LALP+ : um framework para o desenvolvimento de aceleradores de hardware em FPGAs

Cristiano Bacelar de Oliveira

SERVIÇO DE PÓS-GRADUAÇÃO DO ICMC-USP

Data de Depósito:

Assinatura: _____

Cristiano Bacelar de Oliveira

LALP+ : um framework para o desenvolvimento de aceleradores de hardware em FPGAs

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

Área de Concentração: Ciências de Computação e Matemática Computacional

Orientador: Prof. Dr. Eduardo Marques

Coorientador: Prof. Dr. João Manuel Paiva Cardoso

USP – São Carlos
Fevereiro de 2016

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

048l Oliveira, Cristiano Bacelar de
LALP+ : um framework para o desenvolvimento
de aceleradores de hardware em FPGAs / Cristiano
Bacelar de Oliveira; orientador Eduardo Marques;
coorientador João Manuel Paiva Cardoso. - São Carlos
- SP, 2016.
164 p.

Tese (Doutorado - Programa de Pós-Graduação em
Ciências de Computação e Matemática Computacional)
- Instituto de Ciências Matemáticas e de Computação,
Universidade de São Paulo, 2016.

1. Computação Reconfigurável. 2. FPGA.
3. Compiladores. 4. Memória. 5. HLS. I. Marques,
Eduardo, orient. II. Cardoso, João Manuel Paiva,
coorient. III. Título.

Cristiano Bacelar de Oliveira

LALP+ : a framework for developing FPGA-based hardware accelerators

Doctoral dissertation submitted to the Instituto de Ciências Matemáticas e de Computação – ICMC-USP, in partial fulfillment of the requirements for the degree of the Doctorate Program in Computer Science and Computational Mathematics. *FINAL VERSION*

Concentration Area: Computer Science and Computational Mathematics

Advisor: Prof. Dr. Eduardo Marques

Co-advisor: Prof. Dr. João Manuel Paiva Cardoso

USP – São Carlos
February 2016

Este trabalho é dedicado à minha esposa, Ívian, e aos meus pais, Tarcísio e Liana.

AGRADECIMENTOS

À Deus, primeiramente, por Sua graça e por ter colocado em meu caminho pessoas que me ajudaram e apoiaram ao longo da minha vida.

À minha esposa, Ívian, por ser minha amiga e companheira há quase uma década, pela força e incentivo que me deu nesses anos e pelo seu sacrifício pessoal diante das adversidades enfrentadas para que eu pudesse concluir este projeto de doutorado.

Aos meus pais, Tarcísio e Liana, por todo amor e cuidado, pelos ensinamentos, orações e conselhos que moldaram minha formação pessoal.

Ao meu irmão, por todo apoio e ajuda ao longo da minha vida.

Ao meu orientador Prof. Dr. Eduardo Marques, pelas contribuições, orientações e conselhos, tanto em nível acadêmico quanto pessoal. Também por ter aceitado assumir a orientação deste projeto, mesmo diante de circunstâncias não ideais.

Ao meu co-orientador, Prof. Dr. João M. P. Cardoso, por sua hospitalidade e cordialidade durante meu estágio na FEUP, em Portugal, por compartilhar seu conhecimento e me orientar em momentos cruciais de minha pesquisa.

Aos meus amigos, Erinaldo, Gustavo e Marcilyanne, pelas diversas contribuições técnicas, mas, também, pelos momentos de convívio dentro e fora da universidade.

Aos meus demais colegas de laboratório do LCR e do SPeCS, pela convivência harmoniosa durante o período da pesquisa.

Ao meu amigo Isaías, pelo acolhimento e suporte à minha estadia em Portugal.

Ao professor Dr. Jorge Luiz e Silva, por ter iniciado a orientação do meu doutorado.

Aos demais professores do ICMC, pelo compartilhamento de conhecimentos e experiências durante a realização deste trabalho.

À USP e à FEUP, pela infraestrutura e suporte oferecidos para execução deste trabalho.

Por fim, à Fundação de Amparo à Pesquisa do Estado de São Paulo, por todo suporte financeiro que tornou possível a realização desta pesquisa.

“Pois, que adianta ao homem ganhar o mundo inteiro e perder a sua alma?”

Marcos 8:36

RESUMO

OLIVEIRA, C. B.. **LALP+ : um framework para o desenvolvimento de aceleradores de hardware em FPGAs**. 2016. 164 f. Tese (Doutorado em Ciências – Ciências de Computação e Matemática Computacional) – Instituto de Ciências Matemáticas e de Computação (ICMC/USP), São Carlos – SP.

Considerando a crescente demanda por desempenho em sistemas computacionais, a implementação de algoritmos diretamente em *hardware* com o uso de FPGAs (*Field-programmable Gate Arrays*) é uma alternativa que tem apresentado bons resultados. Porém, os desafios de programação envolvidos no uso de FPGAs, de tal forma a explorar eficientemente seus recursos, limita o número de desenvolvedores em função da predominância do paradigma de programação tradicionalmente sequencial, imposto pelas linguagens imperativas. Assim, este trabalho busca desenvolver mecanismos que facilitem o desenvolvimento com FPGAs, otimizando o uso de memória e explorando o paralelismo das operações. Este documento apresenta a tese de doutorado de título *LALP+ : um framework para o desenvolvimento de aceleradores de hardware em FPGAs*. Dado que a latência para leitura e escrita de dados têm sido um gargalo para algumas aplicações de alto desempenho, este trabalho trata do desenvolvimento de técnicas para geração de arquiteturas de *hardware*, considerando aspectos relativos ao mapeamento, gerenciamento e acesso à memória em arquiteturas reconfiguráveis. Para isto, o projeto desenvolvido utiliza como base a linguagem LALP, cujo foco é o tratamento de *loops* com a técnica de *loop pipelining*. As técnicas descritas nesta tese são empregadas no desenvolvimento do *framework* LALP+, o qual estende LALP com a implementação de novas características e funcionalidades, de forma a contribuir para o aumento do seu nível de abstração. As arquiteturas criadas utilizando LALP+ foram comparadas às geradas por ferramentas comerciais e acadêmicas, tendo apresentado, em média, um melhor desempenho, com redução do tempo de execução de $10,01\times$, no melhor caso. Espera-se, por meio das contribuições aqui apresentadas, facilitar a implementação de produtos e projetos relacionados a aplicações de computação de alto desempenho que envolvam o uso de arquiteturas reconfiguráveis, promovendo uma maior absorção desta tecnologia.

Palavras-chave: Computação Reconfigurável, FPGA, Compiladores, Memória, HLS.

ABSTRACT

OLIVEIRA, C. B.. **LALP+ : um framework para o desenvolvimento de aceleradores de hardware em FPGAs**. 2016. 164 f. Tese (Doutorado em Ciências – Ciências de Computação e Matemática Computacional) – Instituto de Ciências Matemáticas e de Computação (ICMC/USP), São Carlos – SP.

Considering the demand for high-performance in computer systems, the implementation of algorithms directly in hardware by using FPGAs (*Field-programmable Gate Arrays*) is an alternative that has shown good results. However, the number of developers is limited due to the challenges faced for efficiently programming FPGAs. In addition to that, developers are more used to the traditional sequential programming paradigm imposed by the imperative languages. This work seeks to develop mechanisms to facilitate the development with FPGAs, by optimizing memory usage and exploiting the parallelism of operations inside a loop. This document presents the doctoral thesis entitled *LALP+ : a framework for developing FPGA-based hardware accelerators*. Since the latency for reading and writing data have been a bottleneck for high performance applications, this work deals with the development of techniques for generation of hardware architectures, considering aspects related to mapping, management and memory access in reconfigurable architectures, using as basis the LALP language, which focuses on the treatment of loops with the technique of loop pipelining. The techniques described in this thesis are employed in the development of the LALP+ framework, which extends LALP by implementing new features and functionalities, in order to contribute to increase its abstraction level. LALP+ architectures were compared to ones generated by using academical and commercial tools, having presented, on average, better performance, with a execution time speedup of $10,01\times$ for the best case. Thus, it is expected that the hereby presented contributions facilitate the implementation of products and projects related to high-performance computing applications with reconfigurable architectures, contributing for the use of such technology.

Key-words: Reconfigurable Computing, FPGA, Compilers, Memory, HLS.

LISTA DE ILUSTRAÇÕES

1	Frequência de <i>clock</i> de CPUs entre os anos de 1992 e 2012	34
2	Uma seleção de projetos com impacto na área de computação reconfigurável, desenvolvidos nos últimos 25 anos	35
3	Aumento da densidade e capacidade de memórias DRAM em relação ao aumento da velocidade de processamento das CPUs entre os anos de 1980 e 2010	36
4	Aumento da latência de acesso a memórias DRAM entre os anos de 1980 e 2010	36
5	Comparação geral entre o uso de HDL, DSL e HLS para desenvolvimento de aceleradores em FPGAs	39
6	Estrutura interna de um FPGA	45
7	Estrutura típica de um compilador	47
8	Exemplos de representações intermediárias.(a) Expressão original;(b) Código de três endereços referente a (a);(c) Árvore sintática abstrata referente a (a) .	48
9	Estrutura da plataforma PAM-Blox	57
10	Estrutura do compilador ASC	58
11	Nó computacional utilizando DFEs	59
12	Arquitetura OpenCL com <i>kernels</i> mapeados em FPGAs da Altera	60
13	Fluxo de compilação FCUDA	61
14	Fluxo de compilação em Haydn-C	63
15	Fluxo de desenvolvimento SA-C	64
16	Fluxo de compilação do LegUp	65
17	Fluxo de desenvolvimento usando o Impulse C	66
18	Fluxo de compilação C-to-Silicon	67
19	Fluxo de compilação do Vivado HLS	72
20	Visão geral do REFLECT	74
21	Exemplo da técnica de <i>loop pipelining</i>	79
22	Fluxo de compilação LALP	80
23	Visualizações de <i>software</i> e <i>hardware</i> correspondentes ao exemplo de código LALP mostrado no Source code 3	84
24	Impacto na execução resultante das diferentes formas de aplicação de <i>delays</i> em LALP: (a) Efeito do código original; (b) Efeito do Source code 5; (c) Efeito do Source code 6; (d) Efeito do Source code 7	85

25	Arquitetura geral de um acelerador LALP	86
26	Posições acessadas em cada iteração para computação do filtro de <i>Sobel</i>	88
27	Representação de um número Real em 32 <i>bits</i> . (a) Ponto flutuante, segundo padrão IEEE 754; (b) Ponto fixo, com 8 <i>bits</i> para a parte fracionária	94
28	Arquitetura com acesso único indexado	101
29	Arquitetura com máquina de estado para controle de múltiplos acessos sequenciais a uma mesma BRAM	102
30	Fluxo de dados gerado na abordagem utilizando FSM em um caso com 4 acessos	102
31	Arquitetura para controle de múltiplos acessos com particionamento de dados	103
32	Exemplos de mapeamento de uma matriz 4x4 em múltiplas BRAMs: (a) Declaração da matriz em código; (b) Partição em 2 BRAMS; (c) Partição em 4 BRAMS	104
33	Exemplo do impacto conjunto de particionamento e endereçamento no número de ciclos de relógio, para o caso onde são realizadas leituras em colunas diferentes: (a) 4 colunas em 2 BRAMS; (b) 4 colunas em 4 BRAMS; (c) 2 colunas em 2 BRAMS; (d) 2 colunas em 4 BRAMS	106
34	Possíveis arquiteturas de um acelerador LALP+: (a) Acelerador LALP+ com FSM; (b) Acelerador LALP+ com MMU	108
35	Diagrama de sinais de instrução customizada do NiosII [®] , estendida para até 8 operações ⁽¹⁾	109
36	Frequência máxima em MHz obtida pelos operadores LALP e Flopoco para adição e multiplicação em ponto flutuante, usando um FPGA StratixIV (modelo EP4SGX530KH40C20) da Altera [®]	116
37	Latência dos operadores LALP e Flopoco para adição e multiplicação em ponto flutuante, usando um FPGA StratixIV (modelo EP4SGX530KH40C20) da Altera [®]	117
38	Tempo(μ s) da operação soma dos operadores LALP e Flopoco para adição e multiplicação em ponto flutuante, usando um FPGA StratixIV (modelo EP4SGX530KH40C20) da Altera [®]	117
39	Tempo(μ s) da operação de multiplicação dos operadores LALP e Flopoco para adição e multiplicação em ponto flutuante, usando um FPGA StratixIV (modelo EP4SGX530KH40C20) da Altera [®]	118
40	Resultados referentes ao uso de recursos dos operadores LALP e Flopoco para adição e multiplicação em ponto flutuante, usando um FPGA StratixIV (device EP4SGX530KH40C20) da Altera [®]	119
41	Frequência (MHz) máxima de <i>clock</i> atingida com o módulo LALP de multiplicação para números variados de estágios de <i>pipeline</i> , usando FPGA Xilinx Virtex7 XC7VX330T	120

42	<i>Speedup</i> de tempo de execução atingido pelas arquiteturas LALP+ em relação às geradas pelo Vivado HLS, usando um FPGA Xilinx Virtex 7 (device XC7VX330t-1-ffg1157)	122
43	Frequência máxima (em MHz) obtida pelas arquiteturas LALP+ e Vivado HLS, usando um FPGA Xilinx Virtex 7 (device XC7VX330t-1-ffg1157) . .	122
44	Latência (em ciclos de relógio) das arquiteturas LALP+ e Vivado HLS, usando um FPGA Xilinx Virtex 7 (device XC7VX330t-1-ffg1157)	123
45	Total de LUTs usadas nas arquiteturas LALP+ e Vivado HLS, usando um FPGA Xilinx Virtex 7 (device XC7VX330t-1-ffg1157)	124
46	Total de Flip-flops usados nas arquiteturas LALP+ e Vivado HLS, usando um FPGA Xilinx Virtex 7 (device XC7VX330t-1-ffg1157)	124
47	Speedups obtidos para as arquiteturas LALP+ em relação às geradas pelo LegUp, para um FPGA StratixIV (device EP4SGX530KH40C20) da Altera®	125
48	Latência (em ciclos de relógio) das arquiteturas LALP+ e LegUp, para um FPGA StratixIV (device EP4SGX530KH40C20) da Altera®	126
49	Frequência máxima (em MHz) obtida pelas arquiteturas LALP+ e LegUp, para um FPGA StratixIV (device EP4SGX530KH40C20) da Altera®	126
50	Total de registradores usados pelas arquiteturas LALP+ e LegUp, para um FPGA StratixIV (device EP4SGX530KH40C20) da Altera®	127
51	Total de ALUTs usadas pelas arquiteturas LALP+ e LegUp, para um FPGA StratixIV (device EP4SGX530KH40C20) da Altera®	128
52	Total de ALUTs usadas pelas diferentes versões LALP+ testadas	131
53	Total de Registradores usados pelas diferentes versões LALP+ testadas . . .	131
54	Latência total (em ciclos de relógio) de cada uma das diferentes versões LALP+ testadas	133
55	Frequência máxima (em MHz) atingida em cada uma das diferentes versões LALP+ testadas	133
56	<i>Speedup</i> das arquiteturas com acessos paralelos (MMUs) em relação à arquitetura com acesso sequencial (FSM)	134
57	Etapas da versão desenvolvida do Sistema de HLS	138
58	Infraestrutura de um tradutor escrito usando ROSE	139
59	Visão geral do sistema de geração de <i>hardware</i>	141
60	Estrutura do compilador proposto	142
61	Diferentes etapas de processamento de um compilador C++ construído usando a infraestrutura do ROSE	144

LISTA DE QUADROS

1	Características gerais de GPLs e DSLs	53
2	Resumo das características gerais das linguagens e <i>frameworks</i> apresentados	65
3	Linguagens de entrada e saída suportadas pelos compiladores das ferramentas de HLS descritas	74
4	Comparação entre as ferramentas HLS apresentadas	75
5	Atributos e sinais do componente contador	86
6	Tipos básicos pré-definidos em LALP+	97
7	Sinais da interface de instruções customizadas estendidas do NiosII® . . .	108
8	Comparação entre LALP e LALP+	114
9	Descrição dos <i>benchmarks</i> utilizados, com indicação das ferramentas onde foram implementados	121
10	Versões LALP+ usadas nos experimentos com <i>benchmarks</i> para acessos múltiplos	129
11	Transformações e parâmetros suportados pela versão atual do compilador desenvolvido	139
12	Lista parcial de transformações suportadas pelo ROSE	145

LISTA DE ALGORITMOS

Algoritmo 1 – Cálculo do número de ciclos de relógios para o incremento do contador . 105

LISTA DE CÓDIGOS-FONTE

Código-fonte 1	Trecho de código C usado para exemplificar a técnica de <i>loop pipelining</i>	78
Código-fonte 2	Estrutura de um código LALP	81
Código-fonte 3	Exemplo de código LALP para multiplicação de dois vetores	82
Código-fonte 4	Código C correspondente ao Código-fonte 3	83
Código-fonte 5	Trecho do Código-fonte 3 modificado para inserção de atraso na multiplicação	84
Código-fonte 6	Trecho do Código-fonte 3 modificado para inserção de atraso na leitura do valor do contador	84
Código-fonte 7	Trecho do Código-fonte 3 modificado para inserção de atraso no incremento do valor do contador	84
Código-fonte 8	Exemplo de código LALP para implementação do filtro de Sobel	88
Código-fonte 9	Definição de tipos com sintaxe LALP	97
Código-fonte 10	Definição de tipos com sintaxe LALP+	97
Código-fonte 11	Trecho de código mostrando o uso do contador simples e endereçamento indireto em LALP	99
Código-fonte 12	Trecho de código mostrando o uso do contador multinível e endereçamento direto em LALP+	99
Código-fonte 13	Código FIR em LALP+ com memória interna	109
Código-fonte 14	Código FIR em LALP+ com fifos	110
Código-fonte 15	Multiplicação de matrizes em LALP	161
Código-fonte 16	Multiplicação de matrizes em LALP+	163
Código-fonte 17	Multiplicação de matrizes em C	164

LISTA DE TABELAS

1	Resultados obtidos para diferentes versões FIR, usando um FPGA StratixIV (modelo EP4SGX530KH40C20	128
2	Resultados obtidos para diferentes arquiteturas com múltiplos acessos, usando um FPGA StratixIV (modelo EP4SGX530KH40C20	130
3	Número de acessos e total de ciclos de relógio por iteração para cada <i>benchmark</i>	132

LISTA DE ABREVIATURAS E SIGLAS

AG		Algoritmo Genético
ALAP		<i>As Late As Possible</i>
ALUT		<i>Adaptative LUT</i>
API		<i>Application Programming Interface</i>
ASAP		<i>As Soon As Possible</i>
ASC		<i>A Stream Compiler</i>
ASIC		<i>Application Specific Integrated Circuit</i>
AST		<i>Abstract Syntax Tree</i>
BRAM	...	<i>Block RAM</i>
CDFG		<i>Control-Data Flow Graph</i>
CPLD		<i>Complex Programmable Logic Device</i>
DDG		<i>Data Dependency Graph</i>
DFE		<i>Data Flow Engine</i>
DSE		<i>Design Space Exploration</i>
DSL		<i>Domain-Specific Language</i>
DSP		<i>Domain-Specific Processor</i>
ECC		<i>Error Correcting Code</i>
FF		<i>Flip-Flop</i>
FIFO		<i>First In First Out</i>
FNC		Forma normal conjuntiva
FPGA		<i>Field-Programmable Gate Array</i>
FPGAs	...	<i>Field-Programmable Gate Arrays</i>
FSM		<i>Finite State Machine</i>
GPL		<i>General Purpose Language</i>
GPP		<i>General Purpose Processor</i>
GPU		<i>Graphic Processor Unit</i>
HDL		<i>Hardware Description Language</i>
HLS		<i>High-Level Synthesis</i>
HPC		<i>High-Performance Computing</i>
IEEE		<i>Institute of Electrical and Electronics Engineers</i>
II		<i>Initiation Interval</i>

IP *Intellectual Property*
IR *Intermediary Representation*
ISA *Instruction Set Architecture*
LALP *Language for Aggressive Loop Pipelining*
LARA *Language for Reconfigurable Architectures*
LCR Laboratório de Computação Reconfigurável
LUT *Look-Up Table*
MCO Módulo de Compilação e Otimização
MDSE ... Módulo de DSE
MGH Módulo Gerador de *Hardware*
MMU *Memory Management Unit*
NSGA *Non-dominated Sorting Genetic Algorithm*
PAL *Programmable Array Logic*
PLA *Programmable Logic Array*
RAM *Random Access Memory*
ROM *Read-Only Memory*
RTL *Register Transfer Level*
SCS *Spatial Computing Substrate*
SoC *System on Chip*
SPLD *Simple Programmable Logic Device*
TTL *Transistor-Transistor Logic*
UFSCAR . Universidade Federal de São Carlos
ULA Unidade Lógica e Aritmética
USP Universidade de São Paulo
VLSI *Very Large Scale Integration*
XNF *Xilinx Netlist Format*

SUMÁRIO

1	INTRODUÇÃO	33
1.1	Contextualização	33
1.2	Objetivo geral	37
1.3	Objetivos específicos	37
1.4	Motivação e Justificativa	38
1.5	Contribuições	40
1.6	Organização do texto	40
2	COMPILAÇÃO PARA ARQUITETURAS RECONFIGURÁVEIS	43
2.1	Considerações iniciais	43
2.2	Sistemas embarcados e Computação Reconfigurável	43
2.3	Compiladores	46
2.3.1	<i>Front-end</i>	47
2.3.2	<i>Middle-end</i>	48
2.3.3	<i>Back-end</i>	51
2.4	Síntese de alto nível	51
2.5	Linguagens de Domínio Específico - DSLs	53
2.6	Considerações finais	54
3	TRABALHOS RELACIONADOS	55
3.1	Considerações iniciais	55
3.2	Linguagens e <i>Frameworks</i>	56
3.2.1	<i>BSAT e StReAm</i>	56
3.2.2	<i>ASC</i>	57
3.2.3	<i>PyHDL</i>	57
3.2.4	<i>MaxJ</i>	58
3.2.5	<i>OpenSPL</i>	59
3.2.6	<i>OpenCL e Silicon OpenCL</i>	59
3.2.7	<i>FCUDA</i>	60
3.2.8	<i>Handel-C</i>	61
3.2.9	<i>Haydn-C</i>	62
3.2.10	<i>Single Assignment C</i>	62
3.2.11	<i>SystemC</i>	63

3.3	Ferramentas para Síntese de Alto Nível	64
3.3.1	<i>LegUp</i>	64
3.3.2	<i>Impulse C</i>	66
3.3.3	<i>Cadence C-to-Silicon</i>	67
3.3.4	<i>C to Fine-Grained Pipeline</i>	67
3.3.5	<i>ROCCC</i>	68
3.3.6	<i>SPC</i>	68
3.3.7	<i>DEFACTO</i>	68
3.3.8	<i>MATCH</i>	69
3.3.9	<i>DeepC</i>	69
3.3.10	<i>Galadriel e Nanya</i>	69
3.3.11	<i>Spark</i>	70
3.3.12	<i>Catapult C</i>	70
3.3.13	<i>GCC2Verilog</i>	70
3.3.14	<i>C2H</i>	71
3.3.15	<i>HDL Coder</i>	71
3.3.16	<i>Vivado HLS</i>	71
3.3.17	<i>CHiMPS</i>	72
3.3.18	<i>REFLECT</i>	73
3.4	Considerações finais	73
4	A LINGUAGEM LALP	77
4.1	Considerações iniciais	77
4.2	Visão geral	77
4.2.1	<i>Fluxo de compilação</i>	79
4.2.2	<i>O código LALP</i>	81
4.2.3	<i>Componentes de hardware</i>	85
4.3	Principais problemas	87
4.3.1	<i>Endereçamento de memória</i>	87
4.4	Abordagens envolvendo LALP	90
4.5	Considerações finais	90
5	LALP+: UMA EXTENSÃO DE LALP	91
5.1	Considerações iniciais	91
5.2	Visão geral da abordagem	91
5.3	Novas características e funcionalidades	92
5.3.1	<i>Suporte a operações de ponto fixo/flutuante</i>	92
5.3.1.1	<i>Componentes para Floating/Fixed Point</i>	92
5.3.1.2	<i>Módulos parametrizáveis para ponto flutuante</i>	95
5.3.1.3	<i>Outras alterações para ponto fixo/flutuante</i>	96

5.3.2	Sintaxe	96
5.3.2.1	<i>Criação e definição de tipos</i>	96
5.3.2.2	<i>Expressões numéricas com constantes</i>	98
5.3.2.3	<i>Suporte a loops aninhados e a endereçamento direto</i>	98
5.3.2.4	<i>Simplificação da sintaxe</i>	100
5.3.3	Gerenciamento de memória e sincronismo	100
5.3.3.1	<i>Indexação para acesso único</i>	101
5.3.3.2	<i>Múltiplos acessos sequenciais</i>	101
5.3.3.3	<i>Múltiplos acessos com particionamento de dados</i>	103
5.3.4	Suporte a co-projetos de hardware e software	107
5.4	Resumo das Contribuições	111
5.5	Considerações finais	114
6	EXPERIMENTOS E RESULTADOS	115
6.1	Considerações iniciais	115
6.2	Testes e resultados comparativos	115
6.2.1	<i>Operadores para ponto flutuante</i>	115
6.2.2	LALP vs. HLS	120
6.2.2.1	<i>LALP vs. Vivado HLS</i>	121
6.2.2.2	<i>LALP vs. LegUp</i>	125
6.2.3	<i>Interfaces para co-projeto</i>	127
6.3	Arquiteturas com múltiplos acessos e particionamento de dados	129
6.4	Considerações finais	135
7	LALP+ NO CONTEXTO DE SÍNTESE DE ALTO NÍVEL	137
7.1	Considerações iniciais	137
7.2	Visão geral da ferramenta atual	137
7.3	Ferramenta de HLS com LALP+	140
7.4	Módulo de compilação e otimização	141
7.4.1	<i>O front-end</i>	143
7.4.2	<i>O middle-end</i>	145
7.4.3	<i>O back-end</i>	146
7.5	Considerações finais	146
8	CONCLUSÃO	147
	Referências	151
	APÊNDICE A CÓDIGOS PARA MULTIPLICAÇÃO DE MATRIZES EM LALP, LALP+ E C	161

INTRODUÇÃO

1.1 Contextualização

Aplicações que trabalham com manipulação massiva de dados estão cada vez mais comuns. Como exemplo destas podemos citar análises meteorológicas, simulação para pesquisas em nível de biologia molecular, processamento e transmissão de áudio e vídeo de alta definição, etc. Diante da crescente demanda por grande capacidade computacional para tais aplicações, estratégias que visam alcançar altas taxas de processamento por meio do aumento de *clock* têm encontrado obstáculos consideráveis devido a limites dos componentes eletrônicos. Tais limitações, que envolvem tanto questões de energia como outras características físicas dos materiais, contribuíram para a desaceleração no aumento das taxas de *clock* (BORKAR; CHIEN, 2011). A Figura 1 ilustra esta desaceleração, em contraste à tendência de aumento de frequência observada há duas décadas.

Com o objetivo de dar suporte a aplicações que envolvem computação de alto desempenho, as comunidades acadêmica e industrial têm desenvolvido máquinas de grande porte capazes de suportar imensa quantidade de cálculos simultâneos, atingindo altas taxas de processamento por meio da execução dos processos em paralelo. Com isto, o uso de arquiteturas de computação paralela tem se tornado frequente em aplicações desta natureza. Entretanto, o uso em larga escala de máquinas altamente concorrentes aumenta não só a necessidade de energia para essas máquinas, mas, também, a dificuldade em programá-las efetivamente. Tais fatos colocam a área de *High-Performance Computing* (HPC) no centro de muitos desafios científicos e tecnológicos atuais (CATANZARO *et al.*, 2010).

Uma abordagem alternativa para contornar alguns desses problemas baseia-se na utilização de *Field-Programmable Gate Arrays* (FPGAs). Os FPGAs são dispositivos de *hardware* programáveis que podem ter sua lógica interna configurada a fim de se desenvolver estruturas de

Figura 1 – Frequência de *clock* de CPUs entre os anos de 1992 e 2012

Fonte: Adaptada de [Smith, Wang e Fujino \(2012\)](#).

hardware para aplicações com domínio muito específico, eficientes em termos de consumo e desempenho ([BOBDA, 2007](#)). Eles possibilitam o desenvolvimento de sistemas que permitem a execução diretamente em *hardware* de algoritmos computacionalmente mais complexos. Desta forma, é possível atender aos requisitos de alto desempenho sem negligenciar fatores como a eficiência de energia.

Em função das constantes melhorias de capacidade dos FPGAs, arquiteturas baseadas em FPGAs são boas candidatas para a construção de sistemas complexos e vistas como potenciais soluções para o desenvolvimento de sistemas computacionais com requisitos de alto desempenho ([MENOTTI, 2010](#)). Deste modo, os FPGAs tornam-se opções de escolha tanto para prototipação como para produtos finais que requeiram, dentre outras características, sistemas heterogêneos, reconfiguráveis e *multi-core on-chip*, onde é possível ter conjuntos de instruções personalizadas, várias RAMs distribuídas e interconexões configuráveis, como o mostrado por [Bertels et al. \(2010\)](#). A [Figura 2](#) apresenta uma seleção de projetos na área de computação reconfigurável, desenvolvidos nos últimos 25 anos.

A flexibilidade obtida com FPGAs permite que o desenvolvedor personalize sua arquitetura de *hardware* de modo a alcançar um desempenho melhor que o de sistemas de computação convencionais ([BORKAR; CHIEN, 2011](#)). Além disto, com a execução dos algoritmos em *hardware* há uma redução considerável do consumo de energia ([CARDOSO; HÜBNER, 2011](#)).

Apesar dos benefícios alcançados com o uso de FPGAs, o custo de programação envolvido é um fator crítico. Isto porque, para efetivamente usufruir destes benefícios, é preciso implementar os algoritmos utilizando linguagens de programação específicas, orientadas a *hardware*, como VHDL ou Verilog. Este tipo de linguagem expõe o programador a um paradigma

Figura 2 – Uma seleção de projetos com impacto na área de computação reconfigurável, desenvolvidos nos últimos 25 anos

Fonte: Tessier, Pocek e DeHon (2015).

de programação e de execução que explora o paralelismo em nível de *hardware*, o que não é facilmente acessível ao programador mediano mais habituado ao paradigma de programação tradicionalmente sequencial (HALSTEAD *et al.*, 2010; ANDREWS *et al.*, 2008). Além disto, o escalonamento, balanceamento de carga, sincronização e *overhead* de comunicação entre processos ainda constituem desafios para a programação paralela (PATTERSON D. A.; HENESSY, 2009).

Uma vez que a largura de banda disponível para acesso à memória é um fator que limita o desempenho geral do *hardware*, desenvolver arquiteturas que permitam o acesso paralelo aos dados contribui para que os algoritmos tratem os dados de forma independente, aumentando o desempenho das aplicações (ASHER; ROTEM, 2010). Assim, questões referentes ao acesso à memória e manipulação de dados também se tornaram um problema para aplicações envolvendo HPC. Parte disto se dá pelo fato de que as melhorias em relação à memória se deram com o aumento da capacidade de armazenamento e nível de integração, em detrimento ao aumento da velocidade de acesso aos dados. Fato que ocorreu por questões principalmente mercadológicas ¹ (BORKAR; CHIEN, 2011).

A Figura 3 mostra o aumento da densidade de memórias DRAM ao longo dos últimos trinta anos, comparando sua velocidade em relação à velocidade das CPUs, enquanto a Figura 4 ilustra o aumento do total de ciclos de *clock* necessários para cada acesso à memória. Observando estas figuras é possível perceber a diferença entre as velocidades de processadores e memórias, o que contribui para aumento do gargalo no acesso aos dados. Este problema é conhecido como “*memory wall*” (MCKEE, 2004). Além disto, estudos mostram o aumento da contribuição da

¹ Detalhes técnicos sobre estas questões podem ser vistos em Drepper (2007)

memória para a dissipação da energia de um sistema ([WEHMEYER; MARWEDEL, 2006](#)).

Figura 3 – Aumento da densidade e capacidade de memórias DRAM em relação ao aumento da velocidade de processamento das CPUs entre os anos de 1980 e 2010

Fonte: Adaptada de [Borkar e Chien \(2011\)](#).

Figura 4 – Aumento da latência de acesso a memórias DRAM entre os anos de 1980 e 2010

Fonte: Adaptada de [Borkar e Chien \(2011\)](#).

No caso de arquiteturas reconfiguráveis, a personalização e o número das possíveis configurações aumenta a complexidade no particionamento, mapeamento e gerenciamento dos dados entre as diversas memórias distribuídas. Além disto, é possível que sejam necessárias quantidades de ciclos de *clock* distintas para acesso a memórias localizadas em diferentes

blocos de um FPGA. Por estas razões, pesquisas envolvendo técnicas de transformações de código orientadas a dados e de gerenciamento de memória para HPC baseada em *hardware* reconfigurável têm sido realizadas, como pode ser visto nos trabalhos de [Cardoso, Diniz e Weinhardt \(2010\)](#), [Baradaran e Diniz \(2008\)](#) e [Baradaran e Diniz \(2006\)](#), dentre outros.

Neste contexto, compiladores que mapeiam estruturas de *software* (como *loops*, *arrays*, etc.) em blocos de *hardware* reconfiguráveis passam a ser ferramentas importantes. Tais ferramentas podem contribuir para o desempenho dos sistemas explorando o paralelismo das operações e permitindo ao programador abstrair aspectos de *hardware* mais específicos. O desenvolvimento de ferramentas de compilação, entretanto, não é simples, e envolve conhecimentos tanto teóricos como práticos de diversas áreas da computação. No caso de compiladores para sistemas reconfiguráveis são muitas as opções de arquitetura, o que aumenta consideravelmente a complexidade da tarefa de gerar o código final ([MEHTA; JONES, 2010](#)).

1.2 Objetivo geral

Este trabalho tem como objetivo geral desenvolver uma nova versão do compilador e da linguagem LALP ([MENOTTI et al., 2012](#)), considerando aspectos como dependência, disponibilidade de dados, latência e velocidade de acesso à memória. LALP é uma DSL (*Domain-Specific Language*) de propósito especial, focada no desenvolvimento de aceleradores em FPGA a partir de *loops*, com base na técnica de *loop pipelining*.

Uma vez que otimizações envolvendo loops, escalonamento de instruções e alocação de dados em registradores estão entre as mais importantes otimizações aplicadas por um compilador ([MUCHNICK, 1997](#)), acredita-se que a combinação destas otimizações em arquiteturas reconfiguráveis deve contribuir para melhorar o desempenho geral do sistema. Assim, nesta nova versão da linguagem, técnicas que envolvem análise de estruturas de repetição (loops) são utilizadas em conjunto com o mapeamento de memória, bem como de transformações de código orientadas aos dados, a fim de combinar seus benefícios.

