DISSERTAÇÃO DE MESTRADO

UNIVERSIDADE FEDERAL DE SÃO CARLOS

CENTRO DE CIÊNCIAS EXATAS E DE TECNOLOGIA

PROGRAMA DE PÓS-GRADUAÇÃO EM

CIÊNCIA DA COMPUTAÇÃO

“LALPC: UMA FERRAMENTA PARA COMPILAÇÃO DE PROGRAMAS EM C PARA EXPLORAÇÃO DO PARALELISMO DE LOOPS EM FPGAS”

ALUNO: Lucas Faria Porto
ORIENTADOR: Prof. Dr. Ricardo Menotti

São Carlos
Fevereiro/2015

CAIXA POSTAL 676
FONE/FAX: (16) 3351-8233
13565-905 - SÃO CARLOS - SP
BRASIL

UNIVERSIDADE FEDERAL DE SÃO CARLOS

CENTRO DE CIÊNCIAS EXATAS E DE TECNOLOGIA

PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

LALPC: UMA FERRAMENTA PARA COMPILAÇÃO DE PROGRAMAS EM C PARA EXPLORAÇÃO DO PARALELISMO DE *LOOPS* EM FPGAS

LUCAS FARIA PORTO

ORIENTADOR: PROF. DR. RICARDO MENOTTI

São Carlos – SP

Fevereiro/2015

Ficha catalográfica elaborada pelo DePT da Biblioteca Comunitária da UFSCar

P853Lf

Porto, Lucas Faria.
LALPC : uma ferramenta para compilação de programas em C para exploração do paralelismo de *loops* em FPGAs / Lucas Faria Porto. -- São Carlos : UFSCar, 2015.
88 f.

Dissertação (Mestrado) -- Universidade Federal de São Carlos, 2015.

1. Compiladores (Computadores). 2. FPGAs. 3. *Loop pipeline*. I. Título.

CDD: 005.453 (20ª)

UNIVERSIDADE FEDERAL DE SÃO CARLOS

Centro de Ciências Exatas e de Tecnologia
Programa de Pós-Graduação em Ciência da Computação

Folha de Aprovação

Assinaturas dos membros da comissão examinadora que avaliou e aprovou a Defesa de Dissertação de Mestrado do candidato Lucas Faria Porto, realizada em 04/02/2015:

Prof. Dr. Ricardo Menotti
UFSCar

Prof. Dr. Edson Borin
UNICAMP

Prof. Dr. Vanderlei Bonato
USP

Dedicado aos meus pais por todo suporte nesta jornada.

AGRADECIMENTOS

Agradeço primeiramente a Deus por ter me dado sabedoria para superar todos os desafios durante esta jornada. Agradeço especialmente aos meus pais, Eloi e Gessiene, sou eternamente grato por me apoiarem em minhas escolhas, por estarem comigo nesta caminhada e principalmente por compreenderem a minha ausência. As minhas irmãs que são um exemplo caráter e determinação para vencer qualquer desafio.

O agradecimento a CAPES e a todos os professores do DC no qual sou grato pelos conhecimentos adquiridos, em especial ao professor Ricardo Menotti, ele foi responsável pelo apoio e orientação que resultou nesta dissertação de mestrado. Aos colegas e companheiros de curso, eles serão sempre lembrados principalmente pela amizade construída; lembrados como uma família, caracterizada pela amizade forte que permanece mesmo com aqueles que por algum motivo mudaram de rumo no meio da jornada.

Algumas pessoas, em especial, que foram importantes nessa jornada em São Carlos: Adam, Faimison,Fernando Antônio, Rayner, Edvar, Elias meu compadre, Guilherme Stefano, Odair, André Abade, Fernando Maranhão, Victor (e toda sua família), Leandro Mundim, Jésus, Aried, Ely, Alan e ao pessoal do Peladeiros.

Muito obrigado a todos.

A persistência é o caminho do êxito.

Charles Chaplin

RESUMO

A limitação física do silício forçou a indústria a desenvolver soluções que explorassem o poder de processamento de combinação de vários processadores de propósito geral. Mesmo os supercomputadores complexos que dispõem de vários processadores, eles ainda são considerados ineficientes para processamentos que exigem grandes quantidades de operações aritméticas utilizando dados em ponto flutuante. A computação reconfigurável vem ganhando cada vez mais espaço por ter um desempenho próximo aos dispositivos de propósito específico (ASIC), e ainda assim, manter a flexibilidade proporcionada pela arquitetura dos processadores de propósito geral. Entretanto, a complexidade das linguagens de descrição de *hardware* se torna muitas vezes uma barreira para o desenvolvimento de novos projetos. Ferramentas de síntese de alto nível vem se popularizando, elas permitem a transformação de códigos em alto nível em *hardware* de maneira simples e rápida. Entretanto, soluções encontradas nas ferramentas atuais, geram *hardware* simples que não exploram as técnicas que permitam melhorar o *pipeline* em *hardware*. Este trabalho apresenta o desenvolvimento de técnicas que permitem explorar o poder do paralelismo nos dispositivos reconfiguráveis por meio de programas descritos em uma linguagem C. Essas técnicas identificam laços de repetição e melhoram o desempenho em *hardware*. Como resultado, temos a melhora no processo de síntese de alto nível gerando *hardware* otimizado.

Palavras-chave: FPGA, Compilador, Loop Pipeline

ABSTRACT

The physical limitations of silicon forced the industry to develop solutions that exploit the processing power of combining several general purpose processors. Even complex supercomputers that have multiple processors, they are still considered to inefficient processes that require large amounts of arithmetic operations using floating point data. Reconfigurable computing is gaining more space to have a performance close to a specific purpose devices (ASIC), and yet keep the flexibility provided by the architecture of general purpose processors. However, the complexity of hardware description languages often becomes a problem to the development of new projects. Tools for high-level synthesis have become more popular, they allow the transformation code in high-level hardware simply and quickly. However, solutions found in current tools generate simple hardware that does not exploit the techniques to improve the pipeline in hardware. This paper presents the development of techniques to exploit processing parallelism of the reconfigurable devices through programs described in language C. These techniques identify loops and improve the performance in hardware. As a result, we have improved in the high-level synthesis process generating optimized hardware.

Keywords: FPGA, Compiler, Loop Pipeline

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ALGORITMOS

GLOSSÁRIO

ALP – *Aggressive Loop Pipelining*

API – *Application Programming Interface*

ASIC – *Application Specific Integrated Circuit*

AST – *Abstract Syntax Tree*

CAD – *Computer-Aided Design*

CHiMPS – *Compiling High level language to Massively Pipelined System*

CPLD – *Complex Programmable Logic Device*

CPU – *Central Processing Unit*

CTL – *CHiMPS Target Language*

CUDA – *Compute Unified Device Architecture*

EDG – *Edison Design Group*

FPGA – *Field-Programmable Gate Array*

GAL – *Generic Array Logic*

GPGPU – *General-Purpose Computing on Graphics Processing Units*

GPU – *Graphics Processing Unit*

HDL – *Hardware Description Language*

HLSL – *High Level Synthesis*

LALPC – *Language for Aggressive Loop Pipelining C*

LALP – *Language for Aggressive Loop Pipelining*

LLVM – *Low Level Virtual Machine*

NUMA – *Non-uniform Memory Access*

OpenCL – *Open Computing Language*

OpenMP – *Open Multi-Processing*

PAL – *Programmable Array Logic*

PRAM – *Parallel Random Access Machine*

RAM – *Random Access Machine*

ROCCC – *Riverside Optimizing Compiler for Configurable Computing*

RTL – *Register-Transfer Level*

SIMD – *Single Instruction Multiple Data*

SPMD – *Single Program, Multiple Data*

SSA – *Static Single Assignment*

ULA – *Unidade Lógica Aritmética*

VHDL – *VHSIC Hardware Description Language*

VLIW – *Very Long Instruction Word*

CONTEÚDO

Capítulo 1

INTRODUÇÃO

Com o passar dos anos, o desenvolvimento de sistemas em *hardware* vem se tornando cada vez mais complexo. De tal forma, a evolução da tecnologia e a necessidade de se produzir dispositivos com diversas funcionalidades tornam os projetos cada vez mais desafiadores. Desenvolver projetos complexos em *hardware* demanda por uma grande quantidade de tempo e testes durante o processo de *design* até se alcançar o resultado desejado.

Para acompanhar a complexidade dos projetos em *hardware* faz-se necessário obter cada vez mais desempenho dos dispositivos. Nas últimas décadas o simples fato de aumentar a frequência de um processador tornou-se uma barreira na evolução dos processadores. Tal fato ocorreu pela limitação física do material usado na construção dos processadores, o silício. Ou seja, aumentar a frequência resultava em grande quantidade de calor e alto consumo de energia. O desafio de se obter mais desempenho fez surgir diversas técnicas de paralelismo. Técnicas como utilização de várias *threads*, múltiplos núcleos em um processador, combinação de processadores, etc. Tais técnicas foram de grande importância no processo de evolução computacional.

Mesmo com a evolução dos processadores e das técnicas de processamento paralelo, havia um grupo de desafios específicos que necessitavam de atenção, problemas contendo milhares de operações algébricas tais como processamento de dados meteorológicos, imagens médicas, criptografia de dados, processamento de imagens, cálculos com dados em ponto flutuante. Esses não tinham desempenhos satisfatórios durante suas execuções nos microprocessadores. O processamento em arquiteturas alternativas surgiu como uma possível solução para tais problemas. Como exemplo, temos o processamento de dados em uma arquitetura de processamento gráfico (GPU[1]) o qual possibilita, muitas das vezes, um desempenho superior, se comparado com o

[1] Do inglês: Graphics Processing Unit

processamento em processadores convencionais (CPUs[2]) (UNDERWOOD, 2004). Outra alternativa é a utilização de circuitos reconfiguráveis, mais especificamente os FPGAs[3] para execução de algoritmos em *hardware*. Os resultados de processamentos obtidos em algumas aplicações utilizando estes dispositivos tem um desempenho claramente superior quando comparado com os CPUs (UNDERWOOD; HEMMERT, 2004; EL-GHAZAWI et al., 2008).

1.1 Motivação

Estudos comprovam a eficiência da utilização de GPUs e FPGAs para executarem tarefas de processamento (COPE et al., 2006; KIRK, 2007). A crescente aceitação de tais arquiteturas disponibilizam uma grande oportunidade de atuação na área para os desenvolvedores de *software*. Porém, desenvolvê-los para arquiteturas de GPUs e FPGAs tornou-se um grande desafio pela grande complexidade exigida das linguagens e principalmente por problemas relacionados às técnicas de processamento paralelo. Logo, é necessário conhecimento avançado de programação paralela para que se utilize todo o poder de processamento disponível no *hardware*.

Diversos problemas computacionais estão resolvidos e implementados em linguagens de alto nível como C e Java. Neste caso, a necessidade de se utilizar um determinado código em um projeto não esbarra em dificuldades de reimplementação, tornando os projetos extremamente reutilizáveis. Entretanto, para sistemas em baixo nível, utilizar algoritmos descritos em linguagens de alto nível demanda conhecimento específico da plataforma e grande trabalho de reescrita do código para a arquitetura alvo.

A computação reconfigurável, mais especificamente o uso dos FPGAs modernos, tem oferecido um enorme potencial para o desenvolvimento de sistemas altamente paralelos e com baixo consumo de energia. A presença de *hardware* e *software*, caracterizada pelas arquiteturas heterogêneas, permite acelerar seções críticas em *hardware*, elevando o desempenho, enquanto mantém um alto nível de produtividade no desenvolvimento já que o restante do sistema é projetado em *software*, muitas vezes já disponível para a aplicação desejada (LIAO et al., 2010b).

As ferramentas de síntese de alto nível permitem facilitar o desenvolvimento de projetos de baixo nível. As técnicas de compilação possibilitaram a transformação automática de códigos de alto nível em *hardware*. Desta forma, foi possível um ganho de tempo significativo durante o processo de desenvolvimento dos projetos, além da utilização de todo conteúdo já implementado existente nas linguagens de alto nível.

[2]Do inglês: Central Processing Unit
[3]Do inglês: Field-Programmable Gate Array

Linguagens específicas para síntese de alto nível surgiram com o propósito de facilitar a geração de sistemas em *hardware*. Como exemplo, neste trabalho, temos a linguagem LALP[4] (MENOTTI, 2011), cuja característica principal é a de utilizar uma linguagem com um nível de abstração próximo ao da linguagem C, permitindo mapeamento de características do *hardware*. Entretanto, por se tratar de uma linguagem específica, LALP obriga o desenvolvedor a aprender uma nova linguagem.

A adoção de GPUs e FPGAs como meios de exploração da computação de alta performance é de grande aceitação, pois ambas permitem um ganho em processamento superior às CPUs (MUNSHI, 2008; BADAL; BADANO, 2009). Porém, a arquitetura baseada em FPGAs tem vantagens quando é comparada com outras arquiteturas. A flexibilidade é uma das grandes vantagens dos FPGAs em comparação com a arquitetura ASICs[5], neste caso, os FPGAs permitem efetuar correções e atualizações no projeto sempre que necessário, o mesmo é impossível na arquitetura ASICs. Outra vantagem dos FPGAs é a portabilidade do código, neste caso, o programador gera o código em uma linguagem de descrição de *hardware* e pode executá-lo em qualquer dispositivo cujo fabricante suporte aquela linguagem. Já no caso das GPUs, a aplicação desenvolvida precisa ser adequada pelo programador para a arquitetura, ou seja, caso haja a necessidade de troca da GPU, o programador deve modificar o código para atender o novo dispositivo (LIU; ZHANG; SHEN, 2009). Além disso, os FPGAs consomem uma quantidade de energia muito menor em relação às GPUs.

Em alguns tipos de aplicações, especialmente as baseadas em ponto flutuante, o desempenho dos GPUs é ligeiramente melhor ou está muito próximo de um FPGA (THOMAS; HOWES; LUK, 2009; FOWERS et al., 2012). Entretanto, quando o assunto está relacionado com a eficiência energética (desempenho por watt), CPUs e GPUs estão a uma distância considerável dos FPGAs, como mostra a literatura disponível que compara o desempenho de CPUs, GPUs, e FPGAs em diferentes aplicações (BACON; RABBAH; SHUKLA, 2013). Atualmente os dispositívos híbridos são apresentados como uma tendência na computação de alto desempenho. Combinar *software*/*hardware* permite gerar soluções heterogêneas quando existem dificuldades para se implementar a solução inteiramente em *hardware*. Neste contexto servidores aceitam o acoplamento de FPGAs via PICe ou interfaces proprietárias, que são utilizados como aceleradores para processar regiões críticas do código (PUTNAM et al., 2014; VANDERBAUWHEDE; BENKRID, 2013). Já em sistemas embarcados formetente acoplados é comum encontrar a combinação de FPGAs com um processador no mesmo dispositivo. Para este caso podemos citar os dispositivos da família Zynq da Xilinx (SANTARINI, 2011) e o kit DE2i-150 da Altera/Intel (Terasic

[4]Do inglês: Language for Aggressive Loop Pipelining
[5]Do inglês: Application Specific Integrated Circuit

Technologies Inc., 2013).

Ainda que sistemas híbridos ofereçam um bom desempenho em termos de paralelismo, estes ainda não são largamente utilizados. Neste contexo a principal razão deste cenário está na dificuldade de se projetar *hardware* e integrá-lo ao *software* (CONG et al., 2011). Afim de amenizar esta dificuldade, as ferramentas de síntese de alto nível propõem facilitar a geração automática de circuitos utilizando descrições mais abstratas. Além do ganho em produtividade o processo de geração automática de *hardware* também auxilía na exploração de espaço de projetos, reúso de componentes, validação do projeto, testes, entre outras etapas. No decorrer deste trabalho são expostas características da arquitetura dos FPGAs, e principalmente a importância das ferramentas de síntese de alto nível. Em geral, projetos resultantes de uma ferramenta de síntese de alto nível ainda apresentam um desempenho inferior quando comparados com versões desenvolvidas por um projetista de *hardware*. Tal deficiência nesta área de pesquisa revela a oportunidade de se desenvolver novas técnicas para alcançar melhores resultados.

1.2 Objetivo

O objetivo deste trabalho de mestrado foi o desenvolvimento de técnicas para síntese de alto nível, as quais permitiram o mapeamento eficiente de *loops* descritos em linguagem C para arquiteturas reconfiguráveis. A utilização de diretivas de marcação por meio dos *pragmas* obteve um avanço relacionado à exploração de espaço de projeto. O resultado de tais técnicas permitiram alcançar um alto nível de paralelismo, consequentemente, melhorar o *pipeline* em *hardware*.

1.3 Justificativa

Soluções em síntese de alto nível são propostas desde os meados dos anos 90. Após mais de 20 anos, finalmente começam a surgir ferramentas mais consistentes, que passaram a ser quase uma necessidade para a indústria. Entretanto, mesmo com os avanços da área, as ferramentas ainda não alcançam o mesmo desempenho que um programador experiente.

Para alcançar o objetivo proposto neste trabalho, foram utilizadas as técnicas implementadas no *backend* da ferramenta *Language for Aggressive Loop Pipelining* (LALP) e também, técnicas implementadas no *frontend* do compilador ROSE. O desenvolvimento das técnicas permitiu efetuar vários testes com diversas ferramentas semelhantes, a fim de comparar o desem-

penho relacionado a otimizações. As arquiteturas obtidas com LALPC[6] alcançaram *speedups* significativos, estes serviram como prova da eficiência das técnicas desenvolvidas.

1.4 Organização

O restante da presente dissertação de mestrado está dividido da seguinte forma: No Capítulo 2 são abordados os conceitos fundamentais e características importantes das arquiteturas que exploram o paralelismo em *hardware*. No Capítulo 3 há uma contextualização do problema deste trabalho, destacando técnicas e trabalhos relacionados. No Capítulo 4 são descritas informações sobre compiladores *source-to-source* e a importância destes no desenvolvimento das técnicas propostas neste trabalho. No Capítulo 5 há uma contextualização do desenvolvimento do trabalho proposto, neste capítulo são descritos as técnicas desenvolvidas e as principais características existentes no compilador LALPC. No Capítulo 6 são apresentados os resultados obtidos neste trabalho. Os resultados são comparados com os dos compiladores LALP e LegUp para confrontar o desempenho das arquiteturas geradas pelo compilador LALPC. No Capítulo 7 é descrita a conclusão deste trabalho e apresentam-se algumas pesquisas que podem ser exploradas com o desenvolvimento do compilador LALPC futuramente.

[6]Do inglês: Language for Aggressive Loop Pipelining C

Capítulo 2

ARQUITETURAS PARALELAS

Atualmente, a maioria das solucões em computação são baseados em dois paradigmas distintos. Processadores de propósito geral são usados para resolver problemas diversos, alterando-se somente seu *software*. Soluções baseadas em ASICs[1] permitem desenvolver circuitos integrados para processar uma tarefa específica. Esta tarefa pode ser uma aplicação que executa uma gama de cálculos definidos, consequentemente, eles conseguem atingir um alto grau de eficiência e velocidade ao executar os cálculos que foram programados (COMPTON; HAUCK, 2002; BACON; GRAHAM; SHARP, 1994).

A utilização de ASIC possibilita o ganho de desempenho com menos recursos de *hardware*, isso só é possível graças às técnicas de desenvolvimento, unidades funcionais personalizadas e principalmente, otimização no circuito para uma determinada aplicação. Porém, os custos para se projetar e implementar um sistema em ASIC são elevados, podendo ser justificados em situações de produções em larga escala. Com a produção de circuitos integrado em massa, o custo inicial de pesquisa e projeto é amortizado em cada pastilha de silício produzida, consequentemente, viabilizando sua produção (SKLIAROVA; FERRARI, 2003).

A falta de flexibilidade dificulta efetuar qualquer modificação após a fabricação do circuito. Neste caso, se houver a necessidade, mesmo que seja pequena, é preciso fazer as modificações no projeto e efetuar a fabricação de novos chips para atender tais customizações. Este processo é caro principalmente pelo fato da dificuldade da substituição de todos os circuitos onde foram implantados.

[1] Do inglês: Application Specific Integrated Circuit

2.1 Dispositivos Reconfiguráveis

Dispositivos reconfiguráveis podem ser programados para terem o funcionamento de um circuito lógico em *hardware*, sendo assim, a computação reconfigurável tem o objetivo de preencher o espaço entre o método baseado em *hardware* e *software*. Os métodos usados na computação reconfigurável visam a flexibilidade de desenvolvimento, como no modelo baseado em *software*, enquanto buscam alcançar o desempenho do modelo baseado em *hardware* (COMPTON; HAUCK, 2002).

Diversos tipos de dispositivos fazem parte da computação reconfigurável, sendo eles: GAL[2], PAL[3], CPLD[4] e FPGA[5]. Neste contexto, temos como destaque o FPGA, dentre os citados anteriormente. Este é o que permite maior flexibilidade e configuração de aplicações com alta complexidade. Isto só é possível pois os FPGAs são circuitos integrados contendo milhares de unidades lógicas idênticas, tais unidades são componentes que têm a capacidade de serem configurados de maneira independente e conectadas umas às outras a partir de trilhas condutoras que também têm a capacidade de configuração.

Essencialmente, a arquitetura de um FPGA é composta por três tipos de componentes básicas: recursos de entrada/saída, blocos lógicos e canais de roteamento/interconexões. Os blocos lógicos ou unidades lógicas são configuráveis e estão organizados de forma similar a uma matriz bidimensional, sendo os blocos, conectados pelos canais de roteamento que também são configuráveis, tais características citadas anteriormente são visíveis na Figura 2.1. Por fim, os blocos de entrada e saída têm como objetivo efetuar a comunicação do FPGA com o meio externo (GONSALES, 2002).

A estrutura interna de um FPGA difere de modelo para modelo, são variações na quantidade de blocos de entrada/saída, interconexões e principalmente blocos lógicos com características diferentes; no caso dos blocos lógicos, variações principalmente em sua granularidade. O contexto de granularidade abrange várias características internas do dispositivo, tais como: o número de portas NANDs, a quantidade de implementações de funções possíveis dentro do bloco lógico, a quantidade de entradas e saídas e quantidade de transistores do bloco. Com estas diferenças de arquiteturas, um bloco lógico pode ser simples como no caso dos FPGAs *Crosspoint*, que contém um par de transistores, ou com alto grau de complexidade como o de um processador (GONSALES, 2002).

[2]Do inglês: Generic Array Logic
[3]Do inglês: Programmable Array Logic
[4]Do inglês: Complex Programmable Logic Device
[5]Do inglês: Field-Programmable Gate Array

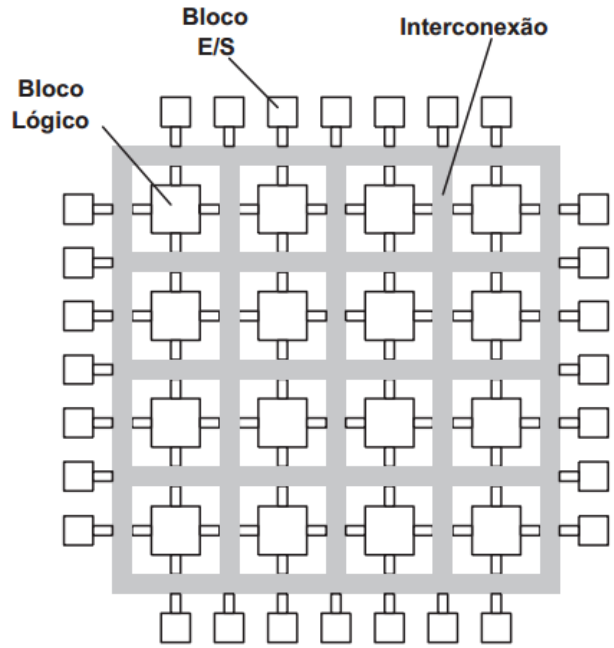
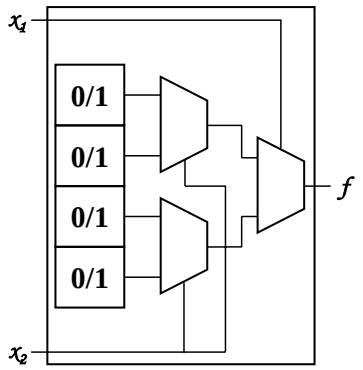


Figura 2.1: Arquitetura padrão de um FPGA.

