempo real, visto que afetam a navegação do robô. Neste caso, nem mesmo a solução utilizando o *desktop* em questão se mostra propícia como sistema de *hardware*. Assim, a solução

embarcada, embora ainda em um estágio inicial, se mostra promissora, e digna de estudos que possibilitem avanços em relação à otimização do tempo de execução. Entretanto, este trabalho de otimização do algoritmo SIFT foi implementado como parte da tese de doutoramento de Bonato (2008), a ser defendida em Fevereiro de 2008.

Gráfico 5.13 – Utilização do *hardware*, em elementos lógicos (não-otimizado vs. otimizado).

6 CONCLUSÃO

A metodologia proposta nesta dissertação de mestrado se mostrou viável e adequada à otimização de algoritmos em código-fonte C executados no processador embarcado Nios II em FPGAs . Esta metodologia foi desenvolvida baseando-se em uma seqüência de passos estabelecidos pela Altera, adaptando-os para incluir novas ferramentas e soluções. Os algoritmos utilizados como casos de estudo para a validação da metodologia proposta foram escolhidos levando-se em consideração sua relevância à projetos de robótica móvel embarcada, área na qual o Laboratório de Computação Reconfigurável, ao qual o mestrando e orientador deste trabalho fazem parte, voltou suas atenções no desenvolvimento de projetos.

Como demonstrado neste projeto, seguindo a metodologia proposta, foram alcançadas otimizações de até 75% no tempo de execução dos sistemas, com um custo de *hardware* 4% maior que o sistema original. Também foram demonstrados as vantagens de se utilizar instruções customizadas do processador Nios II para alcançar tais níveis de otimização. Embora esta solução exija algum conhecimento de linguagens de programação de *hardware*, ainda assim se mostra uma tarefa mais simples quando comparada ao desenvolvimento de um sistema completo em *hardware* equivalente ao algoritmo. Desta forma, foi mostrado que com modificações bem direcionadas, relacionadas unicamente às partes críticas do código, foi possível aumentar em até 4 vezes o desempenho do sistema. O Gráfico 6.1 relaciona o aumento do custo do *hardware* com o *speedup* obtido no melhor caso para cada um dos três estudos de caso apresentados.

Gráfico 6.1 – Comparativo entre Aumento do Custo de *Hardware* e *Speedup*.

Uma etapa importante que foi adicionada à versão inicial desta metodologia foi a utilização do compilador C2H, lançado pela altera em 2006, quando este trabalho já havia sido iniciado. Embora tenha mais restrições quando comparado às instruções customizadas e não possa ser utilizado em todos os casos, o C2H atinge níveis de otimização próximos aos das instruções customizadas quando sua execução é realizada com sucesso, sem um maior acréscimo de *hardware* ao sistema original que a outra solução, como visto nos resultados obtidos.

Em compensação, a menor eficiência alcançada na otimização através do C2H pode ser balanceada pela utilização mais simples desta ferramenta quando comparada com as instruções customizadas. Utilizando o C2H, nenhum conhecimento de desenvolvimento de *hardware* é necessário por parte desenvolvedor de *software*, sem interesse em aprender como desenvolver *hardware*, que queira otimizar suas aplicações embarcadas. Assim, esta solução também provou ter seu valor na metodologia apresentada. Entretanto, para aplicações complexas, como foi o caso do SIFT, uma ferramenta simples de otimização é insuficiente, necessitando ainda de desenvolvimento em

hardware voltado especificamente para otimizar essas funções. O Gráfico 6.2 mostra a relação entre o custo de análise e otimização de uma aplicação, em comparação com a complexidade desta aplicação, aplicando a metodologia apresentada.

Gráfico 6.2 – Custo da Análise de Otimização vs. Complexidade da Aplicação.

A utilização de ferramentas de avaliação e medição do código fonte foi de extrema valia, e mostrou-se não apenas prática, mas até mesmo necessária na metodologia apresentada, visando despendar o menor tempo possível em cada etapa do fluxo de otimização. O *gprof* se mostrou uma ferramenta muito abrangente, que quando utilizada na etapa inicial do processo, traça um perfil fiel das características do código da aplicação. Com isto, candidatos à otimização podem ser pré-estabelecidos, visto que o *overhead* implicado pelo *gprof* é igualmente distribuído, não interferindo proporcionalmente nos resultados com relação à quais são as partes críticas do código.

Por outro lado, o *performance counter*, que também foi uma adição realizada neste projeto à metodologia inicial, embora seja mais limitado na abrangência de cobertura do código, mostra-se mais preciso em suas medições, e próprio para traçar um perfil mais

fiel quando o campo de estudo se limita às partes críticas encontradas com o *gprof*. Além de ser menos intrusivo, o *performance counter* pode ser utilizado para efetuar medições de tempo nas funções mesmo quando são utilizadas as abordagens de otimização, tanto utilizando instruções customizadas quanto o compilador C2H. Ademais, devido à possibilidade de medir apenas as partes desejadas, com o mínimo de *overhead* no sistema, o *performance counter* serviu de meio de validação aos resultados apresentados pelo *gprof*, tornando estas duas ferramentas uma dupla eficiente quando da busca de partes críticas de uma aplicação embarcada.