1.3 Objetivos específicos

A nova linguagem, denominada LALP+, é derivada de LALP, sendo gerada por meio da expansão de suas funcionalidades, incorporando mecanismos que permitem um melhor tratamento da memória e com uma sintaxe mais amigável do que a suportada pelo compilador original. Para isto ser alcançado, os seguintes objetivos específicos são propostos:

- Desenvolver técnicas que auxiliem a implementação de aplicações de alto desempenho por meio do tratamento otimizado de dados para síntese em *hardware* reconfigurável.

- Implementar diretivas e componentes para mapeamento de *arrays* em memória, com suporte a operações de acesso *pipelining* e *non-pipelining*.
- Inserir na linguagem LALP mecanismos que auxiliem a manipulação de dados, a partir das técnicas desenvolvidas, contribuindo para o aprimoramento da linguagem.
- Desenvolver componentes de hardware com suporte a operações de ponto fixo/flutuante e incluí-los na biblioteca LALP.
- Alterar o compilador LALP para suportar tipos e operações com ponto fixo/flutuante.

1.4 Motivação e Justificativa

Uma abordagem comum para o uso de FPGAs é o desenvolvimento de componentes em *hardware* destinados a acelerar tarefas de computação específicas. Tais aceleradores podem ser integrados a outros dispositivos, como processadores, memórias e outros aceleradores, de forma a comporem sistemas computacionais mais complexos. Assim, algoritmos escritos em *software* podem ser substituídos, parcial ou completamente, por circuitos lógicos que realizam a mesma função.

Tipicamente, o desenvolvimento de aceleradores em FPGAs é feito por meio de sua especificação em alguma linguagem de descrição de hardware (HDL - *Hardware Description Language*), como VHDL e System Verilog. Este tipo de especificação ocorre em nível de transferência de registros (RTL - *Register Transfer Level*), o que implica um nível de abstração onde os desenvolvedores devem lidar com características diretamente ligadas ao *hardware*, como sinais, registro de dados e ciclos de relógio.

Apesar de possibilitar o desenvolvimento de aceleradores eficientes, o uso de HDLs impõe dificuldades no tocante ao nível de conhecimento exigido do programador, sendo uma barreira para aqueles habituados a linguagens de alto nível. Nos casos onde o programador detém os conhecimentos necessários, o esforço de programação resulta em potenciais problemas, como tempo de desenvolvimento elevado e possibilidade de erros de codificação, dada a quantidade de linhas de código que precisam ser escritas em comparação ao mesmo algoritmo descrito em *software*.

Como forma de facilitar o processo de desenvolvimento com FPGAs, abordagens que utilizam ferramentas de Síntese de Alto Nível (HLS - *High-level Synthesis*) atuam no sentido de proporcionar um nível de abstração mais elevado para especificação de *hardware*. Tais ferramentas tem por objetivo a compilação automática de programas (*software*) para códigos em nível RTL, permitindo que os desenvolvedores gerem componentes de *hardware* diretamente dos códigos de alto nível escritos em linguagens como C, Java e Matlab. Com isto, espera-se diminuir o esforço de programação necessário aos desenvolvedores de sistemas embarcados.

A facilidade inerente ao uso de ferramentas HLS permite um rápido desenvolvimento de circuitos personalizados, tornando alguns aspectos de implementação de baixo nível, como as várias possibilidades de paralelismo, transparentes ao programador. No entanto, nem sempre as técnicas de compilação automáticas usadas nessas ferramentas tratam estes aspectos de baixo nível de forma otimizada. Como resultado desta limitação, as soluções apresentadas por tais ferramentas são, muitas vezes, consideradas ineficientes, principalmente quando comparadas a soluções obtidas com o uso de HDLs por projetistas de *hardware* especializados.

Neste sentido, linguagens de domínio específico (DSL - *Domain-Specific Languages*), podem ser soluções alternativas para programar FPGAs. Elas podem fornecer níveis de abstração mais elevados do que HDLs, enquanto permitem um maior controle da implementação por parte do programador, fornecendo soluções intermediárias entre o uso de HDLs e ferramentas de HLS, tanto no que diz respeito ao nível de abstração quanto à eficiência das soluções geradas.

Em geral, as DSLs são usadas para fins especiais e seu uso ainda requer esforços para tradução de código de alto nível para o formato de código específico tratado pelo compilador da DSL. Porém, com tais linguagens é possível controlar algumas questões do processo de geração de *hardware* com a inserção de conhecimento especializado de uma forma mais acessível do que usando HDLs. Além disto, as DSLs permitem explorar aspectos específicos da implementação que podem levar a soluções melhores do que as fornecidas por ferramentas HLS.

A [Figura 5](#) traz uma comparação geral entre o uso de HDL, DSL e HLS para desenvolvimento de aceleradores em FPGAs.

Figura 5 – Comparação geral entre o uso de HDL, DSL e HLS para desenvolvimento de aceleradores em FPGAs

Fonte: Adaptada de [Oliveira et al. \(2015\)](#).

Um dos motivos da escolha da DSL LALP como base deste trabalho está no fato de que LALP apresentou bons resultados, mostrados em [Menotti et al. \(2012\)](#), quando comparada com outras ferramentas. Nesses casos, as implementações LALP obtiveram um *speedup* médio de $5.60\times$ sobre C2Verilog ([ROTEM, 2010](#)) para um conjunto de 10 *kernels*, e $1.14\times$ sobre ROCCC ([VILLARREAL et al., 2010](#)) para um conjunto de 3 *kernels*.

Além disto, LALP possui código aberto e foi desenvolvida na Universidade de São Paulo (USP), o que facilita acesso ao suporte por parte dos seus criadores. Espera-se que, futuramente,

LALP possa ser incorporada a uma ferramenta de síntese de alto nível a ser desenvolvida no Laboratório de Computação Reconfigurável da USP.

1.5 Contribuições

LALP foca-se principalmente em fornecer uma linguagem de programação e respectivo suporte de compilação para FPGAs de forma a atingir implementações eficientes de *loops* por meio do alto grau de paralelismo alcançado com a técnica de *loop-pipelining*. Entretanto, a eficiência desse paralelismo é comprometida em alguns casos devido ao acesso sequencial aos dados. A complexidade requerida para a programação em LALP, principalmente no que diz respeito ao sincronismo das operações envolvendo acesso à memória, é um outro fator que limita seu uso.

Dadas essas e outras limitações presentes em LALP, mais detalhadas no [Capítulo 4](#), este trabalho apresenta uma expansão das funcionalidades tanto da linguagem quanto do compilador, sendo implementadas várias melhorias. Em função disto, tem-se como resultado uma versão expandida de LALP, denominada LALP+, que incorpora uma série de novas características, como a criação de novos componentes de *hardware*, otimizações do tratamento de memória e alterações na sintaxe da linguagem. Além disto, algumas das etapas da geração de *hardware* foram automatizadas, o que simplifica a tarefa de definição do sincronismo por parte do programador. As modificações implementadas abrangem diversos aspectos da utilização de LALP, o que contribui para a elevação do nível de abstração e, conseqüentemente, simplificação da programação, resultando em um aumento do escopo da linguagem.

As modificações na linguagem ocorreram de forma a contribuir com o aumento do nível de abstração, bem como do domínio de aplicações suportadas. Em adição a isto tem-se uma reestruturação do modelo arquitetural usado nos aceleradores LALP de forma a facilitar as tarefas de sincronismo e endereçamento. A fim de suportar estas mudanças, o compilador foi alterado a nível de *front-end*, *middle-end* e *back-end*, tendo como resultado uma linguagem mais acessível aos programadores e com novos recursos. Maiores detalhes sobre cada uma das alterações realizadas são mostrados nos capítulos [5](#) e [6](#).

1.6 Organização do texto

O restante deste documento está organizado nos seguintes capítulos:

[Capítulo 2](#) → Descreve uma visão geral da área de computação reconfigurável.

[Capítulo 3](#) → Apresenta os principais trabalhos relacionados com o assunto abordado.

[Capítulo 4](#) → Neste capítulo são apresentados detalhes da linguagem LALP.

Capítulo 5→ Detalhes sobre o desenvolvimento e as contribuições deste trabalho são mostrados neste capítulo.

Capítulo 6→ Este capítulo traz os experimentos e resultados obtidos com a nova versão.

Capítulo 7→ Aqui é apresentado o projeto de uma ferramenta de HLS envolvendo a nova versão de LALP.

Capítulo 8→ Apresenta as conclusões e trabalhos futuros.

COMPILAÇÃO PARA ARQUITETURAS RECONFIGURÁVEIS

2.1 Considerações iniciais

Este capítulo apresenta os principais conceitos relacionados ao assunto desta pesquisa, trazendo uma visão geral do desenvolvimento de compiladores para plataformas reconfiguráveis, em especial FPGAs. O capítulo inclui uma introdução sobre sistemas embarcados e as tecnologias envolvidas em projetos com aplicações reconfiguráveis. O texto aborda, ainda, o uso e desenvolvimento de compiladores, sendo apresentadas as etapas gerais do processo de compilação. Por fim, são mostradas características de ferramentas de HLS e de DSLs, duas das principais abordagens relacionadas ao desenvolvimento com FPGAs.

2.2 Sistemas embarcados e Computação Reconfigurável

De modo geral, sistemas embarcados são sistemas eletrônicos voltados para uma aplicação específica. Outra definição é de que tratam-se de sistemas para processamento de informações encapsulados em produtos, como eletrodomésticos, aparelhos multimídia, computadores de bordo, etc. Tais sistemas possuem, em geral, certas restrições tanto para seu desenvolvimento como para operação. Exemplos dessas restrições são: componentes de baixo custo, resposta em tempo real e baixo consumo de energia ([MARWEDEL, 2010](#); [DUBEY, 2009](#); [WOLF, 2008](#)).

Com o surgimento da tecnologia TTL (*Transistor-Transistor Logic*), componentes digitais passaram a ser utilizados no desenvolvimento de sistemas embarcados, antes compostos predominantemente por componentes analógicos. Essa mudança de tecnologia permitiu o desenvolvimento e a introdução dos microprocessadores programáveis em meados da década de 70 ([DUBEY, 2009](#)). Isto resultou em um novo paradigma para a construção de sistemas computacionais, trazendo uma maior flexibilidade, possível com o uso de *software*, em contraste

com a rigidez imposta pelas soluções compostas integralmente por componentes de *hardware*. Além desta flexibilidade, o uso de *software* permitiu uma maior abstração do *hardware*, cada vez mais complexo, implicando em uma maior velocidade de desenvolvimento (AHO *et al.*, 2001; VAREJÃO, 2004).

Neste cenário, os ASICs (*Application-Specific Integrated Circuits*) e os GPPs (*Generic Purpose Processors*) constituem duas opções para a computação de algoritmos em sistemas embarcados. Os primeiros são mais eficientes para uma dada aplicação, por serem desenvolvidos especificamente para tal, porém com menor flexibilidade. Já um GPP permite uma grande flexibilidade para a execução de vários algoritmos por meio de alterações de *software*, ao custo de um pior desempenho se comparado a um ASIC.

Uma solução intermediária é o uso de Computação Reconfigurável. Bobda (2007) define a Computação Reconfigurável como sendo “o estudo da computação utilizando dispositivos reconfiguráveis”. Tais dispositivos permitem o uso de lógica reconfigurável, onde o *hardware* pode ser alterado após a fabricação, permitindo que o desenvolvedor personalize a lógica do dispositivo como desejar (HSIUNG; SANTAMBROGIO; HUANG, 2009). Com isto, é possível atingir melhor desempenho que os GPPs e obter um ganho de flexibilidade em relação aos ASICs.

Em função da arquitetura empregada, os dispositivos reconfiguráveis podem ser classificados como sendo de grão fino (*fine-grain*) ou de grão grosso (*coarse-grain*). Dispositivos que possuem arquitetura de grão fino permitem alterações no seu circuito em nível de portas lógicas, possibilitando o tratamento de dados a nível de bits. Já nas arquiteturas de grão grosso, as alterações se dão em um nível de unidades funcionais mais complexas, como ULAs, multiplicadores, multiplexadores, etc (HAUCK; DEHON, 2008; BOBDA, 2007).

Dispositivos com arquiteturas de grão-fino têm maior destaque no contexto da Computação Reconfigurável. Exemplos destes dispositivos são os SPLDs¹ (PAL e PLA)², CPLDs³ e FPGAs, sendo estes últimos os principais dispositivos reconfiguráveis usados atualmente. Os FPGAs foram introduzidos pela empresa Xilinx⁴ na década de 80 (BOBDA, 2007; GOKHALE; GRAHAM, 2005).

A Figura 6 mostra uma representação da estrutura interna de um FPGA. Seus principais componentes são: um conjunto de blocos lógicos configuráveis, uma matriz interconexões programável e células de entrada e saída. Os blocos lógicos são usados para implementação das funções no FPGA, sendo interligados pela matriz de interconexões. O usuário pode programar todos estes componentes uma ou mais vezes, de forma total ou parcial, dependendo da tecnologia empregada na fabricação do FPGA (BOBDA, 2007).

Com o aumento da tecnologia empregada nos FPGAs, tornou-se possível implementar

¹ Simple Programmable Logic Device

² Programmable Array Logic e Programmable Logic Array

³ Complex Programmable Logic Device

⁴ <http://www.xilinx.com>

Figura 6 – Estrutura interna de um FPGA

Fonte: Adaptada de Bobda (2007).

sistemas complexos diretamente em *hardware*. Tais sistemas, conhecidos como SoCs (*System-on-Chip*), podem ser compostos por vários blocos de componentes programáveis, processadores, memória distribuída, etc, de acordo com a aplicação do usuário (PASRICHA; DUTT, 2008). Como exemplo da capacidade dos FPGAs atuais, podemos citar os dispositivos da família Stratix® V, desenvolvidos pela Altera⁵. Suas características incluem (Altera Corporation, 2012; Altera Corporation, 2010):

- *Tranceivers* com capacidade de 28Gbps, integrando largura de banda, eficiência de energia e ganho de desempenho.
- Alto nível de integração a 28nm, com cerca de 3,9 bilhão de transistores.
- Um milhão de elementos lógicos.
- Blocos de memória interna distribuídos, com velocidade superior a 600MHz e proteção ECC (*Error Correcting Code*).
- Blocos de DSPs de alta precisão e desempenho de 1 TFLOPS em operações de ponto flutuante.

Uma das etapas de desenvolvimento de *hardware* utilizando FPGA consiste na descrição dos circuitos através de linguagens próprias. Estas linguagens, chamadas HDLs (*Hardware Description Language*)(DUBEY, 2009), requerem que o desenvolvedor conheça e trate detalhes de *hardware* de baixo nível. Com estas linguagens a descrição do sistema é feita em nível RTL (*Register Transfer Level*), onde o circuito é descrito em forma de unidades funcionais

⁵ <http://www.altera.com/>

e de armazenamento, e por elementos de interconexão entre as unidades. Assim, cada um destes componentes é composto por um conjunto de portas lógicas, e modelado em forma de expressões booleanas (GAJSKI *et al.*, 2009). Apesar de não ser necessariamente um problema para desenvolvedores especializados, este nível de abstração restringe o uso dessas linguagens por programadores mais habituados a linguagens de alto nível, como C, Java, etc. Além disto, mesmo que os detalhes de implementação sejam conhecidos pelo programador, o tempo de desenvolvimento necessário pode ser inviável, dada a crescente complexidade dos circuitos, bem como os requisitos de *time-to-market* impostos.

Ao desenvolver um *software* para execução em um GPP, o programador se beneficia com o uso de compiladores que o permitem escrever o programa sem preocupar-se, obrigatoriamente, com a arquitetura do GPP e seu conjunto de instruções. De modo semelhante ao que ocorre no desenvolvimento de *software*, o desenvolvimento de *hardware* pode ser facilitado com o uso de compiladores que gerem descrições de *hardware* a partir de instruções e estruturas típicas de *software*, como vetores e laços de repetição, de forma eficiente. Contudo, o desenvolvimento de compiladores para dispositivos reconfiguráveis é complexo, dado o grau de personalização e as muitas opções possíveis com estes dispositivos (CARDOSO; DINIZ, 2009).

2.3 Compiladores

Um compilador trata-se de um *software* capaz de converter um programa codificado em determinada linguagem em um programa codificado em outra linguagem, mantendo seu significado (GRUNE *et al.*, 2001). O programa de entrada é, geralmente, escrito em linguagem de alto nível (C ou C++, por exemplo) sendo gerado um equivalente em linguagem de mais baixo nível (Assembly, etc.), para execução em uma máquina de determinada arquitetura. Há ainda compiladores que geram o programa de saída na mesma linguagem do código de entrada, após a realização de algumas otimizações de código, denominados de compiladores *source-to-source* (COOPER; TORCZON, 2011).

Compiladores estão diretamente ligados a vários segmentos da Ciência da Computação, como arquitetura de computadores, desenvolvimento de sistemas, metodologias de programação, linguagens formais e autômatos (HOPCROFT; MOTWANI; ULLMAN, 2001), etc., o que os torna parte importante da Ciência da Computação. Outras motivações para trabalhos envolvendo compiladores, citadas em Diniz (2011), são:

- Compiladores incorporam prática e teoria ao realizarem tarefas como verificação de código, seleção de instruções e geração de código otimizado para aumento de desempenho de *cache* na arquitetura alvo, etc.
- Muitas aplicações embarcadas possuem interatividade por meio de linguagens na forma de comandos, macros, *tags*, etc.

- Aplicações como MATLAB e Mathematica, dentre outras, possuem entradas formatadas como linguagens.
- O desenvolvimento de compiladores implica no tratamento de questões práticas como escalabilidade, eficiência, implementação de algoritmos, dentre outras questões ligadas à engenharia.

De forma geral, um compilador possui três componentes principais: *front-end*, *middle-end* e *back-end*. O *front-end* é responsável por verificações do código fonte de entrada, gerando uma nova representação do programa de entrada a partir das análises realizadas, denominada IR (*Intermediate Representation*). O *middle-end*, por sua vez, tem o papel de realizar otimizações no código a partir da IR por meio de uma série de operações como, por exemplo, eliminação de expressões irrelevantes e acessos redundantes à memória, etc. O *middle-end* pode ter como saída a mesma IR fornecida pelo *front-end* ou gerar uma nova IR. Por último, o *back-end* mapeia a IR gerada pelo *middle-end* em um código de saída de acordo com o *hardware* específico, possuindo, em alguns casos, integração com ferramentas e módulo próprios para síntese nas arquiteturas alvo. Cada uma destas etapas pode ser dividida em outras etapas menos complexas (AHO *et al.*, 2001; CARDOSO; DINIZ, 2009; COOPER; TORCZON, 2011). A Figura 7 ilustra a estrutura típica de um compilador.

Figura 7 – Estrutura típica de um compilador

Fonte: Adaptada de Cooper e Torczon (2011).

2.3.1 Front-end

O *front-end* produz como saída uma representação intermediária equivalente ao código de entrada. Para isto, é preciso verificar se o código trata-se de um código válido (AHO *et al.*, 2001). A validação ocorre de acordo com a especificação da linguagem na qual o código foi escrito.

Formalmente, uma linguagem é um conjunto de cadeias de símbolos, geradas a partir de uma gramática. Uma gramática, por sua vez, é uma quádrupla, definida por (Σ, V, S, P) , onde (ROSA, 2010; SIPSER, 2007):

- Σ é um conjunto finito de símbolos terminais, ou alfabeto terminal;
- V é um conjunto de símbolos não terminais, ou alfabeto variável;
- S é o símbolo inicial da gramática;
- P é um conjunto de regras para geração de cadeias, chamadas produções da gramática; e
- $\Sigma \neq \emptyset$, $\Sigma \cap V = \emptyset$, e $S \in V$.

Assim, o *front-end* analisa se os símbolos usados no código de entrada são definidos na gramática (análise léxica), se as cadeias de símbolos obedecem às regras definidas na gramática (análise sintática), e se estas tem significado, considerando o contexto do programa (análise semântica) (AHO *et al.*, 2001). Uma vez realizadas estas análises, o *front-end* gera uma IR equivalente ao código original, a qual será processada pelo *middle-end*.

A IR usada pode variar de acordo com o compilador e linguagem, mas deve ser robusta o suficiente para conter todos os aspectos relevantes do código inicial. As IRs podem ser estruturadas de forma linear (IRs lineares), gráfica (IRs gráficas), ou com uma combinação de elementos destas (IRs híbridas). IRs lineares representam o código original por meio de um pseudo-código que pode ser executado linearmente em uma máquina abstrata (ex. código de três endereços). Já as IRs gráficas utilizam estrutura de grafos, para representar o código por meio de vértices, arestas, árvores, etc (COOPER; TORCZON, 2011). As Figuras 8b e 8c trazem exemplos de IRs para a expressão $c = a * 3 + b * 2 + a$ (Figura 8a).

Figura 8 – Exemplos de representações intermediárias.(a) Expressão original;(b) Código de três endereços referente a (a);(c) Árvore sintática abstrata referente a (a)

Fonte: Adaptada de Cooper e Torczon (2011).

2.3.2 Middle-end

A partir da IR gerada pelo *front-end*, o *middle-end* realiza um conjunto de otimizações no programa. Por este motivo o *middle-end* também é chamado de otimizador (COOPER; TORCZON, 2011). Parte importante deste processo de otimização se dá com a realização de

transformações de código. Estas operações consistem em alterações feitas no código com a finalidade de otimizar o uso de recursos, melhorar desempenho ou tratar particularidades das aplicações (CARDOSO; DINIZ, 2009).

A aplicação de transformações é realizada levando-se em consideração características como dependência e disponibilidade de dados, possibilidade de execução paralela de instruções, *loops*, reuso de dados, etc. As transformações podem ser direcionadas à plataforma alvo a fim de permitir que sejam aproveitadas características como paralelismo, suporte a operações de ponto-fixo, dentre outras, diretamente ligadas à arquitetura. No caso de plataformas reconfiguráveis, podem visar a geração de circuitos lógicos com menor consumo de recursos de *hardware*, e, conseqüentemente, menor consumo de energia (CARDOSO; HÜBNER, 2011).

As transformações podem ser feitas em vários níveis de código e podem envolver tanto mudanças de representação de valores, como no caso de transformações em nível de bits, como alterações na ordem das instruções ou eliminação de *loops*, dentre outras operações. Alguns tipos de transformações de código são: transformações em nível de *bits*, em nível de instrução, em nível de *loop*, orientadas aos dados e orientadas à funções.

Nas transformações em nível de *bits* busca-se otimizar os recursos em nível de *bits*, limitando seu uso somente ao necessário para efetuar a computação. Por exemplo, variáveis inteiras de 16 bits podem ser transformadas em variáveis de 8 bits caso seu valor nunca exceda 255. Outro exemplo deste tipo de transformação seria a simplificação de funções booleanas, reduzindo o número de portas lógicas do circuito. Transformações em nível de instrução buscam minimizar o uso de recursos substituindo um dado conjunto de instruções por outro que gere resultado equivalente a um menor custo. Eliminação de expressões e simplificações algébricas são exemplos de técnicas usadas nestas transformações. Transformações aplicadas em nível de *loop* buscam explorar/viabilizar a execução paralela das instruções que estão dentro do *loop*, bem como reutilizar o resultado de operações entre as iterações do *loop*. Em muitas destas transformações há restrições em relação ao gerenciamento dos dados. Nas transformações orientadas a dados, busca-se gerenciar os dados, organizando-os e armazenando-os de forma a otimizar o acesso. Estas transformações possuem, em muitos casos, relação direta com as transformações de *loops*, principalmente quando se trata de operações envolvendo *arrays* (CARDOSO; DINIZ, 2009).

A seguir são apresentadas algumas transformações, com descrições simplificadas das operações que são realizadas e exemplos de cada transformação. Maiores detalhes sobre estas e outras técnicas de transformação e otimização de código podem ser encontrados em Cardoso e Diniz (2009), Cardoso, Diniz e Weinhardt (2010) e Leupers (2000).

- a) *Bit-Width Narrowing* → Redução do número de bits usados na representação de valores.
- b) *Bit-Level Optimizations* → Simplificação de expressões booleanas.
- c) Alterações na representação de ponto flutuante → Representação de números com ponto

flutuante usando um formato diferente do padrão IEEE 754 (IEEE, 1985). Pode ser usada a representação com ponto fixo ou com outro formato fora do padrão. Deve ser considerado a relação entre precisão da representação e custo computacional das operações numéricas.

- d) *Operator Strength Reduction* → Substituição de um operador por outro equivalente, porém menos custoso computacionalmente.
- e) *Height Reduction* → Reordenação de expressões aritméticas, com base nas propriedades associativa, comutativa e distributiva.
- f) *Code Motion* → Alteração na ordem das instruções buscando explorar a relação entre concorrência e uso de recursos.
- g) *Loop Normalization* → Alteração do valor inicial da variável de controle do *loop* para zero e ajuste do limite da contagem e dos índices no corpo do *loop*.
- h) *Loop Unrolling* → Replicação de instruções do corpo do *loop* e ajuste do índice de controle. Busca explorar o paralelismo e minimizar o *overhead* no controle do *loop*.
- i) *Loop Tiling / Blocking* → Divisão das iterações do *loop*, reestruturando-as em blocos.
- j) *Loop Merging / Fusion* → Fusão de dois *loops* em um único *loop*, concatenando as instruções de ambos os *loops*.
- k) *Loop Distribution* → Distribuição das instruções do corpo de um *loop* em dois *loops* com o mesmo intervalo de controle. Opera de forma inversa à transformação de *Loop Fusion*.
- l) *Loop Folding / Pipelining* → Sobreposição de instruções de um *loop* executadas em iterações diferentes, mantendo a dependência de dados. Com isto, inicia-se uma operação sem que a anterior termine completamente.
- m) *Loop Permutation / Interchange* → Inversão entre os *loops* interno e externo.
- n) *Loop Splitting* → Divisão de um *loop* de N iterações por dois de $N/2$ iterações.
- o) *Distribuição de dados* → Particionamento dos dados de um *array*, distribuindo-os em vários *arrays* e mapeando-os em diferentes memórias.
- p) *Replicação de dados* → Criação de várias cópias dos dados, armazenando-as em memórias distintas. Permite o acesso simultâneo a diferentes dados, originalmente em um único *array* porém com o custo elevado de armazenamento.
- q) *Reuso de dados* → Reutilização de um dado já lido da memória, sem a necessidade de nova leitura.

- r) *Data packing* → Armazenamento de mais de uma variável em um mesmo endereço de memória. Considerando endereços de 32 *bits* é possível, por exemplo, armazenar duas variáveis de 16 *bits* ou quatro de 8 *bits* em um mesmo endereço, reduzindo número de acessos.

Definir quais transformações e análises serão aplicadas durante o processo de compilação é um problema complexo. Esta escolha depende da relação entre os benefícios desejados e os custos associados. Melhorar o acesso aos dados por meio de replicação, por exemplo, tem como benefício aumentar a velocidade de acesso, ao custo de ser necessário maior capacidade de armazenamento (registradores, RAMs, etc.). Além disto, enquanto algumas transformações exploram o paralelismo de forma mais efetiva, como nos casos de transformações envolvendo *loops*, questões como transferência de dados e acesso à memória ainda constituem gargalos no que diz respeito à execução paralela de instruções (LIU *et al.*, 2009). Assim, num cenário onde há várias opções para a geração automática de arquiteturas reconfiguráveis, a escolha de uma solução deve considerar uma série de critérios, como desempenho, consumo de energia, área ocupada, etc., e priorizar aquelas que apresentem a melhor configuração dentro de certas diretrizes.

2.3.3 Back-end

As tarefas finais do processo de compilação são executadas pelo *back-end*. Em suma, o *back-end* produz o código para a arquitetura alvo a partir da IR gerada pelo *middle-end*. O *back-end* seleciona operações na arquitetura alvo que correspondam a cada operação descrita na IR, determinando ordem de execução e alocação de dados em registradores ou memórias (COOPER; TORCZON, 2011). Considerando como arquitetura alvo, por exemplo, um processador x86, o *back-end* geraria o código final contendo instruções *Assembly* pertencentes ao ISA (*Instruction Set Architecture*) deste processador e faria a alocação de dados de acordo com a disponibilidade de registradores e memória.

No caso de compiladores *source-to-source*, o *back-end* gera código de saída escritos em uma linguagem de alto nível. Neste caso a linguagem geralmente é a mesma do código de entrada, mas, dada a segmentação do processo de compilação, é possível gerar o código de saída em uma linguagem de alto nível diferente da usada no código de entrada (COOPER; TORCZON, 2011). No contexto deste trabalho, o *back-end* gerará um código que especificará a arquitetura a ser configurada no FPGA.

2.4 Síntese de alto nível

Na Computação Reconfigurável os compiladores podem ser usados para geração de *hardware* por meio do processo de síntese de alto nível (HLS – *High-Level Synthesis*). Neste

processo, o código em linguagem de alto nível é utilizado para a geração de uma descrição do *hardware* em nível RTL, onde as instruções do programa são mapeadas em componentes de *hardware*, descritos em alguma HDL. Assim, pode-se gerar circuitos de *hardware* que realizem uma computação equivalente ao *software* original.

Durante o processo de HLS, três tarefas são executadas, as quais são: alocação, associação e escalonamento. A tarefa de alocação consiste em selecionar todos os componentes que serão utilizados no circuito e definir a ligação entre os componentes. Para isto é utilizada uma biblioteca de componentes RTL. Outra tarefa do processo é a associação, a qual associa as variáveis aos blocos de memória e registradores, bem como as operações às unidades funcionais do sistema. O sincronismo do sistema é determinado na tarefa de escalonamento. Nesta tarefa define-se a execução das operações em função dos ciclos de relógio. Todas as tarefas devem ser realizadas de forma coordenada, uma vez que decisões tomadas durante uma delas podem afetar as demais (GAJSKI *et al.*, 2009).

Segundo Gupta e Brewer (2008), apesar da potencial melhoria obtida com o processo de HLS, uma série de fatores dificulta sua ampla adoção por parte da comunidade. Dentre estes, destacam-se:

- Resultado final ruim, gerando *hardware* aquém do desejado;
- Falta de ferramentas de verificação;
- Alto erro em processos que envolvem estimação de características físicas;
- Complexidade na descrição das restrições temporais dos circuitos;
- Falta de interesse por parte dos desenvolvedores, dentre outros.

Além destes fatores, Gupta e Brewer (2008) citam, ainda, quatro requisitos básicos a serem cumpridos para viabilizar o processo de gerar uma descrição de *hardware* a partir de um programa em alto nível. São eles:

Concorrência → Formas de especificação para operações concorrentes, incluindo modelos de paralelismo de *hardware* e múltiplos *clocks*.

Determinismo temporal → Garantias de determinismo temporal, permitindo a previsão e simulação do comportamento de partes do circuito mesmo sem a especificação completa do comportamento do circuito;

Programação reativa → Mecanismos para modelagem de interações não-terminais com outros componentes, tratamento de exceções, e eventos; e

Abstração estrutural → Permitir a construção de sistemas complexos a partir da composição de sistemas menores.

2.5 Linguagens de Domínio Específico - DSLs

Como alternativa ao uso de ferramentas de HLS, tem-se a opção por Linguagens de Domínio Específico (DSLs –*Domain-Specific Languages*). Voelter (2013) define uma DSL da seguinte maneira:

Uma DSL é, simplesmente, uma linguagem otimizada para uma determinada classe de problemas, chamada domínio. É baseada em abstrações intimamente alinhadas com o domínio para o qual a linguagem é construída. Linguagens especializadas também trazem sintaxe própria para expressar essas abstrações concisamente. Em muitos casos são notações textuais, mas tabelas, símbolos (como em matemática) ou gráficos também podem ser úteis. Assumindo que a semântica dessas abstrações está bem definida, tratam-se de um bom ponto de partida para a expressão de programas para um domínio especializado efetivamente⁶(VOELTER, 2013, p. 28).

Diferente do que ocorre com as linguagens de propósito geral (GPLs –*General Purpose Languages*), as DSLs são restritas, muitas vezes usadas apenas para o desenvolvimento de partes da aplicação. Seu uso é motivado em casos onde a flexibilidade de desenvolvimento, própria das GPLs, é preterida em relação ao ganho de produtividade e desempenho oferecido por uma DSL com foco no domínio em questão. O Quadro 1 apresenta características que diferenciam DSLs e GPLs(VOELTER, 2013).

Quadro 1 – Características gerais de GPLs e DSLs

	GPLs	DSLs
Domínio	Ampla e complexa	Pequena e bem definida
Tamanho da linguagem	Grande	Pequena
Turing completude	Sempre	Geralmente não
Abstrações definidas pelo usuário	Sofisticadas	Limitadas
Tempo de vida	Anos a décadas	Meses a anos, dependendo do contexto
Desenvolvidas por	Gurus ou comitês	Poucos engenheiros e especialistas no domínio
Comunidade de usuários	Ampla, anônima e difundida	Pequena, acessível e local
Evolução	Lenta, geralmente padronizada	Rápida
Depreciação/ mudanças incompatíveis	Quase impossível	Viável

Fonte: Adaptada de Voelter (2013, p. 31).

As DSLs podem ser classificadas como externas e internas. Uma DSL externa é aquela que separa-se da linguagem principal da aplicação com a qual trabalha, geralmente com sintaxe e

⁶ Tradução nossa

semântica própria. Uma DSL interna consiste em um subconjunto de uma GPL usado para tratar determinados aspectos do sistema de um modo particular. Apesar do código de uma DSL interna ser válido para a GPL da qual deriva, as sequências entre chamadas e métodos obedecem regras e dependências próprias, formando um tipo de sintaxe. Esta característica diferencia uma DSL de uma API, apesar dessa diferenciação ainda gerar discussões (FOWLER, 2010; VOELTER, 2013).

No tocante ao desenvolvimento de *hardware*, DSLs são tidas como soluções com nível de abstração intermediário entre linguagens de descrição de *hardware* e o uso de HLS, fornecendo um bom equilíbrio entre produtividade, desempenho e portabilidade. As restrições impostas pela própria linguagem podem ser exploradas pelo seu compilador de forma a obter um alto nível de otimização. Outra característica das DSLs é restringir a construção dos programas, garantindo a correção destes e evitando análises e otimizações genéricas e sofisticadas, como o que ocorre com compiladores de uma linguagem de propósito geral (MOHAPI; WINBERG; INGG, 2014).

Contudo, o custo de desenvolvimento de uma DSL passa pela implementação de seu compilador. Além disto requerer conhecimentos em várias áreas da computação, também é necessário que o desenvolvedor tenha certo grau de especialização relativo ao domínio específico que está sendo tratado. Uma vez que não é incomum DSLs focarem em arquiteturas próprias, é importante observar as atualizações e as modificações realizadas no sistema, em termos de *hardware* e *software*, de forma manter a linguagem viável. Isto requer um esforço contínuo e pode ser um obstáculo para a adoção da DSL (MOHAPI; WINBERG; INGG, 2014).