A utilização de LUT[6] permite implementar os blocos citados anteriormente. Neste modelo, as funções lógicas são implementadas em memórias, contendo as entradas, multiplexadores e uma saída. Na Figura 2.2 (a) é possível visualizar a estrutura de uma LUT de duas entradas.

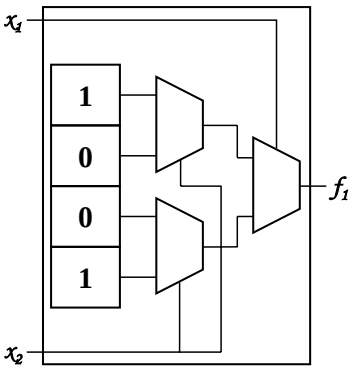
O funcionamento das LUTs se baseia em tabelas de busca, essas tabelas são utilizadas para gerar uma saída de acordo com os sinais de entrada dos blocos lógicos. Na Figura 2.2 (c) é possível visualizar a LUT configurada para implementar a função da Figura 2.2 (b).



(a) Circuito de uma LUT

x_1	x_2	f_1
0	0	1
0	1	0
1	0	0
1	1	1

(b) $f_1 = \bar{x}_1\bar{x}_2 + x_1x_2$



(c) Conteúdo das células

Figura 2.2: Exemplo de utilização de LUTs para implementação de funções lógicas (MENOTTI, 2010).

De acordo com Compton e Hauck (2002), cada bloco lógico varia de sistema para sistema,

[6]Do inglês: Look-Up Table

sendo assim, os mesmos podem ser simples quando implementados em uma tabela LUT de três entradas ou complexos quando implementados em uma ULA. Esta diferença entre os blocos é que define a granularidade da arquitetura, neste caso, uma LUT é um bloco de granularidade fina e a ULA um bloco de granularidade grossa.

Os blocos lógicos de granularidade fina têm como característica principal sua eficiência na utilização, outra vantagem é a facilidade para a ferramenta CAD[7] durante a fase de mapeamento dos blocos. Porém, uma implementação de um dispositivo de granularidade fina necessita de uma quantidade muito maior de segmentos de conexões e chaves programáveis durante o roteamento, aumentando a área e consequentemente o atraso.

Em um comparativo, os FPGAs de granularidade fina são mais lentos e possuem uma densidade maior que os FPGAs de granularidade grossa, isto ocorre pela grande quantidade de conexões entre os pequenos blocos. Por outro lado, os blocos lógicos baseados em granularidade grossa também utilizam multiplexadores em sua implementação, neste caso em maior quantidade; isto proporciona implementar mais funcionalidades para um número pequeno de blocos lógicos. Porém, estes FPGAs demandam de mais recursos de roteamento pela grande quantidade de pinos de entrada (GONSALES, 2002). Alguns sistemas utilizam modelos de blocos lógicos híbridos, neste caso, uma arquitetura heterogênea consegue utilizar as vantagens da granularidade fina e da granularidade grossa ao mesmo tempo (COMPTON; HAUCK, 2002).

Dentro de um *hardware* reconfigurável, o roteamento das conexões entre os blocos lógicos é de grande importância, tal função contribui significativamente para definir a área total do *hardware* reconfigurável. Porém, em um projeto grande, que utiliza uma quantidade elevada de blocos lógicos, as ferramentas de síntese que geram automaticamente o roteamento têm uma grande dificuldade de gerar as ligações necessárias entre os componentes lógicos.

A quantidade de caminhos necessários e de interconexões não cresce de maneira linear acompanhando o aumento na quantidade dos blocos lógicos, isto é, arquiteturas maiores requerem quantidades ainda maiores de encaminhamentos entre os blocos lógicos que outras arquiteturas menores.

Gerar boas interconexões entre os blocos lógicos é fundamental para garantir o sucesso e desempenho dos projetos aplicados nos FPGAs. Isto é justificado pelo fato de que, em uma pastilha de FPGA, os recursos de encaminhamento e interconexões ocupam uma área muito maior quando se comparado com os recursos relacionados à parte lógica. Portanto, arquiteturas mais eficientes são aquelas que conseguem ter uma melhor otimização dos recursos de roteamento (COMPTON; HAUCK, 2002).

[7] Do inglês:Computer-Aided Design

2.1.1 Arquiteturas Heterogêneas

As arquiteturas heterogêneas têm como característica principal o uso de duas ou mais tecnologias distintas. Frequentemente, uma arquitetura heterogênea adota a utilização de um microprocessador de propósito geral junto com outra arquitetura, por exemplo os FPGAs ou GPUs[8] (COMPTON; HAUCK, 2002). Em uma arquitetura heterogênea, formada por um microprocessador e um FPGA, o microprocessador tem como papel executar e gerenciar a lógica do programa; principalmente verificar a necessidade e delegando parte do código para ser executado no dispositivo reconfigurável que age como um coprocessador para um microprocessador central.

Nos sistemas híbridos, a parte reconfigurável pode ser constituída por um ou mais FPGAs ou, em outros casos, pode ser um dispositivo desenvolvido especificamente para atuar como um sistema reconfigurável. Após a finalização da síntese do circuito para o *hardware* reconfigurável, ele está pronto para ser utilizado pelo microprocessador durante a execução do programa.

A operação de síntese do sistema reconfigurável em tempo de execução ocorre em duas fases, são elas, configuração e execução. A operação de configuração está sob o controle do microprocessador. Este direciona o fluxo com os dados de configuração para o *hardware* reconfigurável, definindo o funcionamento do *hardware*. As configurações podem ser carregadas durante a inicialização de um programa ou a cada intervalo de tempo durante a execução, essas condições variam de sistema para sistema.

Sistemas baseados em computação reconfigurável, normalmente utilizam FPGAs ou outra arquitetura baseada em *hardware* programável para acelerar a execução de algoritmos específicos que são mapeados e executados na parte reconfigurável. Quando o FPGA está configurado com alguma função, esse executa as tarefas independentemente do microprocessador e retorna os dados após a conclusão. Esse funcionamento permite à unidade reconfigurável trabalhar por vários ciclos sem a intervenção da unidade principal, neste caso o microprocessador. Essa capacidade garante um grande desempenho das arquiteturas heterogêneas (COMPTON; HAUCK, 2002).

2.1.2 Programabilidade

Os dispositivos baseados em arquiteturas reconfiguráveis, como os FPGAs, são projetados de maneira semelhante aos dispositivos ASICs, neste caso, podem ser projetados com base em modelos esquemáticos ou através da programação usando linguagens de descrição de *hard-*

[8] Do inglês: Graphics Processing Unit

ware(HDL[9]), tais como VHDL[10] e Verilog (MENOTTI, 2010).

Conforme d'Amore (2005), VHDL é uma das linguagens de descrição de *hardware* mais utilizadas, que permite a implementação de circuitos em nível de portas lógicas, bem como a descrição de algoritmos em nível mais alto de abstração. De acordo com Che et al. (2008), Wain et al. (2006), VHDL possibilita a geração de *hardware* eficiente, porém, a mesma exige conhecimento avançado da linguagem.

2.2 Multi/Many-cores

Nos anos 80, em paralelo com a evolução dos processadores, os fabricantes de *hardware* desenvolveram computadores com arquiteturas que exploravam técnicas de paralelismo. A ideia se baseava em computadores com vários processadores que compartilhavam a memória entre eles, visando a um alto poder de processamento.

2.2.1 Tipos de Execução Paralela

De acordo com Kessler e Keller (2007) os estilos de execução paralela são responsáveis por definir diversas maneiras em que atividades paralelas (processos, threads, etc.) são executadas, isto é, definir no código o início e o fim da execução paralela. Em questão, são abordados dois estilos de execução paralela: *fork-join* e o estilo SPMD[11].

Na abordagem *Fork-Join* o programa gera atividades paralelas dinamicamente em determinados trechos do código que marcam o início (*fork*) e o fim (*join*) do paralelismo. No início e no fim da execução do programa existe apenas uma atividade em execução, entretanto durante esse processo, a quantidade de atividades paralelas pode variar consideravelmente dependendo dos recursos de *hardware* disponíveis naquele momento. O mapeamento das atividades paralelas para os processadores físicos pode ser gerenciado em tempo de execução pelo sistema operacional, por gerenciamento de pacotes de *threads* ou pelo gerenciador da linguagem de programação.

No estilo SPMD é definido no início do programa um número N de atividades paralelas, podendo usar processadores físicos ou virtuais, até o fim da execução sem alterar a quantidade de atividades paralelas durante a execução. Nesse estilo o programador é responsável por definir no código em quais dos processadores disponíveis vão ser executadas as tarefas paralelas,

[9] Do inglês: Hardware Description Language
[10] Do inglês: VHSIC Hardware Description Language
[11] Do inglês: Single Program, Multiple Data

sendo assim, possibilita ao programador gerenciar o balanceamento de carga nos processadores, enquanto no estilo *fork-join* essa tarefa é feita de maneira automática sem a intervenção do programador. Tal característica pode ser considerada complexa para o programador, pois ao mesmo tempo que é possível a redução na sobrecarga nos processadores e no tempo de execução, uma decisão errada no balanceamento dos processadores pode gerar perda de desempenho e gerar uma sobrecarga em um determinado processador (KESSLER; KELLER, 2007).

2.2.2 Paralelismo em Nível de Instrução

A necessidade de alto desempenho nos computadores atuais resulta em arquiteturas altamente complexas. Essas arquiteturas são construídas com a combinação de vários componentes lógicos, ou unidades funcionais, que têm a capacidade de funcionamento simultâneo com tarefas específicas. Como exemplo, estas permitem realizar operações aritméticas de dois números e/ou operações lógicas. O resultado é um computador capaz de buscar dados na memória, efetuar a operação aritmética, avaliar se o resultado é maior que outro valor, e tudo isso ao mesmo tempo. Esse processamento paralelo em baixo nível de execução é conhecido como "paralelismo em nível de instrução". Tais características são encontradas nas arquitetura superescalares, arquiteturas VLIW[12] e também, atualmente, são comuns nos microprocessadores de propósito geral encontrados nos computadores pessoais.

De acordo com Chapman, Jost e Pas (2007), a ideia é reordenar as operações em nível de instrução para ocupar o processador, tendo em vista não deixá-lo ocioso. Na arquitetura VLIW, o compilador é responsável pela maior parte do trabalho para encontrar tal ordenação das operações. Este processo utiliza técnicas para identificar dependências entre as operações para possibilitar uma ordenação eficiente mantendo as unidades lógicas do processador sempre em operação. Nas arquiteturas superescalares, o paralelismo em nível de instrução é detectado dinamicamente em baixo nível, com pouca ou quase nenhuma influência do compilador. Assim, as instruções são executadas fora de ordem e posteriormente os resultados são identificados e ordenados para manter a semântica do programa.

Os compiladores modernos realizam um esforço considerável para efetuar a reordenação de instruções. Por exemplo, a técnica de *loop pipelining* que modifica a sequência das instruções em um *loop*. Neste exemplo, o compilador pode sobrepor várias instruções diferentes para garantir a execução do maior número de instruções no mesmo ciclo de *clock*. Entretanto, vários estudos mostram que normalmente o compilador não consegue sobrepor mais do que quatro instruções devido a dependência entre as mesmas, assim, não conseguindo obter uma melhora

[12] Do inglês: Very Long Instruction Word

significativa de desempenho na maioria das aplicações (CHAPMAN; JOST; PAS, 2007).

2.2.3 Memória Distribuída

Um outro modelo de programação diferente foi proposto, baseado em sistema de memória distribuída. Conhecido como modelo de "transmissão de mensagens" ou "*message-passing*". Neste modelo os programas são executados por um ou mais processos, no qual cada um tem o seu próprio espaço de memória privado. Este modelo de programação paralela visa à realização de operações através de envio e recebimento de mensagens que possibilita as trocas de dados entre os processos paralelos, o modelo *message-passing* gerencia os processos que necessitam de uma cooperação entre si, neste caso, as troca de mensagens vão ocorrer quando um determinado processo solicita um determinado dado produzido por outro. Na Figura 2.3 é possível observar a estrutura do modelo de memória distribuída, nele temos os módulos de memórias (M) ligados nos módulos de processamento (P).

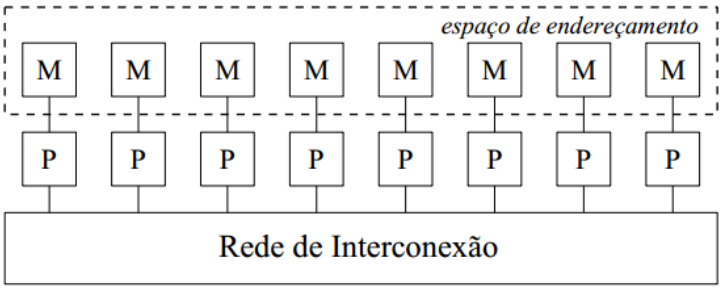


Figura 2.3: Estrutura do modelo baseado em memória distribuída (ROSE; NAVAUX, 2003).

2.2.4 Memória Compartilhada

No paralelismo de memória compartilhada ou multiprocessadores, o computador trabalha com vários processos ao mesmo tempo ou que um processo possa ser dividido, e que cada parte possa ser executada em cada unidade de processamento simultaneamente. Tal abordagem tornou-se popular entre os servidores devido ao desempenho obtido. Permitindo aumentar significativamente a quantidade de processadores e de memória compartilhada, o desenvolvimento e gerenciamento é o mesmo quando se compara com o sistema que contém apenas dois ou quatro processadores, garantindo escalabilidade. Na Figura 2.4 é possível observar a estrutura do modelo de memória compartilhada, neste temos apenas um módulo de memória (M) compartilhado por vários módulos de processamento (P).

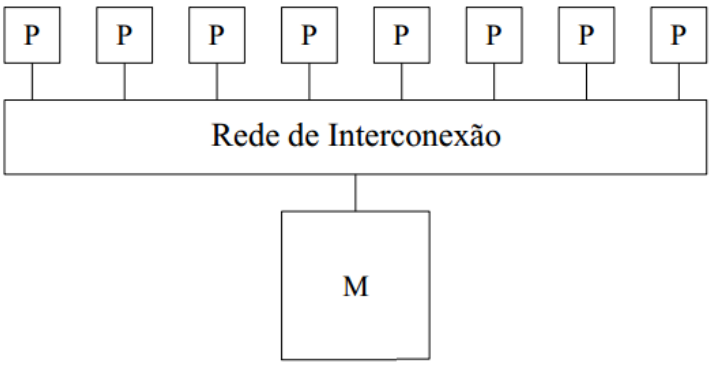


Figura 2.4: Estrutura do modelo baseado em memória compartilhada (ROSE; NAVAUX, 2003).

O modelo PRAM[13] surgiu como proposta de ser uma simples extensão do modelo RAM[14] utilizado em projetos com programação sequencial. O modelo PRAM baseia-se em um conjunto de processadores com acesso a uma memória compartilhada, também há um *clock* global que é usado tanto nos processadores quanto na memória. O tempo de execução das instruções é sempre constante (incluindo instrução de acesso à memória) independente do processador ou qual é o endereço de memória acessado. Outra característica importante é que neste modelo não existe uma limitação para a quantidade de processadores acessando simultaneamente a memória compartilhada (KESSLER; KELLER, 2007).

No modelo com memória compartilhada, sua característica é o acesso em uma memória comum pelas diversas *threads* em execução. Nesse contexto, esse modelo se assemelha ao modelo PRAM. Porém, todas as *threads* de processos são executadas de maneira assíncrona. Sendo assim, o programador deve prever e resolver todos os possíveis conflitos entre as *threads* durante o acesso à memória compartilhada. Outra diferença notável com o modelo PRAM está relacionado à visibilidade de gravações na memória. No modelo PRAM, cada gravação na memória compartilhada era visível para todas as *threads* em execução, chamada de "consistência rigorosa". Com o propósito de gerar uma implementação simples e eficaz, foi desenvolvido um modelo com consistência mais fraca, deixando a responsabilidade da execução correta para o programador.

O custo de soluções usando o modelo baseado em memória compartilhada depende da finalidade da aplicação e do *hardware*. O uso de vários processadores acessando a memória física compartilhada através de um barramento tem um custo semelhante ao modelo RAM, contando com as modificações relacionadas à memória cache, e considerando o custo de sincronização. Essas características são chamadas de multiprocessamento simétrico, pois tanto o tempo no me-

[13]Do inglês: Parallel Random Access Machine
[14]Do inglês: Random Access Machine

canismo de acesso e o tempo de acesso na memória são os mesmos, independente do endereço e do acesso do processador.

Entretanto, se houver apenas uma região da memória que está compartilhada (ocorre no modelo de memória distribuída), o custo de acesso à memória compartilhada está relacionada diretamente com a distância entre a memória compartilhada e o processador. Isto é chamado de NUMA[15], logo o tempo de acesso na cache, memória local ou em uma memória externa podem variar consideravelmente refletindo no custo final.

Atualmente a tecnologia utilizada para interligar os processadores e a memória melhorou significativamente em relação à década de 80. O aprimoramento de técnicas de desenvolvimento de *hardware* com ênfase em paralelismo tem grande importância na computação atual.

A nova geração de computadores que usam memória compartilhada tem preço acessível e destina-se ao uso geral. Existem modelos com processador que têm a capacidade de executar vários fluxos de instrução de maneira intercalada. Esta técnica chamada de *Multithreading* permite a intercalação das instruções de várias aplicações visando utilizar todos os componentes lógicos em qualquer momento. Um exemplo é a tecnologia *hyperthreading* da Intel. Outras plataformas possibilitam colocar vários núcleos de processamento em um único chip, onde cada um comporta-se como uma máquina com memória compartilhada de maneira independente, técnica conhecida como *multicore*.

As arquiteturas com *multithreading*, *multicore* e computadores paralelos com compartilhamento de memória oferecem suporte para execução simultânea de instruções. Outra característica interessante é a possibilidade de combinar essas tecnologias para conseguir aumentar ainda mais o nível de paralelismo e consequentemente o desempenho (CHAPMAN; JOST; PAS, 2007).

2.2.5 Soluções com Memória Compartilhada

2.2.5.1 Pthreads

A API *POSIX threads* (Pthreads) é popular, pois fornece ao programador um ambiente de programação paralela usando o sistema de memória compartilhada através da criação e manipulação das *threads*. Os programas *Pthreads* são desenvolvidos como uma biblioteca estendendo uma linguagem de programação sequencial, como C e Fortran, neste caso o compilador consegue identificar apenas um processador para executar o código, consequentemente não possuem um modelo de memória bem definido. Isso acontece, pois os programas *Pthreads* herdam a consistência de memória do *hardware* onde está sendo executado.

[15]Do inglês: Non-uniform Memory Access

O desenvolvimento de programas paralelos usando *Pthreads* exige muito do programador, isto ocorre pelo baixo nível de abstração da ferramenta. O programador deve especificar no código todo o controle e funcionamento das *threads*, o que exige dele um alto nível de conhecimento da ferramenta e do modelo de programação paralela.

2.2.5.2 OpenMP

O OpenMP é uma API de desenvolvimento baseada em memória compartilhada, de acordo com Kasim et al. (2008), o OpenMP permite a criação de programas utilizando paralelismo através da técnica de compartilhamento de memória. Os autores Chapman, Jost e Pas (2007) deixam claro que o ponto forte desta API é uma interface amigável, eficiente, portátil para qualquer plataforma/SO e, principalmente, facilita para o programador no desenvolvimento de *software*, permitindo gerar aplicações que exploram cada vez mais técnicas de paralelismo. O OpenMP é destinado para o desenvolvimento de aplicações para arquiteturas baseadas em compartilhamento de memória, independente se é uma arquitetura multicore e/ou com *multithreading*, estas atualmente estão bem disseminados no mercado.

O OpenMP utiliza linguagens sequenciais, tais como C e C++, para implementar o código da aplicação. Entretanto, o programador descreve quais as regiões no código serão paralelizáveis e processadas em vários núcleos. Ao adicionar recursos relacionados ao OpenMP em um programa sequencial, este permite que a maioria dos códigos sejam beneficiados com o processamento paralelo. Em muitos casos, pequenas modificações no código resultam em um aumento significativo no desempenho da aplicação.

Um dos fatores do sucesso do OpenMP pode ser atribuído ao fato de que o programador desenvolve o código de maneira estruturada e apenas define quais regiões serão paralelizadas. Tal maneira de programar facilita o desenvolvimento da aplicação, pois o processamento é efetuado de maneira transparente sem a necessidade de o programador ter um conhecimento técnico avançado sobre programação paralela. Outro ponto forte desta API é que a aplicação gerada pelo OpenMP pode ser executada em várias plataformas diferentes, seja de *hardware* e *software*; e novamente, sem a necessidade de o programador ter que adaptar o código quando modifica a plataforma (KESSLER; KELLER, 2007).

Programas que utilizam memória compartilhada normalmente são executados por vários segmentos independentes, tais segmentos gerenciam o estado de execução do programa e também os dados compartilhados. O modelo de programação paralela baseada em compartilhamento de memória deve fornecer recursos de gerenciamento de dados, de memória, coordenar acessos às informações, identificar e gerenciar as operações nos segmentos em tempo de exe-

cução. Tais operações são necessárias para garantir a integridade da semântica do programa escrito de maneira estruturada (CHAPMAN; JOST; PAS, 2007).

A chegada e a disseminação dos processadores multicores têm favorecido a popularidade do OpenMP podendo vir a substituir a Pthreads. O OpenMP disponibiliza para o usuário paralelismo estruturado através da combinação dos estilos SPMD e *fork-join*. Uma das características importantes no OpenMP é a possibilidade de gerar códigos que permitem o compartilhamento de uma determinada tarefa, por exemplo, distribuir as iterações de um laço de repetição de acordo com várias regras de escalonamento pré-definidas na API.

2.3 GPU

Desde os primórdios da computação, os computadores trabalhavam com processadores contendo apenas um núcleo, executando códigos de maneira sequencial. Com a evolução e necessidade de se explorar mais velocidade em processamento, as empresas investiram recursos em pesquisas, desenvolvendo processadores mais densos e com frequências cada vez mais altas chegando no limite físico do silício (ZANOTTO; FERREIRA; MATSUMOTO, 2012). Para contornar o problema da barreira física do silício, a indústria desenvolveu novas alternativas, as quais utilizam conceitos de *threads*, *multicores* entre outras, para possibilitar o aumento do poder de processamento sem a necessidade de aumentar a frequência do processador. Utilizar vários núcleos em um único processador ou processadores em paralelo surgiu como mecanismo alternativo para o aumento do poder de processamento. Entretanto, para alcançar o desempenho desses tipos de processamento é necessário adaptar os códigos sequenciais para explorar ao máximo o paralelismo.

Com os novos métodos de processamento específico, a empresa de fabricação de placas de vídeo, a NVIDIA, em 1999, disponibilizou no mercado sua primeira unidade de processamento gráfico ou *Graphics Processing Unit* (GPU), neste caso o modelo GeForce 256 (NVIDIA CORPORATION, 2007). No início, a construção da GPU era baseada em um pipeline gráfico com exclusividade em processamento de imagens tridimensionais, sendo assim, um processador de função fixa. Com o passar do tempo, a GPU teve uma evolução no *hardware* e no final do ano 2000, ocorreu o lançamento da GeForce 3 que marcou o início da técnicas de programação nas GPUs, conhecidas como GPGPU[16].

As técnicas usando GPGPUs para processamento de imagens, permitiram explorar o grande poder de processamento aritmético proporcionado pela arquitetura em algoritmos fora do es-

[16]Do inglês: General-Purpose Computing on Graphics Processing Units

copo de imagens (ZANOTTO; FERREIRA; MATSUMOTO, 2012); A Figura 2.5 possibilita a visualização das diferenças entre as arquiteturas utilizadas em CPUs[17] e GPUs. Para explorar o poder de processamento desta arquitetura, temos as linguagens de programação voltadas para a arquitetura GPU, são elas: CUDA, *DirectCompute*, OpenCL, entre outras (JACOB et al., 2010). Algumas delas terão características abordadas no decorrer deste trabalho.