Em comparação com um *desktop*, as aplicações quando portadas para sistemas embarcados se vêem em um delicado processo de adaptação. A diferença entre um sistema mais custoso, com maior consumo de potência e com um maior poder de processamento geral é visível quando comparado a um sistema embarcado, mesmo que otimizado, como mostra o Gráfico 6.3. Devido às metas gerais de baixo consumo de potência e baixo custo final que permeiam os sistemas embarcados atuais, são um obstáculo para aplicações muito complexas. Estas demandam um estudo especial em cada caso para uma otimização viável e efetiva, não só para que seu desempenho se equipare ao do *desktop*, mas também para que o supere, devido à utilização de um *hardware* extremamente dedicado unicamente a uma determinada aplicação. Um processo que se distancia ao escopo da metodologia apresentada, mas que ainda é necessário perante as ferramentas de otimização automática atuais .

Com resultados satisfatórios, quando não ótimos em relação à proposta inicial de estudo, este trabalho se mostrou de extremo valor na validação de uma metodologia que pode ser aplicada com sucesso na otimização de sistemas embarcados. Apesar de obstáculos terem surgido no caminho, eles foram transpostos em sua maioria. Nos casos

que não foram possíveis de ser solucionados, foram apresentadas as devidas justificativas ou possíveis soluções a serem abordadas em trabalhos futuros.

Gráfico 6.3 – Comparação do Tempo de Execução das Aplicações (Desktop vs. Embarcado).

6.1 TRABALHOS FUTUROS

Como trabalho futuro fica a ponderação de se investigar mais profundamente as causas da impossibilidade de otimização do algoritmo SIFT por parte do compilador C2H. Entender melhor as limitações do compilador, bem como sua total abrangência, pode resultar em uma solução que contorne o problema, mesmo que para isto tenha que ser alterada a biblioteca *opencv*, influenciando de alguma forma os resultados. A substituição de valores em formato ponto-flutuante por inteiros, ou mesmo ponto-fixado, pode apresentar resultados interessantes de serem avaliados.

Outro ponto que pode ser explorado é com relação às limitações de *hardware* do sistema embarcado que impediram o algoritmo SIFT de ser executado sobre o tamanho total da imagem. Uma abordagem pode ser a troca do tipo de memória utilizada ou do

barramento de comunicação, para encontrar caminhos que indiquem uma solução deste problema, visto que sua transposição é fundamental para uma utilização futura de sucesso do algoritmo SIFT em aplicações de robótica móvel.

Ademais, acrescentar outros estudos de caso, principalmente para aplicações finais diferentes da robótica móvel, pode ajudar a consolidar a metodologia, bem como expandir sua abrangência para os mais diversos campos de aplicação dos sistemas embarcados.

REFERÊNCIAS BIBLIOGRÁFICAS

Aditya, S., V. Kathail, e B. R. Rau. “Elcor’s machine description system: Version 3.0.” *Technical Report HPL-98-128, HP*, 1998.

Altera. *Altera*. <http://www.altera.com> (acesso em 30 de 11 de 2007).

—. “Design with Nios II and SOPC Builder.” *Treinamento Altera*. 2004.

—. “Nios II C2H Compiler User Guide.” *Altera*. Agosto de 2006.
<http://www.altera.com/literature/lit-nio2.jsp> (acesso em 29 de Maio de 2007).

—. “Nios II Custom Instructions User Guide.” *Altera*. Maio de 2007.
<http://www.altera.com/literature/lit-nio2.jsp> (acesso em 15 de Novembro de 2007).

—. “Nios II Processor Reference Handbook.” *Altera*. 2007.
<http://www.altera.com/literature/lit-nio2.jsp> (acesso em 13 de Novembro de 2007).

—. “Optimizing Nios II C2H Compiler Results.” *Altera*. Julho de 2006.
<http://www.altera.com/literature/lit-nio2.jsp> (acesso em 29 de Maio de 2007).

—. “Performance Counter Core.” *Altera*. Maio de 2007.
<http://altera.com/literature/lit-nio2.jsp> (acesso em 29 de Maio de 2007).

—. “Profiling Nios II Systems.” *Altera*. Fevereiro de 2006.
<http://altera.com/literature/lit-nio2.jsp> (acesso em 29 de Maio de 2007).

Aragão, A. *Uma Arquitetura Sistólica para Solução de Sistemas Lineares Implementada com circuitos FPGAs*. Dissertação de Mestrado, 1998.

Bonato, V. *Proposta de uma arquitetura baseada em FPGA ara SLAM com multi-câmeras aplicada à robótica móvel*. Submetido para defesa: Tese de Doutorado, 2008.

—. *Um sistema de mapeamento e localização simultâneos usando multi-câmeras para robôs móveis*. Qualificação de Doutorado, 2005.

Bonato, V., J. A. M. de Holanda, e E. Marques. “An Embedded Multi-Camera System for Simultaneous Localization and Mapping.” *International Workshop on Applied Reconfigurable Computing*, Agosto de 2006: 109-114.