2.6 Considerações finais

Neste capítulo foi mostrada uma visão geral do desenvolvimento de compiladores para FPGAs, com ênfase nos principais conceitos relacionados a projetos de arquiteturas reconfiguráveis. Em função das dificuldades inerentes ao desenvolvimento de *hardware* em plataformas reconfiguráveis, vários esforços têm sido realizados no tocante à disponibilização, tanto no âmbito acadêmico, quanto no comercial, de ferramentas que auxiliem o desenvolvedor nesse processo.

A criação de ferramentas robustas e eficientes requer conhecimentos avançados, dada a complexidade desta tarefa, que, além de requisitos técnicos, envolve diferentes paradigmas de programação. Esta complexidade tem motivado pesquisas voltadas ao desenvolvimento de ferramentas para HLS e/ou DSLs. O Capítulo 3 apresenta algumas ferramentas desenvolvidas neste sentido.

TRABALHOS RELACIONADOS

3.1 Considerações iniciais

Dados os desafios inerentes ao desenvolvimento de arquiteturas reconfiguráveis para computação de alto desempenho, certos projetos buscam soluções para auxiliar o desenvolvedor em nível de implementação. Tais propostas buscam, em geral, facilitar o processo de desenvolvimento por meio da construção de ferramentas especializadas na geração automática de *hardware*.

Este capítulo apresenta uma série de trabalhos que visam contribuir para o desenvolvimento de *hardware* reconfigurável com FPGAs. Outros projetos e ferramentas, não abordadas neste texto, tratam, geralmente, de aspectos pouco relacionados aos objetivos desta proposta, como, por exemplo, arquiteturas de granularidade grossa ou execução em plataformas e ambientes específicos.

Dentre os esforços para facilitar o desenvolvimento de ferramentas para geração de *hardware* reconfigurável, pode-se, segundo [Coutinho, Jiang e Luk \(2005\)](#), distinguir duas linhas principais de abordagens utilizadas, no tocante ao nível de abstração imposto. Na primeira, o programador se preocupa com o comportamento algorítmico do problema, sendo abstraídos os detalhes de baixo nível. Já na segunda, o programador interfere diretamente em detalhes de baixo nível, como sincronismo e sinalização, tendo um maior controle sobre a arquitetura gerada.

Em geral, a abordagem comportamental trata do desenvolvimento de ferramentas para geração de *hardware* a partir da tradução de códigos escritos em linguagens de alto nível. As vantagens desse tipo de abordagem são a facilidade de uso, alta produtividade e manutenibilidade do sistema. Em termos de desvantagens, muitas ferramentas não suportam todos os aspectos típicos das linguagens, como uso de ponteiro, estruturas, *loops* irregulares, etc. Além disso, as opções disponíveis ao programador são limitadas, por tratar-se de uma abordagem que requer pouca intervenção humana. Essa característica impede que o programador atue na correção e/ou

ajuste de arquiteturas pouco eficientes.

Já as abordagens de mais baixo nível fornecem um maior controle ao programador. Estas incluem a criação e definição de novas linguagens, semelhantes a uma linguagem de alto nível, que incluam diretivas específicas para controle do *hardware*. Um problema deste tipo de abordagem é a necessidade de conhecimentos específicos de *hardware*, que nem sempre o programador possui (HUONG; KIM, 2011). Portanto, podem resultar em baixa produtividade e na dificuldade de manutenção do sistema.

Os trabalhos apresentados nesta seção tratam-se de ferramentas de HLS, linguagens e *frameworks* usados para desenvolvimento de *hardware*, considerados relacionadas à linguagem LALP. Apesar deste trabalho focar-se em uma DSL, e não em uma ferramenta de HLS, o capítulo apresenta projetos com ambas as abordagens, uma vez que as duas são alternativas ao uso de linguagens de descrição de *hardware*, como VHDL e Verilog, no que diz respeito a facilitar o desenvolvimento para FPGAs.

Note-se que a classificação entre linguagens/*frameworks* (Seção 3.2) e ferramentas de HLS (Seção 3.3) é arbitrária, baseada nas características principais de cada trabalho. Portanto, pode haver casos onde DSLs são usadas em ferramentas de HLS, como em C-to-Silicon (Seção 3.3.3), por exemplo, ou onde o compilador de uma DSL inclui etapas de HLS, como em FCUDA (Seção 3.2.7), por exemplo.

3.2 Linguagens e *Frameworks*

3.2.1 *BSAT e StReAm*

Em Mencer *et al.* (2001) são mostrados os compiladores BSAT e StReAm (MENCER *et al.*, 2000), dois trabalhos iniciais em relação ao uso de DSL para produção de *hardware*. Ambos os compiladores são construídos sobre a plataforma PAM-Blox (MENCER; MORF; FLYNN, 1998), que implementa uma biblioteca de classes com *templates* C++ para FPGAs da família Xilinx XC4000.

A Figura 9 apresenta a arquitetura usada na plataforma PAM-Blox. Essa plataforma possui três componentes principais: *PamDC*, *PamBlox* e *PaModules*. Note-se que os termos "*PAM-Blox*" e "*PamBlox*" referem-se a conceitos distintos, sendo o primeiro referente a todo o ambiente de desenvolvimento e o segundo aos *templates* de *hardware* fornecidos. *PamDC* trata-se do compilador que fornece primitivas de baixo nível, as quais são compiladas para o formato XNF (*Xilinx Netlist Format*). Nesta arquitetura, a aplicação tem acesso a todos os componentes.

BSAT foca-se em resolver o problema da satisfatibilidade em expressões booleanas descritas na forma normal conjuntiva (FNC). Os circuitos BSAT são descritos em C++, de acordo com os modelos PAM-Blox e as expressões são mapeados em uma matriz de FSMs usando

Figura 9 – Estrutura da plataforma PAM-Blox

Fonte: Adaptada de [Mencer, Morf e Flynn \(1998\)](#).

Verilog. A arquitetura BSAT inclui um bloco de dedução lógica, que calcula o resultado da expressão booleana, dada uma atribuição parcial de uma variável da expressão. BSAT também inclui um controlador global para iniciar a computação e tratar da comunicação externa.

StReAm é destinado a projetar objetos *stream* em C++. Com isto, beneficia-se de características como herança e sobrecarga de operadores. As expressões StReAm são usadas para criar um grafo de fluxo de dados, onde os nós são as unidades de operação aritmética. O escalonador percorre o grafo e calcula automaticamente as dependências de dados e a latência geral da arquitetura, além de criar FIFOs distribuídas e tratar da sinalização para os componentes sequenciais.

3.2.2 ASC

ASC (A *Stream Compiler*) ([MENCER, 2006](#)) traz uma evolução das ideias utilizadas no StReAm. Trata-se de uma biblioteca C++ baseada em uma versão mais recente do PAM-Blox (PAM-Blox II), e fornece suporte para várias famílias de FPGA Xilinx (Xilinx 4000, Xilinx Virtex, Virtex 2, Virtex 4, Spartan 2 and Spartan 3).

Em ASC ([Figura 10](#)), os *loops* são implementados em fluxos (*streams*) que incluem parâmetros para otimização do *throughput*, da latência ou da área consumida. A biblioteca ASC inclui um conjunto de funções pré-definidas para lidar com mapeamento de baixo nível como, por exemplo, registradores, sinais e relógio. Diferente do que ocorre em LALP/LALP+, onde o programador apenas descreve o algoritmo de forma comportamental, o código ASC incorpora um *overhead* que inclui configurações de baixo nível, como o mapeamento de memória e recursos para suporte a exploração do espaço de projeto.

3.2.3 PyHDL

Em [Haglund et al. \(2003\)](#) os autores apresentam PyHDL (*Python Hardware Description Language*), uma interface para criação de circuitos em FPGA baseada em *scripts*. PyHDL tenta

Figura 10 – Estrutura do compilador ASC

Fonte: Adaptada de [Mencer \(2006\)](#).

reduzir o tempo de desenvolvimento de projetos de *hardware* em uma abordagem de alto nível por meio de *scripts* Python.

O processo adotado em PyHDL é composto de duas etapas: 1) projetar os circuitos com os *scripts* e componentes pré-compilados, usando modelos PAM-Blox; e 2) converter e compilar em novos componentes os circuitos descritos. Com esta abordagem, o *hardware* produzido é similar ao gerado pelo PAM-Blox, mas com uma redução do *overhead* de compilação, o que diminui o tempo de implementação e facilita prototipagem rápida.

3.2.4 MaxJ

MaxJ ([FU et al., 2014](#)) é uma DSL baseada em Java e usada no compilador MaxCompiler ([FU et al., 2014](#); [PELL et al., 2013](#)) para programar *kernels* em uma abordagem orientada a fluxo de dados para computação de algoritmos científicos. As soluções de *hardware* geradas com MaxJ para estas aplicações são compostas por CPUs e aceleradores para processamento dos trechos críticos, compostos, principalmente, por *loops*.

As arquiteturas são geradas com base em DFEs (Data Flow Engines), que são blocos reconfiguráveis implementados em placas PCI fornecidas pela Maxeler Technologies¹. Várias DFEs podem ser configuradas em paralelo para geração de nós computacionais, como mostrado na [Figura 11](#), comunicando-se por canais chamados *MaxRings*.

Os *kernels* descrevem como o grafo de fluxo de dados do algoritmo deve ser construído. O compilador usa compressão de dados para armazenamento em bancos de memória *on-chip*.

¹ <http://www.maxeler.com>

Os dados são armazenados em blocos e a organização dos *arrays* é computada automaticamente pelo compilador de forma a otimizar o fluxo de dados.

Figura 11 – Nó computacional utilizando DFEs

Fonte: Adaptada de Pell *et al.* (2013).

3.2.5 OpenSPL

OpenSPL (*Open Spatial Programming Language*) (CONSORTIUM, 2013) é outra linguagem baseada em DFEs que implementa o conceito de computação espacial. Neste paradigma, os fluxos de dados e de controle do programa são desacoplados, com as operações executando, por padrão, em paralelo.

Em termos de arquitetura de *hardware*, OpenSPL baseia-se em unidades de *dataflow*, chamadas *Spatial Computing Substrate* (SCS). Tais unidades são escalonadas de modo a otimizar o acesso, diminuindo o fluxo de dados. As SCS incluem, unidades aritméticas, conexões personalizadas e memórias, podendo ser compostas por um ou mais *kernels* paralelos, interconectados segundo o fluxo de dados comum.

3.2.6 OpenCL e Silicon OpenCL

OpenCL (GROUP, 2014; STONE; GOHARA; SHI, 2010) é um *framework* para programação paralela destinado à computação heterogênea de uso geral. É um padrão adotado pelos principais fabricantes de FPGA, Xilinx e Altera, contando com compiladores e bibliotecas disponíveis para ambas as plataformas (ALTERA, 2015; WIRBEL, 2014).

Em OpenCL a aplicação é segmentada de forma que parte é executada em um programa *host* em outras partes, os *kernels*, são executadas em plataformas específicas, como GPUs e FPGAs. A Figura 12 ilustra essa arquitetura. O *host* comunica-se com os *kernels* através de

comandos, os quais são enfileirados e tratados de acordo com um conjunto de estados pré-definidos.

Figura 12 – Arquitetura OpenCL com *kernels* mapeados em FPGAs da Altera

Fonte: Adaptada de Altera (2013).

Os *kernels* são executados em paralelo e agrupados em *work-groups* independentes para uma computação em grade. O sincronismo é feito utilizando barreiras de sincronização definidas a partir de pontos entre comandos contidos nas diversas filas de comandos.

OpenCL utiliza um modelo de memória segmentada em regiões, a qual é compartilhada entres os componentes. Com isto, pode haver sobreposição de escrita na memória, uma vez que grupos distintos executam concorrentemente.

Silicon OpenCL (SOpenCL) (OWAIDA *et al.*, 2011) é uma abordagem para desenvolvimento de aceleradores baseada em OpenCL. SOpenCL faz uso da infraestrutura do compilador LLVM (LATTNER; ADVE, 2004), com a restrição de suportar apenas aninhamento de um *loop*.

O *front-end* SOpenCL usa transformações *source-to-source* para gerar *kernels* paralelos a partir do programa OpenCL de entrada. Os códigos em em RTL são gerados considerando um modelo de arquitetura onde o acesso aos dados e a computação são desacoplados a fim de reduzir a latência.

Apesar de ser uma iniciativa promissora, utiliza um OpenCL específico, que pode não ser adequado para algumas aplicações. Além disso, o processo de HLS precisa ser melhorado de forma a gerar *hardware* mais eficientes.

3.2.7 FCUDA

FCUDA (PAPAKONSTANTINO *et al.*, 2013; PAPAKONSTANTINO *et al.*, 2009) é uma abordagem para geração de código RTL a partir de código CUDA, uma API desenvolvida pela NVIDIA para programação de GPUs. FCUDA baseia-se na infraestrutura do compilador Cetus (DAVE *et al.*, 2009) de forma a realizar transformações no código de entrada, o qual é

submetido para a ferramenta Autopilot (ZHANG *et al.*, 2008) para a execução da etapa de síntese de alto nível. A Figura 13 mostra este fluxo.

Figura 13 – Fluxo de compilação FCUDA

Fonte: Adaptada de Papakonstantinou *et al.* (2009).

O programador precisa modificar o código dos *kernels* CUDA, inserindo diretivas *#pragmas* próprias, de forma a tornar explícito o paralelismo presente nos *kernels* CUDA. Estes, por sua vez, são convertidos em código RTL, pelo Autopilot, de forma a executarem concorrentemente. Além disto, os *#pragmas* FCUDA são usados para especificar características como o sincronismo entre os *kernels*, granularidade dos blocos, organização de dados e bancos de memória, dentre outras.

3.2.8 Handel-C

Handel-C (Agility Design Solutions Inc., 2007) é uma linguagem de programação imperativa que expõe alguns componentes dos FPGAs ao programador, como memórias e FIFOs. Em Handel-C assume-se que as atribuições possuem latência de um ciclo de *clock*. Desta forma, as operações associadas ao mesmo ciclo em uma expressão são controladas pelo programador através da decomposição de expressões, com a concorrência sendo explicitamente especificada com o uso de instruções *par* para delimitar estas seções de código.

O fluxo de execução é dividido quando são encontradas seções paralelas, as quais são iniciadas simultaneamente. Caso a execução de uma seção termine antes das demais, esta é bloqueada até que todas terminem. Estas seções constituem blocos de *hardware* diferentes, os quais comunicam-se por meios de canais de comunicação. Tais canais podem usar FIFOs para enfileirar os dados, realizando uma comunicação assíncrona. Outra possibilidade é a não utilização de FIFOs, caso onde a comunicação ocorre de forma sincronizada entre os blocos envolvidos.

No tocante à geração de *hardware*, é requerido que o programador especifique detalhes da arquitetura, como fonte de *clock*, sinais e portas. Em termos de memória, deve-se especificar em código os tipos de memória (RAM ou ROM), bem como se estas são *on-chip* ou *off-chip*,

sendo geradas memórias *multiport* para que os acessos ocorram em paralelo. Além disto, cada posição de um *array* é implementada como um registrador.

Apesar da proposta de ser uma linguagem com suporte completo a ANSI-C, Handel-C requer que o programador conheça detalhes de baixo nível. A linguagem impõem, ainda, restrições de formatação de código, além de diferenças sintáticas para incorporar os mecanismos que tratam do *hardware*, requerendo certos arranjos para o correto funcionamento de algumas seções, como no caso dos *loops*.

3.2.9 Haydn-C

Haydn-C (COUTINHO; JIANG; LUK, 2005) é uma linguagem baseada em Handel-C que permite ao programador incluir diretivas de *pipeline* no código. A abordagem utilizada combina abordagens comportamental e de baixo nível (orientada a ciclos), onde o programador pode optar por escrever uma descrição comportamental do código, mas também pode inserir diretivas para controlar o escalonamento da arquitetura a ser gerada. Haydn-C é uma linguagem baseada em componentes e, como Handel-C, usa instruções *par* para definir seções paralelas.

A Figura 14 mostra o ciclo de compilação usado em Haydn-C. Neste processo o compilador busca por porções de código com anotações referentes às ações a serem tomadas, como transformações e escalonamento. Essas porções são então processadas e substituídas pelos novos trechos gerados que darão origem ao *hardware*.

Haydn-C trabalha com dois tipos de regras semânticas de temporização. O primeiro, chamado *strict timing*, define um modelo com um conjunto de regras fixas, relativas ao controle dos fluxos de execução sequencial e paralelo e à duração das operações. O segundo modelo, chamado *flexible timing*, lida com as transformações de código. Neste modelo, algumas regras são flexibilizadas a fim de tornar as transformações viáveis. A condição para esta flexibilização é que o resultado do processo permaneça correto.

3.2.10 Single Assignment C

Single Assignment C (SA-C) (NAJJAR *et al.*, 2003) é uma DSL derivada de C, mas com a restrição, explícita no nome, de permitir que cada variável seja atribuída apenas uma vez. É uma linguagem usada para gerar arquiteturas em FPGAs, com parte do código executada em um computador *host*, em uma abordagem de co-projeto.

O compilador implementa uma série de transformações, em nível de *loops*, dados e expressões. Além disto SA-C trata os *loops* de forma especial, com o resultado computado sendo retornado ao final das iterações.

A Figura 15 mostra o fluxo de desenvolvimento com SA-C. O compilador parte de um código de entrada, o qual é analisado para definir quais *loops* serão executados em FPGA. Além disto, SA-C permite que o programador incorpore código VHDL diretamente, realizando as

Figura 14 – Fluxo de compilação em Haydn-C

Fonte: Adaptada de Coutinho, Jiang e Luk (2005).

devidas conexões automaticamente. As partes sequenciais são traduzidas em código C para execução em CPU.

Para geração de *hardware*, o compilador gera um grafo de dependência de dados e fluxo de controle (DDCF) a partir do código de entrada, o qual é processado para geração de um grafo arquitetural. Este último inclui informações de temporização e é utilizado para gerar código VHDL.

3.2.11 SystemC

SystemC (IEEE Comp. Society, 2012; GRÖTKER *et al.*, 2002) é uma biblioteca de classes C++ para desenvolvimento de sistemas e de *hardware* padronizada pelo IEEE. SystemC opera com a definição de funções C++ como processos concorrentes, utilizando uma hierarquia de módulos. A comunicação entre módulos e processos é realizada por meio de canais próprios a este fim. Após a criação dos módulos, os processos são escalonados e sincronizados durante uma etapa de simulação. O escalonador usa eventos (como escrita e leitura) para determinar a ordem de execução dos processos, incluindo aqueles que devem ser executados concorrentemente.

Figura 15 – Fluxo de desenvolvimento SA-C

Fonte: Adaptada de Najjar *et al.* (2003).

O [Quadro 2](#) traz um resumo comparativo das características gerais das linguagens e *frameworks* apresentados.

3.3 Ferramentas para Síntese de Alto Nível

3.3.1 LegUp

LegUp (Canis, Andrew *et. al.*, 2013b; Canis, Andrew *et. al.*, 2013a) é uma ferramenta para síntese de alto nível desenvolvida pela Universidade de Toronto, que produz código em Verilog a partir de programas em C. Para isto LegUp utiliza implementa um *back-end* utilizando o compilador LLVM e sua IR, seguindo o fluxo mostrado na [Figura 16](#).

A compilação com LegUp pode ser feita com focos distintos, podendo ser apenas em nível de *software*, gerando código *assembly* para um processador *soft-core*, apenas em nível de *hardware*, gerando um circuito referente ao código de entrada, ou híbrida, onde parte da aplicação é mapeada em *hardware* e a outra é compilada em código *assembly*. Neste último caso a integração é transparente ao desenvolvedor. Assim como outras ferramentas de HLS, LegUp não suporta alocação dinâmica nem recursividade.

LegUp está atualmente na versão 4.0 e suporta um conjunto limitado de dispositivos e placas Altera. Para a geração de *hardware* LegUp faz uso de componentes específicos dos

Quadro 2 – Resumo das características gerais das linguagens e *frameworks* apresentados

DSL/ <i>Framework</i>	Linguagem base	Arquitetura alvo	Domínio	Outras características
BSAT	C++	FPGA / Xilinx	Satisfatibilidade de expressões	PAM-Blox
StReAm	C++	FPGA / Xilinx	Objetos stream a fluxo de dados	PAM-Blox
ASC	C++	FPGA / Xilinx	Stream de loops	PAM-Blox II
PyHDL	Python	FPGA / Xilinx	Integração de componentes	PAM-Blox
MaxJ	Java	FPGA / Maxeler	Kernels para aplicações científicas	Uso de DFEs
OpenSPL	N/A	FPGA / Maxeler	Kernels dataflow	Uso de DFEs
OpenCL	C	FPGA / CPU / GPU	Computação heterogênea	Computação em grade
SOpenCL	C	FPGA	Kernels OpenCL	Subconjunto de OpenCL
FCUDA	C / CUDA	FPGA	Kernels CUDA	Inclui HLS
Handel-C	C	FPGA	Aplicações paralelas	Paralelismo explícito
Haydn-C	C	FPGA / CPU	Kernels em pipelining	Inserção de diretivas de pipeline
SA-C	C	FPGA / CPU	Loops com retorno de valor	Atribuição única de variáveis
SystemC	C++	FPGA	Módulos de <i>hardware</i> em baixo nível	Padronizada pelo IEEE

Fonte: Elaborada pelo autor.

Figura 16 – Fluxo de compilação do LegUp

Fonte: Adaptada de Toronto (2015).

FPGAs suportados, o que dificulta seu uso em outras plataformas.

3.3.2 Impulse C

Impulse C (XU *et al.*, 2010; ANTOLA *et al.*, 2007) é uma ferramenta para desenvolvimento de aplicações paralelas em co-projetos de *hardware* e *software* a partir de C. O fluxo de compilação do Impulse C é mostrado na Figura 17.

Figura 17 – Fluxo de desenvolvimento usando o Impulse C

Fonte: Adaptada de Impulse Accelerated Technologies (2015).

Com o Impulse C tem-se a possibilidade de criar componentes e interfaces para aplicações *stream*. Para isto, é preciso particionar a aplicação entre *hardware* e *software*. O *software* é responsável pela leitura e escrita nos blocos de *hardware*. Isto ocorre por meio da especificação de canais de comunicação entre os processos presentes no código. Estes canais são mapeados em *hardware* utilizando FIFOs.

Impulse C fornece uma biblioteca para implementação das arquiteturas e criação de interfaces com *hosts* externos. O compilador baseia-se em diretivas inseridas no código para implementação de transformações, como *loop unrolling/pipelining*, bem como para o mapeamento dos blocos em *hardware* e definição dos canais de comunicação.

Trata-se de uma ferramenta comercial que gera código VHDL para FPGAs tanto da Altera quanto da Xilinx. O ambiente de desenvolvimento inclui, ainda, ferramentas para depuração e simulação das arquiteturas geradas.

3.3.3 Cadence C-to-Silicon

C-to-Silicon (CADENCE, 2015) é uma ferramenta de HLS desenvolvida pela Cadence®. Tem como entrada uma descrição de *hardware* feita em System C, gerando arquiteturas RTL em Verilog, tanto para Xilinx quanto para Altera. O fluxo de compilação utilizado é mostrado na Figura 18.

Figura 18 – Fluxo de compilação C-to-Silicon

Fonte: Adaptada de Cadence (2015).

Esta ferramenta utiliza uma interface gráfica para apresentar um grafo a fluxo de dados ao programador. Desta forma, o programador pode realizar alterações e manipular as micro-arquiteturas geradas.

O mesmo código pode ser usado para gerar arquiteturas diferentes, uma vez que este processo baseia-se em requisitos e restrições próprios para cada plataforma alvo. Assim, o código de entrada trata da funcionalidade, enquanto o compilador implementa a exploração do espaço de soluções, considerando requisitos de potência, desempenho e área.

3.3.4 C to Fine-Grained Pipeline

Foi desenvolvido por Maruyama e Hoshino (2000) um compilador para mapeamento de *loops*, descritos em código C, em FPGAs. O compilador gera soluções de *hardware* para computar as operações dos *loops* em *pipeline*. O compilador decompõe as operações aritméticas de forma a realizar operações de no máximo 8 *bits* cascadeadas, utilizando técnicas de execução especulativa entre as iterações do *loop*. Isto é feito a fim de aumentar o desempenho do *pipeline*. Caso seja necessário acesso ao mesmo banco de memória, o acesso é serializado, comprometendo o *pipeline*. As operações executadas especulativamente são canceladas na ocorrência de dependências de dados entre iterações a fim de garantir a consistência dos dados para as

operações futuras. Neste caso, uma vez finalizadas as operações da iteração anterior, a execução especulativa volta a ser iniciada. Este compilador também provê suporte, embora limitado, a chamadas de funções recursivas, transformando-as em chamadas iterativas. Outra característica do compilador é permitir o uso de anotações no código para especificação de operações concorrentes.

3.3.5 ROCCC

O ROCCC (Riverside Optimizing Compiler for Configurable Computing) (VILLAR-REAL *et al.*, 2010) é um compilador para geração de *hardware* a partir de C voltado para otimização de *loops* e reuso de dados. Além disso, a ferramenta busca otimizar as porções do código usadas mais frequentemente. Em certas operações, o compilador realiza um escalonamento de acesso à memória externa, movendo os dados para blocos de memória internos ao FPGA. O controle é feito por meio de *buffers* e módulos pré-definidos. As otimizações são aplicadas à representação intermediária, sendo que o ROCCC utiliza uma representação intermediária denominada CIRRF. Esta representação é uma variação das plataformas SUIF2 and MachSUIF (CARDOSO; DINIZ, 2009). As transformações executadas incluem *loop-unrolling* e *strip-mining*, bem como reuso de dados.

O código de entrada possui certas restrições, impedindo o uso de funcionalidade como chamadas recursivas e ponteiros. Também possui restrições quanto ao aninhamento dos *loops* e ao número de operações com *arrays* dentro dos *loops*. Tais limitações reduzem o número de programas compatíveis com esta ferramenta.

3.3.6 SPC

Outro compilador que busca realizar otimizações por meio da análise de *loops* é o SPC (*SUIF Pipeline Compiler*) (WEINHAUDT; LUK, 2001; WEINHARDT, 1996). Este compilador gera código VHDL a partir da análise de *loops* e dependência de dados, a fim de explorar o paralelismo em nível de instrução com a execução de operações diretamente em *hardware*. Esta análise é feita em todos os *loops* do programa, com foco nos mais internos (CARDOSO; DINIZ, 2009). Além disto, o SPC busca reduzir o número de operações de acesso à memória através do reuso de dados, utilizando componentes específicos e estratégias de alocação de memória que buscam aumentar a disponibilidade dos dados.

3.3.7 DEFACTO

DEFACTO (*Design Environment For Adaptive Computing TechnOlogy*) (BONDALAPATI *et al.*, 1999; DINIZ *et al.*, 2005) é uma ferramenta de HLS, que utiliza do *framework* SUIF e opera dividindo o programa de entrada (escrito em C ou Fortran) em duas partes. Uma parte do programa é compilada para execução em um GPP e a outra é traduzida em VHDL para execução

em um ou mais FPGAs. Este particionamento entre *hardware* e *software* é realizado utilizando técnicas de compilação paralelizantes em conjunto com ferramentas de HLS comerciais. O gerenciamento de dados, que inclui escrita e leitura de memória, é gerado pelo compilador, o qual sincroniza a execução da computação em cada FPGA utilizando memórias paralelamente distribuídas. Os dados são mapeados em estruturas internas, sendo criadas interfaces de memória personalizadas, que permitem a aplicação de análises de reuso de dados e de transformações em nível de *loops*, como *loop unrolling* e *loop tiling* (SO; HALL; DINIZ, 2002).

3.3.8 MATCH

O *MATlab Compiler for Heterogeneous computing systems* (MATCH) (BANERJEE *et al.*, 2000) é um compilador que visa a geração de código VHDL comportamental a partir de programas escritos em MATLAB². Este compilador utiliza um ambiente distribuído, composto por processadores embarcados, DSPs, e FPGAs, e um repositório de componentes (IPs) para geração da arquitetura.

O compilador realiza várias transformações de *loops*, incluindo a execução em *pipelining* e escalonamento das operações, bem como organiza as operações com vetores e matrizes em função do aninhamento dos *loops*, de forma a permitir a execução concorrente em múltiplos FPGAs. Além disso, também são realizadas análises para inferência do número de *bits* a ser usados para representação, bem como a conversão de tipos de ponto flutuante em ponto fixo e dimensionamento dos *arrays*. Em relação à memória, o acesso é escalonado em função do número de portas, das memórias disponíveis, e do atraso em cada componente (CARDOSO; DINIZ, 2009).

3.3.9 DeepC

O compilador DeepC (BABB; AGARWAL, 2001) gera código Verilog comportamental a partir de C ou Fortran. Na arquitetura gerada pelo DeepC, os programas C ou Fortran são mapeados em blocos lógicos conectados a memórias locais e a um canal para comunicação entre os blocos, controlados por uma unidade de controle baseada em máquinas de estado finito. O particionamento dos dados entre as memórias é realizado considerando a porção de dados que será tratada por cada bloco. Também é realizada a análise do número de *bits* necessários, bem como a conversão de operações de ponto flutuante em operações com inteiros.

3.3.10 Galadriel e NENYA

Os compiladores Galadriel e NENYA (CARDOSO, 2000; CARDOSO; NETO, 1999) operam em conjunto para geração de *hardware* a partir de *bytecodes* Java. Nesta abordagem o compilador Galadriel atua como *front-end* de modo a expor o paralelismo presente em cada

² <http://www.mathworks.com/products/matlab/>

método Java, enquanto que o NENYA foca o escalonamento das operações presentes em cada método. A estrutura gerada destina-se a um único FPGA conectado a uma ou mais memórias. Dada a capacidade limitada do FPGA alvo, o NENYA realiza o particionamento temporal de projetos que excedam essa capacidade. Neste caso, cada partição possui uma interface de memória correspondente, além das unidades de controle (VHDL comportamental) e *data-path* (VHDL estrutural). A geração do *hardware* e o escalonamento das operações também incluem análise do número de *bits* de cada operando do circuito, bem como uma estimativa de área e atraso de cada componente, a partir de uma biblioteca de *hardware* e modelos de cada dispositivo.

3.3.11 Spark

O Spark (GUPTA *et al.*, 2003) trata-se de um projeto para síntese de alto nível a partir de linguagem C, obedecendo restrições como a impossibilidade do uso de ponteiros, recursões, *loops* irregulares, arranjos multidimensionais, dentre outras. O Spark gera código VHDL correspondente ao código C de entrada buscando escalonar as operações para execução paralela. Para isto, realiza de uma série de transformações, como eliminação de código morto, *loop unrolling*, reordenamento e duplicação de instruções, reuso de dados, etc. O *hardware* gerado possui, ainda, um bloco de controle em forma de máquina de estados finitos, usado para sincronizar as demais partes do circuito.

3.3.12 Catapult C

Desenvolvido pela Mentor Graphics³, o Catapult C (BOLLAERT, 2008) gera código VHDL comportamental a partir de código C++. Para isto, o código deve ser escrito em estilo próprio, obedecendo certas regras que permitem a síntese. O Catapult C permite apenas o uso de ponteiros estáticos, não suportando uso de alocação dinâmica (*malloc*, *free*, por exemplo). Algumas transformações, como *loop unrolling*, *loop merging* e *loop pipelining* são implementadas pelo Catapult C, de forma a atingir os requisitos de frequência de operação definidos pelo usuário.

3.3.13 GCC2Verilog

GCC2Verilog (HUONG; KIM, 2011) é um compilador desenvolvido com base no GCC que gera código Verilog a partir de código ANSI C. Este compilador transforma cada uma das funções do código C em um módulo Verilog sintetizável. No fluxo de compilação isso ocorre pouco antes do RTL ser convertido em código *assembly*, sendo a geração do *hardware* realizada a partir da IR gerada pelo GCC. Como todas as análises e otimizações são realizadas pelo GCC, toda a sintaxe ANSI C é suportada, inclusive o uso de ponteiros.

³ <http://www.mentor.com/>

O código gerado tem como plataforma alvo o processador PICO (KATHAIL *et al.*, 2002). Para compilação é necessário passar como parâmetro o nome da função e do arquivo fonte que a contém. O GCC2Verilog utiliza os arquivos de ligação que o GCC gera para endereçar os módulos de *hardware*. Deste modo, as funções convertidas rodam como blocos IPs de *hardware* configuráveis, enquanto que o restante do código executa no processador PICO. Os módulos de *hardware* gerados são endereçados como sendo as funções de *software* chamadas no C, utilizando o mesmo espaço de endereçamento do processador, sendo possível realizar chamadas às demais funções tanto do módulo quanto do processador. O GCC2Verilog não gera módulos para alocação ou desalocação de memória. Estas tarefas são feitas em *software*, considerando que o sistema operacional realiza melhor a tarefa de gerenciamento de memória. O compilador não implementa unidades funcionais para operações de ponto flutuante, sendo estas realizadas por *software* com biblioteca própria do GCC.

3.3.14 C2H

A Altera desenvolveu o compilador C2H para acelerar código em C para o processador Nios II (ALTERA, 2011b). Assim, o foco principal desta ferramenta é melhorar, através da aceleração por *hardware* o desempenho dos programas executados neste processador. O C2H trata o código em nível funções C isoladas, especificadas pelo usuário. A ferramenta opera com base no grafo de dependência de dados (DDG) da função, determinando a melhor sequência para a execução das operações e realiza algumas otimizações não descritas na documentação. Como ocorre em outras ferramentas, o C2H não possui suporte a todas as características da linguagem C, dentre as quais destacam-se o uso de ponto flutuante, funções recursivas, instruções de desvio incondicional (*goto*), etc (MENOTTI, 2010).

3.3.15 HDL Coder

HDL Coder⁴ é um *software* comercial, desenvolvido pela MathWorks Inc.® que permite a geração de código VHDL ou Verilog a partir de outras ferramentas da empresa, como Simulink e Matlab. O HDL Coder permite a aplicação de técnicas como *loop unrolling* e outras otimizações envolvendo *loops*. Além disso, implementa o compartilhamento de recursos buscando uma boa realação entre área ocupada e desempenho. Por se tratar de uma ferramenta comercial, informações detalhadas sobre os algoritmos utilizados não estão disponíveis.

3.3.16 Vivado HLS

Vivado HLS⁵ é outra ferramenta comercial voltada à síntese de alto nível. Vivado HLS utiliza o compilador LLVM, seguindo o fluxo mostrado na Figura 19.

⁴ <http://www.mathworks.com/products/hdl-coder/>

⁵ <http://www.xilinx.com/products/design-tools/vivado/index.htm>

Figura 19 – Fluxo de compilação do Vivado HLS

Fonte: Adaptada de Windh *et al.* (2015).

Desenvolvida pela Xilinx, a ferramenta possui suporte a C e C++, além de SystemC (GRÖTKER *et al.*, 2002). Vivado HLS permite a criação e integração de componentes por meio da inserção de diretivas *#pragma* diretamente em código. Essas diretivas são passadas para o compilador de forma a serem processadas as restrições e transformações de código especificadas. Também é possível usar arquivos de configuração e *scripts* gerados no próprio ambiente de desenvolvimento.