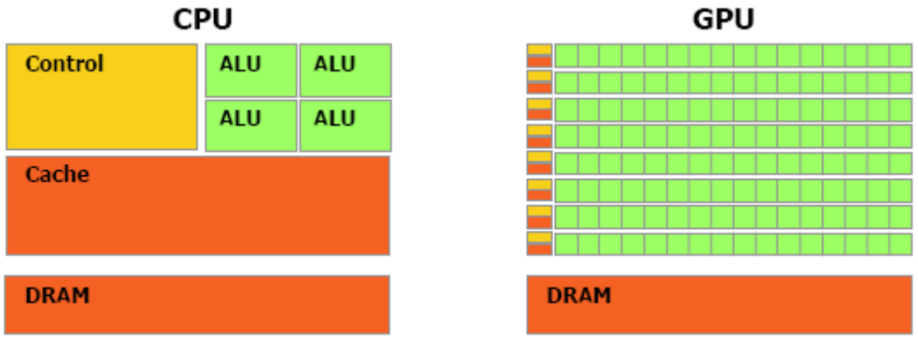


Figura 2.5: Diferença entre as arquiteturas da CPU e GPU (JR et al., 2008).

Atualmente a GPU é um processador dedicado, sua arquitetura é baseada em processamento gráfico da classe SIMD[18] possui grande capacidade de programação, alta taxa de processamento paralelo e desempenho nos cálculos em grandes volumes de dados, resultando em um grande *throughput*. Tais características são resultantes da arquitetura específica para resolver cálculos de problemas em ponto flutuante (ZANOTTO; FERREIRA; MATSUMOTO, 2012).

Com capacidade de grande processamento de funções em ponto flutuante, as GPUs têm se destacado principalmente pelo desempenho superior quando se comparado com a arquitetura baseada em CPU *multicores*. A diferença de desempenho das duas arquiteturas de processadores se justifica pelo fato de que as CPUs foram projetadas para otimizar o desempenho de programas descritos sequencialmente, enquanto a GPU tem a capacidade de executar cálculos de diversas operações em ponto flutuante, utilizando a execução de várias *threads*, podendo executar várias tarefas simultaneamente dependendo do algoritmo (TSUDA, 2012).

De acordo com Che et al. (2008), Jr et al. (2008), Wu e Liu (2008), o grande poder de processamento resultante da utilização das técnicas de programação paralela, das GPUs, está sendo utilizado em aplicações que exigem altas taxas de processamento de dados. Nas GPUs, a arquitetura, como pode ser observado na Figura 2.5, foi projetada para a execução paralela de instruções contendo uma quantidade de ULAs dedicadas muito superior quando se comparado com uma CPU. Essa característica resulta no conceito de intensidade aritmética que, neste caso, é responsável pela relação da quantidade de operações com a quantidade de palavras tranferidas

[17] Do inglês: Central Processing Unit
[18] Do inglês: Single Instruction Multiple Data

com a memória (*bandwidth*). Sendo assim, a intensidade aritmética em GPU é bem superior quando é comparada a uma CPU. Na Figura 2.6 é possível visualizar o grande poder das GPUs em relação às CPUs.

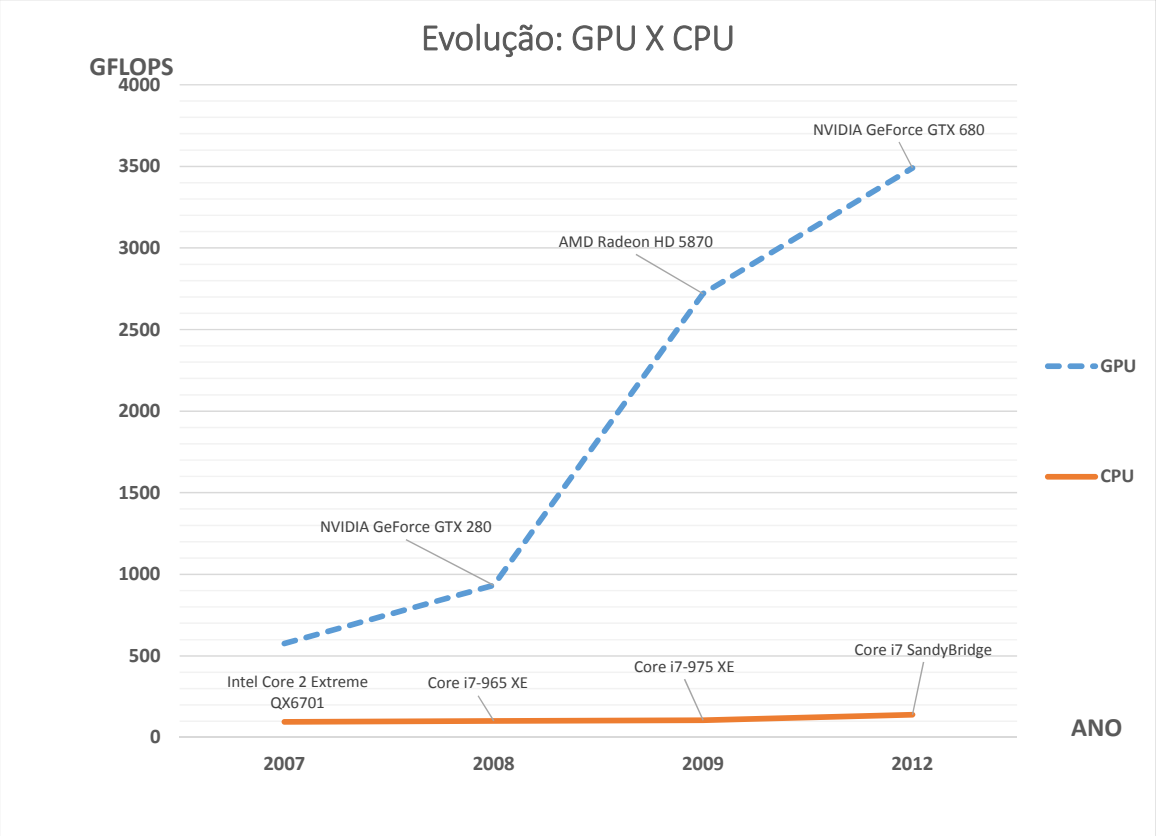


Figura 2.6: Evolução do poder de processamento entre CPU e GPU.

2.3.1 Programabilidade

Com a possibilidade de se programar algoritmos de propósito geral em GPUs, surgiram ferramentas de interfaces de programação de aplicativos, também conhecidas como (API[19]), par este fim. No caso das GPUs, as APIs, por exemplo: *DirectX*, *OpenGL* e *Cg*, possibilitaram a programação e o desenvolvimento de aplicações paralelas nas placas gráficas. Porém, isso requeria muito conhecimento técnico por parte do programador, exigindo do mesmo um grande domínio da arquitetura e principalmente a necessidade de representar o problema utilizando características gráficas, tais como coordenadas cartesianas, texturas, entre outras (ZANOTTO; FERREIRA; MATSUMOTO, 2012).

[19]Do inglês: Application Programming Interface

Visando facilitar o desenvolvimento para os programadores, a NVIDIA desenvolveu a API CUDA[20], uma plataforma de desenvolvimento de computação paralela para GPU. Com essa plataforma foi possível uma maior utilização do poder da GPU e, consequentemente, obter um avanço de desempenho nas aplicações. A plataforma CUDA teve um papel de grande importância na inovação da computação paralela, embora tenha sido criada para o desenvolvimento de aplicações voltadas para as arquiteturas de GPUs.

De acordo com Garland et al. (2008), o modelo de programação paralela em CUDA tem duas características importantes. A primeira está relacionada à linguagem de programação, que neste caso, é uma linguagem de programação sequencial baseada em C/C++, cuja característica é a possibilidade da utilização de paralelismo. A ideia é permitir que o programador se concentre nas questões importantes relacionadas ao paralelismo, garantindo que ele desenvolva algoritmos eficientes, utilizando uma linguagem simples, o que não acontecia nas APIs anteriores. A utilização do paralelismo nas GPUs da NVIDIA é interessante, pois as mesmas variam a quantidade de recursos disponíveis dependendo do modelo da placa de vídeo. A saber, é possível encontrar GPUs com oito núcleos e 768 *threads* até GPUs de 4992 núcleos com mais de 5 milhões de *threads*, sendo essenciais dependendo da aplicação paralela (NVIDIA, 2011; NVIDIA CORPORATION, 2014).

A segunda característica é que a plataforma CUDA foi desenvolvida para a utilização de código paralelo altamente escalável. Isso possibilita a utilização de vários núcleos da GPU e milhares de *threads* simultâneas. Esse processo ocorre de maneira transparente para o programador, que não precisa se preocupar com o gerenciamento das threads durante a aplicação, independente do nível de paralelismo que será utilizado no projeto (GARLAND et al., 2008). Durante o desenvolvimento de um projeto baseado em CUDA, o programador define na implementação trechos no código do programa para serem executados de maneira paralela. Os segmentos do código não paralelizáveis são executados de maneira sequencial pela CPU, enquanto os outros trechos são processados de maneira paralela pela GPU, dependendo do recurso de *hardware* .

O OpenCL[21] surgiu em 2008, com características de programação similares ao CUDA, sendo assim, a plataforma no início dava suporte apenas para a programação em arquiteturas de GPUs. De acordo com Karimi, Dickson e Hamze (2010), enquanto CUDA permite a programação que pode ser executada especificamente em GPUs NVIDIA, o OpenCL é um padrão aberto que permite o desenvolvimento de aplicações da programação em paralelo tanto em CPUs quanto em GPUs independente do fabricante.

[20] Do inglês: Compute Unified Device Architecture
[21] Do inglês: Open Computing Language

Atualmente o desenvolvimento de programas usando GPU é muito mais fácil quando se comparado na época do seu surgimento. Inicialmente, existia apenas as APIs: *Brook-GPU* e a *Close to Metal*; Estas são as interfaces de programação de baixo nível da época, que eram utilizadas para explorar as possibilidades de paralelismo em *hardware*. Com o passar dos anos, surgiram CUDA e OpenCL, que devido a diversas vantagens em relação às antigas APIs, tiveram uma grande aceitação por partes dos programadores. Entretanto, a curva de aprendizagem e a utilização delas pelos programadores iniciantes ainda é considerada lenta. Outro ponto negativo em CUDA e OpenCL é que os programadores gastam uma grande quantia de tempo com código específico do dispositivo comparado àquele gasto implementando a lógica do próprio algoritmo.

Além das APIs CUDA e OpenCL, existe a plataforma de desenvolvimento OpenACC. De acordo com Reyes et al. (2012), a ideia central para a computação heterogênea proposta pela OpenACC é tentar resolver o problema citado anteriormente. Neste caso, o programador não precisa especificar no código qual será o *hardware* de execução. Ao contrário de CUDA que tem melhor desempenho em placas da NVIDIA, a plataforma OpenACC funciona em *hardwares* da NVIDIA, AMD e Intel sem a necessidade de modificação no código fonte. Outra característica dessa plataforma é que OpenACC utiliza *pragmas* de compilação para identificar as regiões do código que serão paralelizáveis, definidas pelo programador. Como ocorre no OpenCL, em OpenACC é possível executar o código tanto em GPU quanto em CPU. Assim, OpenACC possibilita utilizar duas plataformas simultaneamente. A Figura 2.7 mostra o exemplo da estrutura de uma aplicação para OpenACC.

2.3.2 *Pragmas*

Conforme os autores Chapman, Jost e Pas (2007), a definição de uma API ideal para programação paralela tem como base características voltadas na facilidade de uso, permitir a especificação dos algoritmos paralelos e principalmente gerar programas eficientes.

Conforme visto na Seção 2.3.1, diversas soluções foram propostas para aumentar o poder de processamento com a utilização de programação paralela, neste caso utilizando *pragmas* para definir regiões paralelizáveis. *Pragmas* são diretivas de programação que auxiliam o compilador com informações adicionais durante a compilação. No caso da computação paralela, a utilização dos *pragmas* definem explicitamente regiões no código que deverão ser processados de maneira paralela independente da arquitetura. Um exemplo da utilização de *pragmas* pode ser observado na Figura 2.7, nesta temos um código utilizando na plataforma OpenACC, onde os *pragmas* delimitam a região no código fonte que será processada pela GPU de maneira paralela.

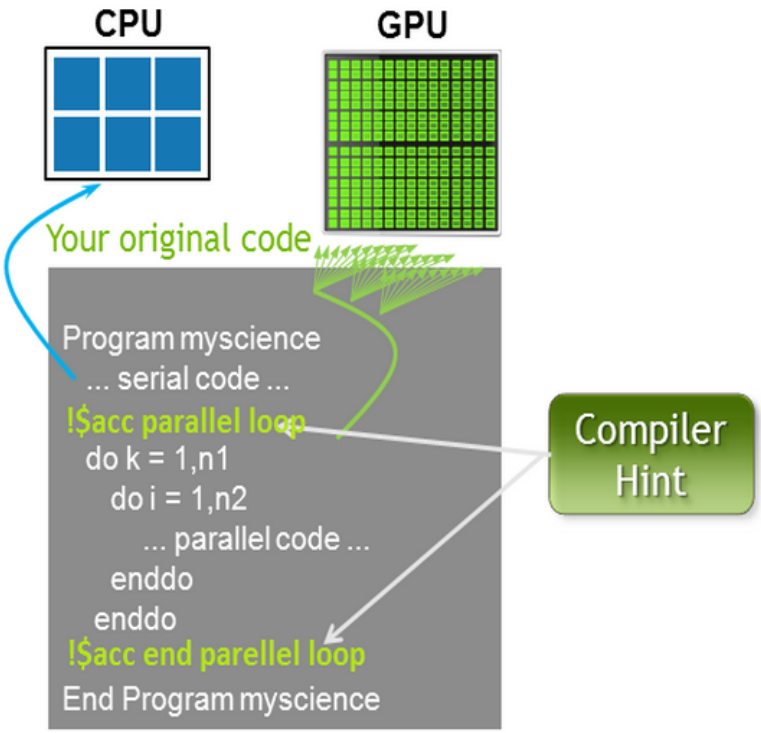


Figura 2.7: Exemplo de codificação em OpenACC (NVIDIA CORPORATION, 2011).

Com a diversidade de arquiteturas e modelos de programação fica evidente a busca de alternativas para melhorar o poder de processamento de dados. Cada solução dispõe de características distitas que podem obter um desempenho satisfatório em uma aplicação específica se sobresaindo sobre as outras soluções.

Ferramentas baseadas em CUDA e OpenCL voltados para FPGAs surgem como uma opção na área, evidenciando o interesse em novas soluções para a geração de *hadware* reconfigurável (LIN; LEBEDEV; WAWRZYNEK, 2010; PAPAKONSTANTINOU et al., 2009; SINGH; ENGINEER, 2011). Neste trabalho é abordado o estudo e a utilização de arquiteturas reconfiguráveis para processamento de alto desempenho, mais informações do trabalho utilizando soluções em arquitetura reconfigurável são descritas nos próximos capítulos.

Capítulo 3

SÍNTESE DE ALTO NÍVEL

A empresa VLSI Technology, fundada no final da década de 70, definiu o rumo do desenvolvimento de circuitos integrados de aplicação específica (ASIC). Como consequência, hoje, temos circuitos complexos contendo milhões de transistores em um único chip. Para o desenvolvimento de tais *chips* com tamanha densidade, os engenheiros contam com ferramentas avançadas de CAD para o gerenciamento e posicionamento dos transistores de maneira eficiente dentro da pastilha. Até recentemente, as grandes empresas utilizavam a metodologia de *design* de circuitos integrados baseada em captura-e-simulação.

Seguindo a metodologia captura-e-simulação, o processo de criação começa no departamento de *design* especificando os requisitos do produto, posteriormente, uma equipe composta por arquitetos desenvolvem um diagrama de blocos da arquitetura do chip gerando uma especificação preliminar do projeto. O próximo passo é executado pelas equipes de lógica e de *designers* de layout, os quais transformam cada bloco funcional do passo anterior em uma lógica ou em um circuito esquemático, o resultado deste passo é enviado para ferramenta de "captura" e posteriormente, é efetuada a simulação para verificar o desempenho, funcionalidades e identificar possíveis falhas do projeto.

A descrição de entrada tem como característica definir quais são as funcionalidades do projeto. Normalmente uma descrição de entrada é definida por algoritmos, neste passo não existem informações estruturais do projeto como interconexões, componentes, estrutura do circuito, estágios *pipeline* e informações de *clock*. A descrição de entrada normalmente é gerada utilizando uma linguagem de descrição de *hardware* (HDL[1]), que possibilita descrever o comportamento em diversos níveis de abstração, e também, suportam atribuições, condicionais, *loops*, entre outros.

[1]Do inglês: Hardware Description Language

A síntese de alto nível é uma sequência de passos que, através da representação textual ou comportamental, possibilita gerar um projeto RTL[2] que pode conter unidades lógicas funcionais, unidades de memória, interconexões como barramento e multiplexadores que possibilitam a geração de *hardware*. Resumidamente, o processo de síntese de alto nível é a transformação automática de um código de alto nível (por exemplo C, C++, Java) em um circuito de *hardware* por meio de compiladores (GAJSKI; RAMACHANDRAN, 1994; CANIS et al., 2011; MCFARLAND; PARKER; CAMPOSANO, 1990).

No processo de compilação, a ferramenta faz a leitura do código de alto nível e gera componentes baseados no código de entrada. Com os componentes definidos, o *netlister* é responsável pelo próximo passo, nesse é efetuado a especificação do *design* de *hardware* relacionada ao funcionamento e comunicação entre os módulos mapeando a entrada e saída dos mesmos. Por fim, é gerada a estrutura RTL final dos componentes e informações de entrada e saída possibilitando uma simulação final do circuito. No RTL é descrito o comportamento de um ASIC utilizando algoritmos, programas, conjunto de instruções e fluxograma de fluxo de dados. Logo, é possível sintetizar os ASICs utilizando as técnicas de síntese de alto nível.

3.1 Ferramentas para geração de *hardware*

Trabalhar com linguagens de descrição de *hardware* tornou-se desafiador, pois exige muito conhecimento técnico da linguagem por parte dos programadores. Nos últimos anos as ferramentas de síntese de alto nível ganharam força e vêm sendo utilizadas como parte importante do processo em projetos de circuitos (SARKAR et al., 2009). A síntese de lógica (que é um processo baseado em síntese de alto nível) é uma evolução importante na metodologia captura-e-simulação. Esse método é vantajoso, pois permite a descrição do comportamento no circuito, independente de detalhes de implementação do *hardware*. Ele permite também que o projetista utilize a metodologia em diversos níveis de abstração, sendo assim, por meio da lógica combinacional é possível sintetizar o projeto em nível de portas lógicas gerenciando unidades funcionais e de controle. Outra característica importante é a possibilidade de sintetizar diagramas baseados em máquinas de estado finito através de síntese sequencial.

McFarland, Parker e Camposano (1990) reforçam que, como em qualquer projeto, independente da sua finalidade, sempre existe a possibilidade de trabalhar com vários níveis de detalhamento, no caso de síntese de alto nível esse processo também ocorre em vários níveis de abstração. Sendo assim, o processo de transformação de código-fonte em alto nível para gerar

[2]Do inglês: Register-Transfer Level

hardware é de suma importância, pois possibilita a diminuição do tempo de desenvolvimento e no custo final do projeto. Trabalhar com uma linguagem de alto nível tal como C/C++, é interessante não apenas pela facilidade relacionada à programação quando se comparada com uma linguagem de geração de *hardware*, mas sim pela quantidade de projetos e problemas já solucionados disponíveis em C/C++ que é muito superior quando se compara com uma linguagem de geração de *hardware*.

Ferramentas de síntese de alto nível possibilitam gerar *hardware* diversificado a partir do mesmo código de entrada dependendo das restrições do projeto, ou seja, características de projeto relacionadas a tempo de execução, área disponível no *chip*, consumo de energia etc, podem afetar o *hardware* gerado pela ferramenta. Essa característica de gerar *hardware* diferente a partir de uma mesma entrada, é possível, pois existem diferentes estruturas pré-definidas que realizam o mesmo comportamento, cada uma com a finalidade de atender uma determinada restrição.

Essa geração automática permite ao desenvolvedor um ganho de tempo de programação considerável, isso também implica menos erros no sistema, pois parte-se do pressuposto de que as estruturas estão livres de erros e otimizadas. Além do mais, esse método possibilita versões diferentes nas quais o programador pode trabalhar e escolher a melhor para o projeto (MCFARLAND; PARKER; CAMPOSANO, 1990).

De acordo com Coussy et al. (2009), aumentar o nível de abstração resulta em um aumento considerável na automação, tanto no processo de desenvolvimento quanto no processo de verificação e simulação. Esses benefícios são as principais vantagens que resultam no ganho de produtividade e capacidade de alcançar um melhor *design* dos componentes no projeto, neste caso, melhorias no *design* impactam diretamente em um ganho no aproveitamento do espaço do *chip*, que pode influenciar no desempenho do circuito. Por outro lado, um desenvolvedor experiente em linguagem HDL consegue criar circuitos que podem ser mais eficiêntes quando comparado com os circuitos gerados pelas feramentas HDL. Essa diferença pode ser relacionada com aproveitamento do espaço no *chip* e/ou em performace de processamento paralelo (BEZERRA, 2002). Mesmo não gerando um resultado tão eficiente comparado com um programador, as ferramentas de síntese de alto nível estão convergindo para alcançar este objetivo.

3.2 Trabalhos Relacionados

Recentes trabalhos de pesquisa buscam melhorar cada vez mais o processo de síntese de alto nível, visando diminuir o tempo de desenvolvimento e garantindo um desempenho no *hardware*

gerado. Este desempenho tende a ser comparado com projeto de *hardware* desenvolvido manualmente por um projetista.

3.2.1 LegUp

O trabalho apresentado por Canis et al. (2011) mostra a ferramenta de código aberto LegUp, voltada para síntese de alto nível utilizando como entrada a linguagem C. O código é processado, compilado automaticamente e tem como saída a linguagem de *hardware*, neste caso Verilog. LegUp é escrito em C++ e utiliza o compilador LLVM[3] para efetuar todos os passos iniciais de análise e otimização posterior de síntese. Nesse compilador é implementado um novo *backend* que usufrui das técnicas de otimizações disponíveis no LLVM, visando melhorias significativas no *hardware*.

A ferramenta LegUp permite ao programador definir onde o código de entrada será executado; isto é, pode ser executado em uma CPU[4] ou em um FPGA[5]. Portanto, o LegUp também permite gerar arquiteturas de estrutura heterogênea, podendo o programador definir trechos no código para serem executados em um processador TigerMIPS e no FPGA. Neste caso, utiliza-se um FPGA para processar um determinado trecho de código funcionando como um acelerador. Assim, o processamento no FPGA pode ser mais eficiente quando é comparado com a mesma execução no processador TigerMIPS. Além disso, ele também permite a integração de vários aceleradores com programas descritos em OpenMP e Pthread.

O LegUp pode sintetizar grande parte do código C para *hardware*, possibilitando trabalhar várias estruturas, tais como matrizes multidimensionais, variáveis globais e ponteiros. Segundo os autores, o *hardware* gerado pela ferramenta LegUp possui qualidade comparável com aquele gerado a partir das ferramentas comerciais disponíveis para síntese de alto nível (CANIS et al., 2011).

O projeto do compilador está ativo e recebe ajuda de vários colaboradores do mundo todo. além disso, diversos projetos de pesquisas estão sendo desenvolvidos para melhorar ainda mais os resultados de desempenho do LegUp. Este conta com o desenvolvimento de técnicas para dar suporte a *loops* para alcançar um nível maior de paralelismo na aplicação.

Alguns pontos são importantes quando se define um projeto em *hardware*, por exemplo desempenho e/ou limitação de recursos. Em alguns casos a velocidade não é um fator relevante ou a limitação de área disponível também não é um problema no projeto. Entretanto, no

[3]Do inglês: Low Level Virtual Machine
[4]Do inglês: Central Processing Unit
[5]Do inglês: Field-Programmable Gate Array

caso de um projeto com área limitada para a aplicação, requer resolução dos desafios para se implementar os componentes do *hardware*, que levam os projetistas a sacrificarem o desempenho do dispositivo. Pesquisas sobre técnicas relacionadas ao aproveitamento de área mostram grande potencial quando implementadas em ferramentas de síntese de alto nível. A técnica de compartilhamento de recursos visa otimizar o uso da área dentro de um determinado chip.

