UNIVERSIDADE FEDERAL DE MINAS GERAIS
Instituto de Ciências Exatas
Programa de Pós-Graduação em Ciência da Computação

Pedro Henrique Moreira Caldeira

Geração de código para arquiteturas reconfiguráveis

Belo Horizonte
2019

Pedro Henrique Moreira Caldeira

Geração de código para arquiteturas reconfiguráveis

Versão Final

Dissertação apresentada ao Programa de Pós-Graduação em Ciência da Computação da Universidade Federal de Minas Gerais, como requisito parcial à obtenção do título de Mestre em Ciência da Computação.

Orientador: Renato Antônio Celso Ferreira

Belo Horizonte
2019

© 2019, Pedro Henrique Moreira Caldeira.
Todos os direitos reservados

Caldeira, Pedro Henrique Moreira.

C146g Geração de código para arquiteturas reconfiguráveis [recurso eletrônico] / Pedro Henrique Moreira Caldeira –2019.
1 recurso online (76 f. il, color.) : pdf.

Orientador: Renato Antônio Celso Ferreira.

Dissertação (mestrado) - Universidade Federal de Minas Gerais, Instituto de Ciências Exatas, Departamento de Ciências da Computação.
Referências: f.49-52

1. Computação – Teses. 2. Arquitetura de computador – Teses. 3. Compiladores (Programas de computador) – Teses. 4. Linguagem de programação (Computadores) – Teses.I. Ferreira, Renato Antônio Celso II. Universidade Federal de Minas Gerais, Instituto de Ciências Exatas, Departamento de Computação. III.Título.

CDU 519.6*22(043)

Ficha catalográfica elaborada pela bibliotecária Belkiz Inez Rezende Costa
CRB 6/1510 Universidade Federal de Minas Gerais - ICEx

UNIVERSIDADE FEDERAL DE MINAS GERAIS
INSTITUTO DE CIÊNCIAS EXATAS
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

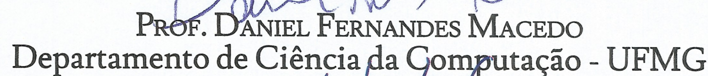
FOLHA DE APROVAÇÃO

GERAÇÃO DE CÓDIGO PARA ARQUITETURAS RECONFIGURÁVEIS

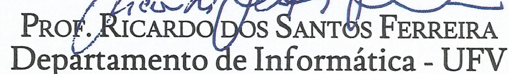
PEDRO HENRIQUE MOREIRA CALDEIRA

Dissertação defendida e aprovada pela banca examinadora constituída pelos Senhores:

Prof. Renato Antônio Celso Ferreira - Orientador
Departamento de Ciência da Computação - UFMG



Prof. Daniel Fernandes Macedo
Departamento de Ciência da Computação - UFMG



Prof. Ricardo dos Santos Ferreira
Departamento de Informática - UFV

Prof. Gilberto Medeiros Ribeiro
Departamento de Ciência da Computação - UFMG

Belo Horizonte, 11 de Março de 2019.

Agradecimentos

Agradeço em primeiro lugar toda mística que me permitiu e me auxiliou realizar o meu trabalho. Agradeço a Deus, Nossa Senhora Aparecida e a São Judas Tadeu.

Dedico o trabalho a minha mãe, Perpétua, pela incansável insistência na minha formação. Agradeço ao meu pai, Antônio, pelo abrigo e por jamais me deixar passar qualquer necessidade.

Agradeço profundamente ao meu Professor Renato Ferreira por sempre apostar em mim e que abriu todas as portas do meu caminho profissional. Gratidão também ao Professor Fernando Quintão pela paciência e toda a imensurável contribuição pra esse trabalho.

Dedico e agradeço a meu grande amor, Priscilla Passos, por tanto carinho, paciência e dedicação. Que essa conquista seja mais uma entre inúmeras que a vida irá nos proporcionar.

Aos meus mais valiosos amigos, Matheus Tymburibá, Stephanie Alves e Paulo Neto que me deram absoluto apoio, noite e dia. Obrigado por serem fiéis aliados, essa conquista é nossa.

Toda realidade foi sonho um dia. Agradeço a esse delírio coletivo de acreditar que um dia eu poderia ser mestre. Em especial ao Esdras Frederico e ao Vinícius Rocha por toda essa motivação particular.

Com muita honra ofereço esse trabalho ao Laboratório de Compiladores da UFMG e toda a sua equipe: Angélica, Andrei, Breno, Bruno, Caio, Carina, Gabriel, Gleison, Guilherme, Junio, Marcus, Marcelo, Pedro, Rafael, Rubens e especialmente a Tarsila que foi fundamental na execução deste trabalho.

Por fim, agradeço a todos meus amigos e colegas que, apesar de não nominados, foram importantes na minha vida, não somente nesse momento. Agradeço a todos meus mestres e as pessoas que anonimamente contribuíram pra que eu estivesse aqui. Muito obrigado!

“*It is victory, victory at all costs,*
victory in spite of all terror,
victory, however long and hard the road may be
for without victory, there is no survival.”
(Winston Churchill)

Resumo

Nos últimos anos observa-se uma ascenção na popularidade das Matrizes de Portas Programáveis em Campo (Field Programmable Gate Array - FPGA). Programadores podem utilizá-las para desenvolver aplicações de alto desempenho que podem ser eficientes em tempo e energia.

Porém, programar para FPGAs permanece uma tarefa difícil. Apesar de existirem interfaces OpenCL para sintetizar esse hardware, linguagens de alto nível como Java, C# ou Python permanecem distantes dessa arquitetura.

Nesse trabalho descrevemos um compilador e um ambiente de execução que reduz essa distância traduzindo código funcional escrito em Java para a plataforma do Intel HARP. Portanto trazemos duas contribuições. Primeiro, mostramos que uma biblioteca de programação funcional é um bom ponto de partida para aproximar linguagens de alto nível e FPGAs. Segundo, a implementação de um arcabouço que inclui um compilador, uma representação intermediária e um ambiente de execução capaz de fazer a transferência de dados entre hospedeiro e acelerador sem intervenção explícita do programador.

Afim de demonstrar a eficácia do nosso sistema, nós o utilizamos para implementar diferentes casos de testes utilizados em processamento de imagens e mineração de dados. Para entradas com grande volume de dados, observamos acelerações estáveis de 20x se comparando com o código original executado na Máquina Virtual Java entre todos nossos casos de teste. Dependendo da função que compilamos essa aceleração pode chegar a 280x.

Palavras-chave: Compiladores. Linguagens de Programação. Arquitetura de Computadores.

Abstract

Recent years have seen a surge in the popularity of Field-Programmable Gate Arrays (FPGAs). Programmers can use them to develop high-performance systems that are not only efficient in time, but also in energy.

Yet, programming FPGAs remains a difficult task. Even though there exist today OpenCL interfaces to synthesize such hardware, higher-level programming languages, such as Java, C# or Python remain distant from them.

In this work, we describe a compiler, and its supporting runtime environment, that reduces this distance, translating functional code written in Java to the Intel HARP platform. Thus, we bring two contributions. First, the insight that a functional-style library is a good starting point to bridge the gap between high-level programming idioms and FPGAs. Second, the implementation of this system itself, including the compiler, its intermediate representation, and all the runtime support necessary to shield developers from the task of transferring data back and forth between the host CPU and the accelerator.

To demonstrate the effectiveness of our system, we have used it to implement different benchmarks, used in image processing and data-mining. For large inputs, we can observe consistent 20x speedups over the Java Virtual Machine across all our benchmarks. Depending on the target function that we compile, this speedup can be as large as 280.

Keywords: Compilers. Programming Languages. Computer Architecture.

Lista de Figuras

2.1 Lista de operadores ADD 20
2.2 Arquiteturas Multicores, GPU e Harp: Modelo de execução assíncrono e transferência de dados 21

3.1 Processo de compilação 30

4.1 Mediana do tempo de execução dos testes 43
4.2 Tempo total de execução 44
4.3 Relação entre aceleração e o volume de dados processados. 44

A.1 Grafo de Representação Intermediária para o Benchmark SubZip 54
A.2 Grafo Final de Representação Intermediária para o caso de teste SubZip . . . 55
A.3 Grafo de Representação Intermediária para o caso de teste SimSearch 62
A.4 Fluxo de Compilação de StringHash 67
A.5 Representação Intermediária de RGB2YUV 76

Lista de Tabelas

4.1 Casos de teste usados neste trabalho. 41
4.2 Performance dos casos de testes executando na plataforma HARP2. 43
4.3 Utilização de recursos para cada caso de teste na plataforma HARP2. 45

Sumário

1 Introdução **12**

1.1 Motivação 12

1.2 Contribuição 13

1.3 Organização do Trabalho 14

2 Revisão Bibliográfica **16**

2.1 Trabalhos Anteriores 16

2.1.1 ParallelME 16

2.1.2 Biblioteca do Usuário 17

2.1.2.1 Filter 18

2.1.2.2 Map 18

2.1.2.3 Reduce 19

2.1.3 ADD 19

2.1.3.1 Operadores ADD 20

2.1.3.2 Geração de Código Verilog 20

2.1.4 Harp 21

2.2 Trabalhos Relacionados 24

3 Solução **26**

3.1 Solução Proposta 26

3.2 Estruturas de Dados ParallelME 27

3.3 Métodos ParallelME 28

3.3.1 Map 28

3.3.2 Zip 28

3.3.3 Filter 28

3.3.4 Reduce 29

3.4 Compilação 29

3.5 Exemplos 31

3.5.1 SubZip 31

3.5.2 SimSearch 33

3.5.3 StringHash 35

3.5.4 RGB2YUV 37

4 Resultados **40**

4.1 RQ1 – Expressividade 40
4.2 RQ2 – Performance 42

5 Trabalhos Futuros e Conclusão **46**

Referências **49**

Apêndice A Código e Representações Intermediárias **53**
A.1 SubZip 53
A.2 SimSearch 57
A.3 StringHash 63
A.4 RGB2YUV 67

Capítulo 1

Introdução

1.1 Motivação

Uma tendência arquitetural é que o computador possua uma gama de aceleradores heterogêneos [6]. Essa tecnologia surge hoje como uma alternativa útil e eficaz para suprir a infinita necessidade de poder computacional que a indústria requer. No contexto desse trabalho nós chamamos de um *acelerador* qualquer processador que pode ser acoplado a uma CPU hospedeira afim de acelerar uma tarefa computacional que, a princípio, a CPU executaria. Aceleradores comuns incluem Unidades de Processamento Gráfico (em inglês, *Graphical Processing Unit - GPU*) [23] e Arranjo de Portas Programáveis em Campo (em inglês, *Field Programmable Gate Array - FPGA*) [20].

É cada vez mais caro a construção de produtos de silício pelos custos de projeto, verificação e implementação. Além do fato que existem sempre soluções de valor mais competitivo feitas em software. Como consequência, o mercado capaz de investir em produtos de silício customizados está encolhendo. Porém, ao mesmo tempo, as aplicacões precisam de desempenho, eficiência energética e preço que os projetos de silício customizados fornecem e que processadores de propósito genérico não alcançam os mesmos resultados. [1]

FPGAs oferecem uma excelente alternativa para aplicações que necessitam de um produto de silício customizado porque compartilham o desempenho e a capacidade de programação do circuito com exatamente a mesma funcionalidade.

Há uma enorme quantidade de aplicações que precisam de alta performance que se mostram suscetíveis a aceleração. São aplicações em áreas do conhecimento distintas como matemática [19], biologia [10] e física [15].

De todo modo, enquanto as GPUs estão cada vez mais acessíveis, o uso de FPGAs se mantém restrito a programadores experientes [9]. Uma das razões para aparente baixa popularidade das FPGAs é a ausência de recursos de programação. GPUs podem ser programadas diretamente utilizando linguagens específicas e de alto nível como OpenCL e C pra CUDA. Existem também anotações de sistema como OpenACC [36] e OpenMP [13].

FPGAs, em contrapartida, ainda são majoritariamente programadas através de linguagens de descrição de hardware como VHDL e Verilog. As primeiras tentativas de trazer uma linguagem de alto nível como OpenCL para FPGAs só chegaram a indústria recentemente [17]. O esforço de trazer linguagens de alto nível para o universo das FPGAs é tão grande quanto a distância semântica entre a descrição de um hardware e as abstrações feitas por linguagens de programação populares como Objetos e Funções de Alta-Ordem [18]. Essa dificuldade acaba comprometendo a eficiência de hardware sintetizado partindo de linguagens como Java [18].