A ferramenta possui suporte a ponto flutuante, uso automático de memória *on-chip* e integração com IPs proprietários. Vivado HLS é voltada para um vasto conjunto de FPGAs fabricados pela Xilinx, possuindo suporte para integração com outros IPs fornecidos pela empresa.

3.3.17 CHiMPS

CHiMPS (PUTNAM *et al.*, 2008) é um compilador HLS para geração de aceleradores FPGA a partir de código ANSI-C. Este compilador executa sobre uma plataforma híbrida, formada por CPU e FPGA, utilizando o microprocessador *soft core* MicroBlaze (XILINX, 2014).

O código de entrada é convertido em uma linguagem variante do *assembly*, a qual produz um grafo de fluxo de dados espacial em que cada instrução gera um bloco VHDL. Neste modelo, o grafo é usado para determinação do paralelismo, onde cada um dos nós é instanciado como um nó físico em um acelerador FPGA.

Esta abordagem baseada no fluxo de dados pode levar a implementações onde os recursos do FPGA não são suficientes. Por isso, CHiMPS inclui a opção de executar partes do programa no MicroBlaze.

A arquitetura faz uso de um modelo baseado em múltiplas memórias *cache* distribuídas, onde cada endereço de memória não pode ser tratado por mais de uma *cache*. O programador pode fazer uso de *#pragmas* para especificar parâmetros como configurações de *cache*, uso de multiplicadores em *hardware* tamanho da palavra binária, etc.

3.3.18 REFLECT

O REFLECT (*Rendering FPGAs to Multi-Core Embedded Computing*) (CARDOSO; HÜBNER, 2011) é um projeto cuja proposta consiste no desenvolvimento de uma nova abordagem para compilação e síntese para FPGAs. A proposta é baseada em especificação orientada a aspectos, o que permite inserir no sistema conhecimentos específicos do domínio, aproveitando as vantagens do paradigma das linguagens de programação imperativas (TUCKER A. B.; NOONAM, 2007). Desta forma, o projeto pretende aumentar a acessibilidade a tecnologias reconfiguráveis, aumentando a produtividade e facilitando a portabilidade de programas para estas arquiteturas.

No sistema REFLECT o código-fonte de entrada é inserido em conjunto com uma especificação complementar na forma de aspectos. Em seguida, técnicas de otimização e transformação são aplicadas, gerando um código transformado para a linguagem LARA (*Language for Reconfigurable Architectures*). As transformações são realizadas a partir de modelos de *hardware* e boas práticas de mapeamento, obtidos de seus respectivos repositórios, considerando os aspectos especificados. Por fim, são aplicadas técnicas de mapeamento e gerada a descrição do *hardware* para síntese na arquitetura alvo (PETROV; DINIZ, 2010). A Figura 20 apresenta uma visão geral do REFLECT.

O Quadro 3 mostra as linguagens suportadas pelas ferramentas de HLS descritas, enquanto que o Quadro 4 traz uma comparação entre as mesmas, considerando suas características gerais em relação às otimizações realizadas.

3.4 Considerações finais

Neste capítulo foram apresentadas os principais esforços de pesquisa e desenvolvimento relacionados à proposta deste trabalho. Dado o número de trabalhos disponíveis, e as características presentes em cada um deles, percebe-se a complexidade do processo de geração de *hardware* a partir de uma especificação de alto nível, quer em uma linguagem amplamente conhecida, quer em linguagens específicas.

Os esforços no desenvolvimento de ferramentas e DSLs buscam, de modo geral, prover alternativas ao uso de HDLs para a especificação de *hardware*. Os trabalhos apresentados nas seções 3.2 e 3.3 contam com técnicas variadas, que incluem, dentre outras facilidades, o uso de

Figura 20 – Visão geral do REFLECT

Fonte: Adaptada de Petrov e Diniz (2010).

Quadro 3 – Linguagens de entrada e saída suportadas pelos compiladores das ferramentas de HLS descritas

Ferramenta	Linguagem de entrada	Linguagem de saída
LegUp	C	Verilog
Impulse-C	C	VHDL
Cadence C-to-Silicon	System C	Verilog
C to Fine-Grained Pipeline	C	VHDL
ROCCC	C	VHDL
SPC	C, Fortran	VHDL
DEFACTO	C	VHDL
MATCH	Matlab	VHDL
DeepC	C, Fortran	Verilog
Galadriel e Nenya	Java bytecodes	VHDL
Spark	C	VHDL
Catapult C	C++	VHDL
GCC2Verilog	ANSI C	Verilog
C2H	C	VHDL ou Verilog
HDL Coder	Matlab	VHDL ou Verilog
Vivado HLS	C/C++, SystemC	Não gera RTL
REFLECT	C, Matlab	VHDL

Fonte: Elaborada pelo autor.

scripts, reuso de componentes, parametrização automática e exploração do espaço de projeto. Isso demonstra a complexidade presente no processo de desenvolvimento de sistemas embarcados de alto desempenho.

Quadro 4 – Comparação entre as ferramentas HLS apresentadas

Ferramenta	Análises e otimizações envolvendo <i>loops</i>	Análises e otimizações relacionadas ao uso e acesso à memória	Outras características
LegUp	Loop unrolling, loop pipelining	–	Suporte a vários níveis de compilação
Impulse-C	Loop unrolling, loop pipelining	–	Uso de FIFOs para comunicação
Cadence C-to-Silicon			Visualização gráfica do CDFG
C to Fine-Grained Pipeline	Loop pipelining	–	Decomposição de operações
ROCCC	Loop unrolling, strip-mining	Reuso de dados, escalonamento de acesso à memória externa	
SPC	Loop pipelining	Reuso de dados e estratégias de alocação	
DEFACTO	Loop pipelining, unrolling, tiling	Reuso e distribuição de dados	Interfaces de memória personalizadas
MATCH	Loop pipelining	Escalonamento de acesso otimizado	Arquitetura composta por GPPs, DSPs e FPGAs
DeepC	–	Particionamento em múltiplas memórias	
Galadriel e Nenyá	–	–	Particionamento temporal e otimização de área ocupada
Spark	Loop unrolling e reordenação de instruções	Reuso de dados	
Catapult C	Loop unrolling, merging e pipelining	–	
GCC2Verilog	–	–	Particionamento <i>hardware</i> e <i>software</i>
C2H	?	?	Análise de dependência e reordenação de instruções
HDL Coder	Loop unrolling	?	Otimização da relação entre área ocupada e desempenho
Vivado HLS	Loop unrolling, pipelining	?	Integração com IPs proprietários
REFLECT	?	?	Transformações realizadas em função das definições do usuário

Fonte: Elaborada pelo autor.

As ferramentas de HLS geralmente delimitam o escopo das linguagens de alto nível de forma a poderem ser utilizadas. Uma característica a ser destacada é que a maior partes destas ferramentas tem em comum o fato de utilizarem, a fim de otimizar o desempenho do programa a ser sintetizado, análises e transformações envolvendo operações em *loops* ou acesso aos dados.

No caso das DSLs, é comum que requeiram que o programador tenha conhecimentos de *hardware* e/ou sobre a plataforma onde o sistema irá executar. Muitas delas preocupam-se em fornecer um ambiente onde é possível realizar algum tipo de partição entre *hardware* e *software*.

Outra característica comum às abordagens envolvendo DSLs é o uso de arquiteturas personalizadas, ou modelos próprios para lidarem com sincronismo e acesso aos dados. Isto implica, em muitos casos, em situações onde é necessário especificar manualmente detalhes referentes às arquiteturas e na forma como os componentes gerados interagem.

A LINGUAGEM LALP

4.1 Considerações iniciais

Neste capítulo são apresentados detalhes sobre a linguagem LALP. Inicialmente, é mostrada uma visão geral da linguagem, seguida por exemplos de código e técnicas utilizadas no processo de compilação em LALP. São mostrados, ainda, os principais problemas presentes na linguagem, além de outros projetos envolvendo LALP.

É importante ressaltar que as características descritas neste capítulo correspondem ao LALP em sua versão original¹, acrescida de modificações realizadas em outros projetos e fora do escopo desta tese. O conteúdo principal, referente ao funcionamento do *framework* LALP, é adaptado de [Menotti et al. \(2012\)](#) e [Menotti \(2010\)](#), onde pode ser encontrada uma descrição mais detalhada de LALP, bem como a especificação formal da linguagem.

4.2 Visão geral

Desde o primeiro compilador Fortran tem-se buscado otimizar códigos dentro de *loops*, uma vez que estas porções de código são consideradas críticas em muitas aplicações, pois executam com mais frequência do que códigos fora de *loops*. Neste sentido, *loop pipelining*² é uma das técnicas mais utilizadas ([COOPER; TORCZON, 2011](#), p. 666).

LALP (*Language for Aggressive Loop Pipelining*) é uma DSL de propósito especial destinada ao desenvolvimento de aceleradores de *hardware* em FPGAs. Foi desenvolvida no Laboratório de Computação Reconfigurável (LCR) da Universidade de São Paulo (USP) com o objetivo de minimizar as limitações impostas pelas linguagens de programação convencio-

¹ A nova versão de LALP, denominada LALP+, usa esta versão como base para implementação das alterações, incorporando um conjunto de novas características e funcionalidades. LALP+ é mostrada detalhadamente no [Capítulo 5](#) e corresponde à versão atual de LALP.

² Também conhecida como *software pipelining*

nais (C, por exemplo), no que diz respeito à implementação de *loop pipelining* para geração de arquiteturas especializadas. LALP foca-se, principalmente, no mapeamento de estruturas de repetição (*loops*) para *hardware* reconfigurável (MENOTTI *et al.*, 2012; MENOTTI, 2010), a fim de explorar o paralelismo presente nestes trechos do programa..

LALP fornece um alto nível de abstração, o que auxilia no desenvolvimento de aceleradores de *hardware*. Neste sentido, LALP facilita que desenvolvedores implementem aplicações de *hardware* altamente paralelas, permitindo o controle no escalonamento de operações de diferentes iterações do corpo de um *loop*, de forma que estas sejam processadas ao mesmo tempo.

Considerando que o trecho mostrado no [Código-fonte 1](#), a [Figura 21](#) ilustra o funcionamento da técnica de *loop pipelining*. Se o referido código for executado de forma sequencial, cada iteração só é iniciada após a execução de todas as operações da iteração anterior (Figura 21a). Quando utilizada a técnica de *loop pipelining*, iterações subsequentes podem ser iniciadas antes do fim da iteração atual (Figura 21b), desde que sejam respeitadas as dependências entre as iterações. O número de ciclos (ou estágios) necessários antes de uma nova iteração ser iniciada é chamado de *intervalo de iniciação*.

Código-fonte 1: Trecho de código C usado para exemplificar a técnica de *loop pipelining*

```

1 for( i = 0; i < M; i++)
2 {
3   X[i] = i + 1;          \\ I1
4   A[i] = X[i] * 2;       \\ I2
5   B[i] = A[i] + X[i];    \\ I3
6   C[i] = A[i] + B[i];    \\ I4
7 }
```

A execução em *pipelining* ocorre em três etapas: o prólogo, que corresponde ao início da execução enquanto nem todas as iterações são iniciadas; o *kernel*, iniciado após o prólogo, quando são executadas operações pertencentes a todas as iterações; e o epílogo, que ocorre no final do processo, iniciado após o término da primeira iteração.

Usando LALP é possível ter um maior controle da execução do programa de forma menos complexa do que com linguagens de descrição de *hardware*, tais como VHDL e Verilog. LALP suporta operações com alto grau de paralelismo, permitindo o controle de sincronismo das operações em nível de ciclos de *clock*. Este nível de controle pode resultar em circuitos altamente eficientes, uma vez que o programador pode explorar o paralelismo controlando a execução de cada estágio do *pipeline*.

LALP utiliza a abordagem de *Aggressive Loop Pipelining* (ALP), cujo objetivo é minimizar a latência no fluxo de dados presentes nas iterações de um loop, preenchendo os espaços no fluxo de dados com o uso de registradores de deslocamento, registros e contadores, de forma a sin-

Figura 21 – Exemplo da técnica de *loop pipelining*

Fonte: Elaborada pelo autor.

cronizar as operações do *pipeline* (MENOTTI; MARQUES; CARDOSO, 2007). Esta abordagem é baseada nas ideias propostas por Cardoso (2005), mas com foco na paralelização de instruções ao invés de utilizar um esquema orientado a dados (MENOTTI *et al.*, 2012; MENOTTI, 2010).

4.2.1 Fluxo de compilação

A Figura 22 ilustra as principais etapas de desenvolvimento e compilação em LALP. Usando como base um programa em linguagem de alto nível, o programador pode especificar o código comportamental em LALP, que será usado como entrada do compilador para geração de código VHDL. Opcionalmente, este código pode ser escrito de forma estrutural, utilizando uma variante do LALP denominada LALP-S.

O *front-end* é responsável pela validação do código de entrada e geração do grafo de fluxo de controle/dados (CDFG – *Control/Data Flow Graph*). Foi desenvolvido usando JavaCC³, um gerador de *parser* em Java, e conta com versões distintas para análise de código LALP ou LALP-S.

O *middle-end* processa o CDFG gerado, executando um conjunto de algoritmos para escalonamento, balanceamento e sincronização do *pipeline*. Este passo é realizado com base em diretivas de baixo nível que podem ser inseridas no código pelo programador. Tais diretivas são usadas como referência para controle do fluxo de dados e dos sinais de sincronização. Uma versão modificada do esquema ASAP (*As Soon As Possible*) / ALAP (*As Late As Possible*), onde

³ Java Compiler Compiler[™] (JavaCC[™]) - The Java Parser Generator (<https://java.net/projects/javacc>)

Figura 22 – Fluxo de compilação LALP

Fonte: Menotti (2010).

a diferença entre os dois tipos escalonamento para cada nó do grafo, chamada mobilidade, é utilizada para ajuste do intervalo de iniciação do *pipeline*.

O compilador usa um algoritmo conservador para evitar a perda de dados devido a diferenças no escalonamento dos vários caminhos de dados, baseando-se na utilização de contadores. Isto resulta em um número invariável de ciclos por iteração e, como consequência, na falta de suporte ao uso de *loops* irregulares. Contudo, isso permite que o compilador identifique se as operações ligadas aos contadores estão balanceadas. Com isto, o número de ciclos de relógio para cada iteração é determinado pelo cálculo de arestas recorrentes e usado na análise de dependência no *loop*.

Por fim, o *back-end* utiliza o novo CDFG processado para gerar código VHDL correspondente ao código LALP de entrada. Em complemento ao VHDL, são gerados arquivos de visualização (*Graphviz*) contendo representações gráficas de *software* e *hardware* do acelerador.

LALP conta com uma biblioteca de componentes de *hardware* personalizados para a geração de VHDL, a qual é derivada da biblioteca *Nenya VHDL Library* (CARDOSO; NETO, 2003; CARDOSO, 2000). Esta biblioteca não explora explicitamente recursos específicos dos FPGAs, tratando-se de código VHDL genérico. Assim, o acelerador gerado pode ser sintetizado em diferentes FPGAs, tanto fornecidos pela Xilinx®, quanto pela Altera®. O compilador LALP conta com a capacidade das ferramentas de síntese RTL no que diz respeito ao uso de forma eficiente dos recursos de um FPGA específico.

A principal abordagem usada por LALP é minimizar a latência em iterações do *loop* usando registradores cascadeados para equilibrar e sincronizar as operações no *datapath*. A fim de implementar esta abordagem, é necessário controlar o número de ciclos de relógio atrasados ou executados para habilitar cada operação. A biblioteca de componentes VHDL incluem contadores e registradores de deslocamento que permitem tal nível de controle.

4.2.2 O código LALP

LALP possui uma sintaxe semelhante à do C, com a adição de diretivas de baixo nível próprias da linguagem. Com essas diretivas o programador pode interferir no fluxo de execução, sendo possível definir o número de ciclos de *clock* a serem aguardados ou executados para a realização de uma determinada operação, ajustando o *pipeline* de acordo com a aplicação.

Todas as instruções de código LALP são, por padrão, consideradas concorrentes, sendo que a sequência de execução de cada operação é determinada pela presença de dependências, tratadas automaticamente pelo compilador, ou de acordo com os atrasos definidos em código com o uso do qualificador @. Este símbolo indica quantos ciclos de *clock* são necessários em uma instrução e é usado para permitir o escalonamento manual das operações. Esta funcionalidade proporciona ao programador um elevado controle do fluxo de execução do programa e o escalonamento das operações de diferentes iterações do *loop* para uma execução em *pipeline*, uma vez que o programador pode definir, manualmente, o número de ciclos de relógio que uma instrução (ou operação) precisa para ser executada ou atrasada. Como atrasos diferentes podem ser especificados para várias operações, a quantidade de ciclos de relógio total do *pipeline* muda de acordo com a especificação do usuário.

A estrutura geral do código-fonte LALP é mostrado no [Código-fonte 2](#). Tipicamente, um código LALP tem três seções principais. A primeira seção é usada para definir constantes e tipos de dados personalizados (linhas 1 a 4). A segunda seção (linhas 6 a 14) é onde o código principal é escrito. Esta é a única seção obrigatória, uma vez que contém a lógica principal do acelerador, incluindo as portas para comunicação externa, bem como as diretivas para ajuste do *pipeline*. O arquivo também pode conter uma terceira seção opcional, destinada à especificação de *testbench* para verificação da assertividade do hardware gerado (linhas 17 a 20).

Código-fonte 2: Estrutura de um código LALP

```
1 // Declaração de constantes e definição de tipos
2 const <identifier> = <value>;
3 typedef fixed(<bits>, <signal>) <identifier>;
4
5 // Lógica principal do acelerador
6 <identifier>(<ports>)
7 {
8     //Variáveis
```

```

9    {
10        ...
11    }
12
13    ...
14 }
15
16 // Especificação de testbench
17 assert
18 {
19     ...
20 }

```

Um exemplo de multiplicação de dois vetores em LALP é mostrado no [Código-fonte 3](#), sendo o [Código-fonte 4](#) seu equivalente em C. Neste exemplo, a constante N é definida como 3 (linha 1), seguida da definição dos tipos *int* e *bit*, sendo o primeiro definido como uma palavra sinalizada de 32 *bits* (linha 3) e o segundo como 1 *bit* (linha 4). Na linha 6 estão definidas a entrada (*init*) e a saídas (*done* e *result*) do acelerador *arr_mult*. Nas linhas 8 a 11 estão declaradas as variáveis. A variável *i* é usada como contador (linha 14) para controlar as iterações do *loop*, iniciando a contagem de acordo com o sinal *init* (linha 15). Na sequência, linhas 17 e 18, os endereços dos vetores *x* e *y* são associados ao valor do contador *i*, de forma que os valores dos operandos da multiplicação (linha 20) mudam a cada incremento do contador. O resultado da operação é atribuído à porta de saída *result* do acelerador (linha 21), podendo ser lido por algum bloco externo que seja conectado. Por fim, o sinal de saída *done* é conectado ao sinal do *i.done* (linha 23), que indica o final da contagem. Neste caso, porém, o uso do qualificador @, seguido do valor 3, indica que o sinal *i.done* será atrasado em três ciclos de relógio. O exemplo mostrado no [Código-fonte 3](#) não contém nenhuma especificação de *testbench*, uma vez que esta seção é opcional.

Código-fonte 3: Exemplo de código LALP para multiplicação de dois vetores

```

1 const N = 3;
2
3 typedef fixed(32, 1) int;
4 typedef fixed(1, 0) bit;
5
6 arr_mult(out bit done, in bit init, out int result)
7 {
8     {
9         int x[N]={4,5,6};
10        int y[N]={1,2,3};
11        int res,i;
12    }
13

```

```
14 counter (i=0; i<N; i++);
15 i.clk_en = init;
16
17 x.address = i;
18 y.address = i;
19
20 res = x * y;
21 result = res;
22
23 done = i.done@3;
24 }
```

Código-fonte 4: Código C correspondente ao [Código-fonte 3](#)

```
1 #define N 3
2
3 void arr_mult(int *result, int *x, int *y)
4 {
5     int res,i;
6
7     for (i=0; i < N; i++)
8     {
9         res = x[i] * y[i];
10        result[i] = res;
11    }
12 }
```

A [Figura 23](#) apresenta duas formas de visualização correspondentes ao exemplo mostrado. Na [Figura 23a](#) é mostrado o CDFG final, enquanto que a [Figura 23b](#) traz as ligações entre os componentes de *hardware* que formam a arquitetura gerada.

Na [Figura 23a](#) é possível perceber a inserção de *delays* nos sinais *done* e *wrEn*. No primeiro caso, este atraso ocorre conforme especificado em código, enquanto que, no segundo, o atraso é inserido automaticamente pelo compilador com o propósito de habilitar a escrita na variável *res* de forma sincronizada. Estes *delays* são implementados em *hardware* com registradores de deslocamento conectados nas respectivas portas de origem e destino dos sinais, conforme visto na [Figura 23b](#).

Conforme visto no exemplo, o qualificador @ pode ser usado para a inserção de atrasos e consequente modificação na forma como o *pipeline* é executado. Para fins de ilustração, trechos do código original mostrado no [Código-fonte 3](#) foram modificados a fim de serem implementados alguns efeitos, como: retardar o resultado da multiplicação ([Código-fonte 5](#)), o endereçamento de memória e acesso aos vetores ([Código-fonte 6](#)) e aumentar o número de ciclos necessários para cada etapa do contador ([Código-fonte 7](#)). Os efeitos correspondentes a cada um destes casos são

Figura 23 – Visualizações de *software* e *hardware* correspondentes ao exemplo de código LALP mostrado no Código-fonte 3

Fonte: Elaborada pelo autor.

apresentados na Figura 24, com alguns sinais omitidos para fins de simplificação da ilustração.

Em nenhum desses casos há mudança nos valores computados, apenas nos ciclos onde ocorrem as operações. Contudo, o uso inadequado do qualificador @ pode levar à perda de sincronismo do *pipeline* e, portanto, a resultados incorretos.

Código-fonte 5: Trecho do Código-fonte 3 modificado para inserção de atraso na multiplicação

```
19 ...
20  c = (x *@2 y);
21 ...
```

Código-fonte 6: Trecho do Código-fonte 3 modificado para inserção de atraso na leitura do valor do contador

```
16 ...
17  x.address = i@2;
18  y.address = i@2;
19 ...
```

Código-fonte 7: Trecho do Código-fonte 3 modificado para inserção de atraso no incremento do valor do contador

```

13 ...
14 counter (i=0; i<N; i++@2);
15 ...

```

Figura 24 – Impacto na execução resultante das diferentes formas de aplicação de *delays* em LALP: (a) Efeito do código original; (b) Efeito do Código-fonte 5; (c) Efeito do Código-fonte 6; (d) Efeito do Código-fonte 7

Fonte: Elaborada pelo autor.

4.2.3 Componentes de hardware

Pode-se agrupar os componentes de *hardware* dos circuitos gerados pelo LALP em dois grupos. No primeiro estão os componentes referentes às operações lógicas e aritméticas, como somadores, multiplicadores e registradores. No segundo grupo estão os componentes de controle, como os contadores e os multiplexadores. As diretivas próprias do LALP (como o @ e o *when*) têm o papel de permitir que o programador configure os componentes de controle.

O principal componente de controle gerado pelo LALP é o contador, uma vez que dele dependem o sincronismo dos operadores lógicos e aritméticos e o acesso à memória. Portanto, o contador exercer três funcionalidades distintas: controlar as iterações do *loop*, fornecer sinais para sincronismo das operações e prover o *offset* para cálculo dos endereços. Ao utilizar o LALP, o programador tem o controle explícito do sincronismo das operações, manipulando o contador por meio da utilização do símbolo @.

O componente contador (*counter*) possui um conjunto de sinais que são usados para sincronizar as operações, dentre os quais os mais utilizados são o *step* e o *done*. O sinal *step* produz um pulso de duração de um ciclo de *clock* sempre que o valor do contador é modificado. Quando o contador usa mais de um ciclo por iteração esse sinal tem nível alto no primeiro ciclo e baixo nos demais. Assim, o sinal *step* pode ser usado para habilitar operações que ocorram

uma única vez a cada iteração. O sinal *done* sinaliza o término da contagem, podendo ser usado, por exemplo, para habilitar operações subsequentes ao *loop* ou outro *loop* que não seja aninhado. Os principais atributos e sinais do contador são mostrados no [Quadro 5](#)).

Quadro 5 – Atributos e sinais do componente contador

Nome	Função
<i>bits</i>	Largura das portas <i>input</i> , <i>out put</i> e <i>termination</i>
<i>steps</i>	Número de ciclos por iteração (padrão: 1)
<i>increment</i>	Valor do incremento/decremento por iteração (padrão: 1)
<i>down</i>	Direção da contagem (padrão: 0, incrementar)
<i>condition</i>	Condição de parada (padrão: 1, <=)
<i>input</i>	Usada para carga do valor inicial
<i>termination</i>	Valor para término da contagem
<i>clk</i>	<i>Clock</i> do sistema
<i>clk_en</i>	Habilita o funcionamento
<i>reset</i>	Reinicia a contagem
<i>load</i>	Realiza a carga do valor inicial
<i>step</i>	Usado para sincronização das operações
<i>done</i>	Indica o término da contagem
<i>output</i>	Valor atual da contagem

Fonte: [Menotti \(2010\)](#).

A [Figura 25](#) mostra a arquitetura geral de um acelerador LALP. Nesta figura é possível perceber que o contador funciona como um controlador estático do restante do circuito. É ele que fornece a base (*offset*) para os endereços de memória que serão acessados durante a execução do *loop*, além dos sinais de sincronismo para habilitar cada operação no momento correto. Pode haver mais de um contador no circuito gerado, sendo necessário que o programador configure cada um deles manualmente.

Figura 25 – Arquitetura geral de um acelerador LALP

Fonte: Elaborada pelo autor.

4.3 Principais problemas

Embora o esforço de programação requerido em LALP seja mais baixo do que o de implementar projetos usando uma HDL, LALP ainda exige mais esforços do que o uso de uma ferramenta de HLS, especialmente em casos onde o compilador LALP não lida totalmente com a complexidade de escalonamento necessária. Em tais casos, cabe ao programador a tarefa de escalonar o *loop* apropriadamente.

Outro ponto a ser considerado é que, para utilizar corretamente o LALP, aproveitando suas vantagens, o programador deve conhecer o momento em que cada operação é iniciada no *pipeline* e qual sua duração. Isto dificulta o uso da linguagem, pois, apesar de possuir uma sintaxe semelhante a uma linguagem de alto nível, seu uso impõe um nível de abstração bem próximo ao *hardware*.

A linguagem LALP possui, ainda, algumas limitações como a falta de instruções para estrutura de decisão (*if then else*) e de suporte a operações de ponto flutuante, bem como a impossibilidade de criação de procedimentos e funções. Além disto, a implementação de *loops* aninhados e/ou irregulares não é suportada nativamente, sendo sua implementação dependente de artifícios de programação que exigem conhecimentos específicos, tanto do código quanto do funcionamento dos componentes da biblioteca VHDL utilizada.

4.3.1 Endereçamento de memória

Em adição aos problemas citados, um ponto crítico do uso de LALP está relacionado à manipulação de dados em *arrays*. Em LALP não é possível, por exemplo, acessar um mesmo vetor com diferentes índices ($x = a[i+1]$; $y = a[i+2]$), bem como não são aceitos *arrays* multidimensionais. O principal fator que leva a este tipo de limitação está na forma como é implementado o endereçamento de memória e o acesso aos dados.

Uma grande dificuldade para usar LALP está relacionada com a utilização de @, especialmente ao lidar com várias operações de memória. O compilador do LALP gera uma memória diferente para cada *array* declarado. O controle de acesso é feito associando o valor do contador ao endereço da memória que se deseja acessar. Assim, o endereço selecionado para leitura/escrita muda a cada incremento ou decremento do contador. Para acessar mais de uma posição sem que haja mudança no valor do contador é preciso especificar manualmente a posição ou o deslocamento desejado, bem como sincronizar este acesso com o uso de @. Uma consequência desta forma de acesso é que a movimentação de dados nos *arrays* depende diretamente da contagem do contador.

Para exemplificar este problema, o [Código-fonte 8](#) traz uma implementação do filtro de *Sobel* em LALP. Neste exemplo, o *array* de entrada precisa ser acessado oito vezes em uma mesma iteração, conforme o padrão mostrado na [Figura 26](#).

Figura 26 – Posições acessadas em cada iteração para computação do filtro de Sobel

Fonte: Elaborada pelo autor.

Nesta possível implementação do filtro de Sobel em LALP o *loop* interno para computar a máscara está desenrolado. Neste caso, a posição da matriz *A* é definida pela variável *addr* (linha 28 linha). No entanto, uma vez que o endereço da matriz é vinculado ao contador, é necessário realizar 8 leituras da memória antes de alterar o valor do contador. Isso é feito atrasando o incremento por 8 ciclos de relógio (linha 18). Assim, um endereço é calculado para cada ciclo (linhas 20 a 27). Para a leitura de cada valor da memória em sincronismo com o endereçamento é preciso escalonar as atribuições aos registradores correspondentes, utilizando palavra reservada *when* como condicional (linhas 30 a 37). o valor de saída é calculado nas linhas 39 a 45, o qual será truncado se for maior do que 255 (linha 47). Esta condição é implementada usando um operador ternário, uma vez que, como previamente mencionado, não há (*if them else*) em LALP.

Código-fonte 8: Exemplo de código LALP para implementação do filtro de Sobel

```

1 const COLS = 10;
2 const SIZE = 100;
3 typedef fixed(8,0) int;
4 typedef fixed(1,0) bit;
5
6 sobel(in bit init, out bit done, out int result)
7 {
8   {
9     fixed(16, 0) H, 0, V, Hpos, Vpos, res;
10    int i, addr;
11    int i00, i01, i02, i10;
12    int i12, i20, i21, i22;
13    int A[SIZE] = {...}; //inicialização omitida

```

```

14     int output[SIZE];
15     bit done_ok = 1;
16 }
17 i.clk_en = init;
18 counter (i=0; i < 78; i++@8);
19
20 addr = i;
21 addr = (i) + 1 when i.step@1;
22 addr = (i) + 2 when i.step@2;
23 addr = (i) + COLS when i.step@3;
24 addr = (i + COLS) + 2 when i.step@4;
25 addr = (i + COLS) + COLS when i.step@5;
26 addr = ((i + COLS) + COLS) + 1 when i.step@6;
27 addr = ((i + COLS) + COLS) + 2 when i.step@7;
28 A.address = addr;
29
30 i00 = A when i.step@2;
31 i01 = A when i.step@3;
32 i02 = A when i.step@4;
33 i10 = A when i.step@5;
34 i12 = A when i.step@6;
35 i20 = A when i.step@7;
36 i21 = A when i.step@8;
37 i22 = A when i.step@9;
38
39 H=((-i00) + (-i01 -i01)) + (((-i02) + i20) + ((i21 + i21) + i22));
40 V=((-i00) + i02) + (((-i10 -i10) + (i12 + i12)) + ((-i20) + i22));
41
42 Hpos = H < 0 ? -H : H;
43 Vpos = V < 0 ? -V : V;
44
45 0 = Hpos + Vpos;
46
47 res = 0 > 255 ? 255: 0;
48
49 output.address = i;
50 output.data_in = res;
51
52 result = output;
53 done = done_ok;
54 }

```

Neste exemplo percebe-se que o programador precisa entender a integração do contador com o restante do circuito, em especial com a memória, para definição correta dos valores usados com @ de forma a manter o correto sincronismo do circuito. Uma vez definido este sincronismo, o compilador escala as demais operações de forma automática.

4.4 Abordagens envolvendo LALP

LALP foi objeto de estudo de, pelo menos, dois projetos de mestrado, desenvolvidos na Universidade Federal de São Carlos (UFSCAR). Ambos os trabalhos atuaram na tentativa de contornar alguns dos problemas do LALP por meio da geração automática de código LALP a partir de C.

O primeiro projeto, desenvolvido por [Rettore \(2012\)](#), utilizou LALP como estudo de caso no desenvolvimento de um *framework* para auxílio na geração de *hardware* reconfigurável para *loop pipelining*. Este trabalho teve como base o compilador Cetus ([DAVE et al., 2009](#)) e tratou da análise de dependências de dados em *loops* e da definição do intervalo de iniciação para geração do *pipeline*. No tocante ao LALP, teve como foco o problema da definição dos valores de @, fornecendo como saída um código LALP preliminar, contendo uma sugestão de escalonamento a ser revisada pelo programador. Apesar de promissora, esta abordagem contou com uma série de simplificações, não tratando de casos onde ocorrem múltiplos acessos em uma mesma iteração.

A segunda abordagem envolvendo LALP consiste no compilador LALPC ([PORTO et al., 2015](#); [PORTO, 2015](#)), o qual utiliza o compilador *source-to-source* ROSE ([QUINLAN; LIAO, 2011](#)) para geração de código VHDL a partir do C. LALPC permite o uso de *pragmas* para aplicações de transformações de *loop unrolling* no código C por meio do ROSE e reimplementa a mesma estrutura do *back-end* do LALP de forma a gerar *hardware* baseado no código transformado. Estes *pragmas* guiam o compilador na aplicação de transformações. A questão dos múltiplos acessos é tratada em LALPC com a geração de memórias *multport*, onde é possível fazer todos os acessos em um único ciclo.

4.5 Considerações finais

Este capítulo apresentou uma visão geral das ideias incorporadas pela linguagem LALP. Foram apresentados trechos de código de forma a exemplificar o uso geral da linguagem, bem como os principais problemas enfrentados pelo desenvolvedor.

Apesar da proposta de LALP ser uma linguagem de alto nível, os problemas apresentados dificultam seu uso. Alguns destes são decorrentes de funcionalidades não implementadas enquanto outros, mais críticos, são inerentes ao funcionamento e às características da própria linguagem, como é o caso da dificuldade no uso correto do qualificador @. Corrigir tais problemas implica em realizar alterações estruturais no compilador e na forma como os componentes interagem.

LALP+: UMA EXTENSÃO DE LALP

5.1 Considerações iniciais

Em face às limitações e problemas do LALP, como mostrado no [Capítulo 4](#), este trabalho focou-se em sua expansão. Assim, foram modificadas certas características do *framework* LALP, tanto da linguagem quanto do compilador, a fim de prover uma versão com maior nível de abstração e que mantém em sua base a abordagem de *Aggressive Loop Pipelining*. O resultado das alterações implementadas deu origem à expansão LALP+, a qual mantém o foco em *loops*.

Neste capítulo são apresentados detalhes do desenvolvimento de LALP+. Inicialmente é apresentada uma visão geral da abordagem adotada durante o processo de desenvolvimento. Em seguida, são apresentadas as principais modificações e novas características incorporadas ao ambiente LALP para criação de LALP+.

5.2 Visão geral da abordagem

Uma das preocupações iniciais referentes ao desenvolvimento de LALP+ foi tentar manter um certo nível de similaridade e compatibilidade com a versão original de LALP. Desta forma, projetos paralelos, como os mencionados no [Capítulo 4](#), poderiam ser integrados sem grandes alterações nos *frameworks* envolvidos.