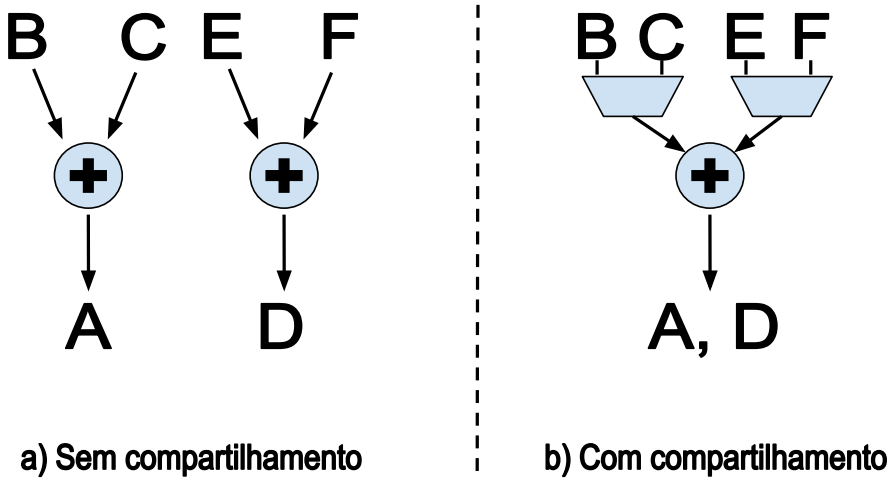


Figura 3.1: Exemplo de compartilhamento de recurso (HADJIS et al., 2012).

No trabalho apresentado por Hadjis et al. (2012), os autores abordam melhorias na ferramenta de síntese de alto nível LegUp, utilizando técnicas de compartilhamento de recursos para conseguir o reaproveitamento de partes funcionais dentro do *hardware*. Considere por exemplo, na Figura 3.1(a) observamos dois processos de adição que devem ser executados em dois momentos distintos. Enquanto na Figura 3.1(b) utiliza os conceitos de compartilhamento de recursos (multiplexadores), sendo assim, tais adições podem utilizar a mesma região de *hardware* para efetuar a soma economizando recursos dentro do chip.

Vale lembrar que a utilização da técnica de compartilhamento de recursos pode impactar no desempenho final do projeto. Voltando no exemplo anterior, os dois processos de soma são executados em dois períodos diferentes durante a execução, caso os processos de soma fossem executados no mesmo ciclo de *clock* seria necessário definir um atraso entre as somas, logo afetando o tempo final de execução.

Na Figura 3.2 é possível visualizar os resultados apresentados no trabalho, neste caso, eles apontam uma redução significativa da área utilizada dentro do dispositivo reconfigurável, nos testes foram utilizados dois modelos diferentes de FPGA da Altera, o modelo Cyclone II e Stratix IV. Nesse trabalho são apresentadas as melhorias da densidade do circuito, por outro lado, o autor não relata informações relacionadas sobre o desempenho dos mesmos.

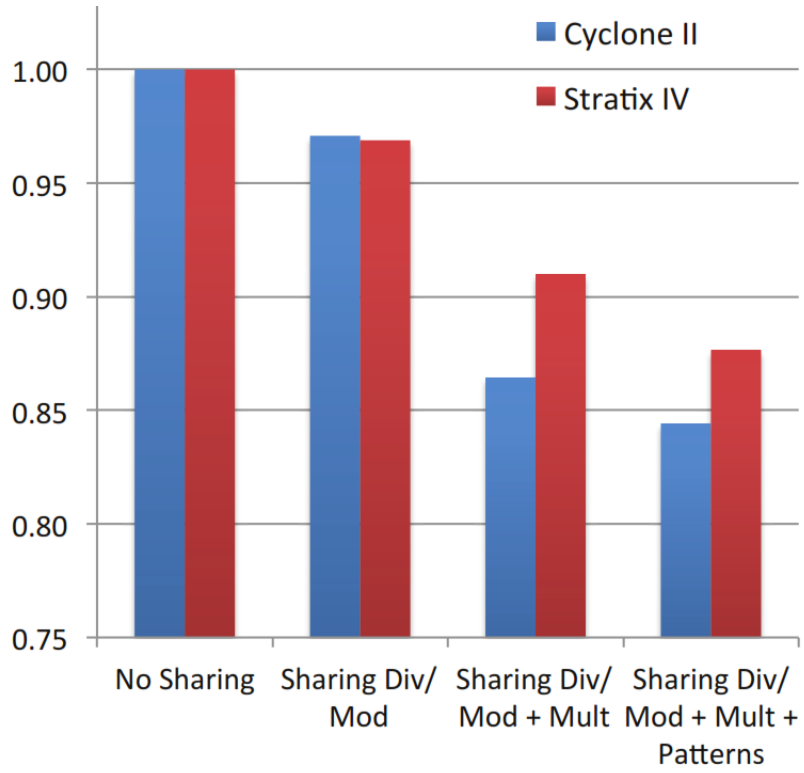


Figura 3.2: Resultados da utilização de compartilhamento de recurso (HADJIS et al., 2012).

3.2.2 LALP

A dificuldade de desenvolver projetos diretamente na linguagem de *hardware* motiva ideias e soluções alternativas utilizando síntese de alto nível. Conforme discutido nos capítulos anteriores, está claro a vantagem de se trabalhar com programação em um nível mais alto. Desenvolver linguagens de alto nível é uma das soluções propostas como alternativa relacionada com o problema de complexidade das linguagens de baixo nível.

Linguagens como LALP e Haydn-C propostas por Menotti (2011) e Coutinho e Luk (2003) respectivamente, propõem uma linguagem de nível de abstração mais elevada, visando ajudar o programador a desenvolver soluções em *hardware* mais facilmente. Tais linguagens possuem sintaxe específica que possibilita ao programador colocar informações no código explicitamente. Estas informações são necessárias para suprir características existentes nas linguagens de baixo nível (informações de barramento, tamanho do registrador, entre outras) que são necessárias para o funcionamento correto do compilador e possibilitar a geração do código de *hardware* automaticamente.

Mesmo atendendo a proposta de desenvolver uma linguagem que facilite o desenvolvimento de projetos em *hardware*, desenvolver uma linguagem específica para síntese de alto nível leva ao surgimento de um problema comum, aprender uma nova linguagem de programação. O

programador precisa aprender toda a sintaxe dessas linguagens para desenvolver seus projetos.

Ainda que essas linguagens tenham semelhanças com linguagens populares, exigem um período de aprendizagem da linguagem. Utilizar projetos existentes por exemplo na linguagem C, demanda de um período de transcrição do algoritmo para a linguagem específica, podendo exigir mais tempo no projeto.

A existência de uma vasta quantidade de linguagens de programação gera diversas dúvidas ao iniciar um determinado projeto. Conforme citado na subseção anterior, a linguagem LALP foi desenvolvida para proporcionar uma alta abstração, visando facilitar o desenvolvimento de *hardware* por meio de síntese de alto nível. Entretanto, mesmo sendo uma linguagem de alto nível, LALP está longe da popularidade entre os programadores de linguagens como Java e C.

Nos últimos anos diversas pesquisas foram desenvolvidas para gerar soluções visando facilitar o desenvolvimento de sistemas. No caso do LALP não foi diferente, no trabalho proposto por Rettore (2012), este descreve o desenvolvimento de um compilador da linguagem C para LALP. O autor justifica o estudo com base no bom desempenho obtido pela ferramenta LALP, e a compara com outras ferramentas de síntese de alto nível. Na Figura 3.3 é possível visualizar um comparativo de tempo de execução na qual mostra claramente um melhor resultado para a ferramenta LALP com outras ferramentas de síntese de alto nível.

O *backend* do compilador LALP tem o foco em *loops* e conta com otimizações satisfatórias nos *hardware* gerado (Figura 3.3). Tais características foram importantes para decidir sua utilização neste trabalho.

3.2.3 CHiMPS

A empresa Xilinx Inc., fundada em 1984, é uma das líderes no mundo no desenvolvimento de soluções em diversas áreas, tais como aeroespacial, medicina, indústria automotiva, data centers, processamento de alta performance, entre outras. A Xilinx vem demonstrando que soluções baseadas em *hardware* reconfigurável é bastante promissora para a computação de alto desempenho.

O departamento de pesquisa da Xilinx desenvolveu um compilador de síntese de alto nível com foco em explorar o poder de processamento paralelo. A ferramenta CHiMPS[6] foi desenvolvida para facilitar o desenvolvimento de sistemas que utilizem o poder de aceleração de processamento baseada em *hardware*, possibilitando a combinação de CPU e FPGA.

[6]Do inglês: Compiling High level language to Massively Pipelined System

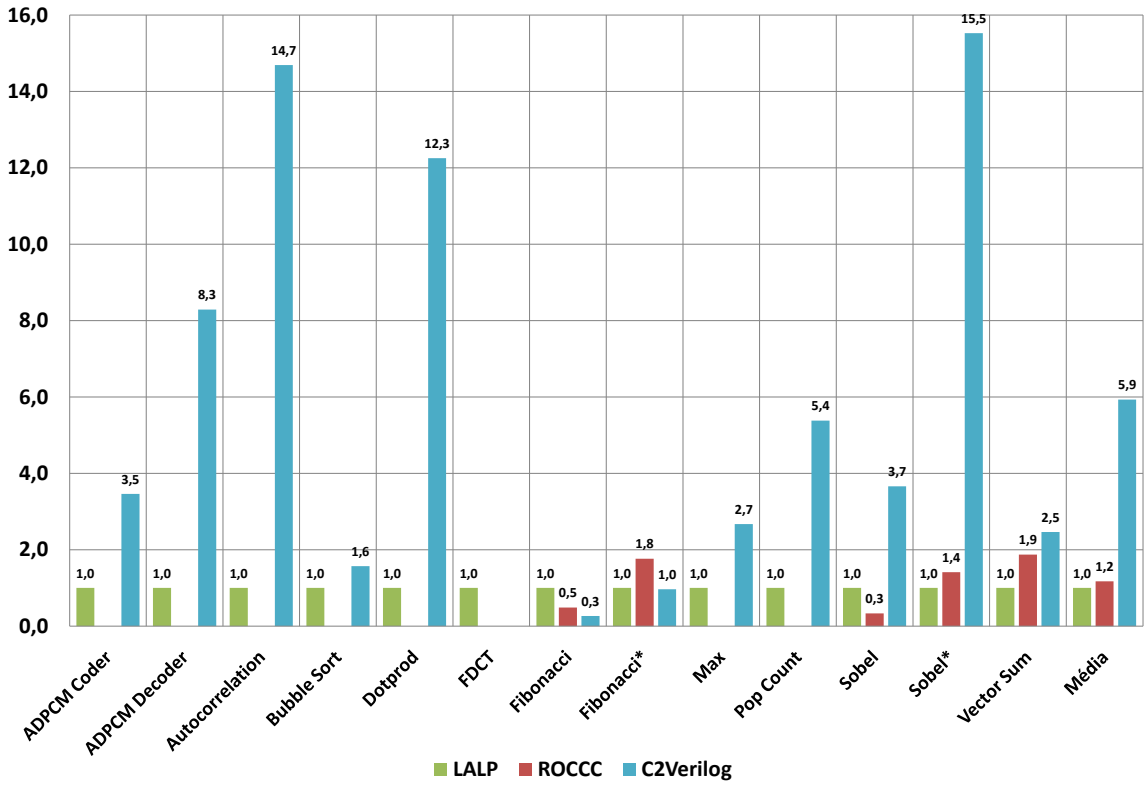


Figura 3.3: Comparativo de tempo de execução (us) entre exemplos sintetizados pelas ferramentas LALP, ROCCC e C2Verilog (MENOTTI, 2010).

A ferramenta CHiMPS utiliza como entrada a linguagem C e gera uma linguagem intermediária, chamada de CTL[7]. Enquanto isso, o *backend* baseado no LLVM, gera o *hardware* em VHDL de acordo com código CTL. Na Figura 3.4 é possível visualizar o processo de síntese da ferramenta CHiMPS.

O foco principal é a utilização de técnicas para otimização do código de entrada para ter um *hardware* na saída com alto nível *pipeline*. Outro ponto interessante dessa ferramenta é a utilização de *pragmas*, que visa permitir ao programador definir características específicas do comportamento do *hardware* de saída, isto é, o programador define explicitamente no código de entrada se uma determinada técnica de otimização deve ser aplicada ou não (PUTNAM et al., 2008).

Algumas técnicas que podem ser definidas explicitamente pelo programador através da utilização de *pragmas* no código de entrada por exemplo: definir manualmente espaço de memória, técnicas de otimização de laços de repetição, definir a estrutura de um bloco funcional, definir a largura de bits.

[7]Do inglês: CHiMPS Targuet Language

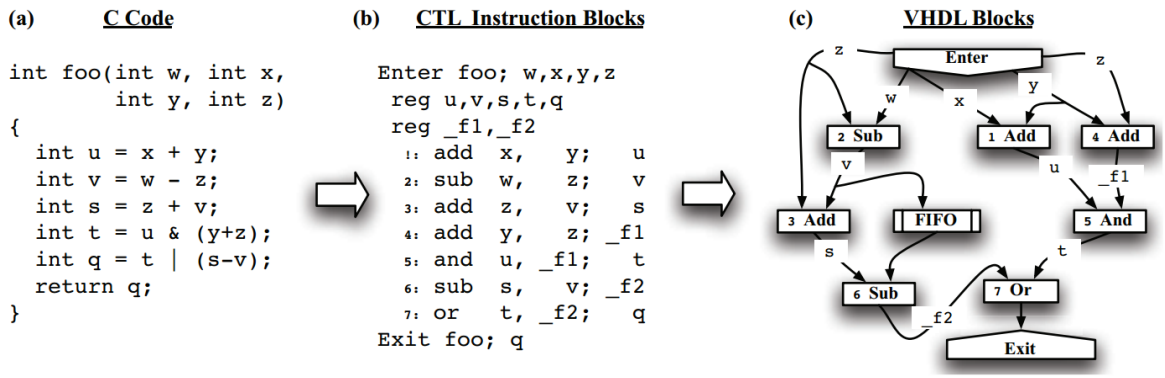


Figura 3.4: Processo de síntese da ferramenta CHiMPS. (a) Exemplo de código em C. (b) Código C transformado em blocos de instruções em CTL. (c) Blocos VHDL gerado no final do processo (PUTNAM et al., 2008).

De acordo com Lee, Raila e Kindratenko (2008), o *frontend* da ferramenta CHiMPS sofre com graves limitações que impedem a geração automática de código. Um exemplo seria uma função que dispõe de dois *return* dentro do escopo, entretanto a ferramenta falha durante o processo de compilação. Outro exemplo de limitação referente à estrutura declarativa que são permitida pela linguagem C mas não é aceita pela ferramenta, este pode ser observado na Figura 3.5.

Figura 3.5: Ferramenta CHiMPS. (a) Exemplo de código C suportado. (b) Exemplo de código C não suportado (PUTNAM et al., 2008).

3.2.4 ROCCC

A ferramenta de síntese de alto nível ROCCC[8], que atualmente se encontra na versão 2.0, usa um compilador para transformar a linguagem C em uma linguagem de descrição de *hardware*, neste caso VHDL.

O objetivo principal do ROCCC é gerar *hardware* otimizado para dispositivos reconfiguráveis. Com a utilização de técnicas de otimização, o *hardware* gerado visa alcançar velocidades acima da média quando se comparados com sistemas similares que geram *hardware* de maneira aleatória (VILLARREAL et al., 2010). Sendo assim, o compilador aproveita de diversas técnicas

[8]Do inglês: Riverside Optimizing Compiler for Configurable Computing

de otimização para alcançar um determinado nível de *pipelining* eficiente, com isso obter uma densidade menor dentro do chip e um ciclo de *clock* mais otimizado.

Outro ponto em destaque do ROCCC é a maneira intuitiva e simples de gerar *hardware* com base no código de entrada na linguagem C, sendo assim, a ideia da ferramenta é facilitar e minimizar o esforço mesmo quando existir a necessidade de uma outra plataforma, ou seja, minimizar que grandes modificações no código sejam necessárias quando migrar para um outro dispositivo FPGA.

A estrutura de funcionamento da ferramenta é dividida em duas partes, a primeira o *frontend*, que é responsável por todas as transformações em alto nível. A segunda parte é o *backend*, responsável por todas as otimizações em baixo nível e geração do *hardware* (BUYUKKURT; GUO; NAJJAR, 2006). O processo de otimização do ROCCC é composto por três recursos importantes, que são responsáveis pelo desempenho final. A saber são:

- **Paralelismo:** Desdobramento de laços de repetição a fim de explorar o poder de execução paralela. Isso quando possível, pois depende da quantidade de elementos lógicos disponíveis no FPGA.
- **Armazenamento:** Otimização para reutilização de dados visando diminuir o acesso à memória.
- ***Pipelining*:** Gerenciar a comunicação de dados eficiente.

Assim como no LegUp, o ROCCC utiliza diretivas do LLVM para a geração da representação intermediária do código de entrada, além das otimizações provenientes do LLVM. Ao se iniciar o processo de compilação, é exibida uma janela na qual o usuário define algumas variáveis de acordo com o objetivo do projeto, um exemplo seria o usuário definir se o código de entrada representa um módulo ou um sistema. De acordo com (CANIS et al., 2011), os módulos são as funções apresentadas no código C utilizados para repetir um determinado cálculo de um fluxo de dados. Já os sistemas são os trechos de código que fazem a instanciação dos módulos, e geralmente consiste em um agrupamento de *loops*.

Durante o processo de compilação, caso seja selecionada a opção módulo, todos os *loops* serão desenrolados automaticamente, substituindo todas as multiplicações de inteiros por *shifts* e adições constantes. Caso o usuário escolha compilar para sistema, o compilador utiliza outras técnicas de otimização, como exemplos temos: transformação de uma matriz bidimensional em uma estrutura unidimensional, detecção de código comum, visando à redução do tamanho do

hardware gerado, para melhorar o desempenho temos técnica de agrupamento de *loops*, entre outras (JACQUARD COMPUTING INC., 2011).

Ao se comparar ROCCC com outras ferramentas, ela se mostra promissora para alguns problemas específicos, porém, quando se compara a ferramenta ROCCC com LegUp utilizando o benchmark CHStone, é possível identificar nos resultados que o compilador ROCCC é inferior e ainda precisa melhorar (CANIS et al., 2011).

3.2.5 C2H

Ferramenta desenvolvida pela a Altera, o compilador C2H faz a leitura de um código em C, e tem o propósito de alcançar alto poder de processamento, através de aceleração de algoritmos, utilizando processador Nios II. Essa ferramenta utiliza o conceito de geração de *hardware* através do código de entrada em alto nível. O programador define quais são as regiões no código que serão executadas em *hardware*. Neste caso, o compilador gera automaticamente componentes que são conectados diretamente ao barramento do processador Nios para executar a aceleração do programa.

O funcionamento do compilador é simples, o usuário coloca dentro de funções separadas os trechos de código que serão acelerados e convertidos pelo compilador em um módulo de aceleração em *hardware*. Caso algum módulo contenha uma chamada de função, esta também será acelerada automaticamente. O processo de geração de aceleração de *hardware* segue os seguintes passos:

- Executa um preprocessamento com o compilador GCC para avaliação do código de entrada.
- Identifica dependências de dados.
- Executa processos de otimização.
- Define o melhor fluxo de execução de cada operação.
- Gera um arquivo HDL com o módulo responsável pela aceleração.
- Gera um arquivo C contendo um *wrapper* que define detalhes de funcionamento da função, e também, ocultando detalhes de interação do módulo como o processador Nios II.

Na Figura 3.6 é possível visualizar a estrutura gerada pelo compilador do módulo de aceleração integrado ao barramento Avalon, e também o processador Nios II. A estrutura utiliza

barramentos exclusivos tanto para dados quanto para instruções. Já a memória do módulo utiliza operações de acesso direto à memória, trocando dados entre o processador e os módulos de aceleração.

As funções mais indicadas para serem aceleradas são aquelas que dispõem de laços de repetição com alguns níveis de interação. Porém funções com várias operações sequenciais, dados com pontos flutuantes, funções recursivas não geram bons resultados de transformação.

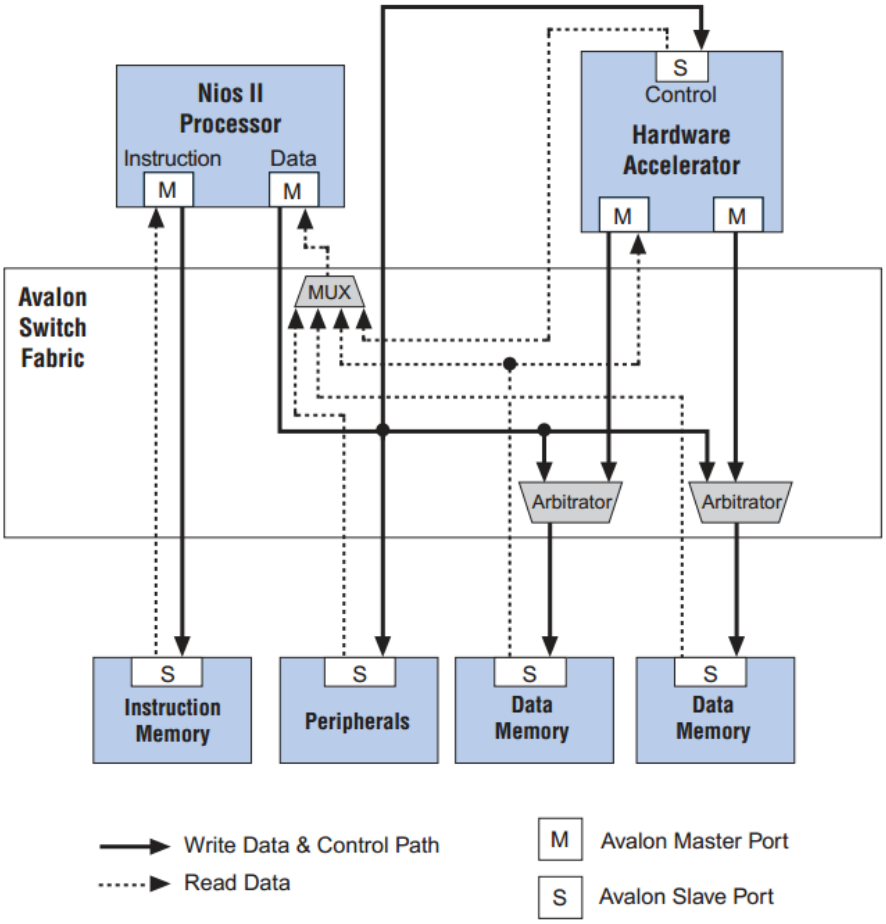


Figura 3.6: Modelo de aceleração utilizado pelo compilador C2H (ALTERA CORPORATION., 2009b).

3.2.6 Outros Trabalhos

Outra abordagem interessante neste contexto é a do ReflectC, uma ferramenta de síntese de alto nível com foco em alta flexibilidade no projeto. O compilador utiliza a ferramenta LARA por meio de programação orientada a aspectos, o que permite o usuário especificar informações adicionais do projeto em um arquivo separado. Essas parametrizações feitas no padrão LARA permitem tanto a aplicação de características específicas quanto a aplicação de otimizações no *hardware* gerado (CARDOSO et al., 2013; COUTINHO et al., 2013). Diversas ferramentas de síntese

de alto nível têm sido propostas, o que demonstra o interesse em pesquisas na área de geração automática de *hardware* para dispositivos reconfiguráveis, entre as quais podemos citar Altera, Catapult C, Synopsys, Vivado (FEIST, 2012; NEUENDORFFER; MARTINEZ-VALLINA, 2013).

Capítulo 4

COMPILADORES *open source*

Os compiladores *souce-to-source* exercem um papel fundamental em ferramentas de otimização de código. Geralmente estes compiladores permitem validar o código de entrada, efetuar análises, otimizações e gerar uma estrutura de dados servindo como base para o desenvolvimento de outras ferramentas. Neste trabalho a ferramenta em questão auxilia na validação do código, dispõe de técnicas de otimização e por fim, gera uma estrutura em árvore que viabiliza a criação dos componentes de *hardware*. A seguir são descritas algumas características de alguns compiladores *souce-to-source*.