1.2 Contribuição

Nesse trabalho tentamos endereçar essa questão – a ausência de uma linguagem mais expressiva que suporte FPGAs. Para esse fim, apresentamos um compilador que sintetiza hardware partindo de código Java. Porém, ao contrário de trabalhos anteriores que tentam preservar o paradigma genérico de Orientação de Objetos natural da linguagem [18], nós traduzimos código escrito em um estilo de paradigma funcional em Java. Para esse efeito, nós decidimos utilizar o padrão map-reduce, [7] uma abstração de programação de alto nível que nos dá grandes oportunidades de paralelismo. Projetamos e implementamos um compilador que traduz métodos chave de uma biblioteca funcional Java chamada ParallelME [2] para o processador Intel Altera HARP. Diversas combinações de métodos bem conhecidos entre programadores funcionais como map, reduce, filter, e zip nos permite escrever algoritmos complexos que podem ser usados, por exemplo, em mineração de dados ou processamento de imagens. Esses algoritmos são escritos em um estilo funcional de alto nível e o nosso compilador garante que eles podem se beneficiar de todo ganho de performance e, inclusive, ganho de energia utilizando de um acelerador FPGA.

Ressalto que o objetivo do trabalho é validar a plataforma como uma ferramenta que auxilia o programador a escrever programas paralelos utilizando FPGAs. As abstrações feitas pelo compilador e as estruturas de dados ParalleME visam atender especificamente o problema de trazer a tradução do código Java, utilizando de MapReduce, para FPGA. É possível implementar as mesmas estruturas utilizando outra linguagem apesar que pode ocorrer de haver abordagens mais eficientes. A comparação de eficiencia entre as possíveis linguagens fontes que se pode converter para código de descrição de hardware foge o escopo do trabalho. O leitor mais atento pode sentir falta de uma comparação dos resultados desse trabalho com outras arquiteturas como, por exemplo, GPUs. Essa comparação não é feita porque o arcabouço ParallelME já foi originalmente implementado

para GPUs e o objetivo desse trabalho era a implementação para a arquitetura nova, FPGAs. Mostramos no trabalho que a conversão utilizando nosso arcabouço além de possível é eficiente pois não há perda de desempenho na tradução do código Java para FPGAs. Inclusive é demonstrado que através do paralelismo do código traduzido temos um ganho de desempenho.

Após a conclusão desse trabalho, o compilador e todas suas bibliotecas estarão publicamente disponíveis e podem ser requisitadas a qualquer momento. Na seção 4 demonstramos que esse compilador é eficaz e útil. Nós utilizamos esse compilador para acelerar três algoritmos bem conhecidos: função Hash de String, conversão de formatos de imagens e busca por similaridade. Em todos algoritmos tivemos um grande aumento na velocidade de execução. Para grandes entradas, podemos observar uma aceleração de 20x comparando com a execução original que executa na máquina virtual. Em implementações altamente paralelas como a busca por similaridade temos acelerações de até 1000x. Para se beneficiar desse ganho em desempenho, os programadores precisam escrever código utilizando as quatro funções previamente citadas. Essas funções estão disponíveis através da biblioteca de usuário ParallelME [2]. Adicionalmente o programador fica isolado dos conceitos arquiteturais do hardware: não é necessário, por exemplo, inserir nenhuma primitiva para transferir os dados entre a CPU e o acelerador. Garantimos que protegemos o programador dessas tarefas muitas vezes consideradas entediantes.

1.3 Organização do Trabalho

No capítulo 1 damos a motivação desse trabalho. Discutimos sobre as tendências arquiteturais dos computadores e a necessidade de utilizar aceleradores. Em especial, sugerimos a utilização de FPGAs como aceleradores para diversas aplicações que necessitam de alto desempenho e eficiência energética. Porém argumentamos que as FPGAs ainda são de difícil adequação pela complexidade de desenvolver aplicações para esses aceleradores. Finalizamos apresentando a nossa solução que visa dar uma linguagem de alto nível para implementações de aplicações que vão se beneficiar das FPGAs como aceleradores. Há também uma apresentação da organização do trabalho.

No capítulo 2 fazemos uma revisão da bibliografia do assunto de aceleradores e bibliotecas de alto nível. Mostramos que a nossa solução utiliza o conceito de um trabalho anterior chamado ParallelME: Uma biblioteca Java, inspirada nas funções de Scala que permite o desenvolvedor escrever suas aplicações em alto nível utilizando de *Map*, *Reduce* e *Filter*. Em sequência apresentamos o Intel HARP, arquitetura alvo da nossa solução. A grande utilidade do Intel HARP nesse trabalho é abstração da transferência de dados

entre a CPU hospedeira e os aceleradores. Descrevemos outro trabalho fundamental que é o ADD. Uma biblioteca com componentes de baixo nível pré definidos acompanhado de uma linguagem para descrevê-los. O ADD também acompanha um módulo complementar para o simulador HADES que permite o desenvolvedor a simular suas aplicações antes de transferí-las para os aceleradores FPGA. Por último fazemos uma comparação da nossa solução com outras que existem na bibliografia sobre o desafio de facilitar o desenvolvimento para FPGAs.

No capítulo 3 damos os detalhes de como funciona a nossa solução. Mostramos quais são os métodos disponíveis e as estruturas de dados que o programador utilizará para desenvolver suas aplicações. Mostramos as nossas aplicações de exemplo e o código que o usuário escreve para cada uma delas para ilustrar ao leitor a simplicidade da nossa plataforma para escrever aplicações bastante distintas. Descrevemos como funciona o compilador que transforma esse código escrito pelo usuário em código Verilog para FPGAs. Mostramos as etapas que constituem o compilador desde a interpretação do código, a geração da representação intermediária, o paralelismo gerado e a conversão final em Verilog.

O capítulo 4 é uma análise de desempenho da nossa solução. Dado os nossos 4 casos de teste escolhidos respondemos duas hipóteses de pesquisa: facilidade de programação e desempenho. Mostramos, através de aplicações distintas, a quantidade de código escrita é simples e os ganhos de desempenho são maiores que 200x. Medimos o tempo de execução de cada caso de teste, mostramos a quantidade de linhas necessária para escrever as aplicações, apresentamos a proporção de utilização dos aceleradores para cada caso de teste em relação a capacidade total da FPGA.

No capítulo 5 concluimos o trabalho e damos perspectivas de como aprimorá-lo. Reforçamos que atingimos o objetivo de melhorar a programabilidade das FPGAs com uma biblioteca simples e totalmente compatível com Java, uma linguagem de alto nível e de extrema adesão. Confirmamos que além da programabilidade obtivemos uma aceleração em todos os nossos casos de teste mostrando ser uma solução de alto desempenho e escalável. Discutimos ainda uma nova arquitetura alvo chamada CGRA que eliminaria o tempo de sintetização do código na FPGA. Sugerimos também a implementação de novos métodos na nossa solução para aprimorar a nossa interface de programação.

Capítulo 2

Revisão Bibliográfica

Neste capítulo iremos fazer uma revisão da bibliografia do assunto de aceleradores e bibliotecas de alto nível. Mostramos que a nossa solução é baseada na interface de programação de um trabalho anterior chamado ParallelME: Uma biblioteca Java, inspirada nas funções de Scala que permite o desenvolvedor escrever suas aplicações em alto nível utilizando de *Map*, *Reduce* e *Filter*. Em sequência apresentamos o Intel HARP, arquitetura alvo da nossa solução. A grande utilidade do Intel HARP nesse trabalho é abstração da transferência de dados entre a CPU hospedeira e os aceleradores. Descrevemos outro trabalho fundamental que é o ADD. Uma biblioteca com componentes de baixo nível pré definidos acompanhado de uma linguagem para descrevê-los. O ADD também acompanha um módulo complementar para o simulador HADES que permite o desenvolvedor a simular suas aplicações antes de transferí-las para os aceleradores FPGA. Por último fazemos uma comparação da nossa solução com outras que existem na bibliografia sobre o desafio de facilitar o desenvolvimento para FPGAs.

2.1 Trabalhos Anteriores

2.1.1 ParallelME

ParallelME é um arcabouço formado por uma biblioteca Java acompanhada de um compilador que permite geração de código de alto desempenho. [2] Originalmente a solução gerava código para o ambiente de execução Android. ParallelME é composto de duas partes:

1. Uma abstração de programação (Biblioteca do Usuário).
2. Um compilador de fonte pra fonte

O modelo de programação utilizado em ParallelME foi inspirado pelas biblioteca de coleções de Scala. [27] Essa biblioteca consiste de um conjunto de diferentes estruturas de dados com suporte intrínseco a computação paralelo. O objetivo do ParallelME é oferecer uma biblioteca orientada a estruturas de dados similar a de Scala porém em Java e que pode ser utilizada para o desenvolvimento de programas estruturados como fluxo de dados. Uma característica importante de ParallelME é o seu viés a programação funcional. O código é escrito em uma combinação de funções de alta ordem como map, filter e reduce. Uma função de alta ordem é apenas uma função que recebe outra como argumento. Desenvolvedores podem descrever seus programas passando suas funções para essas funções de alta ordem. Os programas escritos em ParallelME podem ser executados na máquina virtual Java (JVM), como um código Java comum, ou pode ser traduzido para uma arquitetura paralela. A implementação original de ParalleME tinha como alvo GPUs, nossa arquitetura alvo são FPGAs. Enquanto previamente era produzido código OpenCL nesse trabalho produzimos Verilog. No restante dessa seção iremos descrever o arcabouço ParallelME com o objetivo de justificar algumas decisões de implementação mais tarde.

2.1.2 Biblioteca do Usuário

Programadores interagem com o nosso arcabouço através da Biblioteca do Usuário. Ela nada mais é que uma coleção de estruturas de dados e métodos que podem ser combinados para desenvolver aplicações. A abstração de programação que apresentamos nesse trabalho suporta três tipos de operações, são elas:

- Map
- Filter
- Reduce

Elas seguem o formato $A \overset{f}{\mapsto} B$, onde f representa a função implementada pelo usuário. O único requisito que essa função f deve seguir é que ela deve receber uma coleção Java A e produzir um elemento B. B pode ser uma nova coleção, um conjunto vazio ou um elemento único. A seguir, descrevemos as três operações principais usadas em ParallelME.

2.1.2.1 Filter

Filter $(A : L, f : L \mapsto bool) \mapsto B : L$ Essa operação possibilita que os usuários criem sub conjuntos dos dados. Um filtro recebe uma coleção A do tipo L, mais um predicado $f : L \mapsto bool$, e produz uma nova coleção B do tipo L. Essa nova coleção contém apenas elementos de A em que o predicado é verdadeiro. *Filter* não permite efeitos colaterais, isso significa que a coleção de entrada A é considerada uma estrutura de dados imutável. Em outras palavras, Filter cria uma nova área de memória para salvar a sua saída B. Essa operação é definida pela seguinte equação, onde A é a coleção de entrada, f é a função do usuário que retorna um valor binário e B é a coleção resultante.

$$A = \begin{bmatrix} a^1 \\ \dots \\ a^n \end{bmatrix} \overset{f}{\mapsto} B = \begin{bmatrix} b^1 \\ \dots \\ b^m \end{bmatrix} \quad (2.1)$$

Exemplo 2.1.1 (Filter) *O tipo abstrato T representa o tipo de elemento da coleção. O código do usuário retorna um valor binário. Caso a função retorne verdadeiro para dado elemento ele será inserido na coleção de resultado. Ao contrário, se a função do usuário retorna falso o elemento em questão não estará na coleção de resultado.*

2.1.2.2 Map

Map $(A : L, f : L \mapsto L') \mapsto B : L'$ A operação *Map* aplica a função do usuário em cada elemento da coleção A retornando uma nova coleção B como resultado. Pode ser matematicamente definida com a equação abaixo, onde A é a coleção de entrada, f é o código do usuário e B é o resultado que será a coleção de saída.

$$A = \begin{bmatrix} a^1 \\ \dots \\ a^n \end{bmatrix} \overset{f}{\mapsto} B = \begin{bmatrix} b^1 \\ \dots \\ b^n \end{bmatrix} \quad (2.2)$$

Como em *Filter*, uma operação *Map* não permite efeitos colaterais. Portanto A permanece como uma estrutura de dados imutável. Porém, ao contrário de *Filter*, uma operação *Map* retorna uma coleção B sempre com o mesmo tamanho de A. É permitido inclusive que *Map* retorne elementos de um tipo (L') distinto da sua entrada. O resultado se dá pela aplicação da função f em todo elemento de A. Perceba que a ordem relativa dos elementos é mantida da entrada até a saída. Como *Map* não permite efeitos colaterais, qualquer alteração feita nos elementos de entrada será descartada.