As modificações seguiram no sentido de aumentar o nível de abstração da linguagem. Assim, funções antes dependentes de especificação por meio de diretivas da linguagem (@, when, etc) passaram a estar implícitas no compilador. Isto foi feito com o intuito de fornecer uma ferramenta para desenvolvimento de aceleradores em FPGAs com utilização menos complexa do que LALP.

Em relação ao sincronismo e acesso à memória, tratou-se da criação de componentes que visam incorporar estas funcionalidades de modo a separá-las do contador. Com isto, o acesso

e endereçamento em código passam a exigir um menor nível de conhecimento por parte do programador, sendo utilizado um código mais limpo e com a exposição de menos detalhes relacionados ao *hardware*.

Como consequência, além de novas características, foi realizada também uma mudança estrutural da arquitetura para alteração do subsistema de memória atualmente usado em LALP. Isto resultou em modificar o compilador a nível de *front-end*, *middle-end* e *back-end*.

O *front-end* foi alterado de forma a suportar a nova sintaxe, mantendo, porém, compatibilidade com a anterior. O *middle-end* passou a considerar no cálculo dos *delays* a barreira de sincronização imposta pelo nova arquitetura, enquanto foram desenvolvidos algoritmos para tratamento de alguns casos específicos. Em termos de *back-end*, as alterações incluem a incorporação de novos componentes à biblioteca VHDL usada, bem como a geração de componentes em tempo de compilação, dando uma maior flexibilidade para a geração de arquiteturas. O contador usado em LALP foi estendido para uma versão que suporta *loops* aninhados, passando a fornecer valores para unidades de endereçamento específicas dos subsistemas de memória desenvolvidos.

Outro fator considerado durante o desenvolvimento foi a integração dos aceleradores criados a componentes externos. Foram criadas interfaces de comunicação que permitem a geração de IPs que requerem uma menor intervenção do usuário para utilização em um sistemas mais complexos.

5.3 Novas características e funcionalidades

5.3.1 Suporte a operações de ponto fixo/flutuante

5.3.1.1 Componentes para Floating/Fixed Point

Novos componentes para operações de ponto flutuante foram integrados à biblioteca VHDL usada pelo LALP. Tais operadores foram desenvolvidos segundo o padrão IEEE 754 (IEEE, 1985).

De acordo com este padrão, um número de ponto flutuante é composto por três valores distintos, os quais são: *sinal*, *expoente* e *mantissa*. O valor do número é calculado segundo a Equação 5.1. A precisão dos números Reais que podem ser representados varia de acordo com o número de *bits* utilizados para representação.

$$sinal \times 2^{expoente} \times mantissa \quad (5.1)$$

O padrão estabelece formatos com precisão de 16, 32, 64 e 128 *bits*. Contudo, este projeto focou-se na representação utilizando 32 *bits*, que corresponde a números de ponto flutuante de precisão simples. Nesta representação, os valores de *sinal*, *expoente* e *mantissa* são armazenados conforme Figura 27a, onde:

- O *bit* [31] representa o *senal*. Se este *bit* for 1 implica que o número é negativo; se for zero o número é positivo.
- O *expoente* é calculado a partir dos *bits* [30 : 23], convertendo-se o valor de binário para decimal e subtraindo 127.
- A *mantissa* é definida como $1, \textit{frac}$, sendo que *frac* corresponde à parte fracionária e somente esta é representada pelos *bits* [22 : 0], sendo implícita a parte inteira. Desta forma, o valor da *mantissa* pode variar entre 1 e 2.
- Se todos os *bits* do *expoente* forem 0, a *mantissa* passa a ser calculada de forma não normalizada, com o *bit* [22] representando a parte inteira (0 ou 1), e os demais representando a parte fracionária.
- O número zero possui notação positiva e negativa. O mesmo ocorre para o valor infinito.
- O valor infinito é representado com todos os *bits* do *expoente* iguais a 1 e todos da *mantissa* iguais a 0.
- Os casos em que todos os *bits* do *expoente* sejam 1 e pelo menos um *bit* da *mantissa* seja diferente de 0 são tratados como valores *NaN* (*Not a Number*), geralmente usados para representar valores inválidos, como, por exemplo, uma divisão por zero.

Tomando como exemplo o número Real 10.5, sua representação em 32 *bits* segundo o padrão IEEE 754 seria:

Hexadecimal: 0x41280000 = 01000001001010000000000000000000b

Sinal: *Bit* 31 = 0 $\rightarrow$ +

Expoente: *Bits* [30 : 23] = 10000010 $\rightarrow 2^{130-127} = 2^3$

Mantissa: *Bit* [22 : 0] = 010100000000000000000000 $\rightarrow 1 + 0,3125 = 1,3125$

Real: $+1 \times 2^3 \times 1,3125 = 10,5$

Outra forma de representar números reais é utilizando uma representação de ponto fixo. Neste caso, o número é dividido em duas partes, uma inteira e outra fracionária (Figura 27b), sendo usada a mesma regra de conversão de número binário fracionário, onde os *bits* são deslocados o número casas correspondente ao total de *bits* usados para a parte fracionária, gerando *bits* com índices negativos. Por exemplo, assumindo uma representação de 32 *bits*, com 24 *bits* para a parte inteira e 8 *bits* para a parte fracionária, o número 8,25 seria representado por:

Hexadecimal: 0x00000840 = 00000000000000000000000010000100 0000b

Parte inteira: Bits $[23 : 0] = 0000000000000000000000001000 \rightarrow 2^3 = 8$

Parte fracionária: Bits $[-1 : -8] = 01000000 \rightarrow 2^{-2} = 0,25$

Valor Real: 8,25

Figura 27 – Representação de um número Real em 32 bits. (a) Ponto flutuante, segundo padrão IEEE 754; (b) Ponto fixo, com 8 bits para a parte fracionária

Fonte: Elaborada pelo autor.

Esta forma é mais simples que a representação em ponto flutuante, e, portanto, mais rápida para ser calculada em *hardware*. Por este motivo é uma opção viável para várias aplicações em sistemas embarcados. Uma desvantagem, porém, é a precisão dos números representados, que varia de acordo com número de *bits* reservado para representação da parte fracionária do número.

Para implementação destes componentes foi utilizada uma biblioteca com funções VHDL para ponto fixo/flutuante, disponível gratuitamente em <http://www.eda.org/fphdl/>. Esta biblioteca, conhecida como *IEEE Proposed Library*, foi desenvolvida por David W. Bishop, com apoio de *EDA Industry Working Groups*¹ e Accellera Systems Initiative².

Uma vantagem do uso da biblioteca *IEEE Proposed* é permitir que o programador defina o tamanho de cada campo, mantendo compatibilidade com outros componentes usados pelo LALP. Outra vantagem é que a mesma pode ser utilizada tanto em FPGAs Xilinx, quanto nos fornecidos pela Altera.

Assim como os operadores inteiros previamente existente na biblioteca LALP, os novos operadores suportam operações com dois valores de entrada e um de saída. Em termos de sincronismo, cada operação dura um ciclo de relógio por padrão, sendo implementadas com lógica combinacional e podendo ser deslocadas com o uso do qualificador @, tal como ocorre na versão para número inteiros.

¹ <http://eda.org>

² <http://www.accellera.org/home>

5.3.1.2 Módulos parametrizáveis para ponto flutuante

Além dos componentes criados, foram desenvolvidos dois módulos LALP para cálculo de operações de ponto flutuante no padrão IEEE 754 de 32 *bits*. Diferente dos criados com a biblioteca *IEEE Proposed*, estes componentes implementam operadores em *pipelining* para adição e multiplicação, descritos diretamente em LALP.

O uso de LALP para descrição destes operadores permite ao compilador ajustar a geração de *hardware* com base em parâmetros especificados no código, o que pode levar a circuitos otimizados. Apesar dessa característica também poder ser alcançada com descrições em VHDL, com LALP é possível explorar o grafo *dataflow* que representa os operadores, de forma a especializar a geração de *hardware* em vários níveis e com uma abstração maior do que a necessária quando usado VHDL.

Dado que LALP é fortemente focada na criação de *pipelines*, os módulos foram implementadas em vários estágios de *pipeline*, conforme a operação realizada. Desta forma é possível usar código LALP para explorar o número de ciclos de relógio gastos em cada estágio, a fim proporcionar um incremento de frequência no circuito final gerado. A seguir são mostrados os estágios definidos para cada módulo criado.

Estágios da multiplicação:

- 1 → Leitura dos valores de entrada para sinal, expoente e mantissa;
- 2 → Cálculo do sinal e expoente do resultado;
- 3 → Inserção do *bit* implícito na mantissa;
- 4 → Ajuste da mantissa;
- 5 → Multiplicação das mantissas;
- 6 → Ajuste de *overflow* e composição do resultado final.

Estágios da adição:

- 1 → Leitura dos valores de entrada para sinal, expoente e mantissa;
- 2 → Determinação do maior expoente;
- 3 → Subtração dos expoentes;
- 4 → Deslocamento e ajuste da mantissa;
- 5 → Soma das mantissas;
- 6 → Ajuste de *overflow* e composição do resultado final.

O código VHDL gerado para os módulos está incluso na biblioteca de componentes. Contudo, a opção padrão do compilador é utilizar os componentes citados na Seção 5.3.1.1. O programador deverá informar explicitamente ao compilador, por meio de parâmetro próprio, caso deseje utilizar estes módulos. Para parametrização, é necessário alterar e recompilar o código LALP do operador desejado (p/ multiplicação ou adição).

5.3.1.3 Outras alterações para ponto fixo/flutuante

Alterações no compilador também foram implementadas a fim de suportar operações de ponto fixo/flutuante, tanto no *front-end* quanto no *back-end*. Em termos de *front-end*, foi incluído suporte à inicialização de memórias e registradores com valores em ponto flutuante, convertidos de acordo com os tipos e tamanhos definidos. Outros tipos de declarações e inicializações, como o uso de valores em notação hexadecimal, não sofreram alterações.

O *back-end* foi alterado de forma a identificar automaticamente os operadores a serem usados, de acordo com o tipo de dado tratado. Durante a etapa de geração do circuito em VHDL, o compilador verifica o tipo das variáveis da operação, associando o operador correspondente. Se houver diferença entre os tipos das variáveis de entrada ou saída, todas serão consideradas como sendo de ponto flutuante ou fixo caso pelo menos uma delas possua estes tipos. Tal comportamento pode implicar em erro no resultado final, devendo o programador estar atento a este detalhe.

5.3.2 Sintaxe

5.3.2.1 Criação e definição de tipos

A definição de tipos de tamanho variável possibilita a aplicação de técnicas para redução automática do número de *bits*, o que implica na redução da largura dos barramentos de dados em *hardware*. Com isto, é possível atingir valores mais altos de frequência e menor consumo de recursos do FPGA. Esta otimização é realizada por outras ferramentas de HLS, porém ainda não está implementada em LALP, ficando a cargo do programador definir o tamanho apropriado para cada variável.

A versão original do LALP não conta com tipos pré-definidos, mas suporta a criação de tipos por meio de diretivas genéricas (**typedef** e **fixed**(*<bitwidth>*,*<signal>*)). Um tipo pode ser definido com a especificação do tamanho da palavra binária referente ao tipo, acrescida de parâmetro que identifica se o mesmo corresponde a um tipo com ou sem sinal. Por exemplo, para definir um número inteiro de 32 *bits* com sinal, a sintaxe usada deveria ser:

typedef fixed(32, 1) int;

Dada a vasta ocorrência de declarações de variáveis, é comum que tipos sejam definidos no início de um código LALP. A falta de suporte a tipos pré-definidos implica na necessidade da

definição destes tipos ao início da programação de cada novo acelerador, sob pena do código tornar-se poluído caso sejam usadas apenas diretivas genéricas.

LALP+ inclui o suporte a tipos pré-definidos, mostrados no [Quadro 6](#), a fim de prover uma sintaxe mais limpa e aproximada do C. Tal como em LALP, LALP+ permite criar variações de tipos com tamanhos personalizados a partir destes tipos básicos. O [Código-fonte 9](#) mostra a definição de tipos em LALP, enquanto que o [Código-fonte 10](#) apresenta a versão em LALP+. A sintaxe antiga foi mantida por questões de compatibilidade.

Quadro 6 – Tipos básicos pré-definidos em LALP+

Tipo	Descrição	Tamanho da palavra
<i>signed</i>	Inteiro sinalizado	32 <i>bits</i>
<i>unsigned</i>	Inteiro não sinalizado	32 <i>bits</i>
<i>sfixed</i>	Ponto fixo sinalizado	32 <i>bits</i> (16 <i>bits</i> frac)
<i>ufixed</i>	Ponto fixo sinalizado	32 <i>bits</i> (16 <i>bits</i> frac)
<i>float</i>	Ponto flutuante	32 <i>bits</i> (Single Precision)
<i>bit</i>	Bit / Sinal	1 <i>bit</i>
<i>signed(N)</i>	<i>signed</i> personalizado	<i>N bits</i>
<i>unsigned(N)</i>	<i>unsigned</i> personalizado	<i>N bits</i>
<i>sfixed(M,N)</i>	<i>sfixed</i> personalizado	(<i>M + N</i>) <i>bits</i> , <i>N bits</i> frac
<i>ufixed(M,N)</i>	<i>ufixed</i> personalizado	(<i>M + N</i>) <i>bits</i> , <i>N bits</i> frac
<i>float(M,N)</i>	<i>float</i> personalizado	(<i>M + N</i>) <i>bits</i> , (<i>M – 1</i>) <i>bits</i> p/ a mantissa e <i>N bits</i> p/ o expoente

Fonte: Elaborada pelo autor.

Código-fonte 9: Definição de tipos com sintaxe LALP

```

1 //inteiro de 32 bits
2 typedef fixed(32,1) s_int32;
3
4 //inteiro de 8 bits não sinalizado
5 typedef fixed(8,0) u_int8;
6
7 foo(out s_int32 result, out fixed(1, 0) done)
8 {
9     ...
10 }
```

Código-fonte 10: Definição de tipos com sintaxe LALP+

```

1 //inteiro de 32 bits
2 typedef signed s_int32;
3
4 //inteiro de 8 bits não sinalizado
5 typedef unsigned(8) u_int8;
```

```
6
7 foo(out s_int32 result, out bit done)
8 {
9     ...
10 }
```

5.3.2.2 Expressões numéricas com constantes

O compilador LALP+ implementa um tipo de simplificação de código, denominada desdobramento de constantes (*Constant folding*), que consiste em avaliar expressões compostas por constantes e substituí-las pelo seu resultado calculado em tempo de compilação. O compilador LALP original não faz distinção entre estas expressões, gerando um circuito para realizar o cálculo, o que resulta em um uso desnecessário de recursos do FPGA.

Ainda no contexto de LALP, além de prover uma otimização em relação ao uso de recursos, essa funcionalidade também permite a introdução de *delays* de forma relativa, uma vez que novas constantes podem ser definidas a partir de outras previamente criadas. Por exemplo, supondo uma operação que deve ocorrer no ciclo N e outra no ciclo $N + 2$, não seria possível realizar essa definição de forma genérica anteriormente, sendo necessário modificar cada valor manualmente caso o valor de N mudasse.

Como consequência da introdução desta funcionalidade, foi implementada uma abordagem para definição automática dos *delays* com base no uso de Algoritmos Genéticos (GAs). Nesta abordagem, o programador passaria apenas a definir quais operações teriam a inserção de *delays*, mas não o valor (em ciclos de *clock*) inserido. Foi implementado um GA simples, mono-objetivo, onde cada indivíduo na população contém uma lista de valores inteiros correspondentes aos *delays* desejados. O tamanho dessa lista varia, portanto, de acordo com a quantidade de @ presentes no código.

No decorrer dos experimentos observou-se duas situações comuns aos algoritmos: 1) aquelas onde várias soluções são possíveis, ou seja, os *delays* contribuem para o desempenho mas não comprometem o resultado; e 2) aquelas em que poucas soluções (ou apenas uma) garantem que o resultado computado será correto. Assim, o intervalo de valores definido nas configurações do algoritmo, pode gerar um espaço de busca demasiadamente grande para o número de soluções disponíveis, resultando na falta de soluções válidas, sobretudo no caso 2, e/ou tornando-se inviável pelo tempo de processamento requerido para avaliação de cada solução. Tal abordagem mostrou-se ineficiente, não funcionando bem para os diversos casos.

5.3.2.3 Suporte a loops aninhados e a endereçamento direto

Conforme mostrado no [Capítulo 4](#), LALP utiliza uma forma de endereçamento indireto. Utiliza-se uma variável para o endereçamento, normalmente vinculada ao contador, a qual é

manipulada a cada ciclo de *clock* de forma a indexar diferentes posições do vetor. O *back-end* utiliza um registro conectado à porta de endereço da memória para implementar esse endereçamento.

Outra característica de LALP é que o componente contador implementado na biblioteca VHDL somente realiza contagens simples, com incremento configurável, não sendo destinado à implementação de *loops* aninhados. Este tipo de *loop* requer a utilização de vários contadores independentes. Estes, por sua vez, devem ser sincronizados entre si com o uso dos sinais de controle apropriados (*load*, *reset*, etc.). Isto é feito manualmente pelo programador, que precisa considerar não somente os ciclos de contagem, mas, também, os que envolvem operações de carga, reinício de contagem e sinalização. Essa complexidade, somada ao endereçamento indireto, dificulta a programação de *loops* aninhados.

LALP+ permite uma indexação direta e implementa um contador multinível como formas de minimizar este problema. Com isto, a sintaxe para acesso a um vetor em LALP+ está no formato *Vet*[*lin*][*col*], como ocorre em códigos C. A biblioteca VHDL foi expandida para este fim, sendo incorporados componentes que calculam o endereço de acordo com os índices e com o tamanho do vetor. Uma exceção é o contador multinível, o qual não está incluso na biblioteca, sendo gerado dinamicamente a partir do código LALP recebido pelo compilador. Com isto, a sintaxe LALP+ fornece suporte a *arrays* multidimensionais.

Para fins de diferenciação entre os dois tipos de contadores, optou-se pela utilização de duas sintaxes diferentes para especificação cada um deles, sendo a diferença principal o uso de colchetes ([e]) para delimitar os *loops* aninhados. Os trechos mostrados no [Código-fonte 11](#) e no [Código-fonte 12](#) trazem a sintaxe de cada uma das versões.

Código-fonte 11: Trecho de código mostrando o uso do contador simples e endereçamento indireto em LALP

```
1 ...
2 counter(i=0; i<100; i++@11);
3
4 addr = i;
5 Vet.address = addr
6
7 x = Vet.data_out;
8 ...
```

Código-fonte 12: Trecho de código mostrando o uso do contador multinível e endereçamento direto em LALP+

```
1 ...
2 counter[(i=0; i<10; i++)
3         (j=0; j<10; j++)];
```

```
4  
5 x = Vet[i][j];  
6 ...
```

5.3.2.4 Simplificação da sintaxe

De um modo geral, as alterações realizadas na sintaxe original de LALP buscam simplificar a linguagem e facilitar seu uso por programadores habituados ao uso da linguagem C. Outro efeito desejado é tornar o código mais limpo, não expondo detalhes tão próximos ao *hardware*, uma vez que LALP se propõe a fornecer uma alternativa de programação de mais alto nível que as HDLs.

Como forma de apresentar as diferenças entre LALP e LALP+, o [Apêndice A](#) traz implementações de uma multiplicação de matrizes, escritas em LALP e LALP+, bem como o código equivalente em C.

5.3.3 Gerenciamento de memória e sincronismo

A principal questão ao gerenciar acessos em LALP é sincronizar corretamente as várias operações de leitura e escrita com todos os outros estágios do *pipeline*, já que em LALP o endereçamento de memória é feito usando o valor do contador para definir o índice do vetor. Esta dependência do contador pode levar a casos em que é necessário parar a contagem por mais de um ciclo de relógio. Além disso, as funcionalidades de endereçamento, sincronismo e controle das iterações são todas vinculadas ao contador, e, como consequência, alterações em alguma destas impactam diretamente as demais. Assim, o desenvolvedor precisa levar em conta o número de ciclos para cada operação realizada ao longo do caminho de dados, conhecendo detalhes de *hardware* que, idealmente, não deveriam estar explícitos. Tal nível de detalhamento aumenta a dificuldade em usar LALP para implementar aceleradores muito complexos.

LALP+ trabalha com a hipótese de que o sincronismo entre acesso aos dados e computação pode ser ajustado utilizando barreiras de sincronização. A abordagem usada em LALP+ busca separar essas funcionalidades implementando componentes próprios e tratando a dependência entre elas automaticamente.

Nesta abordagem, os algoritmos de balanceamento e escalonamento executados no compilador LALP passam a considerar apenas as operações no corpo do *loop*, sob a premissa de que os dados estarão disponíveis simultaneamente. Componentes específicos para cálculo dos endereços e gerenciamento de acesso são responsáveis por assegurar esta premissa, enquanto que o compilador realiza o ajuste de *delay* e associação dos sinais de controle.

Para efeito de tratamento automático do sincronismo, LALP+ considera três condições de acesso aos vetores: a) acesso a um único endereço de memória por iteração; b) múltiplos acessos por iteração; e c) múltiplos acessos com particionamento de dados. Cada uma destas condições é

tratada de forma diferente pelo compilador LALP+. Outras abordagens com múltiplas memórias, como, por exemplo, CUDA, usam arquiteturas fixas, sendo necessário adequar o código da aplicação a elas. Isto é diferente do que ocorre em LALP+, onde é possível gerar arquiteturas personalizadas de acordo com a aplicação.

5.3.3.1 Indexação para acesso único

Para os casos onde há apenas um acesso por iteração, o compilador LALP+ realiza tratamento semelhante ao que ocorre em LALP. A principal diferença é a utilização de componentes para cálculo dos índices, mencionados na [subseção 5.3.2.3](#), em substituição ao uso de um registrador conectado à porta *address* da memória.

A utilização destes componentes de índice causa um atraso entre o contador e a memória ([Figura 28](#)), o qual é propagado para o restante do circuito. Este *delay* depende do índice calculado e é considerado pelo compilador para ajuste do sincronismo

Figura 28 – Arquitetura com acesso único indexado

Fonte: Elaborada pelo autor.

5.3.3.2 Múltiplos acessos sequenciais

A segunda condição trata dos casos onde um único *array* é acessado várias vezes na mesma iteração (e.g. *Sobel*). Nestes casos, o incremento do contador é interrompido de acordo com o número de endereços de memória manipulados por iteração. Para n acessos, o valor do contador permanece inalterado por n ciclos de relógio, sendo os endereços acessados sequencialmente, uma posição a cada ciclo.

A abordagem usada em LALP+ para tratamento desta condição é a utilização de uma máquina de estados finitos (*Finite State Machine* – FSM). Esta FSM controla a sincronização, assegurando que os dados serão inseridos no caminho de dados nos tempos corretos. A cada n ciclos de relógio a FSM é reiniciada, habilitando os componentes do circuito que farão a computação, para os quais os dados chegam como se fossem provenientes de uma BRAM com múltiplas portas acessadas paralelamente.

A Figura 29 apresenta a arquitetura de um acelerador LALP+ com o uso de FSM para controle de múltiplos acessos sequenciais a uma mesma BRAM. Nesta figura é utilizado um conjunto de índices que são usados para gerar os endereços. A FSM seleciona, por meio de um multiplexador, qual endereço será acessado a cada ciclo, obedecendo a ordem em que as instruções de acessos aparecem no código-fonte. Os dados lidos a cada ciclo são armazenados nos respectivos registradores, selecionados por meio de um demultiplexador. A Figura 30 mostra o fluxo de dados resultante desta abordagem para um exemplo, sem perda de generalidade, onde ocorrem quatro acessos.

Figura 29 – Arquitetura com máquina de estado para controle de múltiplos acessos sequenciais a uma mesma BRAM

Fonte: Elaborada pelo autor.

Figura 30 – Fluxo de dados gerado na abordagem utilizando FSM em um caso com 4 acessos

Fonte: Elaborada pelo autor.

Acessos à memórias diferentes são sincronizados por FSMs diferentes. Cada FSM é gerada dinamicamente em tempo de compilação, sendo produzido um arquivo VHDL com a descrição RTL do componente que a implementa.

5.3.3.3 Múltiplos acessos com particionamento de dados

Na abordagem utilizando FSMs a latência do circuito depende diretamente do número de acessos, sendo necessário interromper o contador por alguns ciclos, o que implica em um maior intervalo de iniciação do *pipeline*. A terceira abordagem trata-se de uma alternativa que permite múltiplos acessos para um único ciclo de relógio, diminuindo a latência.

Esta abordagem realiza o mapeamento de uma matriz declarada no código em várias BRAMs. Em termos de *hardware*, a arquitetura gerada utiliza uma unidade de gerenciamento de memória (*Memory Management Unit – MMU*), mostrada na Figura 31, para controlar o acesso aos dados. LALP+ inclui um parâmetro de compilação usado para selecionar qual opção (FSM ou MMU) será utilizada em caso de múltiplos acessos.

Figura 31 – Arquitetura para controle de múltiplos acessos com particionamento de dados

Fonte: Elaborada pelo autor.

Para funcionamento desta arquitetura, as matrizes são mapeadas em 2^N BRAMs, sendo N um valor definido pelo usuário em tempo de compilação. Esse mapeamento não afeta o código LALP+. As matrizes podem ser tratadas pelo programador como se fossem mapeadas em uma mesma memória contígua. Assim, os índices acessados em código são convertidos em endereços reais dentro do intervalo de cada BRAM, conforme o mapeamento realizado.

A partição de dados é feita de tal forma que todos os elementos de uma mesma coluna

de uma matriz são armazenados em uma mesma BRAM. Entretanto, uma única BRAM pode armazenar dados de mais de uma coluna. Exemplos de mapeamento de uma matriz 4x4 são mostrados na [Figura 32](#).

Figura 32 – Exemplos de mapeamento de uma matriz 4x4 em múltiplas BRAMs: (a) Declaração da matriz em código; (b) Partição em 2 BRAMS; (c) Partição em 4 BRAMS

Fonte: Elaborada pelo autor.

Os dados referentes a endereços de memórias distintas são indexados através de uma função *hash*, usada para selecionar a memória correspondente a cada índice e gerar o endereço real. Como as memórias BRAMS são independentes, endereços pertencentes a BRAMS diferentes podem ser acessados no mesmo ciclo de relógio.

As BRAMs são controladas por componentes denominados *MemCtrl*, sendo um para cada BRAM. Um *MemCtrl* é responsável por interfacear o acesso de todos os endereços da BRAM que controla, acessando-a conforme a indexação recebida. Se mais de um endereço pertencer a uma única memória, o *MemCtrl* irá tratar um acesso por ciclo de relógio.

Dependendo dos endereços, os dados podem ser lidos fora de ordem. Deste modo, o componente *Data binder* realiza a associação e encaminhamento entre os dados produzidos e os respectivos *datapaths*. A versão atual desta MMU suporta apenas os casos em que a entrada e saída são mapeadas em memórias separadas.

Em termos de sincronização, cada passo do contador é retardado de acordo com a memória que possui o maior número de acessos por iteração do *loop*. Consequentemente, o número de ciclos pode variar não apenas de acordo com os endereços, mas, também, de acordo com o número de BRAMs. Os componentes *Data control* e *Step control* operam conjuntamente, sincronizando os acessos de acordo com os incrementos do contador. O primeiro é responsável por habilitar as leituras e escritas a cada ciclo, enquanto o segundo monitora o passo do contador. Assim, uma nova iteração é iniciada quando todos acessos forem concluídos.

A [Figura 33](#) mostra algumas situações onde é possível perceber as alterações no número de ciclos de relógio requeridos para cada passo do contador, de acordo com os endereços de leitura e o número de BRAMs usadas na partição de dados. As figuras [33a](#) e [33b](#) ilustram um exemplo de quatro leituras na matriz A, onde cada posição acessada corresponde a uma coluna da matriz. No caso da utilização de 2 BRAMs para partição de dados de dados, serão

necessários dois ciclos de relógio para completar todas as leituras (Figura 33a), enquanto que apenas um único ciclo é necessário ao quando utilizadas 4 BRAMs (Figura 33b). Uma forma diferente de realizar quatro leituras é apresentada nas figuras 33c e 33d. Neste caso, 3 endereços acessados estão na mesma coluna, sendo necessários 3 ciclos de relógio para completar a leitura, independentemente se são utilizadas 2 ou 4 BRAMs (Figuras 33c e 33d).

Em alguns casos, como no filtro de *Sobel*, os índices usados para acessar a matriz estão na forma $a_n = x + \delta_n$, onde a_n é a posição acessada, x é o valor atual do contador e δ_n é uma constante. Este formato garante ao índice a propriedade de translação, com o valor no contador correspondendo ao deslocamento do endereço ao longo das iterações. Quando todos os endereços possuem tal propriedade, o compilador calcula automaticamente a quantidade mínima de ciclos de relógio necessária para cada incremento do contador.

A propriedade da translação do endereço permite utilizar o Algoritmo 1 para calcular todos os endereços considerando a primeira iteração e os valores iniciais dos contadores. A ideia geral do algoritmo é calcular quantas vezes cada BRAM precisa ser acessada em cada iteração. Em seguida, o valor do passo (número de ciclos) do contador é configurado de acordo com a maior número de acessos. Por exemplo, se houverem quatro BRAMs, e cada uma for acessada apenas uma vez, o passo do contador é ajustado para 1 ciclo de relógio. Se alguma BRAM for acessada por duas vezes, o passo é configurado para 2 ciclos, e assim por diante.

Algoritmo 1: Cálculo do número de ciclos de relógios para o incremento do contador

input : O contador (*counter*) e o número de BRAMs (*num_mem*)

output : O número de ciclos para o incremento de *counter*

```

1  AI ← Lista de índices;
2  acessos [ ] ← new array(num_mem);
3  initial ← Valor inicial de counter;
4  foreach índice Idx ∈ AI do
5  begin
6      if idx depende de Ctr then
7          begin
8              pos ← CalculaEndereco(idx, initial);
9              mem ← pos mod num_mem;
10             acessos [mem] ← acessos [mem] + 1;
11         end
12     end
13  step ← Passo atual de counter;
14  new_step ← Max(acessos);
15  if step < new_step then
16  begin
17      counter.steps ← new_step;
18  end

```

O algoritmo tem como entrada o contador (*counter*) e o número de BRAMs. Inicial-

Figura 33 – Exemplo do impacto conjunto de particionamento e endereçamento no número de ciclos de relógio, para o caso onde são realizadas leituras em colunas diferentes: (a) 4 colunas em 2 BRAMs; (b) 4 colunas em 4 BRAMs; (c) 2 colunas em 2 BRAMs; (d) 2 colunas em 4 BRAMs

mente, são criadas uma lista (*AI*) contendo todos os índices da matriz acessados no código e um vetor (*acessos*[]) de tamanho igual ao número de BRAMs, bem como recuperado o valor inicial da contagem (linhas 1 a 3). Depois disso, tem-se o laço principal do algoritmo (linhas 4 a 12). Durante este laço é computado, para cada índice em *AI* que depende de *counter*, o endereço acessado na primeira iteração (*pos*). Cada posição *i* de *acessos*[] armazena o número de acessos por iteração da *BRAM_i*, o qual é incrementado caso o endereço retornado pela função *CalculaEndereco*(linha 8) corresponda à *BRAM_i* (linhas 9 e 10).

É possível utilizar um único contador para endereçar várias matrizes, o que pode resultar na situação onde matrizes distintas são acessadas um número diferente de vezes. A fim de evitar erros em tais situações, o algoritmo é conservador e o passo do contador é configurado de acordo com a matriz com o maior número de acessos. Consequentemente, um novo valor só é atribuído se for maior do que o valor corrente, como pode ser visto nas linhas 15 a 18.

O algoritmo não funciona para os casos em que a propriedade de translação não está garantida, isto é, quando o endereço não está no formato $x + \delta$. Assim, nestes casos, o programador deve configurar o passo do contador de acordo com o particionamento e os endereços acessados, sendo que o valor configurado pode variar de 1 ao total de endereços acessados.

A abordagem utilizando MMU permite a implementação de transformações relacionadas à *loops* que se beneficiem de memória distribuída. Apesar disso, o desenvolvedor precisa levar em conta o número de endereços e matrizes e também a forma como a partição de dados é realizada ao definir o total de BRAMs. Estas duas características terão impacto sobre o número de ciclos por iteração e sobre o *throughput* dos dados, afetando o impacto das transformações.

A Figura 34 mostra, de forma geral, as duas possíveis arquiteturas de um acelerador LALP+, uma utilizando FSM (Figura 34a) e outra utilizando MMU (Figura 34b). Em ambas as configurações, diferente do que ocorre em LALP, o contador assume um papel secundário no que diz respeito ao sincronismo e ao endereçamento, sendo responsável, principalmente, pelo controle das iterações.

5.3.4 Suporte a co-projetos de hardware e software

O modelo de aceleradores LALP incorpora as memórias BRAMs de forma que o acesso às mesmas por parte de componentes externos só é possível com a alteração direta do código VHDL produzido pelo compilador. Assim, a escrita e leitura dinâmica de dados precisa ser prevista pelo programador, o qual deve criar, manualmente, interfaces RTL próprias a este fim. Tal característica, porém, impõe mais uma dificuldade ao uso de LALP, agora relacionada à integração dos aceleradores com outros IPs.

Independente da arquitetura de memória utilizada, LALP+, inicialmente, também impõe esta restrição. Contudo, modificações foram incluídas no compilador LALP+ a fim de prover suporte à integração dos aceleradores com blocos externos. Especificamente, tais modificações

Figura 34 – Possíveis arquiteturas de um acelerador LALP+: (a) Acelerador LALP+ com FSM; (b) Acelerador LALP+ com MMU

Fonte: Elaborada pelo autor.

visam a utilização dos aceleradores em co-projetos de *hardware* e *software*.

As principais alterações incluídas para isto consistem na inclusão de parâmetros de compilação que permitem a criação de interfaces para acesso à memória no padrão de instruções customizadas estendidas do processador *soft-core* NiosII, fabricado pela Altera®. Os sinais que compõem este tipo de interface são mostrados no Quadro 7, enquanto a Figura 35 apresenta o diagrama de tempo destes sinais.

Quadro 7 – Sinais da interface de instruções customizadas estendidas do NiosII®

Tipo de IC	Aplicação	Portas
Estendida	Blocos customizáveis capazes de realizar múltiplas operações	$dataa[31:0]$ $datab[31:0]$ $result[31:0]$ clk clk_n $start$ $reset$ $done$ $n[7:0]$

Fonte: Adaptada de Altera (2011a).

O uso da interface para instrução customizada estendida permite a utilização da porta n para selecionar qual operação será realizada pelo bloco. No caso dos aceleradores LALP+, estas operações correspondem à inicialização e *reset* do bloco, acrescidas da escrita e leitura nas memórias internas.