4.1 LLVM

O projeto LLVM, desenvolvido inicialmente na Universidade de Illinois, foi responsável pelo desenvolvimento do compilador que recebe o mesmo nome. Atualmente, o compilador LLVM é um projeto que conta com contribuições de colaboradores de todo o mundo. A ferramenta LLVM é um compilador *souce-to-source* com foco em otimizações e conta com diversas técnicas implementadas, visando melhorar o desempenho da aplicação. Trata-se de um compilador baseado no GCC e inicialmente contava com interface C e C++. Na versão atual, conta com suporte para as linguagens C, C++, Ada, Objective-C e Fortran, atraindo diversos desenvolvedores para a plataforma (LATTNER, 2008).

A ideia por trás do LLVM é facilitar o desenvolvimento de projetos por meio de seus componentes modulares reutilizáveis, que permitem o desenvolvimento de *backends* para uma outra linguagem.

No início do processo de compilação o compilador lê o arquivo fonte de origem, aplica técnicas de otimização no código e gera uma representação intermediária. A representação

intermediária do código de entrada é disponibilizada para o desenvolvedor em uma linguagem de nível mais baixo. Já o *backend* tem como característica suporte para várias arquiteturas incluindo x86, x86-64, PowerPC, ARM, etc.

Além do projeto principal voltado ao compilador LLVM, existem diversos sub-projetos com vários componentes que permitem a compilação de outras linguagens de programação utilizando o compilador LLVM, sendo as principais Ruby, Python, Haskell, Java, D, PHP, Pure entre outras (LLVM PROJECT, 2013).

4.2 Cetus

O Cetus é um compilador *source-to-source* iniciado em 2004 como pesquisa chamada *Cetus Project*. O intuito inicial do projeto era utilizar o poder de processamento por meio das técnicas de paralelização automática de código em projetos de pesquisa (CETUS PROJECT, 2011). O projeto recebeu incentivo financeiro da *U.S. National Science Foundation*, e ainda conta com ajuda de pessoas do mundo todo (DAVE et al., 2009).

O compilador é desenvolvido em Java, tem como entrada um arquivo com o código em linguagem C e conta com diversas técnicas de otimizações de códigos, estas por sua vez, possibilita a geração de código otimizado em C, OpenMP e CUDA.

O trabalho de Johnson et al. (2005) relata as principais características do compilador Cetus, sendo elas:

- **Representação Intermediária**: O compilador Cetus gera uma estrutura de dados em árvore com a representação intermediária do código fonte de entrada mantendo a mesma sintaxe do código fonte.
- **Identificação de Componentes**: Após ser gerada a representação intermediária, o compilador dispõe de uma classe abstrata *IRIterator* que implementa a interface *Iterator* do Java, permitindo percorrer facilmente os nodos da árvore. Na Figura 4.1 é possível visualizar um trecho de código que permite percorrer a árvore utilizando iteradores para a identificação de *loops*.
- **Tabela de Símbolos**: O compilador mantém todas as informações do código em uma tabela de símbolos. Permite acesso simples às informações de identificação, tais como constantes, funções, variáveis, etc.

- **Manipulação e Análise do Código**: A ferramenta permite a análise e manipulação do código gerado, isto facilita o desenvolvimento de novos *parsers* conforme a necessidade do projeto. Essa característica é muito importante para o desenvolvimento de novas técnicas de otimizações e outras soluções.

- **Geração de Grafos**: Permite a geração gráfica do código fonte por meio de grafos, podendo gerar uma representação de todo o código ou de uma determinada parte.

```
BreadthFirstIterator iter = new BreadthFirstIterator(proc);
try {
  while (true)
  {
    Loop loop = (Loop)iter.next(Loop.class);
    // Do something with the loop
  }
} catch (NoSuchElementException e) {
}
```

Figura 4.1: Código de exemplo usando para identificação de *loops* no código (DAVE et al., 2009).

4.3 ROSE

O projeto *ROSE Compiler Framework* é responsável pelo compilador ROSE, o qual é um compilador de código aberto que fornece uma estrutura para leitura e escrita de código-fonte com suporte às linguagens de programação C, C++ e Fortran. Outra característica deste compilador é que, além de leitura e escrita de código-fonte, ele também suporta efetuar análise em binários executáveis, nas arquiteturas x86, Power PC e conjuntos de instruções ARM. O compilador ROSE visa facilitar o processo de tradução *source-to-source* em uma compilação, pois o mesmo, oferece ao desenvolvedor ferramentas de análise e transformação de código.

Compiladores com base em código-fonte oferece funcionalidades de *parsing*, análises de compilação, transformações e geração de código. Já na parte da ferramenta que trabalha com binários, o compilador suporta a desmontagem do arquivo, detecção de funções e análises de código. Além disso, o compilador ROSE, permite o desenvolvedor mesclar código-fonte com binário (ROSE COMPILER INFRASTRUCTURE, 2012).

De acordo com Liao et al. (2010a), o compilador ROSE tem capacidade de trabalhar em conjunto com aplicações paralelas utilizando OpenMP, UPC e MPI. Semelhante aos demais compiladores *source-to-source*, ROSE é composto com *frontend*, um *middle-end* e um *backend*, utiliza em conjunto com várias técnicas de análises de código-fonte e otimizações. Basicamente

o ROSE fornece uma representação intermediária orientada a objetos com um conjunto de análise e de interfaces de transformação que permite o usuário desenvolver tradutores, analisadores, otimizadores e ferramentas especializadas rapidamente.

O funcionamento do compilador é baseado na leitura do código-fonte e/ou do binário para gerar uma árvore de sintaxe abstrata. A árvore é a utilizada para a representação gráfica da estrutura do código de entrada, possibilitando a automatização da representação intermediária do código para análise do fluxo do programa (QUINLAN, 2000).

4.4 Comparativo

De acordo com Dooley (2006), o compilador LLVM consegue gerar boas otimizações no código, entretanto, o código de saída tem uma sintaxe muito diferente da linguagem de entrada, que por sua vez dificulta a manipulação e tomadas de decisões por parte do programador.

No trabalho de King (2012), o autor descreve que a ferramenta LLVM é projetada para a otimização binária e também para gerar uma representação intermediária em um nível muito baixo, o que dificulta diretamente o processo de desenvolvimento.

Na ferramenta ROSE, durante o processo C-to-C, o compilador efetua diversas validações da linguagem e possibilita a otimização do código. O fato importante está na representação intermediária, que mantém a mesma sintaxe do código de entrada, ou seja, a sintaxe na representação intermediária é mantida em um alto nível de abstração, facilitando sua manipulação.

O compilador Cetus tem a mesma característica da representação intermediária do compilador ROSE, e também conta com técnicas de otimização de código. Como citado anteriormente, o compilador Cetus é implementado em Java, isso facilitaria a integração com o *backend* do compilador LALP. Entretanto o compilador ROSE conta com o *frontend* EDG[1], que é considerado robusto e bastante conceituado. Este também dispõe de técnicas de otimizações que não estão disponíveis no compilador Cetus. Essas técnicas são de suma importância para o desenvolvimento deste trabalho. Sendo assim, essa é a principal característica na escolha do compilador ROSE para o desenvolvimento do compilador LALPC.

[1] Do inglês: Edison Design Group

Capítulo 5

LALPC

A utilização de processamento paralelo tem se mostrado uma solução importante para o problema relacionado ao aumento de poder de processamento. Como citado no início deste trabalho, uma maneira de contornar este problema é a utilização de dispositivos reconfiguráveis como aceleradores em regiões críticas no código. Combinar arquiteturas reconfiguráveis com processadores mostra-se uma alternativa promissora.

Neste trabalho é apresentado o compilador LALPC[1]. Esse foi desenvolvido baseando no compilador LALP citado anteriormente na Seção 3.2.2 porém com algumas diferenças. Tanto LALP quanto LALPC utilizam técnicas de paralelismo para acelerar regiões baseadas em laços de repetição. Entretando, LALPC tem como objetivo principal aumentar o nível de abstração aceitando como entrada programas descritos em linguagem C, além de utilizar *pragmas*. Os *pragmas* são necessários para implementar funcionalidades existentes em LALP e também, técnicas de otimização inexistentes no compilador.

Como em LALP, o objetivo deste trabalho é gerar *hardware* automaticamente para FPGAs[2] a partir de descrições em alto nível. Porém, diferente de LALP, em LALPC aceita códigos descritos em C com *pragmas*. Ao utilizar os *pragmas*, estes abrem caminho para um desenvolvimento mais unificado com outras ferramentas e arquiteturas que também utilizam diretivas de marcação para definir explicitamente o paralelismo no código. Na Seção 5.6 descreve-se detalhadamente a funcionalidade dos *pragmas* no compilador LALPC.

Neste capítulo são apresentadas informações detalhadas referentes ao compilador implementado. Essas informações descrevem a estrutura do compilador, o processo de compilação, bem como as semelhanças e diferenças entre LALPC e LALP.

[1] Do inglês: Language for Aggressive Loop Pipelining for C
[2] Do inglês: Field-Programmable Gate Array

5.1 Estrutura do Compilador

O compilador LALPC é composto de cinco partes, a saber: *frontend* do compilador ROSE; módulo de análises e geração dos componentes; módulo de otimizações; *backend* LALP e por fim, a biblioteca de componentes VHDL. Na Figura 5.1 é possível visualizar a estrutura e o fluxo de execução do compilador LALPC.

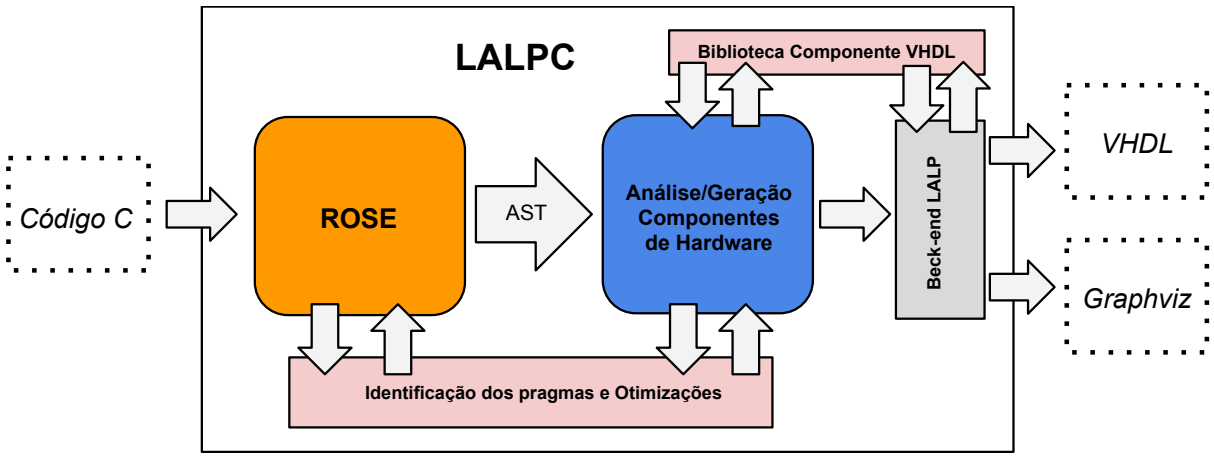


Figura 5.1: Estrutura do compilador LALPC.

O LALPC instancia partes do compilador ROSE para validar o código de entrada em linguagem C. Esse processo utiliza o *frontend* do compilador ROSE efetuando todas as validações necessárias no código de entrada nos padrões ANSI C, tais como a análise léxica, sintática e semântica. O compilador ROSE implementa técnicas de otimização para analisar e modificar laços de repetição, tais como *loop fusion*, *loop fission*, *loop blocking*, *loop interchange* e *loop unrolling* (QUINLAN, 2000). A técnica denominada *loop unrolling* é utilizada no compilador LALPC e é descrita em detalhes na Seção 5.7.

Outra análise do compilador ROSE utilizada no LALPC é a SSA[3]. Este processo modifica o nome das variáveis do código conforme necessário, tornando cada atribuição única, o que facilita o processo de geração dos componentes de *hardware*. Essa característica supriu a necessidade encontrada no compilador LALP. Neste o usuário deve definir no código novas variáveis toda vez que receber uma nova atribuição, essa característica em LALP é detalhada na Seção 5.3.1. Após essa etapa, obtem-se uma AST[4] com a representação intermediária do código. A representação é utilizada para gerar os componentes de *hardware* e também permite definir as ligações de dados e controle entre eles. Na Figura 5.2 é possível visualizar a estrutura da uma árvore de representação intermediária gerada pelo compilador ROSE.

[3]Do inglês: Static Single Assignment
[4]Do inglês: Abstract Syntax Tree

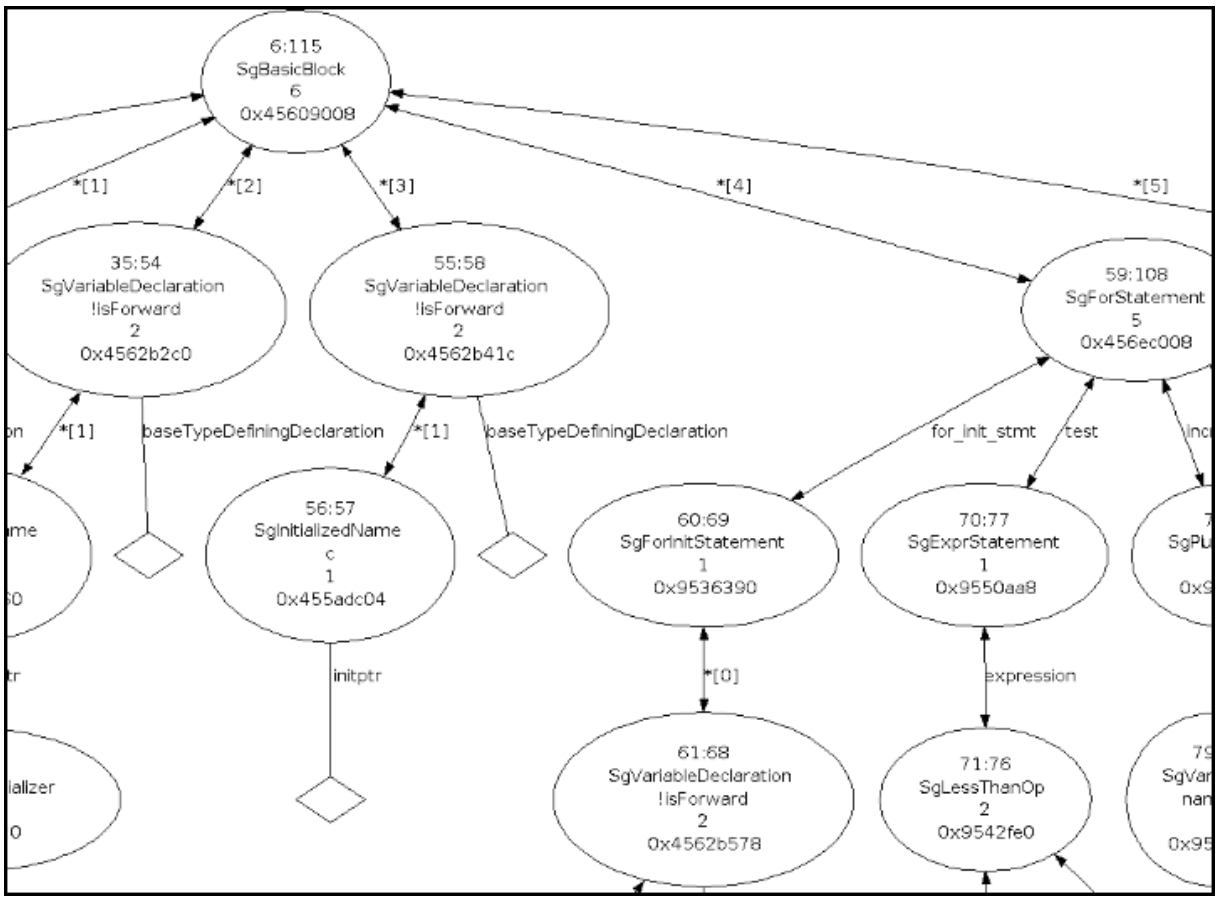


Figura 5.2: Visualização gráfica de uma AST gerada pelo compilador ROSE.

No decorrer do processamento verifica-se a existência dos *pragmas* na AST. Estes como citado anteriormente, são responsáveis pelas otimizações no *hardware*. Ao identificar os *pragmas*, o módulo de otimização do compilador LALPC pode modificar a AST gerada pelo *frontend* do compilador ROSE e/ou modificar os componentes VHDL. Essas modificações na árvore AST afetam o processo de criação dos componentes de *hardware*.

Com a representação intermediária definida, o módulo de criação dos componentes percorre toda a árvore criando os componentes de *hardware*. Após efetuado esse processo, a árvore AST é utilizada novamente para criar as ligações das operações encontradas no código. Essa etapa gera um grafo com os componentes de *hardware* e suas ligações representando preliminarmente a estrutura da arquitetura a ser gerada. Esse processo é descrito com mais detalhes nas Seções 5.6 e 5.7.

Após finalizar as ligações dos sinais entre os componentes, o compilador executa as técnicas de balanceamento dos componentes, provindas do *backend* do compilador LALP, as quais executam o escalonamento dos componentes. Esse escalonamento tenta identificar dependências entre as operações inserindo atrasos nas ligações quando necessário. Sendo assim, o processo

de escalonamento garante que a informação de um componente chegue no momento correto em outro. Além disso, os algoritmos também são responsáveis por inserir novas ligações que permitem ativar a porta *write enable* dos componente a fim de permitir a gravação nestes.

Por fim, com os componentes escalonados e as arestas balanceadas, o compilador gera os arquivos VHDL com os componentes e suas respectivas ligações. Esse processo de geração dos arquivos VHDL também dispõe de geração de um *testbench* para a arquitetura. O *test bench* permite simular a arquitetura a fim de verificar se ela funciona corretamente. Também no final da compilação é gerado um arquivo do tipo Graphviz, que contém a representação gráfica da arquitetura. Esta representação permite visualizar detalhadamente a estrutura do *hardware* contendo os componentes e suas ligações.

O processo de geração do *test bench* e da representação visual da arquitetura foram reimplementados no compilador LALPC semelhantemente ao compilador LALP. A estrutura do *backend* citado anteriormente é descrita com mais detalhes na Seção 5.4.

5.2 Biblioteca de Componentes

Como em alguns compiladores para HLS[5], LALPC utiliza uma biblioteca de componentes que implementa as entidades VHDL. Essas entidades dispõem de características e funcionalidades distintas permitindo gerar sistemas em *hardware* quando combinadas entre elas. VHDL estrutural é o processo de geração de *hardware* utilizando componentes VHDL e ligações entre eles. Essas ligações representam os sinais que permitem a comunicação e tráfego dos dados entre os componentes.

A biblioteca de componentes do compilador LALPC é a mesma utilizada no compilador LALP, que por sua vez é uma versão modificada da biblioteca disponível no compilador NENYA (CARDOSO, 2000; CARDOSO; NETO, 2003). As modificações foram necessárias para atender a linguagem LALP. Essa biblioteca contém componentes de operações lógicas e aritméticas com inteiros, comparadores, contadores, multiplexadores, memórias e registradores que permitem implementar sistemas em *hardware*.

O uso da biblioteca é feito por meio de associação, ou seja, o processo identifica um elemento na árvore com a representação intermediária gerada pelo compilador ROSE e cria um componente de *hardware* respectivo. Essa associação ajuda tanto no processo de criação dos componentes quanto na criação das ligações entre eles.

[5]Do inglês: High Level Synthesis

Código 5.1: Exemplo componente de soma em VHDL

```
1  entity add_op_s is
2    generic (
3      w_in1 : integer := 16;
4      w_in2 : integer := 16;
5      w_out : integer := 32);
6    port (
7      I0    : in  std_logic_vector(w_in1-1 downto 0);
8      I1    : in  std_logic_vector(w_in2-1 downto 0);
9      O0    : out std_logic_vector(w_out-1 downto 0));
10 end add_op_s;
```

No Código 5.1 é possível visualizar a descrição de um componente em VHDL. Nas linhas 3 a 5 encontram-se os parâmetros que determinam a largura de bits das portas (linhas 7 a 9) de entrada e saída do componente.

5.3 LALP

5.3.1 LALP X LALPC

Como citado anteriormente, LALPC é baseado no compilador LALP. Sendo assim, LALPC utiliza os mesmos conceitos de paralelismo por meio do escalonamento dos componentes para otimização do *hardware*. No entanto, LALPC se diferencia do LALP ao utilizar códigos descritos na linguagem C como entrada do compilador, ao invés de utilizar uma linguagem específica. Essa característica é fundamental para sua utilização uma vez que a linguagem C é altamente difundida entre programadores. Ao utilizar a linguagem C como entrada, contribui-se para o desenvolvimento mais rápido do projeto.

As características particulares da linguagem LALP podem elevar o tempo de desenvolvimento e/ou levar o programador ao erro durante o desenvolvimento do código. Isso pode ocorrer porque a LALP dispõe de características específicas e também por ser uma linguagem mais baixo nível que a linguagem C. Para demonstrar as particularidades da linguagem LALP, no Código 5.2 é descrito um exemplo de um código em C com múltiplas leituras em um vetor. No Código 5.3 é apresentado um trecho do código em LALP que implementa o mesmo exemplo. Nas linhas 3 à 10 são definidos os índices de leitura do vetor no multiplexador “add”, enquanto na linha 11 temos a memória “indata” recebendo na porta “address” a saída do multiplexador “add”. Por fim, nas linhas 12 à 19 são efetuadas as atribuições às variáveis.

O compilador LALP permite o sicronismo das operações manualmente por meio do caractere “@” no código. Esse processo auxilia o compilador inserindo atrasos nas operações. O

sincronismo manual é utilizado em operações algumas operações específicas e também quando o programador precisa gerar arquiteturas diferentes para o mesmo código. No Código 5.3 é possível visualizar que a linguagem LALP necessita que o programador utilize as diretivas de marcação. Neste caso, o atraso é necessário para sincronizar a leitura dos valores da memória "input". Nas linhas 4 a 10 é descrito o sincronismo manual que define o endereço da memória, enquanto nas linhas 12 a 19 é descrito o sincronismo manual da gravação dos valores nas variáveis.

O compilador LALPC suporta o uso da técnica de sincronização manual das operações utilizando *pragmas*. Entretanto, a necessidade de definir o sincronismo em exemplos complexos foi resolvida no compilador LALPC modificando o processo de balanceamento em conjunto com a técnica SSA do compilador ROSE. Nos testes realizados neste trabalho não foram necessários os *pragmas* de sincronismo para que o *hardware* gerado calculasse corretamente os resultados, diferentemente do compilador LALP no qual algumas sincronizações manuais eram necessárias.

Código 5.2: Exemplo multiplo acessos em um vetor

```
1  foo() {
2    int i, input[N], a, b;
3    int c, d, e, f, g, h;
4    for (i = 0; i < N; i++)
5      a = input[i];
6      b = input[i+1];
7      c = input[i+2];
8      d = input[i+3];
9      e = input[i+4];
10     f = input[i+5];
11     g = input[i+6];
12     h = input[i+7];
13 }
```

Código 5.3: Exemplo multiplo acessos em um vetor em LALP

```
1  foo() {
2    counter (i=0; i<N; i+=1@8);
3    addr = i;
4    addr = i + 1 when i.step@1;
5    addr = i + 2 when i.step@2;
6    addr = i + 3 when i.step@3;
7    addr = i + 4 when i.step@4;
8    addr = i + 5 when i.step@5;
9    addr = i + 6 when i.step@6;
10   addr = i + 7 when i.step@7;
11   input.address = addr;
12   a = input when i.step@2;
13   b = input when i.step@3;
14   c = input when i.step@4;
15   d = input when i.step@5;
16   e = input when i.step@6;
17   f = input when i.step@7;
18   g = input when i.step@8;
19   h = input when i.step@9;
20 }
```

Conforme citado anteriormente, outra deficiência encontrada em LALP é a necessidade do

programador renomear as variáveis do mesmo nome. Sendo assim, quando existem duas ou mais variáveis sendo atribuídas com o mesmo nome, devem ter seus nomes alterados. Essa característica afeta o processo de ligação dos componentes no compilador LALP, gerando na saída um *hardware* incorreto.