2.1.2.3 Reduce

Reduce: $(A : L, e : L, f : (L, e) \mapsto L') \mapsto b : L'$ A operação *Reduce* é uma função de agregação projetada para combinar todos elementos de uma coleção A em um único. *Reduce* retorna um único valor sumário b do tipo L'. Da mesma maneira que as outras operações, *Reduce* não permite efeitos colaterais. Em outras palavras, a coleção de entrada A é uma estrutura de dados imutável. Toda alteração feita na coleção de entrada é descartada ao final da execução. A operação é descrita pela seguinte equação, onde A é a coleção de entrada, f é o código do usuário com uma função agregadora e b é o valor sumário:

$$A = \begin{bmatrix} a^1 \\ \dots \\ a^n \end{bmatrix} \overset{f}{\mapsto} b(2.3)$$

Exemplo 2.1.2 (Reduce) *O tipo abstrato T representa o tipo de elemento da coleção que inclusive é o tipo do valor agregado na variável de resultado final. A função do usuário, agindo como um agregador, recebe um par de elementos de entrada (elem1 e elem2) e retorna um valor único. Cada elemento retornado será utilizado como entrada para a próxima iteração na coleção formando um novo par de entradas com um elemento ainda não visitado pelo iterador da redução.*

2.1.3 ADD

Desenvolvida pela Universidade Federal de Viçosa, a ferramenta ADD permite a criação, simulação e validação de grafos de fluxo de dados descritos através de código fonte Java ou por interface gráfica. A implementação da ferramenta é feita como uma extensão do editor/simulador HADES através da criação de uma biblioteca com operadores pré definidos descritos em Java. A ferramenta permite a conversão do grafo criado para a linguagem de descrição de hardware, no caso Verilog, para que seja sintetizada em FPGAs usando ferramentas da Xilinx ou Intel/Altera.

Figura 2.1: Lista de operadores ADD

Categoria	Operador	Função
Aritméticos	Abs	$y = \vert a \vert$
	Add	$y = a + b$
	Div	$y = a/b$
	Mod	$y = a\%b$
	Mul	$y = a * b$
	Sub	$y = a - b$
	AddI	$y = a + I$
	DivI	$y = a/I$
	ModI	$y = a\%I$
	MulI	$y = a * I$
	SubI	$y = a - I$
Lógicos	And	$y = a\&b$
	Or	$y = a \vert b$
	Not	$y = \ a$
	AndI	$y = a\&I$
	OrI	$y = a \vert I$
I/O	In 1-32	-
	Out 1-32	-
Memória	Histogram	-
Registrador	Register	$y = a$

Tipo	Operador	Função
Controladores de Fluxo	Beq	$if = (a == b) : 1; 0$ $else = (a == b) : 0; 1$
	Bne	$if = (a! = b) : 1; 0$ $else = (a! = b) : 0; 1$
	BeqI	$if = (a == I) : 1; 0$ $else = (a == I) : 0; 1$
	BneI	$if = (a! = I) : 1; 0$ $else = (a! = I) : 0; 1$
	Merge	$y = a$ se $if == 1$ $y = b$ se $else == 1$
Shift	Shl	$y = a << b$
	Shr	$y = a >> b$
	ShlI	$y = a << I$
	ShrI	$y = a >> I$
Comparadores	Max	$y = Max(a,b)$
	Min	$y = Min(a,b)$
	Slt	$y = (a < b)?1 : 0$
	MaxI	$y = Max(a,I)$
	MinI	$y = Min(a,I)$
	SltI	$y = (a < I)?1 : 0$
Acumuladores	AccAdd	-
	AccMax	-
	AccMin	-
	AccMul	-

Fonte: Elaborado pelo autor(a).

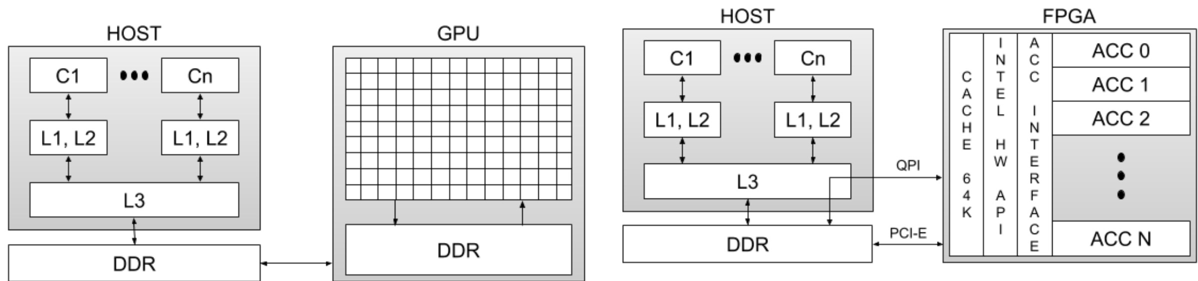
2.1.3.1 Operadores ADD

Para descrição dos grafos de fluxo de dados, a biblioteca de operadores é organizada nas seguintes categorias: acumuladores, aritméticos, branches, comparadores, E/S, lógicos, registrador, memória e shift. Os operadores foram baseados em uma equivalência com instruções RISC. A lista de operadores pode ser observada na Tabela 2.1. Na tabela pode-se ver os nomes dos operadores separados por categorias. Os operadores que possuem a letra “I"no fim do nome são operadores que trabalham com valores imediatos. A ferramenta possui também operadores de uso específico.

2.1.3.2 Geração de Código Verilog

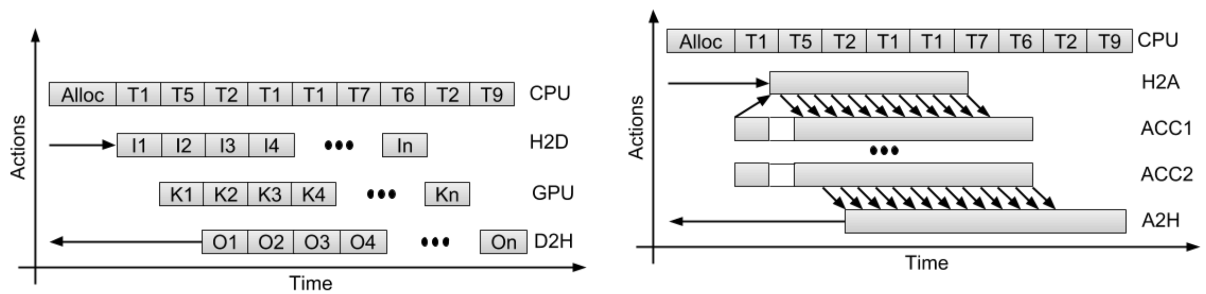
O gerador do ADD permite ao programador abstrair do código Verilog e da complexidade de todos os detalhes de sincronismo no nível de circuito, que envolvem máquinas de estados, sinais de relógio, reset, habilitação, codificação, etc. A descrição Verilog sintetizável é gerada com o auxílio da biblioteca Veriloggen. Semelhante ao fluxo de projeto em OpenCL da Intel, o código gerado pelo ADD deve ser sintetizado com uma ferramenta de projeto de FPGA como o Quartus.

Figura 2.2: Arquiteturas Multicores, GPU e Harp: Modelo de execução assíncrono e transferência de dados



(a) Arquitetura Multicores e GPU;

(b) Arquitetura Multicores e HARP;



(c) GPU com múltiplos streams;

(d) Transferência de dados e execução do Acelerador;

Fonte: Elaborado pelo autor(a).

2.1.4 Harp

Arquiteturas de vários núcleos usam coerência de cache, baixa latência e barramentos de alta velocidade para comunicação entre núcleos. Em 2011, Intel introduziu a Plataforma Aceleradora Heterogênea Reconfigurável (*Heterogeneous Accelerator Reconfigurable Platform (HARP)*) [24] afim de integrar aceleradores FPGA reconfiguráveis em arquiteturas de vários núcleos. Essa plataforma usa a *Intel QuickPath Interconnect Technology (QPI)* para permitir que todos os dispositivos dividam a memória do sistema por intermédio de um protocolo de coerência de cache. Mais tarde, em 2015, a primeira versão acadêmica do HARP estava disponível. No meio de 2017 a segunda versão do HARP foi disponibilizada para a comunidade de pesquisa. Finalmente, no começo de 2018, a Intel anunciou o primeiro processador escalável Intel Xeon com um processador integrado Intel Arria 10 FPGA [11]. Esse último processador está disponível como um produto comercial.

O projeto do HARP carrega várias similaridades com o conjunto CPU-GPU moderno. Porém essas tecnologias também apresentam diferenças muito particulares. Figura 2.2(a)-(d) contrasta a arquitetura do HARP com uma arquitetura que utiliza GPU

no que se trata no acesso de dados pela CPU.

Nos sistemas CPU-GPU, a GPU precisa primeiro transferir os dados para a memória do dispositivo, como podemos ver na Figura 2.2(a). Em contrapartida, na arquitetura HARP utilizamos um protocolo de coerência de cache que facilita o compartilhamento de dados entre hospedeiro e dispositivo, como é mostrado na Figura 2.2(b). O acelerador faz requisições de dados *on-the-fly*, que são entregues pela API HW/SW Intel. Portanto, ao contrário de sistemas de GPU, no HARP a transferência de dados e a computação se sobrepõem sem a intervenção explícita do desenvolvedor. Figura 2.2(c) descreve uma implementação de GPU utilizando várias filas de trabalho hardware (Hyper-Q), onde $I_1, \ldots, I_n$ são os dados enviados para o dispositivo, $K_1, \ldots, K_n$ são os kernels de execução acelerada e $O_1, \ldots, O_n$ são as transferências de dados do dispositivo para o hospedeiro. Para dar suporte a sobreposição, o HARP tem um cache local de 64Kb dentro da FPGA como mostrado na Figura 2.2(d). Dentro das siglas ACC significa Acelerador, H2A Hospedeiro para Acelerador e A2H o contrário. A mesma lógica serve para H2D e D2H onde o H ainda é Hospedeiro e D vem de dispositivo.

Ainda que a Intel API ajude a eliminar a necessidade de criar acesso de dados em baixo nível em HW/SW, várias limitações já impediram que essas ferramentas fossem popularmente aceitas. Primeiro, o projeto do acelerador e a comunicação com o acelerador ainda necessitam de conhecimento em dispositivos FPGA. Apesar da utilização de linguagens de alto nível como a versão da Intel de OpenCL, o projeto é altamente sensível ao estilo de código para extrair um paralelismo que seja efetivo. Para completar, o guia de melhores práticas da Intel recomenda na introdução que os programadores devem saber os detalhes do hardware que existe por baixo dos panos além de otimizações do compilador. Um dos objetivos desse trabalho é exatamente libertar os desenvolvedores dessas nuancias do hardware. Nós trazemos para os desenvolvedores que não conhecem tanto sobre FPGAs os benefícios desse hardware permitindo que eles utilizem uma linguagem de alto nível como Java para esses projetos.

Ressalto que os testes feitos com o Intel HARP foram executados remotamente em resultado de uma parceria com a Universidade Federal de Viçosa que disponibilizou o equipamento para pesquisa. Deixo aqui o meu agradecimento aos meus companheiros de pesquisa da Universidade Federal de Viçosa que foram fundamentais para a execução deste trabalho.

Esta seção descreve as 3 operações que norteam o trabalho *Map; Reduce; Filter*. Descrevemos o arcabouço ParallelME que foi um trabalho, também de minha autoria porém no passado, que disponibiliza as 3 operações *Map; Reduce; Filter* na linguagem Java para o programador escrever suas aplicações. Mostramos a ferramenta ADD desenvolvida pela Universidade Federal de Viçosa que nos fornece funções de baixo nível em Verilog para traduzirmos os programas escritos utilizando ParallelME. O ADD além de fornecer as funções também disponibiliza meios de simular graficamente o circuito Verilog gerado

pela função do usuário em ParallelME. Apresentamos o Intel HARP que é uma arquitetura que possui diversas FPGAs como aceleradores de execução de programas.

2.2 Trabalhos Relacionados

Na última década, tivemos algumas tentativas de abordar o problema de facilitar a programação em FPGAs.

Com a recente popularização dos aceleradores por hardware, essa missão ganhou ainda mais importância e tiveram algumas contribuições nos últimos 2 anos [14, 16, 22, 26, 34].