Uma segunda opção é uso de FIFOs interligadas às portas de entrada e saída do acelerador, o qual neste caso, deve implementar apenas as instruções contidas no corpo do *loop* principal. O controle das iterações deve ser feito pelo processador externo, uma vez que os dados são

Figura 35 – Diagrama de sinais de instrução customizada do NiosII®, estendida para até 8 operações⁽¹⁾

Fonte: Elaborada pelo autor.

(1): Extraída de tela do Qsys®

processados à medida que chegam às FIFOs.

Os códigos mostrados em 13 e 14 trazem diferentes versões de um filtro *fir*. A principal diferença entre essas versões, em termos de código, ocorre na leitura dos dados. Na primeira, os dados são lidos da memória interna e é utilizado o contador para controle das iterações e endereçamento. Na segunda, os dados são lidos a partir das portas de entrada, as quais podem ser conectadas a FIFOs pelo compilador, de acordo com parâmetros de compilação. Note-se que neste último caso, o uso de FIFOs é opcional e esse mesmo código pode ser usado para gerar um acelerador sem FIFOs,

Código-fonte 13: Código FIR em LALP+ com memória interna

```

1 const SIZE = 8;
2 typedef unsigned int;
3
4 fir(out int result)
5 {
6     {
7         int a[SIZE] = {1, 2, 3, 4, 1, 2, 3, 4};
8         int b[SIZE] = {1, 2, 3, 4, 1, 2, 3, 4};
9         int c[SIZE] = {1, 2, 3, 4, 1, 2, 3, 4};
10        int d[SIZE] = {1, 2, 3, 4, 1, 2, 3, 4};
11        int e[SIZE] = {1, 2, 3, 4, 1, 2, 3, 4};
12        int a1, b1, c1, d1, e1, sum;
13        int a2, b2, c2, d2, e2;
14        unsigned(12) i;
15    }
16
17    counter[(i=0;i<SIZE;i++)];
18
19    a1 = a[i];
20    b1 = b[i];
21    c1 = c[i];

```

```
22  d1 = d[i];
23  e1 = e[i];
24
25  a2 = a1 * 3;
26  b2 = b1 * 5;
27  c2 = c1 * 7;
28  d2 = d1 * 9;
29  e2 = e1 * 11;
30
31  sum = ((a2+b2) + c2) + (d2+e2);
32
33  result = sum;
34 }
```

Código-fonte 14: Código FIR em LALP+ com fifos

```
1
2 typedef unsigned int;
3
4 fir(out int result, in int a, in int b, in int c, in int d, in int e)
5 {
6   {
7     int a1, b1, c1, d1, e1, sum;
8     int a2, b2, c2, d2, e2;
9     unsigned(12) i;
10  }
11
12  a1 = a[i];
13  b1 = b[i];
14  c1 = c[i];
15  d1 = d[i];
16  e1 = e[i];
17
18  a2 = a1 * 3;
19  b2 = b1 * 5;
20  c2 = c1 * 7;
21  d2 = d1 * 9;
22  e2 = e1 * 11;
23
24  sum = ((a2+b2) + c2) + (d2+e2);
25
26  result = sum;
27 }
```

Uma limitação para a integração dos aceleradores é esta deve ser feita de forma manual, apesar das interfaces VHDL serem geradas automaticamente. Não é fornecida nenhuma API

para integração a nível de *software*, devendo o programador lidar com esta questão.

No caso do uso de instruções customizadas, ainda não é possível especificar em código LALP+ os valores de n para seleccionar as memórias e operações, sendo estes atribuídos automaticamente de acordo com a ordem em que aparecem no código. Assim, o programador é responsável por conectar os aceleradores ao NiosII utilizando as ferramentas próprias da Altera. No caso do uso de FIFOs a integração pode ser feita com outros processadores, aumentando a flexibilidade para uso dos aceleradores.

5.4 Resumo das Contribuições

Considerando as funcionalidades e aspectos impactados por este trabalho, o restante desta seção apresenta uma descrição geral das suas principais contribuições. O [Quadro 8](#) lista um conjunto de características presentes em LALP e sumariza as diferenças entre as duas versões.

Sintaxe LALP

Tipos pré-definidos → A versão original do LALP suporta a criação de tipos por meio de diretivas genéricas. A nova versão inclui o suporte a tipos pré-definidos, os quais podem ser modificados para a definição de novos tipos personalizados.

Simplificação da sintaxe → Alterações relativas à sintaxe LALP original ocorreram de forma a facilitar o uso da linguagem por programadores habituados ao uso da linguagem C.

Expressões numéricas com constantes → Originalmente, LALP não permitia que novas constantes sejam definidas a partir de outras previamente criadas. Na versão atual isto é permitido, tendo sido incluído o cálculo de expressões numéricas em tempo de compilação.

Suporte a loops aninhados → LALP baseia-se no componente *contador* para controle dos *loops*. Apesar de configurável, este componente limita-se a uma contagem simples. Assim, a fim de implementar *loops* aninhados faz-se necessário o uso de configuração e sincronismo de vários contadores, o que aumenta, significativamente, a complexidade de programação. Como forma de minimizar este problema, a nova versão LALP implementa um contador multinível, gerado dinamicamente a partir do código LALP recebido pelo compilador.

Ponto-flutuante

Novos componentes → A nova versão de LALP inclui operadores descritos em VHDL para suporte a operações básicas de ponto-flutuante. Os componentes seguem o padrão IEEE-754 ([IEEE, 1985](#)) para operações com 32 *bits*, mas podem ser configurados para criação de tipos com tamanhos variados.

Módulos parametrizáveis → Além dos operadores citados, a nova versão inclui módulos parametrizáveis para adição e multiplicação com ponto-flutuante. Tais módulos são descritos em LALP e podem ser usados por meio de parâmetros próprios de compilação.

Sintaxe → Em adição às alterações de sintaxe descritas, mudanças referentes ao uso de ponto-flutuante foram realizadas na linguagem para permitir a utilização dos componentes referidos anteriormente.

Gerenciamento de memória e sincronismo

Acesso aos vetores → A indexação de vetores com múltiplos acessos em código LALP se dá de forma indireta, sendo, usualmente, necessário definir uma variável para o endereçamento e manipulá-la a cada ciclo de *clock* de forma a poder acessar diferentes posições do vetor. Durante a geração de *hardware* esta variável é implementada como um registrador conectado à porta de endereço da memória. Com a nova versão é possível indexar o vetor de forma semelhante ao que ocorre em códigos C, no formato *Vet[lin][col]*, sendo gerados componentes específicos para controle do acesso.

Componentes para gerenciamento de memória → Uma das limitações da versão original do LALP é o acesso a múltiplos endereços de memória em uma mesma iteração do *loop*. Tal limitação é decorrente da impossibilidade de acessos em paralelo, bem como da dificuldade de sincronização dos acessos com o restante do circuito. A fim de contornar estes problemas, foram criados componentes para gerenciar o acesso a memória. O primeiro componente baseia-se em uma máquina de estados para sincronizar o acessos sequenciais, enquanto o outro implementa um conjunto de blocos funcionais para acesso a múltiplos endereços em paralelo. Ambos controlam o acesso de forma a garantir que todos os dados necessários para a computação estarão disponíveis ao mesmo tempo, após um número mínimo de ciclos. Tais componentes são gerados automaticamente e dependem tanto de configurações e parâmetros de compilação, quanto do código em si.

Mapeamento de vetores → Na nova versão do LALP, apresentada neste trabalho, é possível mapear os vetores em múltiplas memórias, ao contrário do que ocorre na versão original, onde cada vetor declarado no código corresponde a uma única memória BRAM no *hardware*. Com isto, o uso de transformações como *loop unrolling* e *loop splitting* torna-se facilmente implementável em LALP.

Cálculo automático de delays ← A versão original de LALP dispõe de uma série de algoritmos para cálculo de *delays* para balanceamento do *hardware* gerado. Contudo, o sincronismo do circuito precisa ser feito manualmente pelo programador nos casos onde múltiplos ciclos de escrita e leitura são requeridos. Com a inclusão dos componentes para gerenciamento de memória é criada uma barreira de sincronização e o compilador passa a calcular

automaticamente os ciclos de espera necessários até que os dados estejam disponíveis. Este sincronismo varia de acordo com o particionamento de memória e com os endereços presentes no código.

Suporte a co-projeto de hardware e software

Canais de comunicação para acesso à memória → A nova versão LALP dispõe de canais de comunicação que permitem a escrita e leitura de dados na memória interna do acelerador LALP gerado. Tal funcionalidade simplifica a integração das arquiteturas geradas com outros IPs, como, por exemplo, o processador *soft-core* NiosII, fabricado pela Altera®.

Integração com o NiosII → Uma das modificações realizadas no compilador LALP trata-se da inclusão de parâmetros para a criação de interfaces usadas na integração com processador *soft-core* NiosII. Com isto, é possível exportar os blocos de *hardware* gerados em LALP para a utilização por meio de *software* desenvolvido para o NiosII. Com isto, permite-se a execução de um programa completo dentro da filosofia de co-projeto de *hardware* e *software*.

Síntese de alto nível

Versão preliminar de ferramenta de HLS → Embora LALP tenha um *framework* independente, uma das motivações para seu desenvolvimento é permitir que as arquiteturas geradas sejam incorporadas a uma ferramenta de Síntese de alto nível. Neste sentido, uma versão preliminar desta ferramenta foi desenvolvida durante a pesquisa. Contudo, esta versão preliminar foi implementada em um escopo diferente do abordado pelo LALP, tratando de otimizações de código, possuindo maior foco na exploração do espaço de projeto do que no desenvolvimento de aceleradores em *hardware* a partir de C. O [Capítulo 7](#) apresenta detalhes desta ferramenta.

Exploração de espaço de projeto → Esta ferramenta realiza uma exploração de espaço de projeto com uma abordagem multi-objetivo. Isto é feito a fim de encontrar soluções com uma boa relação entre desempenho e uso de recursos.

Transformações de código → A ferramenta executa a exploração do espaço de projeto considerando transformações de código e configuração de memória *cache* para um processador *soft-core* customizado. As transformações são realizadas no código C de entrada e aplicadas por um compilador *source-to-source* baseado no compilador ROSE.

Quadro 8 – Comparação entre LALP e LALP+

Característica	LALP	LALP+
Balanceamento automático do <i>datapath</i>	Sim	Sim
Inserção manual de <i>delays</i>	Sim	Sim ⁽¹⁾
Definição de tipos personalizados	Sim	Sim ⁽²⁾
Tipos básicos pré-definidos	Não	Sim
Criação de <i>testbenches</i>	Sim	Sim
Suporte a <i>loops</i> aninhados	Não	Sim
Suporte a operações básicas de ponto-flutuante	Não	Sim
Cálculo de expressões com constantes	Não	Sim
Geração dinâmica de componentes	Não	Sim
Mapeamento de vetores	Fixo	Variável
Particionamento automático de memória	Não	Sim
Sincronização de múltiplos acessos por ciclo	Manual	Automático
Canais de comunicação para acesso externo	Não	Sim
Integração com o NiosII	Não	Sim

Fonte: Elaborada pelo autor.

(1): Requerido apenas em alguns casos.

(2): Alterado para compatibilidade com a nova versão.

5.5 Considerações finais

Apesar de LALP prover uma alternativa para desenvolvimento de aceleradores em FPGA de mais alto nível que as HDLs, ainda mantém explícito o uso de recursos de baixo nível. Isto conflita com a proposta inicial de LALP em facilitar seu uso por meio de programadores não especializados.

Assim, este capítulo apresentou uma nova versão de LALP, denominada LALP+. De modo geral, LALP+ aproveita as ideias contidas em LALP, provendo uma versão estendida da linguagem, incorporando um conjunto de novas funcionalidades e modificações que visam tratar os principais problemas da versão original. Simultaneamente, LALP+ tornou alguns aspectos de *hardware* transparentes ao programador, contendo uma abordagem de mais alto nível do que a usada em LALP.

EXPERIMENTOS E RESULTADOS

6.1 Considerações iniciais

Este capítulo apresenta os experimentos e resultados de LALP+ em comparação a outras ferramentas. Os experimentos ocorreram de forma a avaliar as melhorias propostas em LALP+, considerando as quatro frentes principais seguidas durante o desenvolvimento da pesquisa, as quais são: operadores de ponto flutuante, síntese de alto nível, sincronismo e acesso paralelo aos dados, e co-projeto de *hardware* e *software*.

Foi utilizado, prioritariamente, um dispositivo FPGA da família Stratix® IV da Altera, modelo EP4SGX530KH40C20. Porém, parte dos resultados também inclui o uso de FPGA da Xilinx, família Virtex 7, modelo XC7VX330t-1-ffg1157.

Como métricas principais utilizou-se o desempenho obtido, considerando a frequência máxima atingida e a latência de cada arquitetura, além do total de recursos de *hardware* utilizados. Os dados foram extraídos utilizando as ferramentas QuartusII 14.0, no caso da Altera, e Vivado 2014.3, no caso da Xilinx, tendo sido considerados os valores fornecidos nos relatórios de síntese das mesmas. Parte dos resultados apresentados foram publicados em [Oliveira *et al.* \(2015\)](#), [Oliveira, Cardoso e Marques \(2014\)](#), [Oliveira, Cardoso e Marques \(2014\)](#) e [Oliveira, Cardoso e Marques \(2013\)](#).

6.2 Testes e resultados comparativos

6.2.1 Operadores para ponto flutuante

Os componentes de ponto flutuante desenvolvidos foram avaliados como operadores isolados. Foram consideradas ambas as opções providas, ou seja, utilizando operadores combinacionais desenvolvidos com a biblioteca *IEEE Proposed* ou operadores em *pipelining* criados em LALP. Apesar da primeira opção contemplar as quatro operações básicas, somente é possível

realizar comparações entre as versões dos operadores de adição e multiplicação, uma vez que estes são os únicos implementados como módulos LALP separados. Para os demais operadores (subtração e divisão) foram incluídos resultados apenas da versão combinacional. Em todos os casos são utilizados operadores no padrão IEEE 754 de precisão simples (32 *bits*).

Os operadores LALP foram comparados a componentes similares gerados com a ferramenta Flopoco 3.0 (DINECHIN; PASCA, 2011), um *framework* para geração de operadores aritméticos em *pipelining*. Flopoco permite a criação de operadores de ponto flutuante para várias operações, tendo sido utilizado por tratar-se de ferramenta de código aberto e por fazer uso de uma abordagem em *pipelining*, similar ao que ocorre em LALP/LALP+.

Flopoco utiliza uma representação no formato LNS (CHUGH; PARHAMI, 2013), onde os números são representados em notação logarítmica. Assim, faz-se necessário incluir componentes para conversão deste formato para o padrão IEEE 754. Tais conversores são fornecidos pelo próprio *framework* e estão integrados aos operadores usados nos testes apresentados aqui.

De modo geral, os operadores LALP implementados com a biblioteca *IEEE Proposed* apresentaram os piores resultados dentre as versões testadas. Contudo, estes operadores são implementados com lógica combinacional, gastando apenas 1 ciclo de relógio por operação, com 1 ciclo extra para escrita do resultado em um registrador. Desta forma o caminho crítico do operador é longo, incorporando todas as etapas da operação, contribuindo para a redução da frequência. Isto explica as baixas frequências obtidas nesses casos, como pode ser visto na Figura 36. No caso dos operadores LALP implementados em *pipelining*, ambos gastam 6 ciclos por operação, enquanto que os operadores Flopoco, também implementados em *pipelining*, gastam 20 e 27 ciclos para as operações de multiplicação e adição, respectivamente (Figura 37).

Figura 36 – Frequência máxima em MHz obtida pelos operadores LALP e Flopoco para adição e multiplicação em ponto flutuante, usando um FPGA StratixIV (modelo EP4SGX530KH40C20) da Altera®

Fonte: Elaborada pelo autor.

Os operadores Flopoco apresentam as frequências mais altas, mas com uma diferença

Figura 37 – Latência dos operadores LALP e Flopoco para adição e multiplicação em ponto flutuante, usando um FPGA StratixIV (modelo EP4SGX530KH40C20) da Altera®

Fonte: Elaborada pelo autor.

mínima em relação ao multiplicador LALP em *pipelining* (Figura 36). Mesmo apresentando as frequências mais baixas, os operadores combinacionais executariam o cálculo em menor tempo do que os gerados pelo Flopoco, caso fosse considerada uma única operação. Entretanto, isto não se mantém à medida em que aumenta o número de operações sequenciais. Neste caso, ambos os operadores LALP em *pipelining* mostram resultados similares ao Flopoco em relação ao tempo de execução (Figuras 38 e 39).

Figura 38 – Tempo(μ s) da operação soma dos operadores LALP e Flopoco para adição e multiplicação em ponto flutuante, usando um FPGA StratixIV (modelo EP4SGX530KH40C20) da Altera®

Fonte: Elaborada pelo autor.

Figura 39 – Tempo(μ s) da operação de multiplicação dos operadores LALP e Flopoco para adição e multiplicação em ponto flutuante, usando um FPGA StratixIV (modelo EP4SGX530KH40C20) da Altera®

Fonte: Elaborada pelo autor.

A Figura 40 apresenta os resultados referentes ao uso de recursos pelos operadores. Como mostrado na figuras 40a, os operadores Flopoco utilizam aproximadamente 4 vezes mais registradores do que os operadores LALP em *pipelining* ($3,97 \times$ p/ somador e $4,23 \times$ p/ multiplicador). Em relação ao uso de ALUTs, o somador Flopoco usa 77% menos que o somador LALP, já o multiplicador Flopoco usa $7,94 \times$ mais recursos 40b. Ainda tratando-se desses operadores, o multiplicador LALP usa 2 DSPs enquanto que o Flopoco utiliza 4.

O maior uso de ALUTs pelo somador LALP em *pipelining* da-se por conta da implementação da etapa de deslocamento e ajuste da *mantissa*. Como esta etapa depende da diferença entre os expoentes, a implementação é feita de forma especulativa, com todos os possíveis casos executados em paralelo e apenas o caso correto sendo considerado ao final da etapa. Embora isto simplifique a implementação em LALP, resulta em um alto uso de recursos combinacionais.

Dada que os operadores LALP em *pipelining* apresentam tempos de execução similares aos do Flopoco, a quantidade de recursos utilizados por estes últimos pode ser um fator limitante para seu uso. Além disto, o número de estágios dos operadores LALP pode ser explorado de forma encontrar melhores arquiteturas.

A Figura 41 exemplifica esta exploração mostrando a variação da frequência máxima obtida em relação à quantidade de estágios de um operador LALP de multiplicação. Este exemplo refere-se ao uso do operador LALP em um FPGA Xilinx®, o que demonstra a possibilidade de uso dos operadores em diferentes FPGAs.

Figura 40 – Resultados referentes ao uso de recursos dos operadores LALP e Flopoco para adição e multiplicação em ponto flutuante, usando um FPGA StratixIV (device EP4SGX530KH40C20) da Altera®

Fonte: Elaborada pelo autor.

Figura 41 – Frequência (MHz) máxima de *clock* atingida com o módulo LALP de multiplicação para números variados de estágios de *pipeline*, usando FPGA Xilinx Virtex7 XC7VX330T

Fonte: Elaborada pelo autor.

6.2.2 LALP vs. HLS

A comparação entre essas duas abordagens, trata-se, de fato, de uma comparação entre uma abordagem intrinsecamente baseada em contadores, *delays* enfileirados e controle distribuído (LALP), e outra abordagem orientada a máquinas de estado, com uma unidade de controle centralizada, normalmente obtida a partir do grafo de fluxo de controle referente ao exemplo que estiver sendo compilado (HLS).

LALP+ foi avaliado usando o conjunto de *benchmarks* mostrado no [Quadro 9](#). Estes algoritmos foram codificados em LALP/LALP+¹ a partir do código original equivalente em C. As versões LALP seguem o máximo possível o algoritmo e as estruturas computacionais dos códigos originais em C, o que é importante para tornar a comparação justa.

Estão inclusos resultados para duas ferramentas de HLS: Vivado HLS e LegUp. A primeira tem como alvo FPGAs da Xilinx, tendo sido usado um dispositivo da família Virtex 7 (modelo XC7VX330t-1-ffg1157). A segunda foca-se em dispositivos fabricados pela Altera, sendo apresentados resultados para um FPGA Stratix IV (modelo EP4SGX530KH40C20). Nas próximas subseções são mostrados os resultados para LALP+ em comparação a cada ferramenta separadamente, devido às diferenças entre as tecnologias e capacidades dos dispositivos.

Em ambos os casos foi usada a configuração padrão das ferramentas, sendo incluídas diretivas de *pipelining* com intervalo de iniciação (II) alvo configurado para 1 para os *loops* internos. Apesar dessas ferramentas fornecerem outras opções de transformações, apenas a de

¹ Alguns códigos apresentam sintaxe compatível com ambas as versões. Além disso, para os casos onde não há acessos múltiplos, o código VHDL é gerado utilizando a mesma arquitetura em ambas as versões.

Quadro 9 – Descrição dos *benchmarks* utilizados, com indicação das ferramentas onde foram implementados

Benchmarks	Descrição	LALP+	Vivado HLS	LegUp
<i>adpcm_cod</i>	Codificador de áudio	✓	✓	✓
<i>adpcm_dec</i>	Decodificador de áudio	✓	✓	✓
<i>autocor</i>	Autocorrelação entre 2 vetores	✓	✓	✓
<i>dotprod</i>	Produto escalar de dois vetores	✓	✓	✓
<i>pop_cnt</i>	Algoritmo Pop. count	✓	✓	✓
<i>fdct</i>	Fast Discrete Cosine Transform	✓	✓	✓
<i>fibonacci</i>	Cálculo da sequência de Fibonacci	✓	✓	✓
<i>fir</i>	Filtro FIR	✓	✓	✓
<i>max</i>	Valor máximo de um vetor	✓	✓	✓
<i>sobel</i>	Operador de Sobel	✓	✓	✓
<i>vecsum</i>	Soma de vetores	✓	✓	✓
<i>vecmult</i>	Multiplificação de vetores	✓	✓	✓
<i>matmul</i>	Multiplificação de matrizes	✓	✓	✓
<i>conv_3x3</i>	Convolução 3x3	✓		✓
<i>median_3x3</i>	Mediana 3x3	✓		✓
<i>perimeter</i>	Perímetro de uma imagem	✓		✓
<i>prewitt</i>	Operador de Prewitt	✓		✓
<i>threshold</i>	Algoritmo de halftone	✓		✓
<i>boundary</i>	Computação de pixels internos	✓		✓
<i>change_brightness</i>	Correção de brilho	✓		✓
<i>compositing</i>	Composição de fundo de imagem	✓		✓
<i>pix_sat</i>	Saturação de 16 para 8 bits	✓		✓

Fonte: Elaborada pelo autor.

loop pipelining foi considerada, pois, caso contrário, seria preciso aplicar as demais transformações também em LALP+ para manter a comparação justa. Considere que, mesmo LALP+ requerendo um menor esforço para a implementação de circuitos do que com uso de HDLs, ainda requer mais esforços do que utilizando uma ferramenta de HLS, especialmente porque o compilador LALP+ não é tão sofisticado em termos de otimizações, sendo que, na maioria dos casos, o programador deve realizar as transformações manualmente. Além disso, também não foram utilizadas otimizações específicas para os modelos de FPGAs usados, uma vez que não é nossa intenção comparar as tecnologias suportadas pelos diferentes fabricantes.

6.2.2.1 LALP vs. Vivado HLS

A Figura 42 mostra o *speedup* de tempo de execução atingido pelas arquiteturas LALP+ em relação às geradas pelo Vivado HLS. LALP+ atingiu um *speedup* com média geométrica de $2.66\times$ sobre o Vivado HLS para um conjunto de 13 *benchmarks*.

LALP+ gerou arquiteturas cujos tempos de execução são mais baixos que os obtidos pelo Vivado HLS, com exceção dos casos *fdct* e *vecmult*. No primeiro, LALP+ apresenta maior latência (Figura 44), enquanto que no segundo caso Vivado HLS apresenta maior frequência

Figura 42 – *Speedup* de tempo de execução atingido pelas arquiteturas LALP+ em relação às geradas pelo Vivado HLS, usando um FPGA Xilinx Virtex 7 (device XC7VX330t-1-ffg1157)

Fonte: Elaborada pelo autor.

de *clock* (Figura 43). Apesar destes casos, os resultados mostram que LALP+ obteve melhor desempenho com um menor uso de recursos do que Vivado HLS, conforme mostrado nas figuras 45 e 46).

Figura 43 – Frequência máxima (em MHz) obtida pelas arquiteturas LALP+ e Vivado HLS, usando um FPGA Xilinx Virtex 7 (device XC7VX330t-1-ffg1157)

Fonte: Elaborada pelo autor.

Em média, as arquiteturas LALP+ atingiram valores máximos de frequência maiores ou

Figura 44 – Latência (em ciclos de relógio) das arquiteturas LALP+ e Vivado HLS, usando um FPGA Xilinx Virtex 7 (device XC7VX330t-1-ffg1157)

Fonte: Elaborada pelo autor.

similares aos do Vivado HLS (Figura 43). Contudo, há casos onde LALP+ obteve frequências significativamente menores, como nos casos *autocor*, *dotprod*, *fir*, *vecmult* e *matmul*. Uma razão possível para isso é o uso de multiplicadores sem *pipeline* nesses exemplos LALP+.

O único caso onde LALP+ apresenta uma maior latência é o *fdct* (Figura 44). Isto dá-se devido à limitada exploração do escalonamento e operações nos *loops* dos exemplos. O compilador LALP+ ainda é limitado em relação a escalonar e explorar muitas possibilidades, possuindo maior enfoque em dependências para balanceamento do circuito, e, portanto, não tratando bem algumas situações. Nesse exemplo, fez-se necessário escalonar os *loops* manualmente, sem grandes esforços de otimização, de forma a obter uma arquitetura funcional. Entretanto, é possível que um esforço adicional de implementação e exploração do espaço de soluções leve a arquiteturas com um melhor desempenho.

Em relação aos recursos de *Flip-Flops*(FFs) utilizados, os números variam bastante de acordo com o exemplo, conforme Figura 46. Em 9 dos 13 casos LALP+ gerou arquiteturas com menos FFs que o Vivado HLS, embora este último apresente uma menor (porém similar) média. O uso em LALP+ de contadores, registradores de deslocamento para imposição de *delays*, e componentes sem *pipeline* justifica esses resultados. No tocante ao número de LUTs (Figura 45), as arquiteturas LALP também usaram menos recursos na maioria dos exemplos, embora haja casos (como *autocor* e *fdct*) onde LALP+ precisou de muito mais LUTs.

Em termos de utilização de blocos DSP48E, LALP+ também fez maior uso do que o Vivado HLS, uma vez que não há compartilhamento de recursos em LALP+, diferente do que ocorreu nas arquiteturas Vivado HLS. A maior diferença ocorreu nos casos *fdct* e *fir*, onde LALP+

Figura 45 – Total de LUTs usadas nas arquiteturas LALP+ e Vivado HLS, usando um FPGA Xilinx Virtex 7 (device XC7VX330t-1-ffg1157)

Fonte: Elaborada pelo autor.

Figura 46 – Total de Flip-flops usados nas arquiteturas LALP+ e Vivado HLS, usando um FPGA Xilinx Virtex 7 (device XC7VX330t-1-ffg1157)

Fonte: Elaborada pelo autor.

usou 56 e 10 blocos, respectivamente, enquanto Vivado HLS usou 18 e 2, respectivamente. Ambas as ferramentas obtiveram resultados similares em relação ao uso de BRAMs, com uma pequena variação devida à promoção de certos *arrays* para ROMs no Vivado HLS.

6.2.2.2 LALP vs. LegUp

A Figura 47 mostra o *speedup* de tempo de execução obtido para cada *benchmark* LALP+ em relação aos gerados pelo LegUp. LALP+ atingiu um *speedup* com média geométrica de $1,8\times$ sobre o LegUp para um conjunto de 22 *benchmarks*.

Figura 47 – Speedups obtidos para as arquiteturas LALP+ em relação às geradas pelo LegUp, para um FPGA StratixIV (device EP4SGX530KH40C20) da Altera®

Fonte: Elaborada pelo autor.

Na média, LALP+ gerou arquiteturas que atingiram maior frequência, conforme a Figura 49. Dos casos em que LALP+ apresentou *speedup*, apenas *max* e *perimeter* obtiveram frequências mais baixas, atingindo, em ambos os casos, menor tempo de execução devido à menor latência.

Nota-se, no geral, um aumento da latência das arquiteturas LegUp para os algoritmos onde existem instruções condicionais (*if then else*) aninhadas. Observou-se que essas instruções afetam o cálculo do *pipeline* pelo LegUp de modo a gerar maiores intervalos de iniciação, e, portanto, aumentando a latência do circuito (ver Figura 48). Apesar de LALP+ não suportar as instruções *if then else*, é possível utilizar operadores ternários (*condition?result1:result2*), implementados em LALP+ com componentes VHDL que possuem latência de apenas 1 ciclo, o que contribui para reduzir a latência total do circuito.

Esta característica explica os *speedups* obtidos para os algoritmos *adpcm_coder*, *adpcm_decoder*, *perimeter* e *median_3x3*, os quais estão diretamente relacionados às altas latências das

Figura 48 – Latência (em ciclos de relógio) das arquiteturas LALP+ e LegUp, para um FPGA StratixIV (device EP4SGX530KH40C20) da Altera®

Fonte: Elaborada pelo autor.

arquiteturas LegUp nestes casos (ver Figura 49). Nos casos *adpcm_coder* e *adpcm_decoder*, tem-se, ainda, a obtenção de maiores frequências em LALP+, resultando nos valores de *speedup* de $3,27\times$ e $10,01\times$ para *adpcm_coder* e *adpcm_decoder*, respectivamente. Em adição a isto, o LegUp não conseguiu gerar arquiteturas em *pipelining* com as configurações utilizadas em nenhum destes dois casos.

Figura 49 – Frequência máxima (em MHz) obtida pelas arquiteturas LALP+ e LegUp, para um FPGA StratixIV (device EP4SGX530KH40C20) da Altera®

Fonte: Elaborada pelo autor.

LALP+ apresentou tempos de execução superiores apenas para os algoritmos *fir* e *pix_sat*. Nestes casos LALP+ obteve latências similares porém atingiu frequências mais baixas, mesmo utilizando maior quantidade de ALUTs e registradores (Figuras 51 e 50).

Para os algoritmos *conv_3x3 fibonacci* e *vecsum* LALP+ obteve tempos similares, com *speedups* de $1,01\times$, $1,05\times$ e $1,01\times$, respectivamente. Para o algoritmo *conv_3x3* LALP+ obteve menor latências mas, também, menor frequência. Nos dois casos restantes, os valores de frequência e latência foram similares. Ainda em relação a estes três casos, LALP+ utilizou maior quantidade de recursos (ALUTs e registradores) para *conv_3x3* e *vecsum*, conforme figuras 51 e 50.

Figura 50 – Total de registradores usados pelas arquiteturas LALP+ e LegUp, para um FPGA StratixIV (device EP4SGX530KH40C20) da Altera®

Fonte: Elaborada pelo autor.

No tocante ao uso de ALUTs, LALP+ usou, em média, menos recursos, tendo usado maior quantidade em apenas 5 dos 22, dos quais 2 (*fir* e *pix_sat*) não geraram arquiteturas com *speedup*. Em semelhança ao que ocorreu em comparação ao Vivado HLS, LALP+ usou em média mais registradores, cerca de 13% a mais. Entretanto em apenas 6 casos esse uso foi superior, com destaque para o *fdct*, onde foram usados $3,3\times$ mais registradores.

6.2.3 Interfaces para co-projeto

As interfaces desenvolvidas para integração dos aceleradores LALP+ a co-projetos de *hardware* e *software* inserem componentes e *overheads* próprios de cada abordagem. Assim, diferentes versões de um filtro FIR de 32bits foram implementadas em LALP+ de forma a verificar o impacto, em termos de área e frequência, que cada interface causa nos aceleradores.

Figura 51 – Total de ALUTs usadas pelas arquiteturas LALP+ e LegUp, para um FPGA StratixIV (device EP4SGX530KH40C20) da Altera®

Fonte: Elaborada pelo autor.

A Tabela 1 mostra os resultados obtidos para as quatro versões de filtro FIR testadas. São elas:

fir → Opção padrão, sem interface e com os vetores mapeados em BRAMs.

fir_ic → Versão semelhante a *fir*, mas com interface para uso como instrução customizada do NiosII.

fir_no_mem → Versão sem memória internas. Os dados são lidos/escritos diretamente das/nas portas de entrada/saída do acelerador.

fir_fifo → Semelhante à versão *fir_no_mem* mas com as portas conectadas a FIFOs.

Tabela 1 – Resultados obtidos para diferentes versões FIR, usando um FPGA StratixIV (modelo EP4SGX530KH40C20)

Versão	ALUTs	Registers	Freq. Máxima(MHz)	Latência
<i>fir</i>	119	131	332,34	4
<i>fir_ic</i>	122	136	332,56	7
<i>fir_no_mem</i>	96	130	332,23	4
<i>fir_fifo</i>	113	137	332,34	8

Fonte: Elaborada pelo autor.

Todas as abordagens apresentam, praticamente, a mesma frequência, variando entre 332,26MHz e 332,56MHz da menor para a maior. Isto mostra que as interfaces têm pouco impacto na frequência máxima do circuito.

Considerando os dois casos com BRAMs internas (*fir* e *fir_ic*) há uma variação de 3 ALUTs e 5 registradores, sendo que a versão *fir_ic* usa mais recursos. Esta variação deve-se à

inclusão de sinais de controle e multiplexadores para composição da interface. A versão *fir_ic* apresenta, ainda, uma maior latência, com a inserção ciclos referentes à sincronização.

Semelhantemente, nos casos *fir_no_mem* e *fir_fifo*, onde os dados são inseridos nos aceleradores diretamente nas portas de entrada, a maior latência do último caso deve-se às FIFOs inseridas. Além do aumento da latência, o uso das FIFOs também requer mais recursos, embora essa diferença seja de apenas 7, tanto para as ALUTs como para os registradores.

Diferenças entre as versões são esperadas. Porém, os resultados mostram que trata-se de baixos *overheads* em todos os casos. Embora o suporte a estas interfaces ainda seja limitado, elas facilitam a integração dos aceleradores a outros componentes e o impacto destas interfaces no circuito não afeta seu uso em termos práticos.

6.3 Arquiteturas com múltiplos acessos e particionamento de dados

As abordagens providas por LALP+ para tratamento de casos onde há a ocorrência de múltiplos acessos a um mesmo *array* por iteração (ver [subseção 5.3.3](#)) possibilitam a geração de arquiteturas variadas. Assim, nesta seção são apresentados os resultados comparativos entre as diferentes abordagens LALP+ para *benchmarks* com essa característica. Ressalta-se, porém que faz-se necessário um estudo mais complexo a fim de determinar toda a capacidade e impacto das abordagens adotadas em LALP+, uma vez que a eficiência das arquiteturas depende de outros fatores, como a quantidade de memórias utilizadas e decisões feitas pelo programador no tocante à organização e particionamento dos dados. Desta forma, a principal intenção destes experimentos é mostrar e explorar algumas possibilidades viáveis para a implementação de arquiteturas LALP+ utilizando memórias distribuídas.