Bonato, V., R. Peron, D. F. Wolf, J. A. M. de Holanda, E. Marques, e J. M. P. Cardoso. “An FPGA Implementation for a Kalman Filter with Application to Mobile Robotics.” *IEEE Second International Symposium on Industrial Embedded Systems*, 2007, 2007 ed.: 1-10.

Bondalapati, K., e V. K. Prasana. “Reconfigurable computing systems.” *Proceedings of the IEEE*, 2002: 1201-1217.

Cardoso, J. M. P. *Compilação de Algoritmos em Java para Sistemas Computacionais Reconfiguráveis com Exploração do Paralelismo ao Nível das Operações*. Instituto Superior Técnico de Portugal: Tese de Doutorado, 2000.

Compton, K., e S. Hauck. “An introduction to reconfigurable computing.” *IEEE Computing*, 2000.

Fenlason, J., e R. Stallman. *GNU gprof - The GNU Profiler*. Free Software Foundation, 1998.

Golub, G. H., D. Halperin, J. C. Latombe, and R. Wilson. *Algorithmic Foundation of Robotics*. Massachussets: A. K. Peters Ltda, 1995.

Gonçalves, R. A., e E. Marques. *ARCHITECT-R: Uma Ferramenta para o Desenvolvimento de Robôs Móveis Reconfiguráveis*. Tese de Mestrado, 29 de Julho de 2002.

Holanda, J. A. de, R. Peron, e E. Marques. “Uma Metodologia para Análise de Desempenho em Códigos Executados no Processador Nios II.” *Jornadas em Computação Reconfigurável*, 2006.

Intel Corporation. *Open Source Computer Vision Library*. 2007.
<http://www.intel.com/technology/computing/opencv/index.htm> (acesso em 28 de Novembro de 2007).

Karlsson, N., E. Bernardo, J. Ostrowski, L. Gonçalves, P. Pirjanian, e M. E. Munich. “The vslam algorithm for robust localization and mapping.” *Proceedings of the IEEE International Conference on Robotics & Automations (ICRA)*, 2005: 24-29.

Lowe, D. “Distinctive image features from scale-invariant keypoints.” *International Journal of Computer Vision* vol. 60, n. 2 (2004): 91-110.

Lyons, D. “Holy chip!” 2006. www.forbes.com/forbes/2006/0130/076.html (acesso em Agosto de 2007).

Maybeck, P. *Stochastic models, estimation, and control, vol. 1*. Londres: Academic Press, 1979.

Mazzotti, B. F. “Implementação em FPGA do Gerador de Números Aleatórios Mersenne Twister.” Projeto de Graduação II, 2007.

Oldfield, J. V. *Field-programmable gate arrays: reconfigurable logic for rapid prototyping and implementation of digital systems*. Wiley Interscience, 1995.

Peron, R., V. Bonato, D. Wolf, J. A. de Holanda, e E. Marques. “Otimizando Algoritmos Robóticos em Linguagem C para o Processador Nios II.” *Jornadas em Computação Reconfigurável*, 2007: 66-73.

Ravagnani, G. S. *Projeto de um sistema para monitoramento de hardware/software on-chip baseado em computação reconfigurável*. 2007.

Se, S., D. Lowe, e J. Little. “Mobile robot localization and mapping with uncertainty using scale-invariant visual landmarks.” *The International Journal of Robotics Research*, 2002: 735-758.

Se, S., H. Ng, P. Jasiobedzki, e T. Moyung. "Vision based Modeling and Localization for Planetary Exploration Rovers." *Proceedings of International Astronautical Congress (IAC)*, 2004.

Smith, R., M. Self, e P. Cheeseman. "A stochastic map for uncertain spatial relationships." *The fourth international symposium*. 1988.

Smith, R., M. Self, e P. Cheeseman. "Estimating uncertain spatial relationships in robotics." *Autonomous robot vehicles*, 1990: 167-193.

Synfora Inc. *Synfora Inc.* <http://www.synfora.com/> (acesso em 30 de 11 de 2007).

The Compiler and Architecture Research Group. *CAR Group*. http://www.trimaran.org/car_group.shtml (acesso em 30 de 11 de 2007).

The Trimaran Consortium. *The Trimaran Compiler Research Infrastructure for Instruction Level Paralelism*. <http://www.trimaran.org> (acesso em 30 de 11 de 2007).

Thrun, S. "Robotic mapping: A survey." Relatório técnico CMU-CS-02-1, School of Computer Science, Carnegie Mellon University, 2002.

Welch, G., e G. Bishop. "An Introduction to the Kalman Filter." University of North Carolina, Julho de 2006.

Wolf, D., J. A. de Holanda, V. Bonato, R. Peron, e E. Marques. "Towards FPGA-based mobile robots." *III Jornadas sobre Sistemas Reconfiguráveis (REC2007)*, 2007: 56-59.

Wolf, D., J. A. M. de Holanda, V. Bonato, R. Peron, e E. Marques. "A FPGA-Based Mobile Robot Controller." *III Southern Conference on Programmable Logic*, 2007, 2007 ed.: 1-10.