O trabalho de Prabhakar *et al.* [26], é baseado em linguagens funcionais para descrever uma representação do fluxo de dados que serão mapeadas em hardware. O ponto de partida é uma linguagem de alto nível de domínio específico baseado em Scala. Prabhakar priorizou aplicações de Aprendizado de Máquina. Afim de aprimorar a facilidade de programação nesse contexto, Prabhakar *et al.* propõe uma série de passos de compilação para gerar automaticamente o projeto de hardware partindo de uma representação intermediária em padrões paralelos (Map, Reduce, Zip, Fold, etc). Esses padrões são mapeados em hardware através de templates parametrizados para cada padrão. Eles são implementados usando uma linguagem de geração de hardware (HGL) baseada em Java chamada MaxJ. Os resultados experimentais são de aceleração de até 39.4x nos casos de testes do trabalho. Nós não podemos reproduzir esses resultados porque Prabhakar *et al.* não disponibiliza informações explícitas sobre o tempo de execução e nem qual foram exatamente seus dados de entrada.

Para o caso de teste do k-means, eles declaram uma aceleração de 19.7 vezes em relação ao mesmo caso de teste implementado manualmente. Em contrapartida, um trabalho anterior [16] mostrou uma aceleração de 1.15 quando comparado com uma CPU de 6 núcleos. Koeplinger *et al.* [16] incrementou a contribuição de Prabhakar *et al.* [26]. Nosso trabalho difere de Prabhakar e Koeplinger na escolha da linguagem fonte e na escolha do arquitetura. Também acreditamos que temos um processo de compilação diferente: enquanto Koeplinger e Prabhakar têm componentes pré compilados para funções particulares, nós fazemos a compilação completa. Em outras palavras, nenhuma das nossas aplicações reutiliza partes compiladas anteriormente.

Houveram tentativas de melhorar a programabilidade de FPGAs em linguagens que não executam na JVM, como C e C++. Por exemplo, o trabalho de Wang *et al.* [34] apresenta um arcabouço MapReduce chamado Melia, que abstrai a FPGA utilizando interfaces MapReduce escritas em C e OpenCl. Eles propõem otimizações para aprimorar uma série de parâmetros que afetam significativamente o desempenho e a utilização de recursos da FPGA. Nesse trabalho eles mostraram aceleração de 0.5x ate 3.2x em relação uma abordagem utilizando GPU [34]. O trabalho de Neshatpour *et al.* [22] apresenta uma solução map-reduce para clusters de arquitetura FPGA+CPU que vai de ponta a ponta. Porém, os autores não fizeram experimentos com FPGAs reais, utilizaram simulação da

arquitetura para mostrar os resultados.

Kachris *et al.* [14] propõe hardware aceleradores usando ferramentas de síntese de Alto Nível (HLS) para tarefas que utilizam Map como um fluxo de dados e uma memória aceleradora reconfigurável para as tarefas que utilizam Reduce. Programadores precisam escrever suas aplicações usando uma combinação de código C/C++ usando o arcabouço Phoenix MapReduce. Kachris *et al.* apresenta aceleração na faixa de 1-5x comparando contra um processador de 8 núcleos.

Se tratando de hardware heterogêneo CPU+FPGA, várias empresas estão projetando suas plataformas proprietárias. A IBM já Interface do Processador Acelerador Coerente (CAPI) [32] para integrar os processadores Power8 com FPGAs ou outros aceleradores. Essa interface permite a integração de aceleradores usando pinos de E/S e oferece uma camada de abstração para os aceleradores acessarem a memória do sistema. Em 2016, a Microsoft lançou a Nuvem Configurável [3], uma evolução do projeto original Catapult [28]. Nesta plataforma, as placas aceleradoras FPGA são colocadas entre a placa de interface de rede (NIC) gerando uma comunicação de baixa latência entre as FPGAs. Há também uma interface de comunicação PCI Express entre os processadores Xeon e as placas de aceleradores FPGA. A Amazon também está se tornando um personagem importante no assunto com o lançamento da plataforma EC2 F1 onde aceleradores FPGA estão disponíveis para processamento na nuvem.

Trabalhos anteriores com a primeira versão do Intel Harp demonstraram como acelerar aplicações de propósito específico. [4, 5, 37, 30, 38, 35, 12, 29, 21, 31]. Diferente dessas abordagens, nós oferecemos uma interface de programação genérica baseada em Java para o sistema Intel HARP. Nós traduzimos código diretamente para ambos, seja um binário de execução CPU ou projeto de hardware para FPGA. Essa combinação de CPU-FPGA não assume que o programador tenha algum conhecimento prévio sobre configuração de hardware. Portanto, esse trabalho se intitula o primeiro compilador capaz de traduzir programas escritos em linguagem de alto nível para a arquitetura Intel HARP.

Neste capítulo fizemos uma revisão da bibliografia do assunto de aceleradores e bibliotecas de alto nível. Mostramos a biblioteca ParallelME e definimos as funcões principais da biblioteca: *Map*, *Filter* e *Reduce* Definimos nossa arquitetura alvo como o Intel HARP que proporciona uma facilidade na trasferência de dados entre CPU e Aceleradores FPGA. Apresentamos a biblioteca ADD que possui componentes pré definidos que combinados serão semanticamente equivalentes ao código do usuário. Por último, revisamos trabalhos relacionados ao desafio de melhorar a programabilidade de FPGAs.

Capítulo 3

Solução

Neste capítulo daremos os detalhes de como funciona a nossa solução. Mostramos quais são os métodos disponíveis e as estruturas de dados que o programador utilizará para desenvolver suas aplicações. Mostramos as nossas aplicações de exemplo e o código que o usuário escreve para cada uma delas para ilustrar ao leitor a simplicidade da nossa plataforma para escrever aplicações bastante distintas. Descrevemos como funciona o compilador que transforma esse código escrito pelo usuário em código Verilog para FPGAs. Mostramos as etapas que constituem o compilador desde a interpretação do código, a geração da representação intermediária, o paralelismo gerado e a conversão final em Verilog.

Ressalto que também utilizamos o nome ParallelME para a nossa solução porque o trabalho anterior deixa um legado sobre a abstração de código que proporcionamos ao usuário. O que vem neste capítulo é uma restruturação e aprimoramento do que o ParallelME foi no passado. Foi necessário a completa reescrita do código do arcabouço e principalmente adicionando uma representação intermediária em forma de grafo para suportar a arquitetura alvo desse trabalho, um novo ParallelME.

3.1 Solução Proposta

ParallelME é um arcabouço focado em traduzir código de uma abstração de alto nível em Java para um ambiente de execução de alto desempenho programado por Verilog. O ambiente alvo de alto desempenho é o Intel HARP, que está fora do ambiente da JVM, portanto é necessário código para integrar os dois ambientes. ParallelME é capaz de suprir essa integração gerando código para o usuário afim de deixar essa tarefa completamente transparente na chamada do Intel HARP e na transferência de dados.

3.2 Estruturas de Dados ParallelME

A principal estrutura do arcabouço ParallelME é um Array que chamamos de GenArray. O nome GenArray remete a Generic Array, porque o objetivo inicial do trabalho era traduzir qualquer tipo primitivo Java. Essa implementação genérica ficará como trabalho futuro. Para utilizar o GenArray o programador deve inicializar a estrutura de dados com um Array nativo do Java e será armazenado um apontador para o vetor original nativo, não há cópia de dados nesse processo. Basicamente estamos só encapsulando o vetor nativo de Java para utilizar os métodos do arcabouço ParallelME.

Após utilizar as funções de ParallelME o programador pode decidir trazer de volta o Array nativo em Java para continuar sua utilização da maneira que desejar. Nenhuma cópia de dados é feita também ao retirar os dados de dentro de GenArray. Houve um cuidado muito grande de manipulação de dados nessa parte do arcabouço porque essa tarefa não é traduzida para uma arquitetura de alto desempenho. Caso houvesse cópia de memória nessas etapas acarretaria em um grande gargalo na nossa implementação.

Apesar da Biblioteca de Usuário permitir qualquer tipo de dados dentro de GenArray apenas haverá conversão de código para valores inteiros. A implementação com outras estruturas de dados como ponto flutuante é plenamente possível mas iria demandar um maior esforço de engenharia e excederia o tempo limite para esse trabalho. Além de tipos flutuantes, um objetivo futuro de ParallelME é dar suporte a estruturas cada vez mais genéricas, não só aquelas de tipos primitivos. O esforço pra tipos genéricos porém é menos trivial que para tipos primitivos. Isso acontece porque classes genéricas podem conter atributos de outras classes e tudo terá que ser convertido para tipos primitivos. Além de ineficiente do ponto de vista de geração do código essas classes genéricas podem causar perda de performance já que, provavelmente, será feita a transferência de dados desnecessários na função a ser convertida para FPGA. Como veremos no futuro, a transferência de dados é o maior custo quando executamos operações na FPGA. Portanto temos conversão de código para as seguintes estruturas:

Integer GenArray<Integer> Vetor simples contendo apenas inteiros

Pair GenArray<Pair> Vetor de pares de inteiros, implementa os métodos array.getElementE1() e array.getElementE2() para acessar o primeiro e o segundo elemento do vetor respectivamente. Esses métodos também são úteis na hora de converter o código para Verilog para definir qual elemento do par estará associada a qual operação

Triple GenArray<Triple> Vetor de triplas de inteiros. Semanticamente equivalente ao Pair

O GenArray implementa as funções de ParallelME: map, filter, reduce e zip.

3.3 Métodos ParallelME

3.3.1 Map

O método *Map* aplica uma função implementada pelo usuário em cada elemento do GenArray retornando um novo GenArray como resultado. Pode-se ver a definição de *Map* em 2.1.2.2.

A implementação desse método se dá por uma classe genérica que recebe dois tipos que representam, respectivamente, o tipo de saída e o tipo de entrada da função que é escrita pelo usuário.

O usuário, para escrever o código da sua função Map, deverá sobrescrever o método *function* da classe Map de ParallelME com o código que ele deseja executar em cada elemento, respeitando os tipos definidos por ele mesmo. É possível ver um exemplo de Map no código 3.1

3.3.2 Zip

O método *Zip* consiste de um método auxiliar para unir dois GenArray em um único GenArray<Pair>. O resultado é a associação entre os elementos de cada GenArray de mesma posição formando pares em uma coleção única.

Quando dizemos que é apenas um método auxiliar é porque não é gerado código de baixo nível para esse método. Ele serve apenas para fazer com que dois GenArray sejam enviados para a FPGA na forma de um Array único.

Se as coleções possuem tamanhos diferentes será lançada uma exceção no código. É possível ver um exemplo de Zip no código 3.1

3.3.3 Filter

O método *Filter* gera um novo GenArray partindo do GenArray original de entrada onde os elementos do GenArray gerado satisfazem alguma regra definida pelo usuário.

Pode-se ver a definição de *Filter* em 2.1.2.1.

A implementação desse método é definida por uma função do usuário que retorna um valor booleano. Quando verdadeiro o elemento deve estar na coleção de saída e deve ser ausente quando falso. É possível ver um exemplo de *Filter* no código 3.2.

3.3.4 Reduce

O método *Reduce* retorna um único elemento dado um GenArray de entrada. O usuário irá escrever um método em como combinar o resultado de todos os elementos do GenArray de entrada em único valor que chamamos de acumulador. Pode-se ver a definição de *Reduce* em 2.1.2.3.

A implementação desse método contem uma função que irá receber o elemento a ser processado e o acumulador. O programador definirá qual será as operações que irão executar e acumular o resultado. Quando todos os elementos tiverem sido processados o resultado final do método é o acumulador. É possível ver um exemplo de *Reduce* no código 3.3.

3.4 Compilação

Esse trabalho descreve um compilador que traduz Java em Verilog em um processo de quatro passos:

- Front-End: Código Java é traduzido para uma representação intermediária. Nesse nível os programas ficam representados por grafos de fluxo de dados em que vértices são operações e as arestas representam dependências entre as operações.

- Merging: Grafos independentes representando diferentes funções são fundidos em super grafos. Essa fusão visa reduzir a transferência de dados entre o hospedeiro e o dispositivo.

- Parallelization: O super grafo de fluxo de dados é replicado. Cada instância representa código que pode ser executado em paralelo.

- Back-End: O grafo final é traduzido em Verilog. Código Verilog é sintetizado na FPGA do Intel HARP.