Para este fim foram utilizadas 6 versões LALP+ distintas, mostradas no [Quadro 10](#). Os algoritmos usados correspondem a cinco *benchmarks* utilizados na [subseção 6.2.2](#): *sobel*, *prewitt*, *conv_3x3*, *median_3x3* e *perimeter*, os quais possuem múltiplas operações de acesso por iteração.

Quadro 10 – Versões LALP+ usadas nos experimentos com *benchmarks* para acessos múltiplos

Versão	Arquitetura utilizada	Acesso
<i>FSM</i>	FSM com 1 BRAM, sem partionamento de dados	Sequencial
<i>MMU_1</i>	MMU com 2 BRAMS, com partionamento de dados	Paralelo
<i>MMU_2</i>	MMU com 4 BRAMS, com partionamento de dados	Paralelo
<i>MMU_3</i>	MMU com 8 BRAMS, com partionamento de dados	Paralelo
<i>MMU_4</i>	MMU com 16 BRAMS, com partionamento de dados	Paralelo
<i>MMU_5</i>	MMU com 32 BRAMS, com partionamento de dados	Paralelo

Fonte: Elaborada pelo autor.

Nas versões com particionamento de dados, cada uma das várias BRAMs criadas é,

proporcionalmente, menor que a única BRAM do caso sequencial. Ou seja, o tamanho das BRAMs é igual a $1/N$, arredondado para cima, onde N é o total de BRAMs utilizadas. Apesar disso, o uso de componentes para gerenciamento dos acessos paralelos implica na utilização de uma maior quantidade de recursos de *hardware* do FPGA, à medida que aumenta a quantidade de BRAMs, dado o *overhead* necessário para o controle dos componentes. A Tabela 2 apresenta os resultados obtidos para as diferentes arquiteturas com múltiplos acessos implementadas.

O aumento do número de recursos pode ser visto nas figuras 52 e 53. O comportamento é similar em todos os casos, sendo as versões com FSM aquelas que utilizam menos recursos, tanto em relação ao número de ALUTs (Figura 52) quanto ao de registradores (Figura 53).

Tabela 2 – Resultados obtidos para diferentes arquiteturas com múltiplos acessos, usando um FPGA StratixIV (modelo EP4SGX530KH40C20)

		ALUTs	Registradores	Max. Freq.(MHz)	Latência
<i>sobel</i>	FSM	265	221	285,5	30754
	MMU 1	363	472	178,13	23070
	MMU 2	562	558	179,79	11541
	MMU 3	880	740	163,52	11541
	MMU 4	1589	1094	143,23	11541
	MMU 5	3494	1880	126,58	11541
<i>prewitt</i>	FSM	254	243	324,78	30745
	MMU 1	362	489	191,64	23059
	MMU 2	556	575	171,35	11530
	MMU 3	878	757	152,16	11530
	MMU 4	1608	1111	147,49	11530
	MMU 5	1917	1516	145,94	11530
<i>conv_3x3</i>	FSM	270	316	233,59	1359
	MMU 1	364	640	210,61	916
	MMU 2	627	593	181,88	457
	MMU 3	873	759	154,42	312
	MMU 4	1565	1147	138,93	162
	MMU 5	3117	1865	132,94	162
<i>median_3x3</i>	FSM	319	512	380,23	3019
	MMU 1	298	645	358,68	3019
	MMU 2	359	703	353,48	3019
	MMU 3	467	791	281,06	3019
	MMU 4	626	896	294,99	2019
	MMU 5	1828	1423	213,68	1019
<i>perimeter</i>	FSM	96	134	418,76	2393
	MMU 1	170	326	205,25	1916
	MMU 2	290	382	183,42	962
	MMU 3	457	502	160,51	487
	MMU 4	1261	848	154,32	487
	MMU 5	2016	1430	129,4	487

Fonte: Elaborada pelo autor.

Figura 52 – Total de ALUTs usadas pelas diferentes versões LALP+ testadas

Fonte: Elaborada pelo autor.

Figura 53 – Total de Registradores usados pelas diferentes versões LALP+ testadas

Fonte: Elaborada pelo autor.

Conforme mostrado na [subseção 5.3.3.3](#), as arquiteturas com particionamento de dados dependem dos endereços acessados e do tamanho dos *arrays*. Assim, cada um dos *benchmarks* escolhidos representa uma característica distinta no tocante ao acesso aos dados. As exceções são os casos *prewitt* e *sobel*, onde os algoritmos correspondentes apresentam comportamento similar entre si, uma vez que os dados são acessados seguindo o mesmo padrão em ambos os casos.

A [Tabela 3](#) traz a quantidade de acessos por iteração para cada *benchmark*, seguida da quantidade de ciclos de relógio necessários (por iteração) para realizar estes acessos em cada versão LALP+ usada. Nos casos *prewitt* e *sobel* são realizados 8 acessos por iteração. Nestes casos os dados estão organizados em *arrays* multidimensionais, e os acessos ocorrem em 3 colunas. No algoritmo *perimeter* são realizados 5 acessos, também em 3 colunas. Contudo, os dados estão organizados em um *array* unidimensional. O mesmo ocorre nos casos *conv_3x3* e *median_3x3*, os quais realizam, respectivamente, 9 e 3 acessos por iteração.

Tabela 3 – Número de acessos e total de ciclos de relógio por iteração para cada *benchmark*

<i>Benchmarks</i>	Acessos p/ iter.	Ciclos p/ iteração					
		FSM	MMU 1	MMU 2	MMU 3	MMU 4	MMU 5
<i>sobel</i>	8	8	6	3	3	3	3
<i>prewit</i>	8	8	6	3	3	3	3
<i>conv_3x3</i>	9	9	7	3	2	1	1
<i>median_3x3</i>	3	3	3	3	3	2	1
<i>perimeter</i>	5	5	4	2	1	1	1

Fonte: Elaborada pelo autor.

Como visto na [Tabela 3](#), o uso de particionamento em múltiplas BRAMs ajuda a diminuir o número de ciclos necessários para realizar todos os acessos. Como consequência da redução de ciclos por iteração, proporcionada pelo uso das arquiteturas com MMU, há uma diminuição da latência total do circuito, conforme mostrado na [Figura 54](#). Entretanto, essa diminuição não depende exclusivamente da quantidade de memórias utilizadas, mas, também, dos endereços, tamanho e forma como os dados são organizados.

As frequências máximas atingidas pelas arquiteturas geradas são mostradas na [Figura 55](#). Nota-se que as arquiteturas com FSM atingiram maior frequência que as que utilizam MMU em todos os casos.

A [Figura 56](#) mostra o *speedup* das versões paralelas (com MMU) tomando como referência a versão sequencial (com FSM). O melhor caso varia de acordo com o algoritmos, mas em nenhum caso a versão *MMU_5* mostrou-se a melhor opção, mesmo sendo a opção menos econômica em termos de recursos de *hardware*.

Dentre os algoritmos testados, o caso *conv_3x3* apresenta uma maior diferença à medida em que aumenta o número de BRAMs, proporcionando uma redução da latência de $9\times$ neste caso. Contudo, o total de recursos de *hardware* também aumenta consideravelmente. O maior

Figura 54 – Latência total (em ciclos de relógio) de cada uma das diferentes versões LALP+ testadas

Fonte: Elaborada pelo autor.

Figura 55 – Frequência máxima (em MHz) atingida em cada uma das diferentes versões LALP+ testadas

Fonte: Elaborada pelo autor.

Figura 56 – *Speedup* das arquiteturas com acessos paralelos (MMUs) em relação à arquitetura com acesso sequencial (FSM)

Fonte: Elaborada pelo autor.

speedup ocorre na versão *MMU_4*. Essa redução de latência de $9\times$ ocorre em *conv_3x3*, e não nos demais, porque, quanto mais endereços são acessados por ciclo, maior o potencial de redução da latência. Isto é, se são feitos k acessos por ciclo, a maior redução possível na latência é de k vezes, independente do número de memórias usadas. Além disso, o melhor caso ocorre quando apenas 1 ciclo é necessário, porém, nem sempre o endereçamento permite que isso ocorra. O caso *conv_3x3*, portanto, apresenta o maior número de acessos paralelizáveis, atingindo 1 ciclo de latência por iteração com 16 BRAMs (*MMU_4*). O caso *median_3x3*, mesmo também atingindo de 1 ciclo por iteração, possibilita uma redução de latência de no máximo $3\times$.

Nos casos *sobel* e *prewitt* o maior *speedup* ocorre na versão *MMU_2*. Já para os casos *perimeter* e *median_3x3* as versões *MMU_1* e *MMU_2*, respectivamente, apresentam os melhores resultados.

Como os resultados mostram, o uso das arquiteturas personalizadas proporciona o benefício de permitir múltiplos acessos paralelos. O custo de recursos é aceitável para os casos onde há ganho de desempenho, mas isto varia de acordo com as opções de particionamento escolhidas pelo programador. Com isto, este tipo de abordagem pode ser usado para explorar alternativas de mapeamento de dados para processamento paralelo em LALP+, em adição à técnica de *loop pipelining*. Tendo em vista as várias possibilidades de projeto, a arquitetura final deve ser escolhida considerando a relação custo/benefício de cada abordagem.

6.4 Considerações finais

Este capítulo apresentou os experimentos e resultados obtidos por arquiteturas geradas usando LALP+. Os experimentos focaram as principais funcionalidades implementadas por esta versão, sendo os resultados comparados com outras ferramentas, comerciais e acadêmicas. As arquiteturas LALP+ mostraram um *speedup* com média geométrica de $2,66\times$ sobre arquiteturas geradas pelo Vivado HLS e $1,8\times$ sobre as geradas pelo LegUp, para um conjunto de 13 e 22 *benchmarks*, respectivamente, tendo atingido *speedup* de $10,01\times$, no melhor caso.

LALP+ NO CONTEXTO DE SÍNTESE DE ALTO NÍVEL

7.1 Considerações iniciais

Em adição aos resultados apresentados, este capítulo apresenta o projeto de uma ferramenta para HLS utilizando LALP+. Durante a pesquisa foi desenvolvida uma versão preliminar desta, à parte do escopo principal da tese, e que não está integrada ao LALP+. O desenvolvimento dessa versão preliminar foi motivado pela possibilidade de integração com LALP+, futuramente, considerando que LALP+ foi desenvolvida para auxiliar no desenvolvimento de sistemas embarcados com FPGAs.

A ferramenta trata de transformações de código envolvendo *loops*, sendo que a versão atual foca-se em otimizar código para um processador *softcore* baseado no NiosII® e não tem objetivo de gerar *hardware*. LALP+ opera de forma independente e embora não esteja integrada à ferramenta, os códigos C gerados por esta podem servir como base para os aceleradores LALP+. Além disto, estes aceleradores podem ser integrados como co-processadores do sistema.

7.2 Visão geral da ferramenta atual

A Figura 57 mostra as etapas da versão preliminar da ferramenta desenvolvida, a qual foi inicialmente apresentada por Martins *et al.* (2014). O sistema recebe um arquivo com código C como entrada. A partir deste código busca-se encontrar conjuntos de transformações a serem aplicadas e definir os tamanhos otimizados de memória *cache* de dados e instrução para o processador *soft-core* utilizado a fim de obter uma boa relação entre desempenho (menor tempo de execução) e tamanho de memória. O processador usado, denominado *BSP Processor* (PEREIRA, 2014), baseia-se no conjunto de instruções *assembly* do NiosII da Altera. O mesmo permite, dentre outras configurações, a definição do tamanho das memórias *cache* de dados e de instrução,

sendo esta uma característica explorada neste sistema.

O sistema possui três estágios principais: *Análise de código*, *Exploração de espaço de projeto* (DSE) e *Seleção de soluções*. Inicialmente, o compilador analisa o código para extrair informações sobre os *loops*, como número de iterações, aninhamento, valores inicial e final, etc. Estas informações são extraídas a partir da análise da AST (*Abstract Syntax Tree*) do programa. De posse destas informações, o sistema inicia a exploração do espaço de projeto por meio do algoritmo NSGA II, onde o resultado será um conjunto de soluções não dominadas para os objetivos determinados.

Figura 57 – Etapas da versão desenvolvida do Sistema de HLS

Fonte: Adaptada de Oliveira, Cardoso e Marques (2014).

O processo de DSE inicia com a geração de um conjunto de possíveis soluções (população inicial), sendo que cada solução corresponde a um indivíduo da população e consiste em uma lista de transformações, com seus respectivos parâmetros, e definições de memória *cache* de dados e instrução. Inicia-se, então, uma busca iterativa para determinar o conjunto de soluções correspondente à população final. Nesta etapa, para cada indivíduo da população, um compilador *source-to-source* aplica as transformações ao código C original, gerando um novo código C otimizado. Em seguida, o processador é configurado conforme os parâmetros de *cache* definido

em conjunto com as transformações. O novo código C é então compilado para o processador com estas configurações.

O compilador desenvolvido trata das etapas de geração e análise da AST, geração das informações sobre *loops* e otimização por meio da aplicação de transformações. Para isto é utilizada a infraestrutura do compilador ROSE (Figura 58), sendo provido suporte para as transformações de código descritas no Quadro 11.

Figura 58 – Infraestrutura de um tradutor escrito usando ROSE

Fonte: Adaptada de Liao *et al.* (2010).

Quadro 11 – Transformações e parâmetros suportados pela versão atual do compilador desenvolvido

Descrição	Parâmetros definidos pelo DSE
<i>Loop interchange</i>	<i>Yes / No</i>
<i>Loop blocking</i>	<i>None, Outermost, Innermost or All loops</i>
<i>Blocking size</i>	<i>1 to N</i>
<i>Loop fusion</i>	<i>None, Single or Multi-level</i>
<i>Loop fission</i>	<i>Yes / No</i>
<i>Loop splitting</i>	<i>Yes / No</i>
<i>Loop unrolling</i>	<i>1 to N</i>
<i>Optmizaion level</i>	<i>1 to Max depth</i>
<i>Data Cache word size</i>	<i>Num. of bytes</i>
<i>Data Cache line size</i>	<i>Num. of lines</i>
<i>Instruction Cache word size</i>	<i>Num. of bytes</i>
<i>Instruction Cache line size</i>	<i>Num. of lines</i>

Fonte: Elaborada pelo autor.

A avaliação das soluções é feita por meio de simulação da execução do código C otimizado no processador. Para isto é utilizado um simulador com precisão de ciclos, sendo gerado como resultado da simulação o número de ciclos gastos para executar o código e o número de *cache hit* e *cache miss* para as memórias *cache* de dados e instruções. Os valores de *cache hit* e *cache miss* são usados para calcular a métrica de eficiência de *cache* (E_{cache}) mostrada na Equação 7.1, onde D_s , D_h e D_m são o tamanho e o total de *hits* e *misses* da *cache* de dados, enquanto que I_s , I_h e I_m são o tamanho e total de *hits* e *misses* da *cache* de instruções, respectivamente.

$$E_{cache} = \frac{D_s}{\left(\frac{D_h}{D_h + D_m}\right)} + \frac{I_s}{\left(\frac{I_h}{I_h + I_m}\right)} \quad (7.1)$$

Ao final do processo, é gerado um conjunto de soluções ótimas equivalentes. A escolha da solução que será sintetizada no FPGA fica a critério do usuário, o qual decide qual a solução que melhor atende aos requisitos de tempo e área para uma dada aplicação.

Esta versão encontra-se em desenvolvimento no Laboratório de Computação Reconfigurável (LCR) da Universidade de São Paulo em São Carlos e servirá de base para uma ferramenta de compilação para arquiteturas reconfiguráveis, voltada a aceleração de *hardware* para aplicações de HPC. Esta ferramenta tem como finalidade a geração otimizada de *hardware* a partir de código C e de especificações escritas em alto nível. Assim como os projetos apresentados no Capítulo 3, o sistema busca possibilitar ao programador um maior nível abstração para o desenvolvimento de *hardware*.

7.3 Ferramenta de HLS com LALP+

A Figura 59 apresenta uma visão geral de uma ferramenta de HLS com LALP+. Seguindo um fluxo semelhante à versão corrente, a mesma deverá explorar o espaço de projeto e realizar otimizações no código considerando métricas e requisitos previamente definidos pelo desenvolvedor. A fim de permitir que estes critérios sejam alcançados de forma eficiente, o sistema inclui o uso de Algoritmos Genéticos (AG's) para exploração do espaço de projeto e definição das otimizações a serem aplicadas ao código-fonte de entrada.

A ferramenta é composta por três módulos principais, os quais são: Módulo de Exploração de Espaço de Projeto, Módulo Gerador de *Hardware* e Módulo de Compilação e Otimização, o qual inclui o uso de LALP+. Dado este escopo, este capítulo inclui uma breve descrição do funcionamento geral da ferramenta para fins de contextualização, enquanto que detalhes do Módulo de compilação são apresentados na Seção 7.4.

Como entrada, a ferramenta recebe o código-fonte (em C) a ser convertido em *hardware* e uma especificação de critérios de otimização e restrições, definidos em alto nível na forma

Figura 59 – Visão geral do sistema de geração de *hardware*

de aspectos. Esta especificação é submetida a um *parser* e analisador, e, após validada, será usada para exploração do espaço de projeto por meio de AGs. A cada iteração, o MDSE gera parâmetros de otimização e transformações de código que são passados ao MCO. Este, por sua vez, aplica as otimizações gerando uma descrição de *hardware* condizente com os parâmetros recebidos. Esta descrição, que inclui instruções e diretivas de mapeamento em FPGA, é enviada aos módulos MGH e MDSE. Este último avalia o *hardware* gerado com base nos aspectos especificados. Devido ao tempo necessário para gerar e mapear em *hardware* cada uma das possíveis soluções ser demasiadamente longo, uma estimativa deste mapeamento será realizada a partir de bibliotecas e modelos de *hardware* específicos da plataforma alvo. Ao final do processo, o código otimizado de acordo com os aspectos previamente definidos será mapeado em *hardware*, sendo gerado um *bitstream* para síntese no FPGA alvo.

7.4 Módulo de compilação e otimização

Nesta ferramenta o compilador (MCO) é responsável por realizar as transformações no código de entrada e gerar uma descrição do *hardware* a ser sintetizado. O compilador recebe como entrada o código-fonte escrito em C e um conjunto de parâmetros que serão usados para a aplicação de transformações de código. Após as verificações iniciais feitos pelo *front-end*, o *middle-end* realiza a análise de dependência de dados, gerando o grafo de dependência de dados, ou DDG (*Data Dependency Graph*). A partir desta análise, as transformações são aplicadas e são geradas anotações correspondentes ao código LALP+ que será processado pelo *back-end*. Como saída do compilador, tem-se um código VHDL contendo a descrição do *hardware* em nível RTL e um código específico para configuração e controle do processador *soft-core* desenvolvido.

Além das dificuldades de programação envolvidas no desenvolvimento de *hardware*, o

compilador deverá tratar, ainda, problemas referentes ao acesso aos dados. O uso das arquiteturas usadas em LALP+ para gerenciamento de acesso (FSM e MMU) auxilia nesta questão. O desenvolvimento do compilador inclui a integração com o *framework* LALP+ para tratar deste aspecto, bem como utilizar o compilador ROSE (QUINLAN; LIAO, 2011) como ferramenta para aplicação das transformações. A estrutura geral do compilador proposto é mostrada na Figura 60. Neste escopo, o MCO trata-se do compilador apresentado na seção 7.2, o qual deve ser expandido com o uso da linguagem LALP+ para a inclusão um *back-end* gerador de *hardware*.

Figura 60 – Estrutura do compilador proposto

Fonte: Elaborada pelo autor.

Propõe-se que o MCO seja desenvolvido com base na estrutura do ROSE (Figura 58). O ROSE (LIAO *et al.*, 2010; QUINLAN *et al.*, 2012) é um compilador *source-to-source* de código aberto, desenvolvido e mantido pelo *Lawrence Livermore National Laboratory*, em Livermore, Califórnia. É destinado à análise e otimização de código, possuindo versões para Linux e Mac OS X, em plataformas de 32 e 64 *bits*. A opção pela utilização do ROSE se dá não só por suas características técnicas, mas, também, por se tratar de um produto de código aberto, e por ser um projeto ativo e em constante atualização por um grande laboratório. Segundo a página oficial do projeto¹, dezenas de grupos têm mostrado interesse pelo ROSE, incluindo grupos de pesquisas, empresas e universidades, sendo o mesmo utilizado, inclusive, para desenvolvimento de aplicações comerciais. Por esta razão, acredita-se que o ciclo de vida do ROSE será bastante extenso, o que nem sempre ocorre com ferramentas de código aberto. A robustez e o potencial suporte oferecidos pelo ROSE são fatores que levaram à escolha do ROSE em detrimento de outros compiladores *source-to-source*, como o Cetus (DAVE *et al.*, 2009). Além destas características, o ROSE possui as seguintes funcionalidades:

- Análise e geração de grafos de chamadas de funções e fluxos de dados;
- Ferramentas de depuração;
- Suporte a bancos SQLite;

¹ <http://www.rosecompiler.org>

- Geração de grafos personalizados;
- Suporte à análise de código binário;
- Suporte a diversas linguagens como Fortran, PHP, Cuda, C/C++, OpenMP, etc;
- Suporte à integração com ferramentas externas;
- Diretivas para construção de linguagens;
- Suporte a paralelismo e uso de memória compartilhada;
- Interface para uso da IR SageIII com Haskell.

A Figura 61 apresenta as etapas de um compilador construído utilizando o ROSE. Primeiramente, o código original é submetido a um *parser* sendo então gerada uma representação intermediária na forma de uma Árvore Sintática Abstrata (AST – *Abstract Syntax Tree*), a partir da qual são realizadas as otimizações. Em seguida é feito o reconhecimento de estruturas, contidas na AST, próprias da linguagem processada, como *loops*, código morto, diretivas de pré-processamento, etc. A próxima etapa consiste em processar a AST, aplicando transformações e gerando uma AST otimizada. A partir da AST otimizada, o ROSE gera um código na mesma linguagem de entrada, contendo as transformações realizadas e preservando comentários e estruturas de controle do código original. O código otimizado pode ser compilado por outro compilador para geração do executável/binário final. Neste caso, propõe-se que seja utilizado o compilador LALP+ com as devidas modificações.

7.4.1 O front-end

O *front-end* utilizado pelo ROSE (EDG *Front-end*) gera uma IR gráfica na forma de AST. Esta AST, por sua vez, é traduzida automaticamente por uma ferramenta chamada ROSETTA, a qual gera a IR usada de fato no ROSE, denominada SAGE III. Tanto a ROSETTA, quanto a SAGE III são desenvolvidas pela própria equipe do ROSE. Esta tradução é feita para permitir a disponibilização, em conjunto com o código-fonte do ROSE, somente do binário do EDG *Front-end* e não seu código-fonte. Assim, os usuários do ROSE não possuem acesso ao código-fonte do *front-end*, embora seja possível obtê-lo por meio da aquisição de uma licença de pesquisa (QUINLAN *et al.*, 2012).

O desenvolvimento do MCO proposto foca-se na otimização e geração de *hardware*, que correspondem às etapas de *middle-end* e *back-end*, uma vez que a validação de código e geração da AST inicial são funcionalidades já disponibilizadas pelo ROSE. Assim, não há interesse, em princípio, em realizar alterações no código-fonte do EDG *Front-end*.

Figura 61 – Diferentes etapas de processamento de um compilador C++ construído usando as infraestrutura do ROSE

Fonte: Adaptada de [Quinlan et al. \(2012\)](#).

7.4.2 O middle-end

A etapa de análise e otimização de código realizada pelo MCO consiste na aplicação de transformações para geração de um código C otimizado, equivalente ao originalmente recebido. As otimizações serão realizadas de acordo com os parâmetros de transformações recebidos. Com isto, é possível direcionar a compilação para atender certos requisitos definidos pelo usuário.

O ROSE possui implementações de algumas transformações em nível de *loops*, sendo possível configurar o ROSE para que inclua automaticamente, em conjunto com as transformações, o reuso dos dados e técnicas que aumentem a localidade dos dados, bem como outras otimizações, como eliminação parcial de redundância e eliminação de código morto.

O conjunto de transformações cujos parâmetros serão definidos pelo MDSE é mostrado no Quadro 12, sendo estas suportadas pelo ROSE. Dadas as particularidades das transformações, os parâmetros de cada uma delas são definidos individualmente. Em várias destas transformações é possível aplicar outros parâmetros de otimização, que, em princípio, não são aplicáveis à proposta, como otimização de uso de *cache*, por exemplo.

Quadro 12 – Lista parcial de transformações suportadas pelo ROSE

Descrição	Parâmetros necessários
<i>Loop unrolling</i>	Identificador do <i>loop</i> e fator de desenrolamento
<i>Loop blocking / tiling</i>	Identificador do <i>loop</i> , nível do aninhamento, tamanho dos blocos
<i>Loop permutation / interchange</i>	Identificador do <i>loop</i> mais externo, nível do aninhamento, ordem da permutação
<i>Loop distribution</i>	Identificador do <i>loop</i>
<i>Loop fusion</i>	Identificador do <i>loop</i> , nível de reuso dos dados
<i>Loop splitting</i>	Identificador do <i>loop</i>

O ROSE fornece uma interface padrão para aplicação das otimizações suportadas, mas também permite que o utilizador crie suas próprias funções. Neste caso, porém, as devidas análises para garantir que o código transformado é equivalente ao original não são feitas automaticamente, devendo o usuário implementá-las. Uma vez que várias transformações não são suportadas pelo ROSE, será necessário implementá-las a fim de complementar o conjunto de transformações suportadas. Algumas técnicas úteis na implementação de transformações envolvendo o gerenciamento personalizado de memória são mostradas por (CATTHOOR *et al.*, 1998).

O MDSE fornecerá uma lista contendo as transformações a serem aplicadas, onde cada elemento é composto pela identificação da transformação e seus parâmetros. Uma vez realizadas as transformações, o *back-end* do ROSE utiliza a AST otimizada para gerar código final na mesma linguagem do código de entrada (neste caso é C). Contudo, é possível utilizar um outro *back-end* para geração de código em outra linguagem. No caso desta proposta, isto será feito para permitir a geração de código LALP+ a partir da AST otimizada gerada pelo ROSE.

7.4.3 O back-end

O *back-end* do MCO irá gerar uma descrição do *hardware*, em VHDL, correspondente ao código otimizado pelo *middle-end*. Os componentes de *hardware* serão gerados a partir da biblioteca usada pelo LALP+, acrescida dos componentes para gerenciamento de memória implementados pelo LALP+, bem como de novos componentes a serem potencialmente desenvolvidos. Além da descrição dos componentes em VHDL, o *back-end* deverá gerar instruções de mapeamento dos componentes e dos dados em uma ou mais memórias, de acordo com as otimizações realizadas, eliminando a necessidade do programador realizar esta definição. Em termos de geração de *software*, o *back-end* irá gerar código para integração com um processador, o qual será responsável pelo controle da aplicação.

A integração entre ROSE e LALP se dá pela a conversão de C para LALP, sendo este processo de conversão alvo de um outro trabalho (PORTO, 2015) de pesquisa desenvolvido na Universidade Federal de São Carlos, que focou em uma versão preliminar do LALP. Além disto, um processo adequado de conversão C para LALP depende da implementação de suporte em LALP a algumas características, como *loops* irregulares, chamadas de funções, etc.

7.5 Considerações finais

A ferramenta proposta neste capítulo visa auxiliar no desenvolvimento de aplicações envolvendo SoC. A proposta baseia-se em uma versão protótipo, a qual apresentou bons resultados, como mostrado em (MARTINS *et al.*, 2014), mas cujo foco está na otimização de código para um processador *softcore*. Assim, propõe-se alterações na arquitetura do sistema, de forma a incorporar etapas para a geração de *hardware*. O uso de LALP+ é uma alternativa viável para este fim, dados os resultados alcançados até o momento.

A versão proposta inclui etapas que envolvem múltiplas áreas, como otimizações de código, Algoritmos Genéticos, integração com outras ferramentas, etc. Desta forma tal sistema, além do foco de auxiliar na implementação de *hardware*, pode ser considerado, também, um ambiente de desenvolvimento que oferece oportunidades em potencial para pesquisas futuras.

CONCLUSÃO

Este documento apresentou a nova versão da linguagem LALP, denominada LALP+, que trata-se de uma extensão de LALP para desenvolvimento de aplicações com acesso paralelo aos dados. Apesar da capacidade em gerar arquiteturas eficientes, LALP impõe uma complexidade de programação que requer conhecimentos específicos de características de baixo nível, como paralelismo e sincronização de operações. Isto dificulta sua utilização, afastando desenvolvedores não habituados aos paradigmas impostos pela linguagem.

De modo geral, este projeto tem como objetivo fornecer uma linguagem que abrange uma vasta gama de possibilidades para o desenvolvimento de aceleradores baseados em FPGA. Considerando as características de LALP+ mostradas neste trabalho, tais como nível de abstração (sintaxe semelhante a C) e arquiteturas personalizadas, espera-se que os desenvolvedores LALP+ sejam capazes de lidar com questões que não são atualmente tratadas por outras abordagens de desenvolvimento com FPGA, como o uso de ferramentas de HLS, e/ou quando estas abordagens não são capazes de gerar arquiteturas com um desempenho aceitável.

Este trabalho tratou de limitações e funcionalidades não suportadas em LALP, de forma a contribuir para o aumento do nível de abstração da linguagem. Isto é feito por meio de uma simplificação da linguagem LALP, com alterações em vários níveis do compilador, tendo sido incorporados mecanismos e opções de compilação que simplificam seu uso, incluindo a automatização de etapas antes realizadas manualmente. Apesar disso, LALP+ mantém o suporte às funcionalidades LALP, permitindo um nível de flexibilidade que pode ser usada por desenvolvedores mais experientes.

LALP+ opera utilizando estruturas e componentes modificados para a criação de barreiras de sincronização, as quais facilitam o escalonamento de operações ao longo do fluxo de dados, permitindo o cálculo automático do sincronismo em determinadas situações. Foram incluídas opções para mapeamento e particionamento de dados de forma a permitir o acesso paralelo, diminuindo a latência dos circuitos gerados.

As estruturas de memórias desenvolvidas facilitam o uso de múltiplas BRAMs, o que permite que os usuários possam implementar transformações de *loop* relacionadas com memórias distribuídas com maior facilidade. No entanto, o desenvolvedor ainda deve ter em conta o número de endereços e matrizes, bem como a forma como os dados são particionados, ao definir o total de BRAMs, uma vez que estes parâmetros irão influenciar o número de ciclos por iteração e a taxa de transferência de dados. Mesmo com esse requisito, o uso de LALP+ impõe um menor esforço que o uso de HDLs para a implementação de projetos com FPGAs.

Os resultados apresentados nos encorajam a continuar melhorando a linguagem. LALP+ apresentou arquiteturas com melhor desempenho que as fornecidas por ferramentas HLS, cujo uso é igualmente simplificado, tendo utilizados recursos de *hardware* em quantidade equivalente.

Apesar dos benefícios alcançados nesta nova versão, há ainda situações onde LALP+ mostra-se ineficiente. O suporte a operações de ponto flutuante ainda é limitado, sendo providos apenas componentes para operações básicas. Outra limitação é que as estruturas para controle de acessos paralelos (FSM e MMU) não suportam operações de escrita e leituras para uma mesma matriz, sendo viáveis apenas para aplicações onde os dados são escritos em matrizes diferentes das matrizes de entrada.

Dadas as limitações presentes em LALP+, propõe-se como trabalho futuro:

Suporte completo a operações de ponto flutuante → Os resultados mostraram um melhor desempenho para os módulos criados diretamente em LALP. Por esta razão, uma opção viável é a extensão da biblioteca de operadores LALP+ para FP de forma a incluir outros operadores, a fim de proporcionar um maior suporte a aplicações que envolvem cálculos com ponto flutuante (FP). Outra possível abordagem é incluir operadores gerados pelo Flopoco, o qual também utiliza uma abordagem com *pipelining*, tornando-o compatível com a filosofia da linguagem.

Novos componentes para gerenciamento de memória → Incluir suporte a operações de escrita e leitura no mesmo vetor a fim de aumentar a quantidade de aplicações suportadas. Outra melhoria possível é a revisão da lógica e da estrutura interna desses componentes, de forma a gerar um VHDL otimizado, com menor *overhead* e melhor desempenho. É possível, ainda, que FSM e MMU sejam integrados de forma a comporem um único componente para gerenciamento de memória e/ou suportarem controle a blocos de memória externa.

Exploração de novas formas de particionamento de dados → Uma exploração detalhada das possibilidades de particionamento de dados é necessária, uma vez que novos e algoritmos para mapeamento e organização dos dados podem ajudar na geração de arquiteturas mais eficientes. A adoção de arquiteturas fixas, como o uso de invariável de n BRAMs, é outra possibilidade, mas isso limitaria a flexibilidade na exploração de arquiteturas hoje presente em LALP+.

Criação de uma API para co-projeto de hardware e software → Apesar de LALP+ gerar interfaces para integração das arquiteturas com o processador NiosII, a integração com este é feita manualmente. Assim, para facilitar o uso dos componentes criados, pretende-se desenvolver uma API que permita a programação de módulos LALP+ em conjunto com programas escritos em linguagem de alto nível. Adicionalmente, a API deverá incorporar o suporte a FIFOs e a *streaming* de dados via FIFOs, além de suporte a paralelismo a nível de tarefas, funcionalidades a serem implementadas no compilador LALP+.

Integração de LALP+ a uma ferramenta de HLS → Em conjunto com o desenvolvimento da API citada anteriormente, pretende-se integrar LALP+ a uma ferramenta de HLS, onde porções de código relativas a *loops* possam ser convertidas em blocos de *hardware* com o uso de LALP+. Esta ferramenta já teve seu desenvolvimento iniciado e permite a exploração do espaço de projeto e transformações de código envolvendo *loops* e dados.

Por fim, espera-se que os mecanismos desenvolvidos facilitem o desenvolvimento de *hardware* reconfigurável, promovendo uma maior absorção das tecnologias desenvolvidas ao longo desta pesquisa. Tecnologias desta natureza são importantes dados os desafios presentes na área de computação de alto desempenho.

REFERÊNCIAS

Agility Design Solutions Inc. Handel-C Language Reference Manual. White paper. 2007. Citado na página 61.

AHO, A.; LAM, M.; SETHI, R.; ULLMAN, J. **Compiler: Principles, Techniques and Tools**. 2. ed. [S.l.]: Pearson Addison Wesley, 2001. 1033 p. Citado 3 vezes nas páginas 44, 47 e 48.

ALTERA. **Nios II Custom Instruction - User Guide**. [S.l.], 2011. Online. Acesso em 4/10/2015. Disponível em: <https://www.altera.com/content/dam/altera-www/global/en_US/pdfs/literature/ug/ug_nios2_custom_instruction.pdf>. Citado na página 108.