O compilador LALP durante sua execução cria componentes únicos conforme as variáveis declaradas no código. Sendo assim, quando é identificado uma nova variável e já existe um componente com o mesmo nome, LALP não cria um novo componente, mas simplesmente cria uma nova ligação para o componente existente. Ao tentar utilizar o componente antigo, ele modifica toda a estrutura lógica da aplicação gerando um resultado errado para o *hardware*. No Código 5.4 é descrito um exemplo em C para a situação citada anteriormente com duas atribuições para duas variáveis com nome "a". Para implementar esse mesmo exemplo corretamente no compilador LALP, o programador deve modificar o nome de uma variável. Neste caso, no Código 5.5 descreve-se o exemplo em LALP. Note que, para o funcionamento correto, uma das variáveis com nome "a" foi renomeada para "a1". Sendo assim, ao renomear a variável, o programador deve modificar todas as referências dessa no restante do código. Entretanto, ao renomear as variáveis, o programador precisa ter cuidado para manter a semântica da aplicação, pois em códigos complexos esse processo é uma atividade passível de erro.

Conforme citado anteriormente, em LALPC o problema é contornado com o uso do *frontend* do compilador ROSE. Ao gerar a árvore AST o compilador ROSE gera um nodo no grafo para cada variável utilizada no código fonte. Dessa maneira, a criação de um componente de *hardware* é definida pela identificação de um nodo na árvore AST, independente se o nodo tem o mesmo nome de outro. Logo o exemplo apresentado no Código 5.4 funciona perfeitamente em LALPC.

Código 5.4: Exemplo uso de variáveis em C

```
1 foo() {
2   a = 3 + 1;
3   a = a + 2;
4   b = a;
5 }
```

Código 5.5: Exemplo uso de variáveis em LALP

```
1 foo() {
2   a = 3 + 1;
3   a1= a + 2;
4   b = a1;
5 }
```

A utilização das características supracitadas permitem ao programador gerar de maneira simples sistemas de *hardware* de alto desempenho baseados em programas descritos em C. O compilador LALPC constitui uma alternativa para a simplificação do processo de geração automática de *hardware* mantendo a exploração de processamento paralelo em arquiteturas

reconfiguráveis proposto em LALP.

5.4 *Backend* LALP

O *backend* do compilador LALP foi reimplementado no compilador LALPC. Os resultados apresentados no trabalho de Menotti (2010) justificam a decisão de reimplementá-lo no compilador LALPC. A seguir são descritas as técnicas disponíveis no *backend* LALP, implementadas no compilador LALPC.

5.4.1 Visualização

LALP permite ao fim da compilação gerar grafos da arquitetura de *hardware*. Visualizar graficamente a representação textual da arquitetura facilita ao programador abstrair as informações geradas pelo compilador. Sem a representação gráfica o programador precisa rastrear uma determinada informação diretamente no código VHDL. Esse processo pode ser lento e exaustivo dependendo da complexidade da arquitetura gerada.

O modelo gráfico gerado pelo compilador dispõe de informações visuais tais como: nome do componente, portas, sinais entre os componentes, largura de bits, atrasos entre os componentes, entre outras. Os componentes registrados são representados na cor cinza, os não registrados são brancos, constantes têm a forma de elipses e os pinos de entrada e saída têm a forma retangular. Na Figura 5.3 é possível visualizar detalhadamente os componentes e as ligações de um sistema de *hardware* gerados pelo compilador LALPC.

5.4.2 Balanceamento e Geração de *Hardware*

O processo de geração de *hardware* executa as técnicas de balanceamento reimplementadas no compilador LALPC, as quais identificam dependências entre os componentes escalonando-os de maneira correta. Esse escalonamento insere componentes do tipo *delay* definindo os atrasos para execução correta das operações. Nesse passo, também são criadas as ligações das portas *write enable* de componentes que efetuam gravações, que por sua vez recebem atrasos permitindo que o registrador/memória grave o dado no momento correto. Sendo assim, os atrasos permitem que tanto os dados cheguem no momento correto na porta de entrada de um componente quanto o exato momento que este deve efetuar a gravação.

Para exemplificar o processo de balanceamento, na Figura 5.3 é possível visualizar uma arquitetura desbalanceada. Esta imagem foi gerada com o estrutura da arquitetura antes da

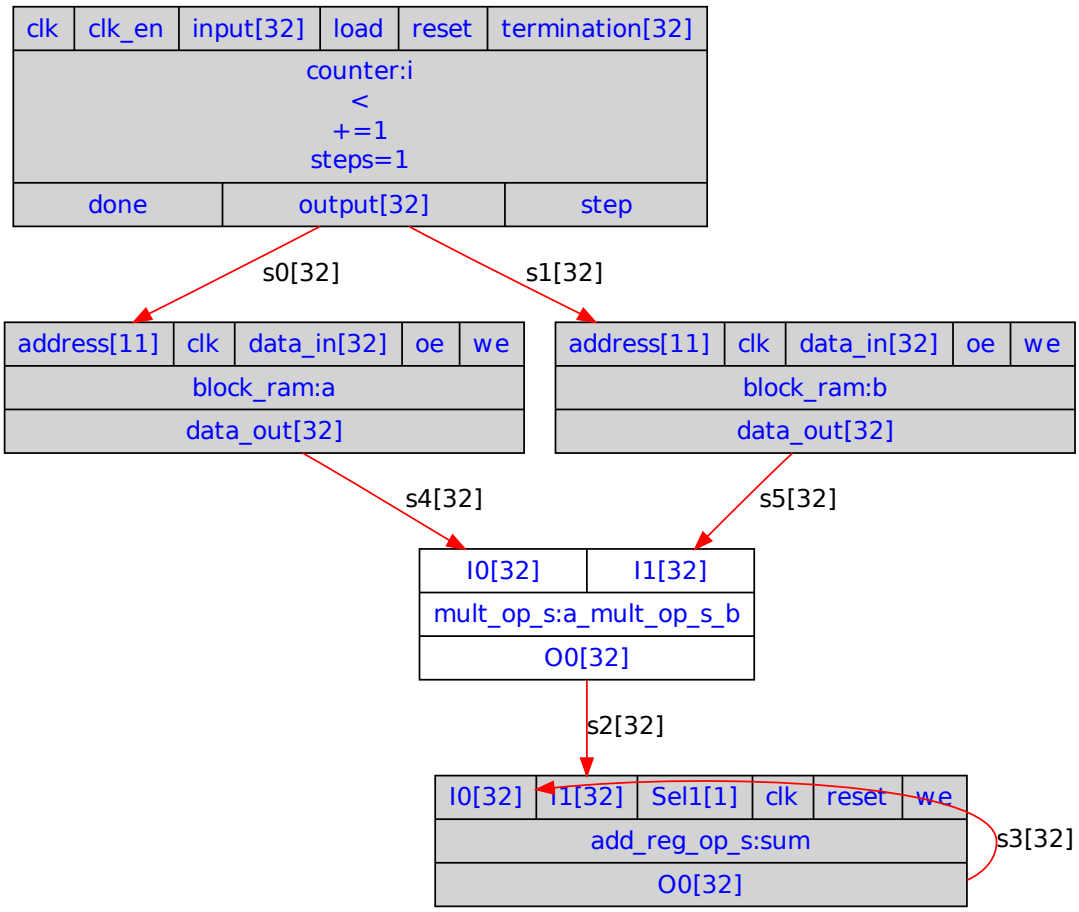


Figura 5.3: Visualização gerada pelo compilador LALPC.

execução do processo de escalonamento dos componentes. Enquanto na Figura 5.4 temos a visualização após o processo de balanceamento. Neste caso, o processo de balanceamento criou uma ligação na porta *write enable* do registrador "sum" com um componente do tipo *delay* atrasando dois ciclos. Esse atraso é necessário uma vez que os componentes do tipo memória necessitam de dois ciclos para a leitura. Ou seja, neste exemplo o escalonamento identificou a necessidade de inserir um atraso de dois ciclos de *clock*. O atraso permite a leitura dos valores das memórias "a" e "b" gravando o resultado da operação de soma no momento correto no registrador "sum".

5.5 Subconjunto Suportado

Diversas ferramentas de síntese de alto nível, algumas usadas comercialmente, têm limitações oriundas das diferenças entre os paradigmas de *software* e *hardware*. As limitações podem

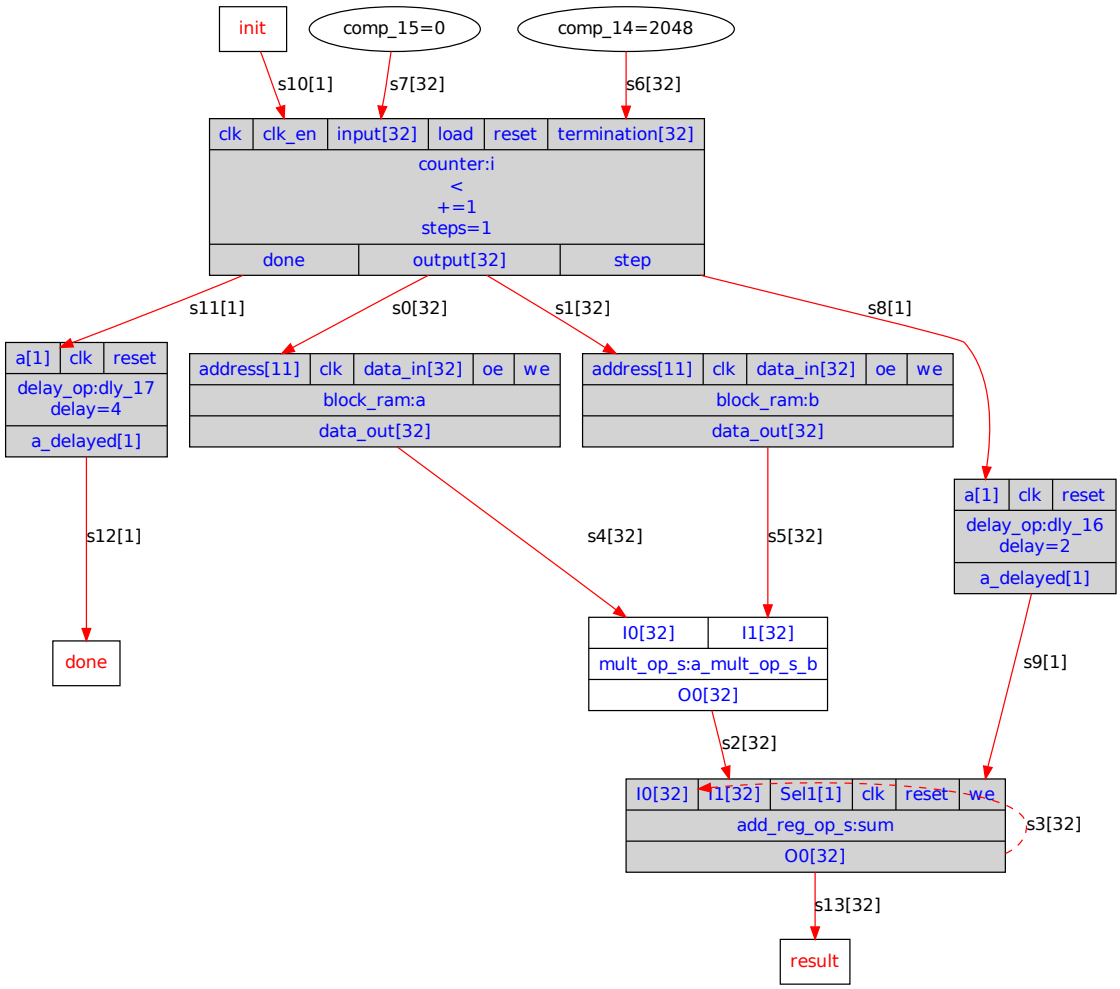


Figura 5.4: Visualização arquitetura balanceada.

não permitir o uso total da sintaxe baseada em *software*. Essas diferenças podem fazer a ferramenta de síntese gerar resultados incorretos e gerar arquiteturas que necessitam de uma grande quantidade de recursos de *hardware*.

Como citado anteriormente, LALPC tem como objetivo gerar sistemas de alto desempenho utilizando técnicas de paralelismo em laços de repetição em C. Neste caso, o programador pode utilizar as seguintes estruturas no processo de geração de *hardware*: verificação condicional IF em bloco e ternário, execução de operações aritméticas, acumuladores, vetores, deslocamento de bits, desdobramento de laços de repetição utilizando *loop unrolling*, memórias RAM customizadas para acessos simultâneos, diretivas de marcação utilizando *pragmas*, definir largura de bits da aplicação, definir pinos de saídas no *hardware*.

Como em outros compiladores, o compilador LALPC tem algumas limitações impedindo a utilização de todos os recursos disponíveis na linguagem C, sendo elas: dados em ponto

flutuante, chamadas de funções, *struct*, ponteiros, matrizes, recursão, entre outras. Para esses casos, é necessário que o programador efetue modificações no código de entrada adaptando-o para que ele seja processado corretamente pelo compilador. Tais limitações de entrada são muito comuns em ferramentas de síntese de alto nível. Modificar a sintaxe ou simplesmente reestruturar o código original costuma ser uma solução para esses casos. Entretanto, conhecer as funcionalidades e as limitações do compilador é de grande importância para obter sucesso no processo de geração de *hardware*.

5.6 *Pragmas*

Na Seção 2.3.2 descreve-se o uso das diretivas de marcação no processo de geração de soluções de alto desempenho. Diversos compiladores utilizam como entrada programas descritos em C com diretivas de marcação, podemos citar: OpenACC, OpenCL, OpenMP e Vivado HLS. O compilador LALPC também utiliza os *pragmas* para auxiliar o processo de compilação.

O processo de utilização das diretivas de marcação por meio dos *pragmas* tem como objetivo permitir ao programador definir algumas características específicas durante o processo de compilação. Neste caso, o uso dos *pragmas* modifica o *hardware* gerado pelo compilador, podendo gerar variações no *hardware* para o mesmo código de entrada.

O uso dos *pragmas* é justificado não apenas pelas características e limitações existentes da linguagem de alto nível - Linguagem C - para a plataforma alvo (VHDL), mas também permitir ao programador utilizar otimizações específicas para suprir uma necessidade em um projeto. As diferenças entre os paradigmas de programação limitam o uso de recursos importantes da arquitetura de *hardware*. Um exemplo entre a linguagem C e uma arquitetura em *hardware* é a necessidade de utilizar várias saídas para a mesma função. No compilador LALPC é possível definir com os *pragmas* quais variáveis poderão ter pinos de saída nos seus respectivos componentes. Outro exemplo é a necessidade de definir a largura de bits nos componentes de *hardware*, para este caso foi criado um *pragma* específico pelo fato de não existir tal informação na linguagem C.

A utilização dos *pragmas* depende exclusivamente de uma funcionalidade implementada pelo compilador. Nesse processo, o compilador tenta identificar a existência de um determinado *pragma* no código, associando-o à função implementada no compilador. A seguir são descritos os *pragmas* tratados pelo compilador LALPC atualmente:

- **#pragma alp unroll** - Aplica no código a otimização de *loop unrolling*, permitindo um

ganho em processamento paralelo na aplicação.

- **#pragma alp data_width** - Define largura de bits utilizados pelos componentes VHDL.
- **#pragma alp bit** - Define se um registrador será do tipo booleano.
- **#pragma alp multiport** - Verifica a existência de múltiplos acessos em uma memória permitindo criar uma memória RAM customizada.
- **#pragma alp out** - Criar pinos de saída nos componentes VHDL, isso facilita a análise dos dados e permite gerar um *hardware* com várias saídas.
- **#pragma alp delay** - Controla os ciclos de *clock* dos componentes, permitindo o sincronismo manual das operações.

5.7 Exploração de Espaço de Projeto

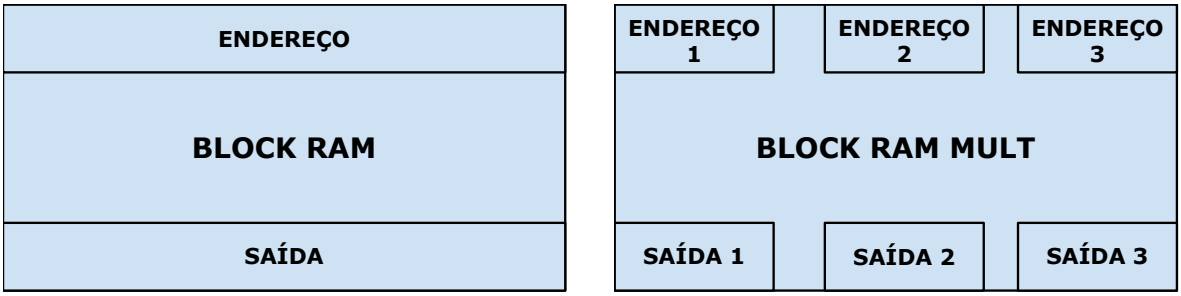
Em projetos de *hardware* ou *software* os programadores definem características importantes como prioridades. Essas características podem ser definidas como restrições que o programador deve respeitar para alcançar o sucesso do projeto. Em sistemas baseados em *hardware*, normalmente essas restrições são: quantidade limitada de recursos de *hardware*, consumo de energia, desempenho da aplicação, entre outras.

Como descrito anteriormente, a utilização dos *pragmas* permite alterar o comportamento do compilador a fim de gerar variações da arquitetura de *hardware*. Gerar arquiteturas diferentes com o mesmo código tem o intuito de atender as restrições do projeto. Os *pragmas* **multiport** e **unroll** são usados para aplicar técnicas de otimização visando acelerar o processamento da aplicação. Entretanto, o uso desses *pragmas* interfere na quantidade necessária de recursos do dispositivo reconfigurável. Neste caso, o uso destas é uma maneira de atender restrições em um projeto relacionado ao desempenho. Sendo assim, ao utilizar os *pragmas* multiport e unroll, o programador deixa explícita a necessidade de desempenho, e de maneira consciente, ele aceita que esses *pragmas* afetam significativamente a quantidade de recursos do dispositivo reconfigurável.

A seguir uma descrição detalhada dos *pragmas* multiport e unroll no compilador LALPC:

#pragma alp multiport - Esse *pragma* permite gerar um componente de memória RAM com múltiplas portas para acessos simultâneos. É uma técnica que melhora o desempenho final da aplicação em muitos casos, pois sem o acesso simultâneo é preciso utilizar um multiplexador para controlar o acesso ao conteúdo da memória sequencialmente (como mostrado no

Código 5.3). Ao utilizar esse *pragma*, durante o processo compilação, ele mapeia todos os acessos ao arranjo e cria uma memória customizada com os endereçamentos e os componentes que vão receber os valores, cabe ressaltar que a memória customizada não pertence à biblioteca de componentes VHDL. Na Figura 5.5a é possível visualizar a representação gráfica de uma memória RAM com uma única porta, enquanto na Figura 5.5b é possível visualizar uma memória RAM multiport gerada pelo compilador LALPC. A customização na memória permite três acessos simultâneos no mesmo ciclo de *clock*.



(a) Memória RAM de uma porta. (b) Memória RAM customizada de três portas.

Figura 5.5: Diferença entre memória RAM e a memória RAM customizada.

Tabela 5.1: Recursos em memórias multiport geradas pelo compilador LALPC.

Portas	Elem. Lóg	Comb.	Reg.	Mem. (bits)
1	55	43	46	32768
2	63	53	47	27648
4	69	58	55	24832
8	107	87	87	20096

Na Tabela 5.1 é possível constatar a diferença em termos de recursos de *hardware* necessários quando memórias customizadas são usadas. A diferença é significativa quando se compara a memória com apenas uma porta a uma memória que permite 2, 4 ou 8 leituras/escritas simultâneas. Isso ocorre porque, durante o processo de síntese, a ferramenta Quartus II move os valores da memória RAM, realocando-os para elementos lógicos (ALTERA CORPORATION., 2009a). Essa diferença pode ser constatada observando-se a última coluna da Tabela 5.1. Quanto mais portas na memória RAM, menos bits são alocados para esta memória e mais elementos lógicos são necessários para sintetizar o *hardware*.

Nos testes iniciais utilizando as memórias customizadas, a ferramenta Quartus II sintetizou facilmente as memórias com várias portas que eram apenas para leitura dos dados. Entretanto, o mesmo não ocorreu em memórias que permitiam a escrita de dados. A ferramenta necessitou de mais tempo para sintetizar os exemplos que utilizavam memórias com duas portas quando comparado com o mesmo exemplo com gravação em uma memória de uma única porta. Este

problema agravou ao tentar sintetizar memórias de escrita com três portas. Consequentemente, todos os testes realizados com memória que recebia gravação foram realizados com apenas memórias com duas portas. Mesmo assim diversas aplicações podem ser implementadas aproveitando das vantagens das memórias com várias portas.

#pragma alp unroll - Ao utilizar esse *pragma*, o compilador analisa as operações dentro do laço tentando identificar dependências entre as mesmas. Tais dependências impedem aplicar a técnica de desenrolar o laço de repetição pois uma operação depende do valor da interação anterior. Por consequência, o processamento desse laço de repetição é obrigatoriamente sequencial.

Não existindo nenhuma dependência de dados entre as operações, o compilador efetua cópias das operações internas do laço de repetição. O processo permite que as operações sejam executadas de maneira paralela. O resultado é um ganho significativo no processamento da aplicação quando comparado com a aplicação sequencial. Por outro lado, o processo de replicar as operações internas interfere na quantidade de recursos necessários do dispositivo reconfigurável, pois são necessários mais componentes de *hardware* para as novas operações.

O Código 5.6 exemplifica a utilização desse *pragma*, na Linha 4 temos a informação do *pragma* e o fator que deve ser aplicado para o desdobramento do laço de repetição. Já no Código 5.7 temos o resultado deste desdobramento por parte do *frontend* do compilador ROSE. Consequentemente, esta passa a ser a nova estrutura na representação intermediária do código que será utilizada na geração dos componentes de *hardware* no compilador LALPC.

Código 5.6: Repetição inicial

```
1 foo() {
2   int i, x[N], y[N], z[N];
3   #pragma alp multiport
4   #pragma alp unroll 2
5   for (i = 0; i < N; i++)
6     z[i] = x[i] + y[i];
7 }
```

Código 5.7: Repetição resultante

```
1 foo() {
2   int i, x[N], y[N], z[N];
3   for (i = 0; i < N; i+=2)
4     z[i]   = x[i]   + y[i];
5     z[i+1] = x[i+1] + y[i+1];
6 }
```

No Código 5.7 é possível constatar que a linha 5 foi gerada a partir da linha 4, porém com acessos diferentes aos vetores. Sendo assim, para a mesma iteração do laço de repetição é possível calcular duas operações de soma com valores diferentes usando o mesmo vetor o qual permite o processamento paralelo de duas iterações a cada ciclo.

Ainda nesse exemplo, ao aplicar o *pragma* unroll no código, esse gera acessos simultâneos para os vetores “x”, “y” e “z” que são representados pelos índices dos vetores “i” e “i+1”.

Desta maneira, ao criar os componentes de *hardware* é possível combinar o *pragma* multiport para acesso simultâneo a esta memória. No Código 5.6 na Linha 3 temos o *pragma* que indica ao compilador LALPC customizar as memórias quando encontrar acessos simultâneos para elas independente de ser de leitura ou escrita.

A combinação dos dois *pragmas* supracitados acarreta em ganho significativo de desempenho para a aplicação, principalmente se comparado ao processamento de uma aplicação sequencial. Isso é possível pois temos o processamento paralelo das operações internas do laço de repetição combinado com as leituras simultaneas dos valores da memória.

Capítulo 6

RESULTADOS

Neste capítulo são apresentados os resultados experimentais obtidos com o compilador LALPC. Estes resultados demonstram que ferramentas com o mesmo propósito geram arquiteturas distintas para o mesmo código de entrada. Para os testes, os compiladores LALPC, LALP e LegUp foram submetidos a testes utilizando um conjunto de *benchmarks*. Cada *benchmark* possui uma característica distinta que permite testar os compiladores. O objetivo das comparações foi apenas o de verificar o desempenho das técnicas implementadas no compilador LALPC, e não comparar a eficácia das ferramentas.