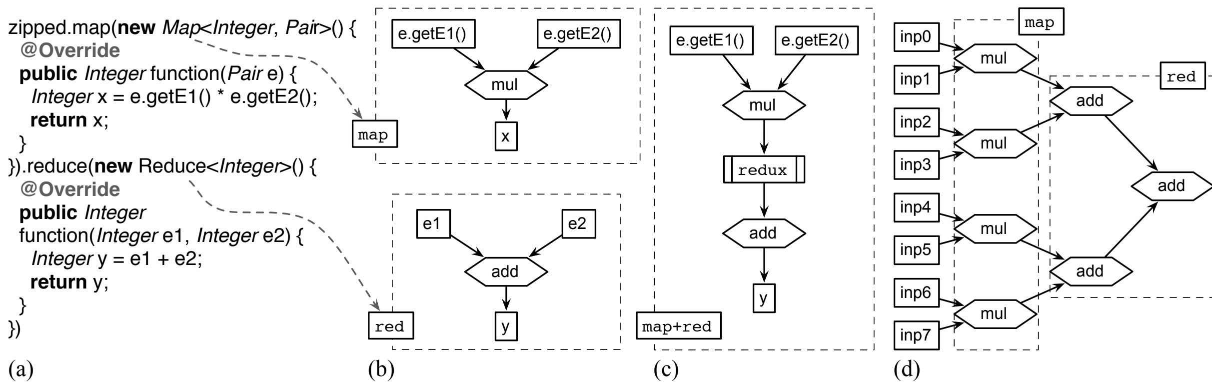
Figura 3.1: A entrada de cada fase do nosso processo de compilação.

a Programa Java que usa as bibliotecas ParallelME.

b Grafo de fluxo de dados produzido para cada função compilável.

c Super grafo produzido após a etapa de fusão.

d Grafo de fluxo de dados replicado.



Fonte: Elaborado pelo autor(a).

A Figura 3.1 mostra nossas várias representações intermediárias que usamos antes de gerar código em Verilog. No restante da seção iremos descrever cada uma dessas representações e explicar como o processo de tradução acontece.

Front-End: A entrada do nosso compilador é o código Java de um programa. Entretanto, apenas algumas partes do programa são traduzidas. Como mencionamos na Seção 1, nós traduzimos quatro operações funcionais básicas: Map, Reduce, Filter e Zip. Essas operações precisam ser escritas como uma extensão das classes de ParallelME. Na Figura 3.1 nós utilizamos classes anônimas para extender as classes Map e Reduce. A tradução acontece como descrito anteriormente, transformamos o código original em uma AST (Abstract Syntax Tree) utilizando o ANTLR [25] e identificamos as funções que precisam ser traduzidas. O código original da função será substituído pelas chamadas ao Intel HARP. A saída dessa etapa de compilação é um grafo de fluxo de dados para cada método ParallelME do nosso programa de entrada. A Figura 3.1(b) mostra um exemplo.

Merging: O objetivo desse estágio é unir dois grafos de fluxo de dados que executam em sequência. Essa união reduz a transferência de dados entre a CPU e a FPGA. Essas sequências são descobertas através da primeira fase do nosso processo de compilação, a etapa de Front-End. Se a saída de uma função f é usada como entrada de uma função g e ambas serão traduzidas então podemos unir as duas. A otimização substitui a representação intermediária individual de cada função por uma única que é a união das duas, um grafo único. Operacionalmente, unir os dois grafos é uma operação simples que consiste em alterar para onde as aretas apontam: arestas que apontam para o nó de saída da função f são redirecionados a apontar diretamente para o nó de entrada em g. A Figura 3.1

(c) mostra o grafo resultante após unir duas funções vistas em 3.1 (b).
PARALLELIZATION: Quando temos os grafos resultados da fase de união –os chamados super grafos– nós partimos para replicá-los. Replicação é a estratégia que utilizamos para obter paralelismo. Cada uma das operações que compilamos: Map, Reduce, Filter e Zip contêm um laço que itera através de uma estrutura de dado. A princípio, nós poderíamos simplesmente sintetizar esse laço na FPGA. Porém, essa estratégia não nos proporciona o paralelismo que precisamos. Portanto, ao invés de sintetizar o laço diretamente nós o *desenrolamos*. O fator de desenrolamento é determinado pela quantidade de recursos que podemos alocar na FPGA. Por exemplo, se a FPGA permite processar 32 elementos por ciclo, um grafo com apenas 2 entradas pode ser replicado 16 vezes para alcançar o limite da FPGA. Tipicamente, os laços que produzimos processam de 16 a 64 elementos do laço por ciclo. Concretamente, um laço desenrolado consiste de uma série de super grafos similares como a Figura 3.1 (d) mostra. Nós também damos suporte a laços em que a quantidade de iterações não é um múltiplo do fator de desenrolamento. A estratégia utilizada para essas iterações restantes é basicamente um laço sequencial que processa os elementos restantes.
BACK-END: A etapa final do nosso processo consiste na geração do hardware. Para converter o grafo de fluxo de dados em Verilog nós utilizamos o Veriloggen [33]. Nós não convertemos o grafo de fluxo de dados visto na Figura 3.1 diretamente para Verilog mas sim uma entrada para o Veriloggen. Essa ferramenta recebe uma descrição do hardware, escrita como um conjunto de funções Python, e produzimos finalmente código Verilog. Essa descrição usa uma série de componentes pré definidos em um arcabouço desenvolvido em um trabalho anterior [8]. Veriloggen lê esses módulos e irá traduzi-los em componentes Verilog os quais sintetizamos no HARP.

3.5 Exemplos

3.5.1 SubZip

O primeiro exemplo a ser dado da nossa solução será o SubZip. Você pode ver o código original e a representação intermediária no apêndice A.1. Esse algoritmo faz a subtração par a par entre valores de dois vetores e é calculado o valor absoluto do resultado. Nesse caso de teste são utilizadas as funções *Zip* e *Map* do nosso arcabouço. *Zip* é utilizado para juntar os dois vetores em um único vetor para ser transferido para a

FPGA e *Map* é utilizado para fazer a subtração par a par e calcular o valor absoluto.

Código 3.1: Código SubZip

```
GenArray<Integer> a = new GenArray<Integer>(primitive_a);
GenArray<Integer> b = new GenArray<Integer>(primitive_b);

GenArray<Pair> zip;

zip = a.zip(b);

GenArray<Integer> subArray = zip.map(new Map<Integer, Pair>() {
       public Integer function(Pair element) {
              int retorno, abs;
              retorno = (Integer) element.getE2() - (Integer) element.
                 ↪ getE1();
              abs = Math.abs(retorno);
              return abs;
       }
});
```

Podemos ver nesse código a declaração de duas estruturas GenArray<Integer>que serão mescladas utilizando o método *Zip*. Com os dois elementos em um GenArray<Pair> único é utilizada a função *Map* para subtrair os dois valores e calcular o valor absoluto do resultado.

A primeira etapa da nossa solução irá gerar uma *Abstract Syntax Tree* desse código e percorrê-la. Ao encontrar a utilização da estrutura de dados GenArray acompanhada de uma invocação da função *Map* o compilador irá gerar uma representação intermediária em forma de grafo. Esse grafo consiste nas operações e variáveis intermediárias do código do usuário partindo da entrada até a variável de retorno do método do usuário.

Como nesse caso de teste não é necessário a etapa de *Merging* iremos direto para *Parallelization*. Para gerar a versão final da representação intermediária iremos avaliar a oportunidade de ampliar o nosso grafo afim de otimizar ao máximo a transferência de dados para a FPGA. Nesse caso de teste temos 2 entradas de 4 bytes cada uma. A nossa arquitetura de teste permite a transferência de 128 bytes por ciclo, portanto podemos replicar a nossa função 16 vezes. Ao fim da etapa de Paralelização teremos uma representação intermediária como visto no Apêndice na figura A.2

Tendo a Representação Intermediária final iremos finalmente gerar código. Em primeiro lugar é gerado código para o simulador Hades, onde o programador pode testar e depurar a corretude da sua solução e do código gerado pelo compilador.

A implementação do algoritmo para o Simulador Hades para o SubZip é feita dina-

micamente através do código A.2 que se encontra no Apêndice. Note que, por simplicidade e melhorar a exibição, foi exibido código gerado sem a etapa de paralelização.

Além da representação no simulador Hades é gerado também o código que faz a transferência dos dados do usuário da JVM para o Acelerador. Logo em seguida também temos efetivamente a chamada do Acelerador. Quando dizemos aqui genericamente Acelerador é porque ele pode ser simulado ou não. Podemos ver o código de chamada de um Acelerador simulado em A.3 que se encontra no Apêndice. Para trocar de um Acelerador simulado para um Acelerador real utilizando FPGA basta alterar a classe AfuManagerSimul para AfuManager. Essa abstração acontece devido a biblioteca ADD.

3.5.2 SimSearch

O segundo exemplo será o SimSearch. O código original e a representação intermediária estão no Apêndice A.2. Dado um vetor de números inteiros, o algoritmo irá buscar por todos os números mais próximos de uma requisição dado uma distância máxima. Por exemplo, dado o vetor [0,1,2,3,4,5], e uma requisição por 3 e distância 1 o resultado será [2,3,4].

O algoritmo irá calcular a distância de todos os números utilizando a função *Map*. Logo em seguida são filtradas, utilizando *Filter*, todas as distâncias menores ou iguais a distância máxima. Os índices que foram selecionados pelo filtro constroem o vetor de saída do nosso algoritmo.

Código 3.2: Código SimSearch

```
public static GenArray<Integer> Search(GenArray<Integer> d, int Q, int
    ↪ q_r) {
                // Map used to calculate the distance between each
                    ↪ element in d and
                // the query q:
                GenArray<Integer> d_map = d.map(new Map<Integer, Integer
                    ↪ >() {
                  @Override
                  public Integer function(Integer element) {
                    int Q_less_element, abs;
                    Q_less_element = Q - element;
                    abs = Math.abs(Q_less_element);
                    return abs; //need to apply absolute value.
```

```
                        }
                }).filter(new Filter<Integer>() {
                        @Override
                        public boolean function(Integer element) {
                          boolean compare_return;
                          compare_return = element < q_r;
                          return compare_return;
                        }
                });

                return d_map;
        }
```

Nesse método Search podemos ver o GenArray<Integer> que é o vetor onde irá ocorrer a busca. A requisição Q e a distância máxima q_r.

O diferencial dessa função é a demonstração do método *Filter*. Outro fator importante nesse código é a sobreposição das funções *Map* e *Filter*. Quando é escrito dessa maneira, caso seja possível, o compilador irá transformar as duas funções em um único dataflow, evitando uma transferência de dados desnecessária entre a FPGA e o Host. Nesse caso, a saída do dataflow de Map é enviada direto para o dataflow de Filter, dentro da FGPA.

Abaixo temos a representação intermediária final, após a união dos dois dataflows. O leitor pode perceber que a função Map termina exatamente após o componente abs0, que calcula o valor absoluto da entrada. Logo em seguida temos a operação de Filter. A nossa função Filter, dentro do dataflow consistem em um multiplexador final que decide qual será a saída. No nosso caso a saída pode ser o valor absoluto calculado pela função Map ou pode ser -1. A função controle do multiplexador é o caminho que passa pelo componente Set Less Than (slt0) e Branch Equals (beq0). O componente slt0 define sua saída como 1 caso o seu valor de entrada for menor que o valor definido por q_r. O outro segmento do dataflow, que passa pelos registradores apenas replica o valor até o multiplexador. O segmento do dataflow, que passa pelos dois subtratores serve apenas para gerar o valor -1. Perceba que o primeiro subtrator faz uma subtração de dois valores iguais, gerando 0. O subtrator sequente decrementa em 1 esse valor gerando o -1.

Nesse caso de teste também é possível fazer a etapa de *Parallelization*. Podemos multiplicar esse dataflow por 32x. Basta notar que existe exatamente uma única entrada para uma única saída. Lembrando que podemos transferir 128 bytes por ciclo, que são exatamente 32 entradas.

Por fim iremos mostrar o código final gerado, após a representação intermediária. Assim como no SubZip o código é gerado, primeiramente, para o simulador Hades para

testes e depuração antes de gerar código para a FPGA.

A implementação do algoritmo para o Simulador Hades para o SimSearch é feita dinamicamente através do código A.5 que se encontra no Apêndice A para melhor visualização. Note que, por simplicidade e melhor exibição, foi exibido código gerado sem a etapa de *Parallelization*.

Além da representação no simulador Hades é gerado também o código que faz a transferência dos dados do usuário da JVM para o Acelerador. Logo em seguida também temos efetivamente a chamada do Acelerador. Quando dizemos aqui genericamente Acelerador é porque ele pode ser simulado ou não. Podemos ver o código de chamada de um Acelerador simulado em A.6. Para trocar de um Acelerador simulado para um Acelerador real utilizando FPGA basta alterar a classe AfuManagerSimul para AfuManager. Essa abstração acontece devido a biblioteca ADD.