_____. **Nios II Processor Reference Handbook**. [S.l.], 2011. Online. Acesso em 10/09/2012. Disponível em: <http://www.altera.com/literature/hb/nios2/n2cpu_nii5v1.pdf>. Citado na página 71.

_____. Implementing FPGA Design with the OpenCL Standard. White paper. 2013. Citado na página 60.

_____. Altera SDK for OpenCL: Getting Started Guide. 2015. Citado na página 59.

Altera Corporation. **Introducing Innovations at 28nm to Move Beyond Moore's Law**. [S.l.], 2010. 15 p. Citado na página 45.

_____. **Stratix V Device Overview**. [S.l.], 2012. 22 p. 22 p. Citado na página 45.

ANDREWS, D.; SASS, R.; ANDERSON, E.; AGRON, J.; PECK, W.; STEVENS, J.; BAIJOT, F.; KOMP, E. Achieving programming model abstractions for reconfigurable computing. **IEEE Transactions on Very Large Scale Integration (VLSI) Systems**, IEEE, v. 16, n. 1, p. 34–44, 2008. Citado na página 35.

ANTOLA, A.; SANTAMBROGIO, M.; FRACASSI, M.; GOTTI, P.; SANDIONIGI, C. A novel hardware/software codesign methodology based on dynamic reconfiguration with impulse c and codeveloper. In: **Programmable Logic, 2007. SPL '07. 2007 3rd Southern Conference on**. [S.l.: s.n.], 2007. p. 221–224. Citado na página 66.

ASHER, Y. B.; ROTEM, N. Automatic memory partitioning: increasing memory parallelism via data structure partitioning. In: IEEE. **2010 IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis**. [S.l.], 2010. p. 155–161. Citado na página 35.

BABB, J.; AGARWAL, A. **High level compilation for gate reconfigurable architectures**. Tese (Doutorado) — Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science, 2001. 215 p. Citado na página 69.

BANERJEE, P.; SHENOY, N.; CHOUDHARY, A.; HAUCK, S.; BACHMANN, C.; HALDAR, M.; JOISHA, P.; JONES, A.; KANHARE, A.; NAYAK, A. *et al.* A MATLAB compiler for distributed, heterogeneous, reconfigurable computing systems. In: IEEE. **Field-Programmable**

Custom Computing Machines, 2000 IEEE Symposium on. [S.l.], 2000. p. 39–48. Citado na página 69.

BARADARAN, N.; DINIZ, P. Memory parallelism using custom array mapping to heterogeneous storage structures. In: IEEE. **International Conference on Field Programmable Logic and Applications, 2006. FPL 2006.** [S.l.], 2006. p. 1–6. Citado na página 37.

_____. A compiler approach to managing storage and memory bandwidth in configurable architectures. **ACM Transactions on Design Automation of Electronic Systems (TODAES)**, ACM, v. 13, n. 4, p. 61:1–61:26, 2008. Citado na página 37.

BERTELS, K.; SIMA, V.; YANKOVA, Y.; KUZMANOV, G.; LUK, W.; COUTINHO, G.; FERRANDI, F.; PILATO, C.; LATTUADA, M.; SCIUTO, D. *et al.* Hartes: Hardware-software codesign for heterogeneous multicore platforms. **IEEE Micro**, IEEE, v. 30, n. 5, p. 88–97, 2010. Citado na página 34.

BOBDA, C. **Introduction to Reconfigurable Computing: Architectures, Algorithms, and Applications.** 1. ed. [S.l.]: Springer Publishing Company, Incorporated, 2007. 362 p. Citado 3 vezes nas páginas 34, 44 e 45.

BOLLAERT, T. Catapult synthesis: a practical introduction to interactive c synthesis. In: **High-Level Synthesis from Algorithm to Digital Circuit.** [S.l.]: Springer Verlag, 2008. p. 29–52. Citado na página 70.

BONDALAPATI, K.; DINIZ, P.; DUNCAN, P.; GRANACKI, J.; HALL, M.; JAIN, R.; ZIEGLER, H. DEFACTO: A design environment for adaptive computing technology. **Parallel and Distributed Processing**, Springer Berlin / Heidelberg, v. 1586, p. 570–578, 1999. Citado na página 68.

BORKAR, S.; CHIEN, A. The future of microprocessors. **Communications of the ACM**, ACM, v. 54, n. 5, p. 67–77, 2011. Citado 4 vezes nas páginas 33, 34, 35 e 36.

CADENCE. **C-to-Silicon Compiler High-Level Synthesis.** 2015. Disponível em: <http://www.cadence.com/products/sd/silicon_compiler/pages/default.aspx>. Citado na página 67.

Canis, Andrew et. al. From Software to Accelerators with LegUp High-level Synthesis. In: **Proceedings of the 2013 International Conference on Compilers, Architectures and Synthesis for Embedded Systems (CASES '13).** Piscataway, NJ, USA: IEEE Press, 2013. p. 18:1–18:9. ISBN 978-1-4799-1400-5. Disponível em: <<http://dl.acm.org/citation.cfm?id=2555729.2555747>>. Citado na página 64.

_____. LegUp: An Open-source High-level Synthesis Tool for FPGA-based Processor/Accelerator Systems. **ACM Trans. Embed. Comput. Syst.**, ACM, New York, NY, USA, v. 13, n. 2, p. 24:1–24:27, set. 2013. ISSN 1539-9087. Disponível em: <<http://doi.acm.org/10.1145/2514740>>. Citado na página 64.

CARDOSO, J.; NETO, H. Macro-based hardware compilation of javatm bytecodes into a dynamic reconfigurable computing system. In: IEEE. **Field-Programmable Custom Computing Machines, 1999. FCCM'99. Proceedings. 7th Annual IEEE Symposium on.** [S.l.], 1999. p. 2–11. Citado na página 69.

_____. Compilation for fpga-based reconfigurable hardware. **IEEE Design & Test of Computers**, v. 20, n. 2, p. 65–75, Mar 2003. ISSN 0740-7475. Citado na página 80.

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. 250 p. Citado 2 vezes nas páginas 69 e 80.

CARDOSO, J. M. P. Dynamic loop pipelining in data-driven architectures. In: **ACM. Proceedings of the 2nd conference on Computing frontiers**. [S.l.], 2005. p. 106–115. Citado na página 79.

CARDOSO, J. M. P.; DINIZ, P. C. **Compilation Techniques for Reconfigurable Architectures**. [S.l.]: Springer, 2009. 230 p. Citado 5 vezes nas páginas 46, 47, 49, 68 e 69.

CARDOSO, J. M. P.; DINIZ, P. C.; WEINHARDT, M. Compiling for reconfigurable computing: A survey. **ACM Computing Surveys (CSUR)**, ACM, v. 42, n. 4, p. 13:1–13:65, 2010. Citado 2 vezes nas páginas 37 e 49.

CARDOSO, J. M. P.; HÜBNER, M. **Reconfigurable Computing: From FPGAs to Hardware/Software Codesign**. [S.l.]: Springer Verlag, 2011. 296 p. Citado 3 vezes nas páginas 34, 49 e 73.

CATANZARO, B.; FOX, A.; KEUTZER, K.; PATTERSON, D.; SU, B.; SNIR, M.; OLUKOTUN, K.; HANRAHAN, P.; CHAFI, H. Ubiquitous parallel computing from berkeley, illinois, and stanford. **IEEE Micro**, IEEE, v. 30, n. 2, p. 41–55, 2010. Citado na página 33.

CATTHOOR, F.; DEGREEF, E.; WUYTACK, S.; BALASA, F.; NACHTERGAELE, L. **Custom Memory Management Methodology: Exploration of Memory Organisation for Embedded Multimedia System Design**. [S.l.]: Kluwer Academic Publishers, Boston, 1998. 364 p. Citado na página 145.

CHUGH, M.; PARHAMI, B. Logarithmic arithmetic as an alternative to floating-point: A review. In: **Signals, Systems and Computers, 2013 Asilomar Conference on**. [S.l.: s.n.], 2013. p. 1139–1143. Citado na página 116.

CONSORTIUM, T. O. OpenSPL: Revealing the Power of Spatial Computing. White Paper, The OpenSPL Consortium. 2013. Citado na página 59.

COOPER, K.; TORCZON, L. **Engineering a compiler**. 2. ed. [S.l.]: Morgan Kaufmann, 2011. 824 p. Citado 5 vezes nas páginas 46, 47, 48, 51 e 77.

COUTINHO, J.; JIANG, J.; LUK, W. Interleaving behavioral and cycle-accurate descriptions for reconfigurable hardware compilation. In: **Field-Programmable Custom Computing Machines, 2005. FCCM 2005. 13th Annual IEEE Symposium on**. [S.l.: s.n.], 2005. p. 245–254. Citado 3 vezes nas páginas 55, 62 e 63.

DAVE, C.; BAE, H.; MIN, S.; LEE, S.; EIGENMANN, R.; MIDKIFF, S. Cetus: A source-to-source compiler infrastructure for multicores. **Computer**, IEEE Computer Society, v. 42, n. 12, p. 36–42, 2009. Citado 3 vezes nas páginas 60, 90 e 142.

DINECHIN, F. de; PASCA, B. Designing custom arithmetic data paths with FloPoCo. **IEEE Design & Test of Computers**, v. 28, n. 4, p. 18–27, jul. 2011. Citado na página 116.

DINIZ, P.; HALL, M.; PARK, J.; SO, B.; ZIEGLER, H. Automatic mapping of C to FPGAs with the DEFECTO compilation and synthesis system. **Microprocessors and Microsystems**, v. 29, n. 2–3, p. 51 – 62, 2005. Citado na página 68.

DINIZ, P. C. **Configurable Techniques for Reconfigurable Computing**. 2011. Notas de aula. 278 Slides. Online. Acesso em 15/05/2011. Disponível em: <<http://www.icmc.usp.br/~lcr/en/ICMC-USP-PedroDiniz.Final.pdf>>. Citado na página 46.

DREPPER, U. **What Every Programmer Should Know About Memory**. [S.l.], 2007. 114 p. Disponível em: <<http://people.redhat.com/drepper/cpumemory.pdf>>. Citado na página 35.

DUBEY, R. **Introduction to embedded system design using field programmable gate arrays**. 2. ed. [S.l.]: Springer Verlag, 2009. 158 p. Citado 2 vezes nas páginas 43 e 45.

FOWLER, M. **Domain Specific Languages**. 1. ed. [S.l.]: Addison-Wesley Professional, 2010. 640 p. Citado na página 54.

FU, H.; GAN, L.; CLAPP, R.; RUAN, H.; PELL, O.; MENCER, O.; FLYNN, M.; HUANG, X.; YANG, G. Scaling Reverse Time Migration Performance through Reconfigurable Dataflow Engines. **Micro, IEEE**, v. 34, n. 1, p. 30–40, Jan 2014. ISSN 0272-1732. Citado na página 58.

GAJSKI, D.; ABDI, S.; GERSTLAUER, A.; SCHIRNER, G. **Embedded System Design: Modeling, Synthesis and Verification**. [S.l.]: Springer Publishing Company, Incorporated, 2009. 358 p. Citado 2 vezes nas páginas 46 e 52.

GOKHALE, M.; GRAHAM, P. S. **Reconfigurable Computing Accelerating Computation with Field-Programmable Gate Arrays**. 1. ed. Dordrecht, Netherlands: Springer, 2005. 238 p. Citado na página 44.

GRÖTKER, T.; LIAO, S.; MARTIN, G.; SWAN, S. **System design with SystemC**. [S.l.]: Springer, 2002. 240 p. Citado 2 vezes nas páginas 63 e 72.

GROUP, K. O. W. **The OpenCL Specification**. [S.l.], 2014. Citado na página 59.

GRUNE, D.; BAL, H.; JACOBS, C.; LANGENDOEN, K. **Projeto Moderno de Compiladores: Implementação e Aplicações**. [S.l.]: Editora Campus, 2001. 684 p. Citado na página 46.

GUPTA, R.; BREWER, F. High-level synthesis: A retrospective. In: **High-Level Synthesis from Algorithm to Digital Circuit**. [S.l.]: Springer Verlag, 2008. p. 13–28. Citado na página 52.

GUPTA, S.; DUTT, N.; GUPTA, R.; NICOLAU, A. Spark: A high-level synthesis framework for applying parallelizing compiler transformations. In: IEEE. **Proceedings of 16th International Conference on VLSI Design**. [S.l.], 2003. p. 461–466. Citado na página 70.

HAGLUND, P.; MENCER, O.; LUK, W.; TAI, B. Hardware Design with a Scripting Language. In: CHEUNG, P. Y. K.; CONSTANTINIDES, G. (Ed.). **Field Programmable Logic and Application**. [S.l.]: Springer Berlin Heidelberg, 2003, (Lecture Notes in Computer Science, v. 2778). p. 1040–1043. ISBN 978-3-540-40822-2. Citado na página 57.

HALSTEAD, R.; VILLARREAL, J.; MOUSSALLI, R.; NAJJAR, W. Is there a tradeoff between programmability and performance? In: IEEE. **Signals, Systems and Computers (ASILOMAR), 2010 Conference Record of the Forty Fourth Asilomar Conference on**. [S.l.], 2010. p. 836–840. Citado na página 35.

HAUCK, S.; DEHON, A. **Reconfigurable Computing: the Theory and Practice of FPGA-Based Computation**. 1. ed. Burlington, United States: Elsevier, 2008. 944 p. Citado na página 44.

HOPCROFT, J. E.; MOTWANI, R.; ULLMAN, J. D. **Introduction to Automata Theory, Languages, and Computation**. 2. ed. [S.l.]: Addison Wesley, 2001. 535 p. Citado na página 46.

HSIUNG, P.-A.; SANTAMBROGIO, M. D.; HUANG, C.-H. **Reconfigurable System Design and Verification**. 1. ed. London, New York: Taylor & Francis Group, CRC Press, 2009. 268 p. Citado na página 44.

HUONG, G.; KIM, S. Gcc2verilog compiler toolset for complete translation of c programming language into verilog hdl. **ETRI Journal**, v. 33, n. 5, p. 731–740, 2011. Citado 2 vezes nas páginas 56 e 70.

IEEE. **IEEE Standard for Binary Floating-Point Arithmetic, ANSI/IEEE Std 754-1985**. 1985. Citado 3 vezes nas páginas 50, 92 e 111.

IEEE Comp. Society. IEEE Standard for Standard SystemC Language Reference Manual. IEEE Std 1666-2011. 2012. Citado na página 63.

Impulse Accelerated Technologies. [online]:**Impulse CoDeveloper C-to-FPGA Tools**. 2015. Disponível em: <http://www.impulseaccelerated.com/products_universal.htm>. Citado na página 66.

KATHAIL, V.; ADITYA, S.; SCHREIBER, R.; RAU, B. R.; CRONQUIST, D.; SIVARAMAN, M. PICO: automatically designing custom computers. **Computer**, IEEE, v. 35, n. 9, p. 39–47, 2002. Citado na página 71.

LATTNER, C.; ADVE, V. Llvm: A compilation framework for lifelong program analysis & transformation. In: **Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)**. Palo Alto, California: [s.n.], 2004. Citado na página 60.

LEUPERS, R. **Code Optimization Techniques for Embedded Processors: Methods, Algorithms, and Tools**. [S.l.]: Kluwer Academic Publishers, 2000. 224 p. Citado na página 49.

LIAO, C.; QUINLAN, D.; PANAS, T.; SUPINSKI, B. de. A ROSE-based OpenMP 3.0 research compiler supporting multiple runtime libraries. **Beyond Loop Level Parallelism in OpenMP: Accelerators, Tasking and More**, Springer, v. 6132, p. 15–28, 2010. Citado 2 vezes nas páginas 139 e 142.

LIU, Q.; CONSTANTINIDES, G.; MASSELOS, K.; CHEUNG, P. Combining data reuse with data-level parallelization for fpga-targeted hardware compilation: a geometric programming framework. **IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems**, IEEE, v. 28, n. 3, p. 305–315, 2009. Citado na página 51.

MARTINS, L. G. A.; OLIVEIRA, C. B. de; PEREIRA, E.; MARQUES, E. A Design Space Exploration tool for combine code transformations and cache configuration for a open-source softcore processor. In: **X Jornadas sobre Sistemas Reconfiguráveis**. [S.l.: s.n.], 2014. Citado 2 vezes nas páginas 137 e 146.

MARUYAMA, T.; HOSHINO, T. A C to HDL compiler for pipeline processing on FPGAs. In: IEEE. **Field-Programmable Custom Computing Machines, 2000 IEEE Symposium on**. [S.l.], 2000. p. 101–110. Citado na página 67.

MARWEDEL, P. **Embedded System Design: Embedded Systems Foundations of Cyber-Physical Systems**. [S.l.]: Springer Verlag, 2010. 400 p. Citado na página 43.

MCKEE, S. Reflections on the memory wall. In: ACM. **Proceedings of the 1st conference on Computing frontiers**. [S.l.], 2004. p. 162. Citado na página 35.

MEHTA, G.; JONES, A. An architectural space exploration tool for domain specific reconfigurable computing. In: IEEE. **IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum**. [S.l.], 2010. p. 1–8. Citado na página 37.

MENCER, O. Asc: a stream compiler for computing with fpgas. **Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on**, v. 25, n. 9, p. 1603–1617, Sept 2006. ISSN 0278-0070. Citado 2 vezes nas páginas 57 e 58.

MENCER, O.; HUBERT, H.; MORF, M.; FLYNN, M. Stream: object-oriented programming of stream architectures using pam-blox. In: **Field-Programmable Custom Computing Machines, 2000 IEEE Symposium on**. [S.l.: s.n.], 2000. p. 309–310. Citado na página 56.

MENCER, O.; MORF, M.; FLYNN, M. Pam-blox: high performance fpga design for adaptive computing. In: **FPGAs for Custom Computing Machines, 1998. Proceedings. IEEE Symposium on**. [S.l.: s.n.], 1998. p. 167–174. Citado 2 vezes nas páginas 56 e 57.

MENCER, O.; PLATZNER, M.; MORF, M.; FLYNN, M. Object-oriented domain specific compilers for programming fpgas. **Very Large Scale Integration (VLSI) Systems, IEEE Transactions on**, v. 9, n. 1, p. 205–210, Feb 2001. ISSN 1063-8210. Citado na página 56.

MENOTTI, R. **LALP: uma linguagem para exploração do paralelismo de loops em computação reconfigurável**. Tese (Doutorado) — Universidade de São Paulo, 2010. 161 p. Citado 7 vezes nas páginas 34, 71, 77, 78, 79, 80 e 86.

MENOTTI, R.; CARDOSO, J. M. P.; FERNANDES, M.; MARQUES, E. Lalp: A language to program custom fpga-based acceleration engines. **International Journal of Parallel Programming**, Springer Netherlands, v. 40, p. 262–289, 2012. ISSN 0885-7458. Citado 5 vezes nas páginas 37, 39, 77, 78 e 79.

MENOTTI, R.; MARQUES, E.; CARDOSO, J. M. P. Aggressive loop pipelining for reconfigurable architectures. In: IEEE. **International Conference on Field Programmable Logic and Applications, 2007. FPL 2007**. [S.l.], 2007. p. 501–502. Citado na página 79.

MOHAPI, L.; WINBERG, S.; INGGS, M. A domain-specific language to facilitate software defined radio parallel executable patterns deployment on heterogeneous architectures. In: **Performance Computing and Communications Conference (IPCCC), 2014 IEEE International**. [S.l.: s.n.], 2014. p. 1–8. Citado na página 54.

MUCHNICK, S. S. **Advanced compiler design and implementation**. [S.l.]: Morgan Kaufmann, 1997. 856 p. Citado na página 37.

NAJJAR, W.; BOHM, W.; DRAPER, B.; HAMMES, J.; RINKER, R.; BEVERIDGE, J.; CHAWATHE, M.; ROSS, C. High-level language abstraction for reconfigurable computing. **Computer**, v. 36, n. 8, p. 63–69, Aug 2003. ISSN 0018-9162. Citado 2 vezes nas páginas 62 e 64.

OLIVEIRA, C. B. de; CARDOSO, J. M. P.; MARQUES, E. LALP Extensions for Supporting Floating-point Operations. In: **Workshop on Research Projects Focusing on High Performance Computing - HPCW / FPL 2013**. [S.l.: s.n.], 2013. Citado na página 115.

_____. Comparando a geração de hardware a partir de uma Linguagem de Domínio Específico à uma abordagem de Síntese de Alto Nível a partir de C. In: **X Jornadas sobre Sistemas Reconfiguráveis**. [S.l.: s.n.], 2014. Citado 2 vezes nas páginas 115 e 138.

OLIVEIRA, C. B. de; MENOTTI, R.; CARDOSO, J. M. P.; MARQUES, E. A Special-Purpose Language for Implementing Pipelined FPGA-based Accelerators. In: IEEE. **Forum on specification & Design Languages, FDL 2015**. [S.l.], 2015. Citado 2 vezes nas páginas 39 e 115.

OLIVEIRA, C. Bacelar de; CARDOSO, J.; MARQUES, E. High-level synthesis from c vs. a dsl-based approach. In: **Parallel Distributed Processing Symposium Workshops (IPDPSW), 2014 IEEE International**. [S.l.: s.n.], 2014. p. 257–262. Citado na página 115.

OWAIDA, M.; BELLAS, N.; DALOUKAS, K.; ANTONOPOULOS, C. Synthesis of platform architectures from opencl programs. In: **Field-Programmable Custom Computing Machines (FCCM), 2011 IEEE 19th Annual International Symposium on**. [S.l.: s.n.], 2011. p. 186–193. Citado na página 60.

PAPAKONSTANTINO, A.; GURURAJ, K.; STRATTON, J. A.; CHEN, D.; CONG, J.; HWU, W.-M. W. High-performance cuda kernel execution on fpgas. In: **Proceedings of the 23rd International Conference on Supercomputing**. New York, NY, USA: ACM, 2009. (ICS '09), p. 515–516. ISBN 978-1-60558-498-0. Disponível em: <<http://doi.acm.org/10.1145/1542275.1542357>>. Citado 2 vezes nas páginas 60 e 61.

_____. Efficient compilation of cuda kernels for high-performance computing on fpgas. **ACM Trans. Embed. Comput. Syst.**, ACM, New York, NY, USA, v. 13, n. 2, p. 25:1–25:26, set. 2013. ISSN 1539-9087. Disponível em: <<http://doi.acm.org/10.1145/2514641.2514652>>. Citado na página 60.

PASRICHA, S.; DUTT, N. **On-Chip Communication Architectures: System on chip interconnect**. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2008. 522 p. Citado na página 45.

PATTERSON D. A.; HENESSY, J. L. **Computer Organization and Design: The Hardware/Software Interface**. 4. ed. [S.l.]: Morgan Kaufmann, 2009. 912 p. Citado na página 35.

PELL, O.; BOWER, J.; DIMOND, R.; MENCER, O.; FLYNN, M. J. Finite-difference wave propagation modeling on special-purpose dataflow machines. **Parallel and Distributed Systems, IEEE Transactions on**, v. 24, n. 5, p. 906–915, May 2013. ISSN 1045-9219. Citado 2 vezes nas páginas 58 e 59.

PEREIRA, E. da S. **Projeto de um processador open source em Bluespec baseado no processador soft-core Nios II da Altera**. 88 p. Dissertação (Mestrado) — Instituto de Ciências Matemáticas e de Computação, Universidade de São Paulo, 2014. Acesso em: 2015-10-22. Disponível em: <<http://www.teses.usp.br/teses/disponiveis/55/55134/tde-25092014-094648/>>. Citado na página 137.

PETROV, Z.; DINIZ, P. C. Fp7 reflect project: Rendering fpgas to multi-core embedded computing. **HiPEAC - European Network of Excellence on High Performance and Embedded Architecture and Compilation**, v. 22, p. 10–11, 2010. Citado 2 vezes nas páginas 73 e 74.

PORTO, L. F. **LALPC: Uma ferramenta para compilação de programas em C para exploração de paralelismo de loops em FPGAs**. 88 p. Dissertação (Mestrado) — Universidade Federal de São Carlos, 2015. Citado 2 vezes nas páginas 90 e 146.

PORTO, L. F.; FERNANDES, M. M.; BONATO, V.; MENOTTI, R. Lalpc: Exploiting parallelism from fpgas using c language. **Journal of Physics: Conference Series**, v. 649, n. 1, p. 012001, 2015. Disponível em: <<http://stacks.iop.org/1742-6596/649/i=1/a=012001>>. Citado na página 90.

PUTNAM, A.; BENNETT, D.; DELLINGER, E.; MASON, J.; SUNDARARAJAN, P.; EGGERS, S. Chimps: A c-level compilation flow for hybrid cpu-fpga architectures. In: **Field Programmable Logic and Applications, 2008. FPL 2008. International Conference on**. [S.l.: s.n.], 2008. p. 173–178. Citado na página 72.

QUINLAN, D.; LIAO, C. The rose source-to-source compiler infrastructure. In: **Cetus Users and Compiler Infrastructure Workshop, in conjunction with PACT 2011**. [S.l.: s.n.], 2011. p. 1–3. Citado 2 vezes nas páginas 90 e 142.

QUINLAN, D.; LIAO, C.; PANAS, T.; MATZKE, R.; SCHORDAN, M.; VUDUC, R.; YI, Q. **ROSE User Manual: A Tool for Building Source-to-Source Translators**. [S.l.], 2012. 349 p. 349 p. Citado 3 vezes nas páginas 142, 143 e 144.

RETTORE, P. H. L. **Geração automática de código LALP para implementação de aceleradores em FPGAs**. Dissertação (Mestrado) — Universidade Federal de São Carlos, 2012. Citado na página 90.

ROSA, J. L. G. **Linguagens Formais e Autômatos**. [S.l.]: Editora LTC. Rio de Janeiro, 2010. 164 p. Citado na página 47.

ROTEM, N. [Online]: <http://www.c-to-verilog.com/>. 2010. Citado na página 39.

SIPSER, M. **Introdução à teoria da computação**. 2. ed. [S.l.]: Thomson Learning, 2007. 488 p. Citado na página 47.

SMITH, K.; WANG, A.; FUJINO, L. Through the looking glass: Trend tracking for ISSCC 2012. **IEEE Solid-State Circuits Magazine**, IEEE, v. 4, n. 1, p. 4–20, 2012. Citado na página 34.

SO, B.; HALL, M.; DINIZ, P. A compiler approach to fast hardware design space exploration in fpga-based systems. In: **ACM. ACM SIGPLAN Notices**. [S.l.], 2002. v. 37, n. 5, p. 165–176. Citado na página 69.

STONE, J. E.; GOHARA, D.; SHI, G. Opencl: A parallel programming standard for heterogeneous computing systems. **IEEE Des. Test**, IEEE Computer Society Press, Los Alamitos, CA, USA, v. 12, n. 3, p. 66–73, maio 2010. ISSN 0740-7475. Disponível em: <<http://dx.doi.org/10.1109/MCSE.2010.69>>. Citado na página 59.

TESSIER, R.; POCEK, K.; DEHON, A. Reconfigurable computing architectures. **Proceedings of the IEEE**, v. 103, n. 3, p. 332–354, March 2015. ISSN 0018-9219. Citado na página 35.

TORONTO, U. of. [S.l.], 2015. Online. Acesso em 22/10/2015. Disponível em: <<http://legup.eecg.utoronto.ca/docs/4.0/legup-4.0-doc.pdf>>. Citado na página 65.

TUCKER A. B.; NOONAM, R. E. **Programming Languages: Principles and Paradigms**. 2. ed. [S.l.]: McGraw-Hill, 2007. 624 p. Citado na página 73.

- VAREJÃO, F. **Linguagens de programação: Conceitos e técnicas**. [S.l.]: Editora Campus, Elsevier, 2004. 335 p. Citado na página 44.
- VILLARREAL, J.; PARK, A.; NAJJAR, W.; HALSTEAD, R. Designing modular hardware accelerators in c with roccc 2.0. In: IEEE. **18th IEEE Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)**. [S.l.], 2010. p. 127–134. Citado 2 vezes nas páginas 39 e 68.
- VOELTER, M. **DSL Engineering: Designing, Implementing and Using Domain-Specific Languages**. 1. ed. [S.l.]: CreateSpace Independent Publishing Platform, 2013. 558 p. Citado 2 vezes nas páginas 53 e 54.
- WEHMEYER, L.; MARWEDEL, P. **Fast, Efficient and Predictable Memory Accesses: Optimization Algorithms for Memory Architecture Aware Compilation**. 1. ed. [S.l.]: Springer, 2006. 270 p. Citado na página 36.
- WEINHARDT, M. Portable pipeline synthesis for fccms. **Field-Programmable Logic Smart Applications, New Paradigms and Compilers**, Springer, p. 1–13, 1996. Citado na página 68.
- WEINHAUDT, M.; LUK, W. Memory access optimisation for reconfigurable systems. In: IET. **Computers and Digital Techniques, IEE Proceedings**. [S.l.], 2001. v. 148, n. 3, p. 105–112. Citado na página 68.
- WINDH, S.; MA, X.; HALSTEAD, R.; BUDHKAR, P.; LUNA, Z.; HUSSAINI, O.; NAJJAR, W. High-level language tools for reconfigurable computing. **Proceedings of the IEEE**, v. 103, n. 3, p. 390–408, March 2015. ISSN 0018-9219. Citado na página 72.
- WIRBEL, L. Xilinx SDAccel: A Unified Development Environment for Tomorrow's Data Center. 2014. Citado na página 59.
- WOLF, W. **Computers as components: principles of embedded computing system design**. [S.l.]: Morgan Kaufmann, 2008. 544 p. Citado na página 43.
- XILINX. **MicroBlaze Processor Reference Guide**. 2014. Disponível em: <http://www.xilinx.com/support/documentation/sw_manuals/xilinx2014_2/ug984-vivado-microblaze-ref.pdf>. Acesso em: 22/05/2015. Citado na página 72.
- XU, J.; SUBRAMANIAN, N.; ALESSIO, A.; HAUCK, S. Impulse c vs. vhdl for accelerating tomographic reconstruction. In: **Field-Programmable Custom Computing Machines (FCCM), 2010 18th IEEE Annual International Symposium on**. [S.l.: s.n.], 2010. p. 171–174. Citado na página 66.
- ZHANG, Z.; FAN, Y.; JIANG, W.; HAN, G.; YANG, C.; CONG, J. AutoPilot: A Platform-Based ESL Synthesis System. In: COUSSY, P.; MORAWIEC, A. (Ed.). **High-Level Synthesis**. [S.l.]: Springer Netherlands, 2008. p. 99–112. Citado na página 61.

A

CÓDIGOS PARA MULTIPLICAÇÃO DE MATRIZES EM LALP, LALP+ E C

Código-fonte 15: Multiplicação de matrizes em LALP

```

1 const DATA_WIDTH = 32;
2 const SIZE = 25;
3
4 typedef fixed(DATA_WIDTH, 0) int;
5
6 matmult(in bit init, out bit done, out int result)
7 {
8     {
9         int a[SIZE] = {
10             1, 2, 3, 4, 6,
11             6, 1, 5, 3, 8,
12             2, 6, 4, 9, 9,
13             1, 3, 8, 3, 4,
14             5, 7, 8, 2, 5
15         };
16
17         int b[SIZE] = {
18             3, 5, 0, 8, 7,
19             2, 2, 4, 8, 3,
20             0, 2, 5, 1, 2,
21             1, 4, 0, 5, 1,
22             3, 4, 8, 2, 3
23         };
24
25         int s[SIZE];
26         int sum, sum1;
27         fixed(5,0) i, j, k, l, m, l_plus_20, m_plus_5, reg;

```

```
28     fixed(5,0) addra, addrb, addrs;
29 }
30
31 counter (m=0; m<SIZE; m+=5@30);
32 m.clk_en = init;
33
34 counter(k=0; k < SIZE; k++@6);
35 k.clk_en =  init;
36
37     counter (j=1; j<l_plus_20; j+=5);
38 j.clk_en =init;
39 j.load = k.step;
40
41 l= reg when k.step;
42
43 reg = l>3 ? 0: l + 1;
44
45 l_plus_20 = l + 20;
46
47 counter (i=m; i< m_plus_5; i++);
48 i.clk_en = init;
49 i.load= k.step;
50
51 m_plus_5 = m + 5 when m.step;
52
53 addra = i;
54 addrb = j;
55 addrs = k;
56
57 a.address = addra;
58 b.address = addrb;
59 s.address = addrs@4;
60
61 sum1 = (i.step@5) ? sum1 + sum : 0 when init;
62 sum = a * b when i.step@4;
63 s.data_in = sum1 when !i.step@5;
64
65 result = s.data_out ;
66 }
```

Código-fonte 16: Multiplicação de matrizes em LALP+

```

1 const N = 5;
2 typedef unsigned(32) int;
3
4 matmult(in bit init, out bit done, out int result)
5 {
6     {
7         int a[SIZE] = {
8             1, 2, 3, 4, 6,
9             6, 1, 5, 3, 8,
10            2, 6, 4, 9, 9,
11            1, 3, 8, 3, 4,
12            5, 7, 8, 2, 5
13        };
14
15        int b[SIZE] = {
16            3, 5, 0, 8, 7,
17            2, 2, 4, 8, 3,
18            0, 2, 5, 1, 2,
19            1, 4, 0, 5, 1,
20            3, 4, 8, 2, 3
21        };
22
23        int c[N][N]= {0};
24        int acc = 0;
25        int prod = 0;
26        unsigned(8) i,j,k;
27        bit done_ok = 1;
28        int a_ik, b_kj;
29    }
30
31    counter[(i=0; i<N; i+=1)
32            (j=0; j<N; j+=1)
33            (k=0; k<=N; k+=1)];
34
35    a_ik = a[i][k];
36    b_kj = b[k][j];
37
38    prod = (a_ik * b_kj);
39    acc = ((k==N)? 0: (prod + acc)) when init@6;
40
41    c[i][j] = acc;
42
43    result = acc;
44 }

```

Código-fonte 17: Multiplicação de matrizes em C

```
1 #define    N    5
2
3 int a[N][N] =
4 {{1, 2, 3, 4, 6},
5 {6, 1, 5, 3, 8},
6 {2, 6, 4, 9, 9},
7 {1, 3, 8, 3, 4},
8 {5, 7, 8, 2, 5}};
9
10 int b[N][N] =
11 {{3, 5, 0, 8, 7},
12 {2, 2, 4, 8, 3},
13 {0, 2, 5, 1, 2},
14 {1, 4, 0, 5, 1},
15 {3, 4, 8, 2, 3}};
16
17 int c[N][N];
18
19 int matmult(void)
20 {
21     int i,j,k;
22
23     for(i=0; i<N; i++)
24     {
25         for(j=0; j<N; j++)
26         {
27             c[i][j] = 0;
28
29             for(k=0; k<N; k++)
30             {
31                 c[i][j] += a[i][k] * b[k][j];
32             }
33         }
34     }
35
36     return 0;
37 }
```