Na Seção 6.1 é apresentada a lista com os *benchmarks* utilizados como entrada nos testes dos compiladores. Na Seção 6.2 é apresentado um comparativo entre os compiladores. Na Seção 6.3 é apresentada uma introdução dos testes iniciais das técnicas desenvolvidas neste trabalho. Nos testes foram obtidas informações relacionadas ao tempo de execução e aos recursos necessários para síntese dos *benchmarks*. Vale ressaltar que os testes foram divididos em duas partes, na primeira são apresentados testes sem a utilização dos *pragmas*. Esta parte tem como objetivo verificar se os resultados obtidos com LALPC são similares aos obtidos com LALP. Pois, como citado anteriormente, LALPC dispões das técnicas de balanceamento existentes em LALP, os resultados são apresentados na Seção 6.2. Já na segunda parte dos testes, na Seção 6.4, são apresentados os resultados dos testes com os *benchmarks* que permitem a utilização dos *pragmas*. Nessa segunda etapa, o objetivo foi verificar se as técnicas de otimização implementadas produzem uma melhora de desempenho quando comparado com a versão sem os *pragmas*.

6.1 *Benchmarks*

O método para avaliar o desempenho do *hardware* gerado pelo compilador LALPC é baseado na comparação de desempenho e recursos com o *hardware* gerado pelos compiladores LALP e LegUp. Uma lista com *benchmarks* com características distintas representam os códigos de entrada para estes compiladores, sendo eles, os algoritmos de áudio *ADPCM Coder* e *ADPCM Decoder* (GUTHAUS et al., 2001); o algoritmo de processamento de imagem *Sobel* (Texas Instruments Incorporated, 2003b); os algoritmos *Dotprod*, *Max* e *Vecsum* utilizados em DSPs (Texas Instruments Incorporated, 2003a); o algoritmo *Accumulator* presente dentro da pasta de testes do compilador LegUp (CANIS et al., 2011). Para o processo de síntese foi utilizada a ferramenta Quartus II 13.0 (64-bit) e o dispositivo reconfigurável (FPGA[1]) Altera EP4CGX150DF31C7 da Família Cyclone IV GX.

No trabalho proposto em LALP, o autor também demonstra o ganho de desempenho da ferramenta comparando-a com outras ferramentas de síntese de alto nível. Nos testes com os mesmos *benchmarks*, o compilador LALPC demonstrou ter resultados bastante semelhantes aos obtidos com o LALP (MENOTTI et al., 2012). O compilador LALPC se utiliza de técnicas reimplementadas do compilador LALP, o que justifica os resultados semelhantes quando não se utilizam os *pragmas*.

6.2 LALPC Comparado ao LALP e LegUp

Conforme citado no Capítulo 5, o compilador LALPC utiliza as técnicas de escalonamento e o mesmo princípio para criação dos componentes de *hardware* proposto em LALP. Neste caso, ao utilizar as técnicas de geração de *hardware* disponível no *backend* do compilador LALP, esperava-se alcançar resultados semelhantes no compilador LALPC. Os resultados de LALP demonstram a capacidade de explorar o *loop pipelining* em *hardware*.

Antes de abordar as técnicas de otimização em LALPC são apresentados os resultados obtidos com o compilador LALPC sem o uso dos *pragmas*. A seguir são descritos os resultados de desempenho e recursos necessários para os *benchmarks* comparando com LALP e LegUp. Os resultados são descritos na Tabela 6.1. Esta tabela contém as informações de recursos necessários para síntese e tempo de execução em *hardware*. Enquanto isso, na Figura 6.1 é visível o comparativo do tempo de execução entre os compiladores por *benchmark*, enquanto na Figura 6.2 é visível o comparativo dos recursos necessários para síntetizá-los.

[1] Do inglês: Field-Programmable Gate Array

Nas Figuras 6.1 e 6.2 são visíveis as semelhanças dos valores entre LALP e LALPC. Apesar de praticamente não haver ganho em relação aos resultados obtidos com LALP, cabe ressaltar a facilidade em se descrever os algoritmos em C. Além disso, parâmetros de sincronismo informados manualmente em LALP puderam ser inferidos automaticamente no compilador LALPC. Sendo assim, esses benefícios citados acima confirmam um avanço alcançado com o compilador LALPC.

Tabela 6.1: Recursos ocupados e tempo de execução nos *benchmarks*.

Benchmark	Comp	Elementos Lógicos	Comb	Reg	Memória (Bits)	Tempo Exec. (us)
Accumulator	LEGUP	658	564	541	640	0.44
	LALP	163	109	138	512	0.11
	LALPC	159	109	139	512	0.10
ADPCM	LEGUP	1067	995	661	36128	172.10
Coder	LALP	800	638	37	40960	79.72
	LALPC	986	673	837	41460	119.30
ADPCM	LEGUP	1049	913	683	52512	95.97
Decoder	LALP	507	388	464	41090	16.61
	LALPC	628	420	574	41248	39.00
Dotprod	LEGUP	852	696	590	131072	152,09
	LALP	95	81	65	131072	28.92
	LALPC	118	102	82	131072	28.50
Max	LegUP	294	272	234	65356	31.46
	LALP	66	47	62	65356	12.79
	LALPC	94	75	84	65356	12.89
Sobel	LEGUP	1266	1156	820	6400	4.16
	LALP	797	683	504	8269	5.67
	LALPC	1158	754	991	8435	4.40
Vecsum	LEGUP	891	788	551	196608	117.85
	LALP	101	53	67	196608	18.95
	LALPC	124	74	83	196608	17.20

A seguir são apresentados os testes iniciais utilizando os *pragmas* apresentados na Seção 5.6.

6.3 Testes Iniciais

A partir dos testes citados na Seção 6.2, observamos um avaço relacionado ao nível de abstração no código de entrada. Neste ponto do trabalho alguns testes inciais foram executados para avaliar as otimizações propostas.

Os *pragmas* citados no capítulo anterior resultam na aplicação das otimizações no código

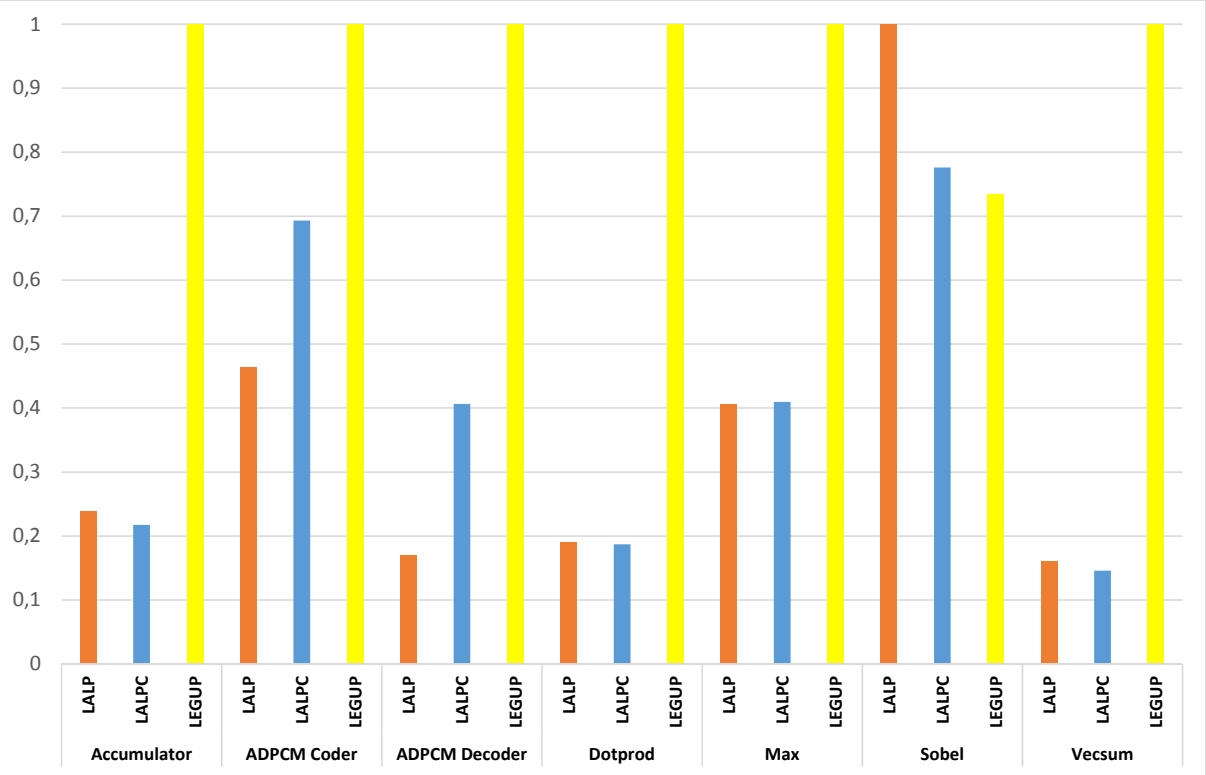


Figura 6.1: Comparativo do tempo de execução normalizado com todos os *benchmarks* entre os compiladores LegUp, LALP e LALPC.

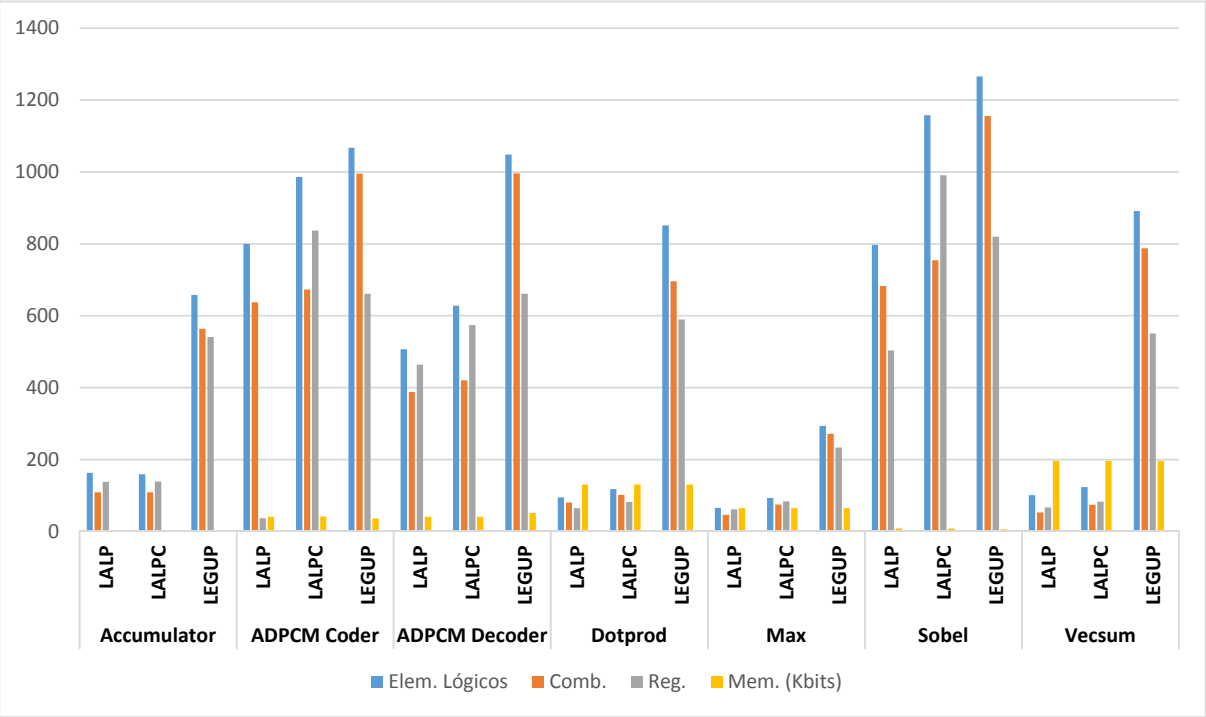


Figura 6.2: Comparativo de recursos de *hardware* necessários em todos os *benchmarks* entre os compiladores LegUp, LALP e LALPC.

de entrada, quando possível. Para avaliar inicialmente a eficiência das técnicas implementadas no compilador, foi utilizado o *benchmark Sobel*. Os outros *benchmarks* dispõem de apenas um acesso ao vetor de leitura, neste caso, é necessário utilizar o *pragma* unroll para criar os novos acessos ao vetor permitindo a utilização do *pragma* multiport. Entretanto, o *benchmark Sobel* realiza 8 acessos de leitura no vetor de leitura e um único acesso de gravação no vetor de saída com o resultado do processamento. Esses acessos na memória permitem a execução dos testes utilizando o *pragma* multiport sem a necessidade de utilizar esse combinado com o *pragma* unroll. Sendo assim, o *benchmark Sobel* é ideal para os testes iniciais das técnicas implementadas.

Como citado anteriormente, no *benchmark Sobel* existem 8 acessos no mesmo vetor, no Algoritmo 6.1 da Linha 8 a 10 é possível visualizar os acessos. Sem o uso do *pragma* multiport, o *hardware* gerado necessida de um multiplexador para gerenciar o controle de acesso na memória. O processo torna a leitura dos dados um processo sequencial. Na Figura 6.3 é possível visualizar esse processo, do qual para cada iteração do contador são necessários 8 ciclos de *clock* para obter os valores da memória.

Nos testes com a memória multiport é visível o resultado positivo no ganho de desempenho da arquitetura gerada com esta otimização. Ao utilizar o *pragma* multiport o compilador transforma essa memória RAM com uma única porta de endereçamento e de saída, para uma memória RAM com 8 portas de endereçamento e 8 portas de saída. Consequentemente, essa memória customizada permite que todos os acessos sejam executados em um único ciclo *clock* por iteração do contador. Na Figura 6.4 é possível visualizar o resultado do processo de leitura dos valores na memória. Com as leituras em paralelo o ganho de desempenho na aplicacão é expressivo.

Código 6.1: Algoritmo *Sobel* para processamento de imagens

```
1  sobel() {
2    int out[100];
3    int H,O,V,i;
4    int i00,i01,i02;
5    int i10,i12;
6    int i20,i21,i22;
7    for (i = 0; i < 78; i++)
8      i00 = in[i];      i01 = in[i+1];    i02=in[i+2];
9      i10 = in[i+10];   i12 = in[i+12];
10     i20 = in[i+20];   i21 = in[i+21];   i22=in[i+22];
11     H = ((-i00) + (-i01 -i01)) + (((-i02) + i20) + ((i21 + i21) + i22));
12     V = ((-i00) + i02) + (((-i10 -i10) + (i12 + i12)) + ((-i20) + i22));
13     if (H<0)
```

```
14          H = -H;
15        if (V<0)
16          V = -V;
17        O = H + V;
18        if (O > 255)
19          O = 255;
20        out[i] = O;
21  }
```

Outra otimização possível de ser aplicada nesse exemplo é a de *loop unrolling* com fator 2, pois não existem dependências dentro do *kernel* do laço de repetição. Ao combinar essa otimização com o *pragma* multiport o *hardware* gerado garante um ganho do tempo de execução quando se compara no *hardware* gerado apenas com o *pragma* multiport. Essa combinação duplica todas as operações existentes dentro do *kernel* do laço, gerando 16 acessos simultâneos na memória contendo a imagem de entrada e outros 2 acessos de gravação na memória de saída.

Testes inciais demonstraram o potencial das técnicas de otimização implementadas no compilador LALPC. Neste caso, para testar as otimizações o *benchmark Sobel* é avaliado sem a utilização de *pragmas* de otimização, utilizando o *pragma* responsável pela técnica multiport e por fim, o resultado utilizando os *pragmas* multiport e o *pragma* unroll com fator 2. Na Tabela 6.2 são apresentados os restultados dos processos de síntese das arquiteturas geradas pelos compiladores LegUp, LALP e LALPC. Por fim, na Figura 6.5 é possível visualizar o impacto no tempo de execução das arquiteturas geradas pelo compilador. Este gráfico permite demonstrar o ganho em otimização alcançada pelo compilador LALPC comparando-o com ele mesmo, e também, com os outros compiladores. Sendo assim, o ganho de desempenho é expressivo e confirma a capacidade de otimização das técnicas propostas em uma aplicação real utilizando o compilador.

Tabela 6.2: Recursos ocupados e tempo de execução no *benchmark Sobel*.

Comp	Elementos Lógicos	Comb	Reg	Memória (Bits)	Tempo Exec. (us)
LEGUP	1266	1156	820	6400	4,16
LALP	797	683	504	8269	5,7
LALPC	1158	754	991	8435	4,40
LALPC *	922	643	549	36387	0,78
LALPC +	7263	6201	4786	23902	0,45

* *pragma* multiport
\+ *pragmas* multiport e unroll com fator 2

As técnicas de otimização afetam tanto o desempenho da arquitetura como a quantidade de

Ciclo	Iteração 0	Iteração 1	Iteração 2
0	Acesso Mem. 1		
1	Acesso Mem. 2		
2	Acesso Mem. 3		
3	Acesso Mem. 4		
4	Acesso Mem. 5		
5	Acesso Mem. 6		
6	Acesso Mem. 7		
7	Acesso Mem. 8		
8		Acesso Mem. 1	
9		Acesso Mem. 2	
10		Acesso Mem. 3	
11		Acesso Mem. 4	
12		Acesso Mem. 5	
13		Acesso Mem. 6	
14		Acesso Mem. 7	
15		Acesso Mem. 8	
16			Acesso Mem. 1
17			Acesso Mem. 2
18			Acesso Mem. 3
19			Acesso Mem. 4
20			Acesso Mem. 5
21			Acesso Mem. 6
22			Acesso Mem. 7
23			Acesso Mem. 8

Figura 6.3: Exemplo escalonamento para leitura na memória no *benchmark Sobel*.

recursos necessários. Na Tabela 6.2, além de dispor do tempo de execução, também é possível visualizar os recursos necessários para sintetizar as arquiteturas geradas pelos compiladores. Sendo assim, é notável a diferença de recursos necessários quando é utilizado um ou dois *pragmas* de otimização em LALPC. Na Figura 6.5 é possível visualizar os recursos necessários de *hardware* comparando o teste sem o uso dos *pragmas* com o teste utilizando dois *pragmas* de

Ciclo	Iteração 0	Iteração 1	Iteração 2
0	Acessos à Mem.		
1		Acessos à Mem.	
2			Acessos à Mem.

Figura 6.4: Exemplo escalonamento para leitura na memória no *benchmark Sobel* utilizando o *pragma* multiport.

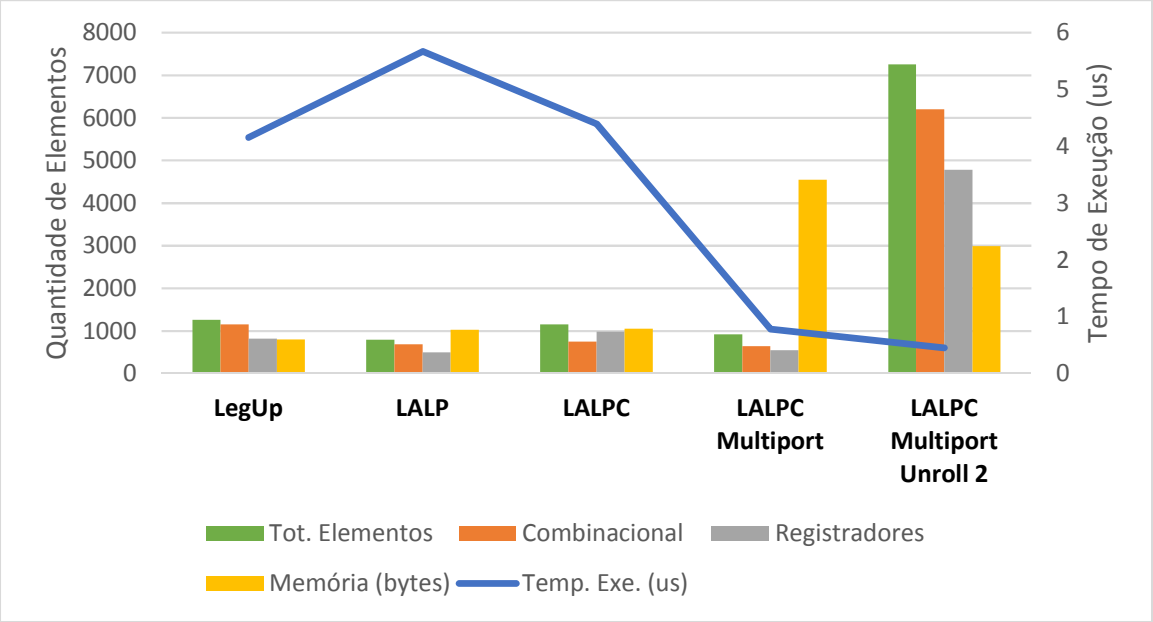


Figura 6.5: Comparativo do tempo de execução e de recursos no *benchmark Sobel* entre os compiladores LegUp, LALP e LALPC.

otimização no *benchmark Sobel*. Cabe ressaltar que ao utilizar os *pragmas*, pode-se inviabilizar sua utilização dependendo da limitação de recursos do dispositivo reconfigurável utilizado no projeto.

6.4 Testes com os *Pragmas*

Após a conclusão dos testes iniciais de avaliação dos *pragmas* com *benchmark Sobel* foram efetuados novos testes com outros *benchmarks* citados anteriormente. Entretanto, aplicações que dispõem de laços com dependência de dados não permite a aplicação do *pragma* unroll. Por existir esse tipo de dependência, alguns *benchmarks* da lista não formam avaliados com o uso dos *pragmas* de otimização. A seguir são descritos os resultados dos *benchmarks* que permitiram a utilização dos *pragmas* de otimização.

6.4.1 *Accumulator*

Nesta Subseção são apresentados os resultados dos testes com o *benchmark Accumulator*. A Tabela 6.3 apresenta os resultados dos testes entre os três compiladores, e também, o teste utilizando os *pragmas* de otimização.

Tabela 6.3: Recursos ocupados e tempo de execução no *benchmark Accumulator*.

Comp	Elementos Lógicos	Comb	Reg	Memória (Bits)	Tempo Exec. (us)
LEGUP	1035	927	668	640	0,46
LALP	153	114	142	0	0,07
LALPC	149	114	143	0	0,07
LALPC *	242	211	176	0	0,05

* *pragmas* multiport e unroll com fator 2

6.4.2 *Dotprod*

Nesta Subseção são apresentados os resultados dos testes com o *benchmark Dotprod*. Na Tabela 6.4 apresenta os resultados dos testes entre os três compiladores, e também, os testes utilizando os *pragmas* de otimização.

Tabela 6.4: Recursos ocupados e tempo de execução no *benchmark Dotprod*.

Comp	Elementos Lógicos	Comb	Reg	Memória (Bits)	Tempo Exec. (us)
LEGUP	802	685	526	131072	130,57
LALP	96	82	65	131072	26,79
LALPC	120	104	83	131072	26,47
LALPC *	211	196	114	131072	13,58
LALPC -	395	380	177	131072	7,06
LALPC +	762	749	304	131072	3,43

* *pragmas* multiport e unroll com fator 2
- *pragmas* multiport e unroll com fator 4
+ *pragmas* multiport e unroll com fator 8

6.4.3 *Vecsum*

Nesta Subseção são apresentados os resultados dos testes com o *benchmark Vecsum*. Na Tabela 6.5 descreve os resultados dos testes entre os três compiladores e também, o teste utilizando os *pragmas* de otimização.

Tabela 6.5: Recursos ocupados e tempo de execução no *benchmark Vecsum*.

Comp	Elementos Lógicos	Comb	Reg	Memória (Bits)	Tempo Exec. (us)
LEGUP	903	802	551	196608	135,90
LALP	101	53	67	196608	17,52
LALPC	124	75	84	196608	17,87
LALPC *	270	186	243	196608	8,88

* *pragmas* multiport e unroll com fator 2

6.4.4 Ganho de Desempenho

A Tabela 6.6 dispõe os resultados relacionados com o *speedup* para os testes utilizando os *pragmas*. Os resultados para os *benchmarks* testados e comparados com o compilador LALP obtivemos a média do *speedup* de 6.03x, enquanto para o compilador LegUp esse valor é de 17.27x. Os testes demonstraram a eficiência das técnicas de otimização implementadas neste trabalho. Vale ressaltar também que houve aumento no resultado da quantidade de recursos necessários para sintetizar o *hardware* gerado com os *pragmas*. Concluíndo, a decisão de utilizar os *pragmas* é toda do programador, pois ele conhece as limitações do projeto e os recursos disponíveis no dispositivo reconfigurável. Sendo assim, quando a quantidade de recurso do *hardware* reconfigurável não for uma restrição é evidente o ganho de desempenho proporcionado pela aplicação dos *pragmas*.

Tabela 6.6: *Speedup* obtained using optimization *pragmas*.