3.5.3 StringHash

Esse caso de teste calcula o valor Hash de uma String exatamente do mesmo jeito que o método padrão implementado pela classe String da JVM. Basicamente consiste em multiplicar cada caracter, da esquerda para a direita, pelas respectivas potências de 31 e somar todas as multiplicações ao final. O código original e as representações intermediárias podem ser vistas no Apêndice A.3.

Código 3.3: Código String Hash

```
public int hashCode() {
		if(this.hash == 0) {
			Integer[] multipliers = new Integer[this.value.
			    ↪ getLength()];
			int multiplier_base = 1;
			for(int n = multipliers.length; n >= 1; n--) {
				multipliers[n-1] = multiplier_base;
				multiplier_base *= 31;
			}

			GenArray<Integer> genArray_multipliers = new
			    ↪ GenArray<Integer>(multipliers);
			GenArray<Pair> zipped = this.value.zip(
			    ↪ genArray_multipliers);
```

```
                this.hash = zipped.map(new Map<Integer, Pair>() {
                    ↪
                        @Override
                        public Integer function(Pair element) {
                                int retorno;
                                retorno = (Integer) element.getE1() *
                                    ↪  (Integer) element.getE2();
                                return retorno;
                        }

                }).reduce(new Reduce<Integer>() {
                        @Override
                        public Integer function(Integer element1,
                            ↪ Integer element2) {
                                element1 = element1 + element2;
                                return element1;
                        }

                });
        }
        return this.hash;
}
```

No código acima temos a função que calcula o hash da String utilizando a nossa solução escrita em ParallelME.

As potências de 31 são calculadas previamente e sequencialmente pela máquina hospedeira, como podemos ver no código.

É enviado para a FPGA um par com o valor do caracter como inteiro e sua respectiva potência de 31 para ser multiplicado. Vemos uma função Map que faz a multiplicação par a par.

Por fim, o resultado das multiplicações é somado utilizando o método Reduce.

Nesse caso de teste também ocorre a fusão das duas funções em único dataflow.

O diferencial desse caso de teste é a replicação de dataflows que contém a função Reduce.

O dataflow é replicado para maximizar a entrada de dados, portando é multiplicado 16 vezes para obtermos a entrada máxima por ciclo.

Porém, na etapa de replicação, iremos adicionar uma camada de somadores para cada dupla de multiplicações para fazer as reduções, como podemos ver na imagem.

Como resultado final teremos uma quantidade de camadas definida pelo resultado do logaritmo na base 2 das entradas, ou seja, 4 camadas de somadores até o somador acumulador final.

Tendo a Representação Intermediária final iremos finalmente gerar código. Em primeiro lugar é gerado código para o simulador Hades, onde o programador pode testar e depurar a corretude da sua solução e do código gerado pelo compilador.

A implementação do algoritmo para o Simulador Hades para o String Hash é feita dinamicamente através do código A.8 que está no Apêndice para melhor visualização. Note que, por simplicidade e melhorar a exibição, foi exibido código gerado sem a etapa de paralelização.

Além da representação no simulador Hades é gerado também o código que faz a transferência dos dados do usuário da JVM para o Acelerador. Logo em seguida também temos efetivamente a chamada do Acelerador. Quando dizemos aqui genericamente Acelerador é porque ele pode ser simulado ou não. Podemos ver o código de chamada de um Acelerador simulado em A.9. Para trocar de um Acelerador simulado para um Acelerador real utilizando FPGA basta alterar a classe AfuManagerSimul para AfuManager. Essa abstração acontece devido a biblioteca ADD.

3.5.4 RGB2YUV

Esse caso de teste transforma uma imagem que está no formato RGB para o formato YUV. Constitui apenas de uma função Map que faz transformações nas cores de cada pixel. Apesar de utilizar apenas a função Map esse é o caso de teste que mais possui componentes e que reutiliza operações. O código original e as representações intermediárias estação no Apêndice A.4.

Código 3.4: Código RGB2YUV

```
GenArray<Triple> image_array = new GenArray<Triple>(
    ↪ primitive_image);
image_array = image_array.map(new Map<Triple, Triple
    ↪ >(){

        @Override
        public Triple function(Triple element) {
                int Y, U, V, y1, y2, y3, u1, u2, u3,
                    ↪ v1, v2, v3;
```

```
                y1 = 9798 * (Integer) element.getE1()
                    ↪ ;
                u1 = 21208 * (Integer) element.getE1
                    ↪ ();
                v1 = -4784 * (Integer) element.getE1
                    ↪ ();

                y2 = 19235 * (Integer) element.getE2
                    ↪ ();
                u2 = 16941 * (Integer) element.getE2
                    ↪ ();
                v2 = 9437 * (Integer) element.getE2()
                    ↪ ;

                y3 = 3736 * (Integer) element.getE3()
                    ↪ ;
                u3 = 3277 * (Integer) element.getE3()
                    ↪ ;
                v3 = 4221 * (Integer) element.getE3()
                    ↪ ;

                Y = y1 + y2;
                Y = Y + y3;
                Y = Y >> 15;

                U = u1 + u2;
                U = U + u3;
                U = U >> 15;
                U = U + 128;

                V = v1 + v2;
                V = V + v3;
                V = V >> 15;
                V = V + 128;

                return new Triple(Y,U,V);
            }
```

```
                    });
```

No código acima temos a função que faz as transformações necessárias para converter um pixel em RGB para YUV. Por simplicidade, o código que faz a leitura da arquivo imagem em RGB foi omitido. Outro diferencial desse caso de teste é que utilizamos GenArray<Triple> para salvar os valores, seja RGB ou YUV do Pixel.

O dataflow é replicado para maximizar a entrada de dados, portanto é multiplicado 10 vezes. Não é possível alcançar o valor máximo de 32 entradas pela característica do problema de receber 3 entradas por pixel.

Tendo a Representação Intermediária final iremos finalmente gerar código. Em primeiro lugar é gerado código para o simulador Hades, onde o programador pode testar e depurar a corretude da sua solução e do código gerado pelo compilador.

A implementação do algoritmo para o Simulador Hades para o RGB2YUV é feita dinamicamente através do código A.11. Note que, por simplicidade e melhorar a exibição, foi exibido código gerado sem a etapa de paralelização.

Além da representação no simulador Hades é gerado também o código que faz a transferência dos dados do usuário da JVM para o Acelerador. Logo em seguida também temos efetivamente a chamada do Acelerador. Quando dizemos aqui genericamente Acelerador é porque ele pode ser simulado ou não. Podemos ver o código de chamada de um Acelerador simulado em A.12. Para trocar de um Acelerador simulado para um Acelerador real utilizando FPGA basta alterar a classe AfuManagerSimul para AfuManager. Essa abstração acontece devido a biblioteca ADD.

Detalhamos neste capítulo a implementação da nossa solução. Apresentamos os métodos que podem ser utilizados pelo desenvolvedor: *Map*, *Reduce*, *Filter* e *Zip*. Junto com os métodos, apresentamos também a estrutura de dados *GenArray<>* que define o vetor de dados de ParallelME. E mostramos como utilizá-lo para conjunto de valores unitários, duplos ou triplos. Apresentamos as quatro etapas necessárias para converter o código: *Front-End*, *Merging*, *Parallelization* e *Back-End*. Também apresentamos os nossos quatro casos de testes juntamente com o seu código fonte escrito pelo usuário. Os casos de teste são SubZip, SimSearch, StringHash e RGB2YUV.

Capítulo 4

Resultados

O objetivo dessa seção é responder duas perguntas de pesquisa (RQs), que são:

- **RQ1**: a nossa solução proposta é expressivamente suficiente para permitir desenvolvedores escrever programas eficientes e úteis ?
- **RQ2**: qual é a aceleração que podemos esperar traduzindo funções Java para a arquitetura do Intel HARP ?

4.1 RQ1 – Expressividade

É difícil medir expressividade, medir quão fácil e elegante são os algoritmos que nós permitimos que os desenvolvedores escrevam. Afim de endereçar essa pergunta de pesquisa, ainda que de um ponto de vista subjetivo, nós devemos descrever nossos quatro casos de teste, os quais escrevemos para demonstrar a efetividade do nosso arcabouço. Os casos de teste são algoritmos recorrentes, utilizados em mineração de dados e processamento de imagens. A tabela 4.1 resume os casos de teste. Na tabela, as colunas **Map**, **Fil**, and **Red** representam, respectivamente, a quantidade de operações Map, Filter e Reduce de cada programa.

A coluna **Loc** significa Linhas de Código em Java, e **CLoC** a quantidade de linhas de código compiladas. Em outras palavras, **CLoC** representa a quantidade de linhas de código que efetivamente foram traduzidas pelo nosso compilador para Verilog. Apesar da quantidade ser pequena o seu tamanho é enganoso. Uma sentença como `list.reduce(0, addition)` que possui apenas uma linha é uma semântica complexa e é traduzido para várias outras linhas. Vamos descrever cada aplicação individualmente no restante da seção.

Caso de Teste 1: SubZip. Esse caso de teste executa a subtração entre valores de mesma posição em dois vetores. É retornado o valor absoluto da substração para cada índice. Se os vetores possuem tamanhos diferentes apenas o prefixo comum entre eles é

Tabela 4.1: Casos de teste usados neste trabalho.

Caso de Teste	**Map**	**Fil**	**Red**	**LoC**	**CLoC**
SubZip	1			50	8
SimSearch	1	1		81	16
StringHash	1		1	115	15
RGB2YUV	1			101	28

processado. Na prática, esse caso de teste consiste da combinação de uma operação Zip e Map: A primeira combina os elementos, transformando em um vetor único de pares de elementos. A última faz a operação: $f(x_1, x_2) = |x_1 - x_2|$.

Caso de Teste 2: SimSearch. Esse algoritmo de busca de similaridade recebe um arranjo v de inteiros, uma consulta q e uma distância máxima d. Partindo dessas entradas, o algoritmo encontra os elemento em v que são mais próximos de q respeitando a distância máxima d. Se nenhum elemento obedece os critérios é retornado vazio. Nós chamamos de distância nesse trabalho o valor absoluto entre os números. A nossa implementação utiliza uma função Map para calcular a distância de cada elemento em v até q. Depois dessa primeira fase, uma função Filter retorna apenas os elementos que possuem distância máxima d.

Caso de Teste 3: StringHash. Esse caso de teste calcula o valor hash para uma String Java através de uma operação Map e uma Reduce. Na prática, nós multiplicamos cada caracter na String pela sua respectiva potência de 31, e uma redução para calcular a soma de todos esses elementos:

$$f(s[n]) = s[0] * 31^{(n-1)} + s[1] * 31^{(n-2)} + ... + s[n-1]$$

Caso de Teste 4: RGB2YUV. Esse caso de teste converte uma imagem no formato RGB no formato YUV. Na prática, esse caso de teste consiste apenas de uma operação Map. Entretanto, nesse caso de teste utilizamos uma estrutura de dados mais complexa. Nesse caso de testes utilizamos um arranjo de triplas para armazenar os valores dos Pixels. Portanto, dada uma entrada (R, G, B), nossa operação de Map aplica a seguinte transformação em cada uma dessas triplas:

$$\begin{pmatrix} Y & = & (9798R + 19235G + 3736B) >> 15 \\ U & = & (21208R + 16941G + 3277B) >> 15 + 128 \\ V & = & (-4784R + 9437G + 4221B) >> 15 + 128 \end{pmatrix}$$

Discussão A tabela 4.1 mostra o tamanho de cada caso de teste. Eles são pequenos e foram escritos utilizando uma linguagem de programação de alto nível, Java em conjunto com ParallelME. Mais importante, eles foram escritos sem levar em conta o fato que após compilados serão transformados em código para uma arquitetura massivamente paralela. Como veremos na Seção 4.2, todos esses casos de teste podem executar sem nenhuma

alteração no seu código original em um computador comum ou no Intel HARP. Portanto, não somente sua semântica é portável entre as arquiteturas, um fato esperado já que esses programas são executados em um ambiente virtual. Porém, eles também mantém a sua desempenho portável. Isso quer dizer que não há perda de desempenho ao transformar o código original para executar em FPGAs. Pelo contrário, quando fazemos essa portabilidade temos um ganho impressionante de desempenho, como vamos discutir mais adiante. Dessa forma, acreditamos que podemos responder com positivo para a nossa primeira pergunta de pesquisa.

4.2 RQ2 – Performance

Metodologia Nessa seção, nós avaliamos a desempenho do código que produzimos quando comparado com a sua implementação original em Java executando na JVM. Os números representam o tempo real de execução. Cada caso de teste foi executado e medido 10 vezes. Afim de reduzir o tempo de aquecimento da JVM nós descartamos o resultado das três primeiras execuções. O tempo de aquecimento da JVM é o tempo gasto para a JVM chegar a um estado estável, onde todo o código já foi compilado, por exemplo. Todas as medidas consideram o tempo de transferência da memória da CPU para a FPGA.

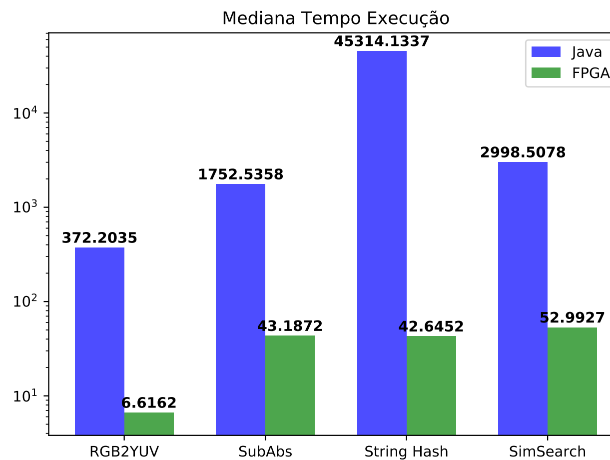
Taxa de Transferência Tabela4.2 mostra o resultado do tempo de execução para grandes entradas em cada um dos nossos casos de teste. A taxa de transferência é medida pela quantidade de dado processada por unidade de tempo. É limitada pela quantidade de dados que podemos transferir para a FPGA e pela quantidade de operações simultâneas que podemos executar na FPGA. O caso de teste RGB2YUV possui a maior taxa de transferência, um fato natural já que esse é o caso de teste que contem a maior quantidade de operações por dado. A menor taxa de transferência acontece com o SubZip, outro fato natural já que esse é o caso de teste com a menor quantidade de operações por dado. Perceba que a quantidade de dados varia entre os casos de teste. Por exemplo, o RGB2YUV envia para a FPGA um arranjo em que cada célula consiste de 3×32 bits e lê de volta um arranjo de tamanho similar. StringHash, em contrapartida, lê um arranjo de caracteres e retorna um valor primitivo consistido por 64 bits. No caso do SubZip, a entrada é o dobro do tamanho da saída: lê dois arranjos e retorna apenas um. Finalmente, a nossa implementação de SimSearch tem entrada e saída com o mesmo tamanho. Ao invés de retornar apenas um elemento, a nossa implementação utiliza um sinalizador para indicar, em cada elemento do arranjo, qual célula retorna verdadeiro ou falso.

Aceleração As figuras 4.1 e 4.2 comparam o tempo de execução dos nossos casos de teste com e sem aceleração. O objetivo desses experimentos é demonstrar que

Tabela 4.2: Performance dos casos de testes executando na plataforma HARP2.

Caso de teste	Entrada GB	Taxa de Transferência GB/s	Tempo ms	GOPs
SubZip	4	17.72	340.75	2.93
SimSearch	4	19.06	420.88	11.88
StringHash	4	11.87	337.67	2.96
RGB2YUV	4	19.52	340.75	19.56

Figura 4.1: Mediana do tempo de execução para cada caso de teste. As barras estão em escala logaritma e o tempo está em Milissegundos



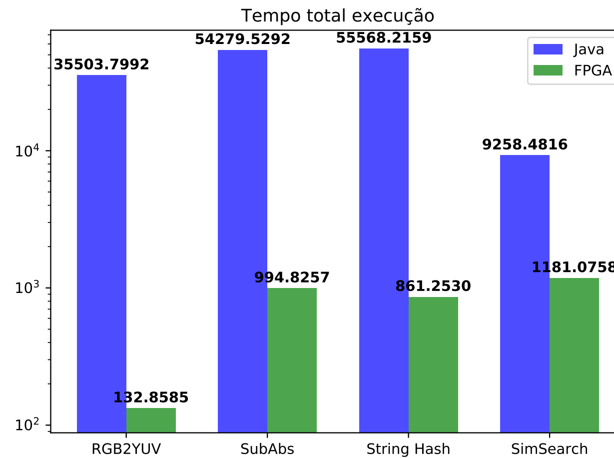
Fonte: Elaborado pelo autor(a).

conseguimos aumentar a desempenho dos nossos casos de teste utilizando o Intel HARP. Os gráficos mostram que nossa técnica melhora o tempo de execução de todos os nossos casos de teste e, em alguns casos, com melhoria muito significativa.

A figura 4.3 mostra quanto a nossa aceleração aumenta com o incremento do tamanho da entrada. Quanto maior a entrada no caso de teste maior é a nossa aceleração observada. Os ganhos de desempenho mais notaveis foram no RGB2YUV. Para imagens com 4M $\times$ 4M pixels, nós observamos uma aceleração de 270 vezes ao se comparar com o mesmo programa executando na JVM sem nenhum suporte ao HARP. Era esperado pra esse caso de teste o maior nível de aceleração porque ele é o que possui maior reuso dos dados de entrada: o mesmo pixel é processado multiplas vezes durante a conversão. Outra observação interessante é sobre a limitação dos nossos ganhos de desempenho: para entradas muito pequenas o uso da FPGA se torna uma vulnerabilidade. Nos nossos experimentos, entradas com tamanho menor que 32.768 elementos tiveram uma desaceleração quando compiladas para utilizar o Intel Harp. Essa observação foi válida para todos os casos de testes exceto o RGB2YUV. Para entradas pequenas, o tempo de transferência dos dados para a FPGA não compensa o ganho de desempenho utilizado no paralelismo.

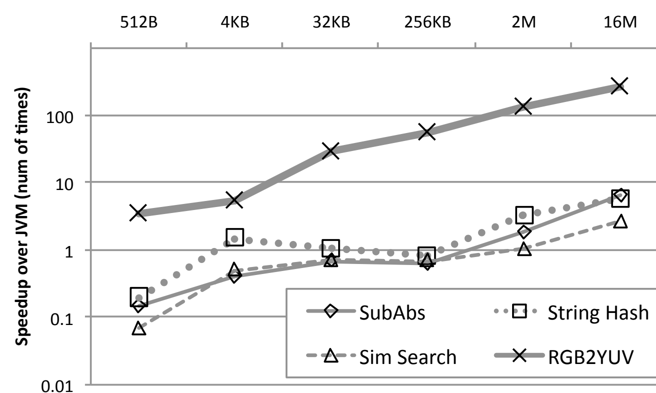
Utilização dos Recursos A Tabela 4.3 mostra a utilização dos recursos pela

Figura 4.2: Tempo total de execução. A inicialização da JVM não é considerada



Fonte: Elaborado pelo autor(a).

Figura 4.3: Relação entre aceleração e o volume de dados processados.



Fonte: Elaborado pelo autor(a).

FPGA do HARP depois da síntese de cada caso de teste. Cada caso de teste foi sintetizado com um clock de 200MHz. Consequentemente, nossos ganhos em desempenho, mostrado nos nossos experimentos, vem através de paralelismo e não de poder computacional. O número de Módulos Logico Adaptaveis (ALMs), junto com os blocos de Processamento de Sinal Digital (DSP) nos da uma ideia da complexidade dos nossos casos de teste. Unidades ALM implementam as operações básicas executadas pelo nosso hardware sintetizado; Os DSPs implementam algumas operações em lote, como blocos de multiplicações e blocos de adição de números de ponto flutuante. Quanto maior a presença dessas unidades mais complexo tende a ser o caso de teste. A última coluna da Tabela 4.3 mostra o tempo em horas necessário para sintetizar esse hardware. A compilação de cada caso de teste leva cerca de duas horas para finalizar. Esse oneroso custo de tempo é pago uma vez só, durante a síntese das operações que constituem as funções map, filter e reduce do caso de teste. Esse custo também está presente em trabalhos recentes que também compilam linguagens de programação de alto nível para FPGAs [16, 26]. Chamadas futuras dos

nossos casos de teste não possuem tal custo.

Tabela 4.3: Utilização de recursos para cada caso de teste na plataforma HARP2.

Algoritmo	**Cópias p/ ACC**	**ALMs**	**Memória bits**	**DSP Blocks**	**Tempo (h)**
SubZip	16	110,405 (26%)	24,344,080 (44%)	0 (0%)	2:08
SimSearch	1	131,326 (31%)	24,344,080 (44%)	0 (0%)	2:02
StringHash	8	112,152 (26%)	24,344,080 (44%)	128 (8%)	1:53
RGB2YUV	15	119,595 (28%)	24,344,080 (44%)	720 (47%)	2:03

Discussão Os experimentos nessa seção demonstram que nossa solução permite entregar uma enorme aceleração para típicos programas escritos em Java. Perceba que baseamos os nossos testes comparando-os com a versão sequencial de cada caso de teste. Pode-se argumentar que deveríamos ter comparado os nossos casos de teste com suas versões paralelas ao invés de sequenciais. Entretanto, queremos ressaltar que o desenvolvedor não precisou se preocupar em escrever código paralelo para ser traduzido para o Intel HARP. Em outras palavras, o desenvolvedor escreve código sequencial. Tendo isso em mente, estamos entregando ao programador um imenso ganho de desempenho sem nenhum custo de desenvolvimento. Poderia ter sido pago um custo de desenvolvimento, por exemplo, se os desenvolvedores tivessem que reescrever o código da aplicação utilizando a biblioteca de programação paralela Java (Java Concurrent Library).

Capítulo 5

Trabalhos Futuros e Conclusão

Neste trabalho, discutimos sobre as tendências arquiteturais dos computadores e a necessidade de utilizar aceleradores. Em especial, sugerimos a utilização de FPGAs como aceleradores para diversas aplicações que necessitam de alto desempenho e eficiência energética. Argumentamos que as FPGAs ainda são de difícil adequação pela complexidade de desenvolver aplicações para esses aceleradores. Finalizamos apresentando a nossa solução que visa dar uma linguagem de alto nível para implementações de aplicações que vão se beneficiar das FPGAs como aceleradores.

Apresentamos então um arcabouço completo, baseado em trabalhos anteriores como o ParellelME, um arcabouço completo que traduz métodos específicos, escritos em Java até Verilog, permitindo sua execução em FPGAs. Os métodos que serão traduzidos devem ser escritos, utilizando a nossa biblioteca de programação e os métodos que serão traduzidos devem seguir um modelo *Map*, *Reduce* e *Filter*. A arquitetura alvo do nosso trabalho é o Intel HARP que possui aceleradores FPGA acoplados junto ao processador principal da máquina. Além do ganho de desempenho pela localidade, o Intel HARP também possui uma implementação eficiente para transferir de dados entre a memória e os aceleradores FPGA através de filas. Utilizamos a ferramenta ADD, desenvolvida pela Universidade Federal de Viçosa para fazer a ponte entre o nosso código gerado, ainda em Java, até a descrição de hardware em Verilog. O ADD possui componentes pré definidos como operações básicas e *branching* que são convertidas para Verilog. A ferramenta desenvolvida nesse trabalho irá gerar um grafo de fluxo de dados utilizando as operações do ADD que irá transformar o código em sua etapa final em Verilog.

A nossa biblioteca de usuário permite que o desenvolvedor escreva programas genéricos utilizando os métodos *Map*, *Reduce* e *Filter* através de uma estrutura de dados de arranjo que permite armazenar valores singulares, pares e triplas. Partindo dessa abstração desenvolvemos quatro casos de teste com aplicações distintas: Subtração entre arranjos, Busca por similaridade, Hash de String e um conversor de pixels em RGB para YUV. Através da implementação desses casos de teste mostramos o quanto a nossa biblioteca de usuário é abrangente e genérica o suficiente para implementar aplicações diversas. Os casos de teste são algoritmos recorrentes, utilizados em mineração de dados e processamento de imagens. Demonstramos também como o compilador implementado nesse

trabalho funciona descrito em quatro etapas distintas: Front-End, Merging, Parallelization e Back-End. O Front-End transforma o código original Java em uma representação intermediária como um grafo de fluxo de dados. A etapa de Merging faz a fusão de métodos distintos mas que são chamados imediatamente em sequência. Essa fusão transforma os dois métodos em único grafo de fluxo de dados e o objetivo é melhorar o desempenho. Isso acontece porque quando temos um único grafo para os dois métodos teremos apenas uma única chamada e transferência de dados entre hospedeiro e FPGA. Em Parallelization replicamos o grafo de fluxo de dados considerando várias entradas distintas ganhando desempenho através de paralelismo. Por último, temos a etapa de Back-End que transforma o grafo de fluxo de dados parelelizado em um formato compatível com a ferramenta ADD que irá fornecer meios de simular e finalmente transformar o nosso código em Verilog para execução em FPGAs.