Benchmark	Melhor Tempo LALPC (us)	Tempo LALP (us)	Speedup	Tempo LegUp (us)	Speedup
Accumulator	0.05	0.11	**2.37**	0.44	**9.34**
Dotprod	3.95	28.92	**7.32**	152.09	**38.50**
Sobel	0.45	5.67	**12.49**	4.16	**9.16**
Vecsum	9.73	18.95	**1.95**	117.85	**12.11**

Capítulo 7

CONCLUSÃO

Neste trabalho foi apresentada uma metodologia para geração de *hardware* a partir de algoritmos descritos em C, baseada no compilador LALP. Entretanto, a abordagem utilizada neste trabalho se difere do compilador LALP basicamente em dois pontos: (i) LALPC permite o uso de códigos descritos na liguagem C para entrada do compilador; (ii) técnicas de otimização por meio de *pragmas*.

A primeira diferença relacionada à linguagem do código de entrada é importante. Considerando outros compiladores para síntese de alto nível, a maioria utiliza a linguagem C como entrada por ser uma linguagem mais difundida no meio computacional. Apesar das limitações em relação ao subconjunto da linguagem C suportado, o compilador LALPC gerou resultados com uma evolução de desempenho significativa em relação ao compilador LALP.

O uso dos *pragmas* é a segunda diferença entre os compiladores LALP e LALPC. O compilador LALP dispõe de diretivas de marcação que permitem ao programador controlar os ciclos de *clock* no código de entrada, possibilitando maior controle sobre o escalonamento das operações em *hardware*. Já no compilador LALPC, as diretivas de marcação (*pragmas*) permitem o controle dos ciclos de *clock* semelhante ao compilador LALP. Entretanto, o uso de *pragmas* no código fonte possibilita a rápida exploração do espaço de projeto, pois com pequenas modificações na entrada são obtidas arquiteturas diferentes, dentre as quais se pode selecionar a mais adequada em termos de desempenho e recursos ocupados no dispositivo. Consequentemente, as diretivas de marcação existentes no compilador LALPC abrangem uma quantidade maior de funcionalidades permitindo que o programador utilize técnicas diferentes no processo de compilação.

Nos testes, desconsiderando os *pragmas* de otimização, o compilador LALPC obteve resultados semelhantes ao compilador LALP. Mesmo que para esse caso não haja um ganho relacio-

nado ao desempenho da aplicação, vale ressaltar que existe um ganho relacionado ao aumento no nível de abstração da linguagem aceita pelo compilador. É importante frisar que para estes testes, o compilador LALPC não precisou do auxílio das diretivas de marcação para auxiliar no sincronismo das operações como ocorre em LALP. Para este caso, durante o processo de compilação, LALP emite mensagens para o usuário informando-o a necessidade de inserção de atrasos nos componentes, quando necessário. Já no compilador LALPC esse processo acontece automaticamente. Como pode se verificar, essas características citadas acima tornam mais simples o uso do compilador LALPC quando comparado com o compilador LALP.

As técnicas de otimização existentes no compilador LALPC são a principal contribuição deste trabalho em relação ao compilador LALP. A técnica de *loop unrolling* permitiu explorar ainda mais o processamento paralelo em laços de repetição. Ao replicar as operações dentro do kernel do laço de repetição, o ganho relacionando ao tempo de execução do *hardware* foi significativo. Enquanto a outra técnica permite a geração de memórias customizadas. Esta permite gerar memórias com várias portas para acesso simultâneo na memória. O ganho de desempenho utilizando essa técnica também foram expressivos. Valendo lembrar que em LALPC é possível combinar as técnicas para melhorar o tempo de execução da aplicação. Nos testes utilizando as técnicas de otimização, os resultados apresentados no capítulo anterior demonstrou o potencial da utilização dos *pragmas* no compilador LALPC. As implementações obtidas com LALPC obtiveram *speedups* médios de 6,03x e 17,27x comparadas àquelas obtidas com o LALP e o LegUp respectivamente. É importante ressaltar que como em LALP, em LALPC é possível alterar a implementação sempre que o resultado não for satisfatório utilizando as diretivas de marcação.

Resumidamente, o trabalho desenvolvido atendeu às expectativas iniciais de se criar uma ferramenta de síntese de alto nível, aproveitando as técnicas existentes no compilador LALP, porém usando como entrada programas descritos em linguagem C. As técnicas de memórias customizadas e *loop unrolling*, implementadas com o *frontend* do compilador ROSE, permitiram gerar arquiteturas com desempenho superior ao obtido com o compilador LALP. Diversas técnicas implementadas no compilador ROSE podem ainda ser aplicadas à geração de *hardware* no contexto deste trabalho. Além das transformações relativas aos *loops*, as técnicas de reuso de dados podem ser exploradas para aumentar o paralelismo sem a necessidade de mais acessos à memória.

O código fonte do compilador LALPC encontra-se disponível em **http://lalp.dc.ufscar.br/**.

7.1 Trabalhos Futuros

O trabalho apresentado oferece oportunidades para novas pesquisas na área de compiladores para arquiteturas reconfiguráveis. A seguir, são apresentadas sugestões de trabalhos futuros para estender e aperfeiçoar os avanços alcançados pelo compilador LALPC.

- Implementar no compilador LALPC outras estruturas existentes na liguagem C sendo elas: while, structs, funções, etc. Essas estruturas permitirão ao compilador ser mais completo, e também, utilizar como entrada do compilador exemplos complexos.

- Implementar no compilador LALPC operações com valores em ponto flutuante. O trabalho apresentado por Oliveira J. M. P. Cardoso (2013) permite o uso de dados em ponto flutuante no compilador LALP.

- As memórias utilizadas pelo compilador LALPC são implementadas com os recursos disponíveis na placa do dispositivo reconfigurável. Alguns destes dispositivos dispõe de memórias externas que podem ser utilizadas sem a necessidade de serem implementadas com os recursos reconfiguráveis. Permitir ao programador utilizar desse recurso proporcionará um ganho na exploração de projeto no compilador.

- A técnica de *loop unroll* é implementada no *frontend* do compilador LALPC utilizando as técnicas disponíveis no compilador ROSE. Como citado na Seção 5.1 o compilador ROSE contempla outras técnicas voltadas para otimizações de laços de repetição. Elas podem ser implementadas no compilador LALPC semelhante à técnica *loop unroll* permitindo gerar *hardware* com caracteristicas específicas, as quais permitirão um avanço na área relacionada à exploração de espaço de projeto do compilador.

- Aplicar outras técnicas existentes no compilador ROSE. Por exemplo resolver expressões matemáticas. Esta permitirá gerar *hardware* mais otimizado, pois em alguns casos, resolver as operações matemáticas resulta em menos elementos lógicos necessários para implementar o *harware*.

- Implementar no compilador LALPC relatórios completos com informações relacionadas à arquitetura gerada. Essas informações dispõe de dados relacionados à quantidade de recursos, frequência, tempo de execução entre outras. O compilador LegUp dispõe de algo semelhante, esses relatórios evitam que o programador necessite de criar e sintetizar manualmente projetos nas ferramentas de síntese de alto nível. Essa funcionalidade pretende uma melhora no processo de geração automática de *hardware*.

- Implementar a geração automática de aceleradores utilizando *hardware*. Algumas estruturas baseadas em *software* podem ser difíceis de serem implementadas em FPGA[1]. Para estes casos o particionamento do processamento entre *hardware*/*software* é uma solução. Permitir gerar um acelerador de uma determinada parte do código enquanto o restante é processado em uma CPU[2] é uma tendência na área de processamento de alto desempenho.

[1] Do inglês: Field-Programmable Gate Array
[2] Do inglês: Central Processing Unit

REFERÊNCIAS

ALTERA CORPORATION. *Best HDL Design Practices for Timing Closure (OHDL1130)*. [S.l.], 2009. Disponível em: <http://www.altera.com/customertraining/webex/TimingClosureBest/presentation.html>.

ALTERA CORPORATION. *Nios II C2H Compiler - User Guide - Revision 9.1*. [S.l.], 2009. Disponível em: <http://www.altera.com/literature/ug/ug_nios2_c2h_compiler.pdf>.

BACON, D. F.; GRAHAM, S. L.; SHARP, O. J. Compiler transformations for high-performance computing. *ACM Comput. Surv.*, ACM Press, New York, NY, USA, v. 26, n. 4, p. 345–420, 1994. ISSN 0360-0300.

BACON, D. F.; RABBAH, R.; SHUKLA, S. Fpga programming for the masses. *Communications of the ACM*, ACM, v. 56, n. 4, p. 56–63, 2013.

BADAL, A.; BADANO, A. Accelerating monte carlo simulations of photon transport in a voxelized geometry using a massively parallel graphics processing unit. *Medical physics*, American Association of Physicists in Medicine, v. 36, n. 11, p. 4878–4880, 2009.

BEZERRA, E. A. Selecting a hardware description language for the design of an on-board scientific instrument processing module. In: *2nd UK ACM SIGDA Workshop on Electronic Design Automation*. [S.l.: s.n.], 2002. p. 16–17.

BUYUKKURT, B.; GUO, Z.; NAJJAR, W. Impact of loop unrolling on area, throughput and clock frequency in roccc: C to vhdl compiler for fpgas. *Reconfigurable Computing: Architectures and Applications*, Springer, p. 401–412, 2006.

CANIS, A. et al. Legup: high-level synthesis for fpga-based processor/accelerator systems. In: ACM. *Proceedings of the 19th ACM/SIGDA international symposium on Field programmable gate arrays*. [S.l.], 2011. p. 33–36.

CARDOSO, J. et al. *Compilation and Synthesis for Embedded Reconfigurable Systems: An Aspect-Oriented Approach*. Springer London, Limited, 2013. ISBN 9781461448938. Disponível em: <http://books.google.com.br/books?id=PRlgLwEACAAJ>.

CARDOSO, J. M. P. *Compilação de Algoritmos em Java para Sistemas Computacionais Reconfiguráveis com Exploração do Paralelismo ao Nível das Operações*. Tese (Doutorado) — Universidade Técnica de Lisboa, 2000.

CARDOSO, J. M. P.; NETO, H. C. Compilation for FPGA-based reconfigurable hardware. *IEEE Design & Test of Computers*, v. 20, n. 2, p. 65–75, 2003. ISSN 0740-7475.

CETUS PROJECT. *The Cetus Compiler Manual*. [S.l.], 2011. Disponível em: <http://cetus.ecn.purdue.edu/Documentation/manual/ch01.html>.

CHAPMAN, B.; JOST, G.; PAS, R. V. D. *Using OpenMP: portable shared memory parallel programming*. [S.l.]: MIT press, 2007.

CHE, S. et al. Accelerating compute-intensive applications with gpus and fpgas. In: IEEE. *Application Specific Processors, 2008. SASP 2008. Symposium on*. [S.l.], 2008. p. 101–107. ISBN 978-1-4244-2333-0.

COMPTON, K.; HAUCK, S. Reconfigurable computing: a survey of systems and software. *ACM Comput. Surv.*, ACM Press, New York, NY, USA, v. 34, n. 2, p. 171–210, 2002. ISSN 0360-0300.

CONG, J. et al. High-level synthesis for fpgas: From prototyping to deployment. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, v. 30, n. 4, p. 473–491, April 2011. ISSN 0278-0070.

COPE, B. et al. Implementation of 2d convolution on fpga, gpu and cpu. *Imperial College Report*, 2006.

COUSSY, P. et al. An Introduction to High-Level Synthesis. *IEEE Design & Test*, IEEE Computer Society Press, v. 26, n. 4, p. 8–17, 2009.

COUTINHO, J. et al. Deriving resource efficient designs using the reflect aspect-oriented approach. In: BRISK, P.; COUTINHO, J. F.; DINIZ, P. (Ed.). *Reconfigurable Computing: Architectures, Tools and Applications*. Springer Berlin Heidelberg, 2013, (Lecture Notes in Computer Science, v. 7806). p. 226–228. ISBN 978-3-642-36811-0. Disponível em: <http://dx.doi.org/10.1007/978-3-642-36812-7_29>.

COUTINHO, J. G. F.; LUK, W. Source-directed transformations for hardware compilation. In: IEEE. *Field-Programmable Technology (FPT), 2003. Proceedings. 2003 IEEE International Conference on*. [S.l.], 2003. p. 278–285.

D'AMORE, R. *VHDL: Descrição e Síntese de Circuitos Digitais*. [S.l.]: LTC, 2005.

DAVE, C. et al. Cetus: A source-to-source compiler infrastructure for multicores. *Computer*, IEEE, v. 42, n. 12, p. 36–42, 2009.

DOOLEY, I. J. *Automated source-to-source translations to assist parallel programmers*. Tese (Doutorado) — University of Illinois at Urbana-Champaign, 2006.

EL-GHAZAWI, T. et al. The promise of high-performance reconfigurable computing. *IEEE Computer*, v. 41, n. 2, p. 69–76, 2008.

FEIST, T. Vivado design suite. *Xilinx, White Paper Version*, v. 1, 2012.

FOWERS, J. et al. A performance and energy comparison of fpgas, gpus, and multicores for sliding-window applications. In: ACM. *Proceedings of the ACM/SIGDA international symposium on Field Programmable Gate Arrays*. [S.l.], 2012. p. 47–56.

GAJSKI, D.; RAMACHANDRAN, L. Introduction to high-level synthesis. *Design & Test of Computers, IEEE*, IEEE, v. 11, n. 4, p. 44–54, 1994.

GARLAND, M. et al. Parallel computing experiences with cuda. *Micro, IEEE*, IEEE, v. 28, n. 4, p. 13–27, 2008.

GONSALES, A. D. *Projeto de uma Nova Arquitetura de FPGA para Aplicações BIST e DSP*. Dissertação (Mestrado) — Departamento de Ciência da Computação, Universidade Federal do Rio Grande do Sul, 2002.

GUTHAUS, M. R. et al. MiBench: A free, commercially representative embedded benchmark suite. In: *WWC '01: Proceedings of the Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop*. Washington, DC, USA: IEEE Computer Society, 2001. p. 3–14. ISBN 0-7803-7315-4.

HADJIS, S. et al. Impact of fpga architecture on resource sharing in high-level synthesis. In: ACM. *Proceedings of the ACM/SIGDA international symposium on Field Programmable Gate Arrays*. [S.l.], 2012. p. 111–114.

JACOB, F. et al. Cudacl: A tool for cuda and opencl programmers. In: *High Performance Computing (HiPC), 2010 International Conference on*. [S.l.]: IEEE, 2010. p. 1–11.

JACQUARD COMPUTING INC. *ROCCC 2.0 User's Manual - Revision 0.6*. [S.l.], 2011. Disponível em: <http://www.jacquardcomputing.com/downloads/documentation/UserManual-0.6.pdf>.

JOHNSON, T. et al. Experiences in using cetus for source-to-source transformations. *Languages and Compilers for High Performance Computing*, Springer, p. 922–922, 2005.

JR, A. C. et al. Aplicando model-driven development à plataforma gpgpu. 2008.

KARIMI, K.; DICKSON, N.; HAMZE, F. A performance comparison of cuda and opencl. *arXiv preprint arXiv:1005.2581*, 2010.

KASIM, H. et al. Survey on parallel programming model. *Network and Parallel Computing*, Springer, p. 266–275, 2008.

KESSLER, C.; KELLER, J. Models for parallel computing: Review and perspectives. *Mitteilungen-Gesellschaft für Informatik eV, Parallel-Algorithmen und Rechnerstrukturen*, v. 24, 2007.

KING, A. *Compile-time Optimization of a Scientific Library through Domain-Specific Source-to-Source Translation*. Tese (Doutorado) — UNIVERSITY OF CALIFORNIA, SAN DIEGO, 2012.

KIRK, D. Nvidia cuda software and gpu parallel computing architecture. In: *ISMM*. [S.l.: s.n.], 2007. v. 7, p. 103–104.

LATTNER, C. Introduction to the llvm compiler system. In: *Proceedings of International Workshop on Advanced Computing and Analysis Techniques in Physics Research, Erice, Sicily, Italy*. [S.l.: s.n.], 2008.

LEE, S. J.; RAILA, D. K.; KINDRATENKO, V. V. Llvm-chimps: Compilation environment for fpgas using llvm compiler infrastructure and chimps computational model. *Reconfigurable Systems Summer Institute (RSSI 2008), Champaign, USA*, 2008.

LIAO, C. et al. A ROSE-Based OpenMP 3.0 Research Compiler Supporting Multiple Runtime Libraries. *Lecture Notes in Computer Science*, v. 6132, p. 15–28, 2010.

LIAO, C. et al. Semantic-aware automatic parallelization of modern applications using high-level abstractions. *International Journal of Parallel Programming*, Springer, v. 38, n. 5-6, p. 361–378, 2010.

LIN, M.; LEBEDEV, I.; WAWRZYNEK, J. OpenRCL: Low-Power High-Performance Computing with Reconfigurable Devices. In: *Field Programmable Logic and Applications (FPL), 2010 International Conference on*. [S.l.: s.n.], 2010. p. 458 –463. ISSN 1946-1488.

LIU, Y.; ZHANG, E. Z.; SHEN, X. A cross-input adaptive framework for gpu program optimizations. In: IEEE. *Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*. [S.l.], 2009. p. 1–10.

LLVM PROJECT. *The LLVM Compiler Infrastructure*. [S.l.], 2013. Disponível em: <http://llvm.org/>.

MCFARLAND, M.; PARKER, A.; CAMPOSANO, R. The high-level synthesis of digital systems. *Proceedings of the IEEE*, IEEE, v. 78, n. 2, p. 301–318, 1990.

MENOTTI, R. *LALP: uma linguagem para exploração do paralelismo de loops em computação reconfigurável*. Tese (Doutorado) — Universidade de São Paulo, 2010.

MENOTTI, R. *LALP: a language for loop parallelism exploitation in reconfigurable computing*. [S.l.]: VDM Verlag Dr. Müller, 2011. ISBN 3639339592, 9783639339598.

MENOTTI, R. et al. LALP: A Language to Program Custom FPGA-based Acceleration Engines. *International Journal of Parallel Programming*, Springer, v. 40, n. 3, p. 262–289, 2012.

MUNSHI, A. Opencl: Parallel computing on the gpu and cpu. *SIGGRAPH, Tutorial*, 2008.

NEUENDORFFER, S.; MARTINEZ-VALLINA, F. Building zynq® accelerators with vivado® high level synthesis. In: *FPGA*. [S.l.: s.n.], 2013. p. 1–2.

NVIDIA, C. Nvidia cuda c programming guide. *NVIDIA Corporation*, v. 120, 2011.

NVIDIA CORPORATION. *GeForce 256*. [S.l.], 2007. Disponível em: <http://www.nvidia.com/page/geforce256.html>.

NVIDIA CORPORATION. *OpenACC*. [S.l.], 2011. Disponível em: <https://developer.nvidia.com/openacc>.

NVIDIA CORPORATION. *Board Specification: Tesla K80 GPU Accelerator*. [S.l.], 2014. Disponível em: <http://international.download.nvidia.com/pdf/kepler/BD-07317-001_v04.pdf>.

OLIVEIRA J. M. P. CARDOSO, E. M. C. B. Lalp extensions for supporting floating-point operations. In: *Workshop on Research Projects Focusing High Performance Computing (HPCW 13)*. Porto: 23rd International Conference on Field Programmable Logic and Applications (FPL'13), 2013.

PAPAKONSTANTINOU, A. et al. FCUDA: Enabling efficient compilation of CUDA kernels onto FPGAs. In: *Application Specific Processors, 2009. SASP '09. IEEE 7th Symposium on*. [S.l.: s.n.], 2009. p. 35 –42.

PUTNAM, A. et al. Chimps: A c-level compilation flow for hybrid cpu-fpga architectures. In: IEEE. *Field Programmable Logic and Applications, 2008. FPL 2008. International Conference on*. [S.l.], 2008. p. 173–178.

PUTNAM, A. et al. A reconfigurable fabric for accelerating large-scale datacenter services. In: *Computer Architecture (ISCA), 2014 ACM/IEEE 41st International Symposium on*. [S.l.: s.n.], 2014. p. 13–24.

QUINLAN, D. Rose: Compiler support for object-oriented frameworks. *Parallel Processing Letters*, World Scientific, v. 10, n. 02n03, p. 215–226, 2000.

RETTORE, P. H. L. *Infraestrutura de Compilação para a Implementação de Aceleradores em FPGA*. Dissertação (Mestrado) — Departamento de Ciência da Computação, Universidade Federal de São Carlos, 2012.

REYES, R. et al. accull: An openacc implementation with cuda and opencl support. *Euro-Par 2012 Parallel Processing*, Springer, p. 871–882, 2012.

ROSE, C. A. D.; NAVAUX, P. O. Arquiteturas paralelas. *Instituto de Informática da UFRGS, Editora Sagra Luzzatto, Porto Alegre*, 2003.

ROSE COMPILER INFRASTRUCTURE. *ROSE User Manual: A Tool for Building Source-to-Source Translators*. [S.l.], 2012. Disponível em: <http://rosecompiler.org/ROSE_UserManual/ROSE-UserManual.pdf>.

SANTARINI, M. Zynq-7000 EPP sets stage for new era of innovations. *Xcell J*, v. 75, n. 2, p. 8–13, 2011.

SARKAR, S. et al. Lessons and Experiences with High-Level Synthesis. *Design and Test of Computers, IEEE*, v. 26, n. 4, p. 34–45, July-Aug. 2009. ISSN 0740-7475.

SINGH, D.; ENGINEER, S. P. Higher level programming abstractions for fpgas using opencl. In: *Workshop on Design Methods and Tools for FPGA-Based Acceleration of Scientific Computing*. [S.l.: s.n.], 2011.

SKLIAROVA, I.; FERRARI, A. B. Introdução à computação reconfigurável. *Revista do DETUA*, v. 2, 2003.

Terasic Technologies Inc. *DE2i-150 FPGA System User Manual*. [S.l.], 2013. Disponível em: <http://www.terasic.com/>.

Texas Instruments Incorporated. *TMS320C64x DSP Library: Programmer's Reference*. [S.l.], 2003. Disponível em: <http://www.ti.com/lit/ug/spru565b/spru565b.pdf>.

Texas Instruments Incorporated. *TMS320C64x ImageVideo Processing Library: Programmer's Reference*. [S.l.], 2003. Disponível em: <http://www.ti.com/lit/ug/spru023b/spru023b.pdf>.

THOMAS, D. B.; HOWES, L.; LUK, W. A comparison of cpus, gpus, fpgas, and massively parallel processor arrays for random number generation. In: ACM. *Proceedings of the ACM/SIGDA international symposium on Field programmable gate arrays*. [S.l.], 2009. p. 63–72.

TSUDA, F. *Utilização de Técnincas de GPGPU em Sistema de Vídeo-Avatar*. Dissertação (Mestrado) — Escola Politécnica, Universidade de São Paulo, 2012.

UNDERWOOD, K. Fpgas vs. cpus: trends in peak floating-point performance. In: ACM. *Proceedings of the 2004 ACM/SIGDA 12th international symposium on Field programmable gate arrays*. [S.l.], 2004. p. 171–180.

UNDERWOOD, K. D.; HEMMERT, K. S. Closing the gap: Cpu and fpga trends in sustainable floating-point blas performance. In: IEEE. *Field-Programmable Custom Computing Machines, 2004. FCCM 2004. 12th Annual IEEE Symposium on*. [S.l.], 2004. p. 219–228.

VANDERBAUWHEDE, W.; BENKRID, K. *High-Performance Computing Using FPGAs*. [S.l.]: Springer, 2013.

VILLARREAL, J. et al. Designing modular hardware accelerators in c with roccc 2.0. In: IEEE. *Field-Programmable Custom Computing Machines (FCCM), 2010 18th IEEE Annual International Symposium on*. [S.l.], 2010. p. 127–134.

WAIN, R. et al. *An overview of FPGAs and FPGA programming: Initial experiences at Daresbury*. [S.l.]: Council for the Central Laboratory of the Research Councils, 2006.

WU, E.; LIU, Y. Emerging technology about gpgpu. In: IEEE. *Circuits and Systems, 2008. APCCAS 2008. IEEE Asia Pacific Conference on*. [S.l.], 2008. p. 618–622.

ZANOTTO, L.; FERREIRA, A.; MATSUMOTO, M. Arquitetura e programação de gpu nvidia. *Revista do DETUA*, 2012.