Demonstramos a eficiência da nossa solução respondendo duas perguntas principais:

- **RQ1**: a nossa solução proposta é expressivamente suficiente para permitir desenvolvedores escrever programas eficientes e úteis ?
- **RQ2**: qual é a aceleração que podemos esperar traduzindo funções Java para a arquitetura do Intel HARP ?

Para a RQ1 mostramos que este trabalho permite que o usuário escreva casos de testes genéricos e com certa simplicidade mostrando exatamente o código de cada caso de teste e ressaltando a clareza e a objetividade da linguagem. Mais importante, eles foram escritos sem levar em conta o fato que após compilados serão transformados em código para uma arquitetura massivamente paralela. Leva-se em conta também que qualquer caso de teste pode ser executado na Java Virtual Machine até o Intel HARP.

Para a RQ2 demonstramos que não há perda de desempenho ao transformar o código original para executar em FPGAs. Pelo contrário, quando fazemos essa portabilidade temos um ganho impressionante de desempenho. Ganho esse causado, majoritariamente, pelo paralelismo adicionado no código. Mostramos que quanto mais parelelismo conseguimos adicionar maior será ganho de desempenho. O grande gargalo do tempo de execução dos nossos casos de teste, utilizando os aceleradores FPGA foi a transferência de dados da memória até os aceleradores. Analisamos a utilização de recursos espaciais dos aceleradores FPGA pela quantidade de componentes que os nossos casos de testes utilizam e percebe-se que podemos colocar casos de testes ainda muito mais complexos, ou seja, grafos de fluxos de dados ainda maiores.

Apesar de utilizável e efetivo ainda há muito o que se extender no trabalho. Existem outras operações funcionais como flap-map e group-by que deixaria a biblioteca do usuário ainda mais expressiva. Podemos fazer um estudo ainda mais elaborado da ferramenta implementando e medindo o desempenho de uma aplicação completa ao invés

de *microbenchmarks*. Inclusive, a implementação de casos de testes novos em aplicações como de aprendizado de máquina enriqueceriam ainda mais o trabalho.

É possível também melhorar o tempo gasto para sintetizar o código do usuário na FPGA se ampliarmos a nossa solução para dar suporte a CGRAs (Coarse Grained Reconfigurable Architectures). CGRAs consistem de um arranjo de várias unidades funcionais todas conectadas entre si além de registradores exclusivos entre unidades funcionais. As unidades funcionais podem executar operações básicas como adição, subtração e multiplicação, muito parecido com os componentes que possuimos através da ferramenta ADD. Em constraste com as FPGAs o tempo de reconfiguração das CGRAs é mínimo e permitiria inclusive que uma mesma arquitetura CGRA executasse códigos distintos. Na nossa solução, a FPGA irá conter apenas código estático já sintetizado previamente a execução do programa e não pode ser alterado durante a execução. As CGRAs inclusive necessitam de um bom compilador para explorar com eficiência os recursos disponíveis pelo CGRA. É uma tarefa desafiadora conectar as unidades funcionais entre si e alocar os registradores de maneira eficiente.

Referências

[1] Sagheer Ahmad, Vamsi Boppana, Ilya Ganusov, Vinod Kathail, Vidya Rajagopalan, and Ralph Wittig. A 16-nm multiprocessing system-on-chip field-programmable gate array platform. *IEEE Micro*, 36(2):48–62, 2016.

[2] Guilherme Andrade, Wilson de Carvalho, Renato Utsch, Pedro Caldeira, Alberto Alburquerque, Fabricio Ferracioli, Leonardo Rocha, Michael Frank, Dorgival Guedes, and Renato Ferreira. ParallelME: A parallel mobile engine to explore heterogeneity in mobile computing architectures. In *Euro-Par*, pages 447–459, Berlin, Heidelberg, 2016. Springer.

[3] A. M. Caulfield, E. S. Chung, A. Putnam, H. Angepat, J. Fowers, M. Haselman, S. Heil, M. Humphrey, P. Kaur, J. Y. Kim, D. Lo, T. Massengill, K. Ovtcharov, M. Papamichael, L. Woods, S. Lanka, D. Chiou, and D. Burger. A cloud-scale acceleration architecture. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1–13, Oct 2016.

[4] M. C. F. Chang, Y. T. Chen, J. Cong, P. T. Huang, C. L. Kuo, and C. H. Yu. The smem seeding acceleration for dna sequence alignment. In *2016 IEEE 24th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 32–39, May 2016.

[5] P. Colangelo, E. Luebbers, R. Huang, M. Margala, and K. Nealis. Application of convolutional neural networks on intel xeon processor with integrated fpga. In *2017 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–7, Sept 2017.

[6] Jason Cong, Hui Huang, Chiyuan Ma, Bingjun Xiao, and Peipei Zhou. A fully pipelined and dynamically composable architecture of cgra. In *Field-Programmable Custom Computing Machines (FCCM), 2014 IEEE 22nd Annual International Symposium on*, pages 9–16. IEEE, 2014.

[7] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, 2008.

[8] Jeronimo Penha e Lucas Bragança e Danilo Almeida e Jose Nacif e Ricardo Ferreira. Add - uma ferramenta de projeto de aceleradores com dataflow para alto desempenho. *Simpósio em Sistemas Computacionais de Alto Desempenho (WSCAD)*, 2017.

[9] Jeremy Fowers, Greg Brown, Patrick Cooke, and Greg Stitt. A performance and energy comparison of fpgas, gpus, and multicores for sliding-window applications. In *FPGA*, pages 47–56, New York, NY, USA, 2012. ACM.

[10] Pascale Guerdoux-Jamet and Dominique Lavenier. SAMBA: hardware accelerator for biological sequence comparison. *Computer Applications in the Biosciences*, 13(6):609–615, 1997.

[11] Jennifer Huffstetler. Intel processors and fpgas—better together, 2018. https://itpeernetwork.intel.com/intel-processors-fpga-better-together/, Last accessed on 2018-05-30.

[12] Z. Istvan, D. Sidler, and G. Alonso. Runtime parameterizable regular expression operators for databases. In *2016 IEEE 24th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 204–211, May 2016.

[13] Julien Jaeger, Patrick Carribault, and Marc Pérache. Fine-grain data management directory for openmp 4.0 and openacc. *Concurr. Comput. : Pract. Exper.*, 27(6):1528–1539, 2015.

[14] Christoforos Kachris, Dionysios Diamantopoulos, Georgios Ch. Sirakoulis, and Dimitrios Soudris. An fpga-based integrated mapreduce accelerator platform. *Journal of Signal Processing Systems*, 87(3):357–369, Jun 2017.

[15] Gota Kikugawa, Rossen Apostolov, Narutoshi Kamiya, Makoto Taiji, Ryutaro Himeno, Haruki Nakamura, and Yasushige Yonezawa. Application of MDGRAPE-3, a special purpose board for molecular dynamics simulations, to periodic biomolecular systems. *Journal of Computational Chemistry*, 30(1):110–118, 2009.

[16] D. Koeplinger, R. Prabhakar, Y. Zhang, C. Delimitrou, C. Kozyrakis, and K. Olukotun. Automatic generation of efficient accelerators for reconfigurable hardware. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, pages 115–127, June 2016.

[17] Andrew C. Ling, Utku Aydonat, Shane O'Connell, Davor Capalija, and Gordon R. Chiu. Creating high performance applications with intel's FPGA OpenCL&Trade; SDK. In *IWOCL*, pages 11:1–11:1, New York, NY, USA, 2017. ACM.

[18] Robert Macketanz and Wolfgang Karl. JVX - A rapid prototyping system based on java and fpgas. In *FPL*, pages 99–108, 1998.

[19] Svetlin A. Manavski. CUDA compatible GPU as an efficient hardware accelerator for AES cryptography. In *ICSCN*, pages 65–68. IEEE, 2007.

[20] Eric Monmasson and Marcian Cirstea. FPGA design methodology for industrial control systems – a review. *Transactions on Industrial Electronics*, 54(4):1824–1842, 2007.

[21] Duncan J.M Moss, Srivatsan Krishnan, Eriko Nurvitadhi, Piotr Ratuszniak, Chris Johnson, Jaewoong Sim, Asit Mishra, Debbie Marr, Suchit Subhaschandra, and Philip H.W. Leong. A customizable matrix multiplication framework for the intel harpv2 xeon+fpga platform: A deep learning case study. In *Proceedings of the 2018 ACM/-SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA, pages 107–116. ACM, 2018.

[22] Katayoun Neshatpour, Maria Malik, Avesta Sasan, Setareh Rafatirad, Tinoush Mohsenin, Hassan Ghasemzadeh, and Houman Homayoun. Energy-efficient acceleration of mapreduce applications using fpgas. *Journal of Parallel and Distributed Computing*, 119:1 – 17, 2018.

[23] John Nickolls and William J. Dally. The GPU computing era. *Micro*, 30:56–69, 2010.

[24] N. Oliver, R. R. Sharma, S. Chang, B. Chitlur, E. Garcia, J. Grecco, A. Grier, N. Ijih, Y. Liu, P. Marolia, H. Mitchel, S. Subhaschandra, A. Sheiman, T. Whisonant, and P. Gupta. A reconfigurable computing system based on a cache-coherent fabric. In *2011 International Conference on Reconfigurable Computing and FPGAs*, pages 80–85, Nov 2011.

[25] Terence J. Parr and Russell W. Quong. Antlr: A predicated-ll (k) parser generator. *Software: Practice and Experience*, 25(7):789–810, 1995.

[26] Raghu Prabhakar, David Koeplinger, Kevin J. Brown, HyoukJoong Lee, Christopher De Sa, Christos Kozyrakis, and Kunle Olukotun. Generating configurable hardware from parallel patterns. *SIGPLAN Not.*, 51(4):651–665, March 2016.

[27] Aleksandar Prokopec, Phil Bagwell, Tiark Rompf, and Martin Odersky. A generic parallel collection framework. In *Euro-Par*, pages 136–147, Berlin, Heidelberg, 2011. Springer-Verlag.

[28] A. Putnam, A. M. Caulfield, E. S. Chung, D. Chiou, K. Constantinides, J. Demme, H. Esmaeilzadeh, J. Fowers, G. P. Gopal, J. Gray, M. Haselman, S. Hauck, S. Heil, A. Hormati, J. Y. Kim, S. Lanka, J. Larus, E. Peterson, S. Pope, A. Smith, J. Thong, P. Y. Xiao, and D. Burger. A reconfigurable fabric for accelerating large-scale datacenter services. *IEEE Micro*, 35(3):10–22, May 2015.

[29] T. S. Accelerating k-means clustering on a tightly-coupled processor-fpga heterogeneous system. In *2016 IEEE 27th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, pages 176–181, 2016.

[30] David Sidler, Zsolt István, Muhsen Owaida, and Gustavo Alonso. Accelerating pattern matching queries in hybrid cpu-fpga architectures. In *Proceedings of the 2017 ACM International Conference on Management of Data*, SIGMOD '17, pages 403–415. ACM, 2017.

[31] Greg Stitt, Abhay Gupta, Madison N. Emas, David Wilson, and Austin Baylis. Scalable window generation for the intel broadwell+arria 10 and high-bandwidth fpga systems. In *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA, pages 173–182. ACM, 2018.

[32] J. Stuecheli, B. Blaner, C. R. Johns, and M. S. Siegel. Capi: A coherent accelerator processor interface. *IBM Journal of Research and Development*, 59(1):7:1–7:7, Jan 2015.

[33] Shinya Takamaeda-Yamazaki. Veriloggen: A library for constructing a verilog hdl source code in python. https://github.com/PyHDI/veriloggen. Accessed: 2017-07-20.

[34] Z. Wang, S. Zhang, B. He, and W. Zhang. Melia: A mapreduce framework on opencl-based fpgas. *IEEE Transactions on Parallel and Distributed Systems*, 27(12):3547–3560, Dec 2016.

[35] Gabriel Weisz, Joseph Melber, Yu Wang, Kermin Fleming, Eriko Nurvitadhi, and James C. Hoe. A study of pointer-chasing performance on shared-memory processor-fpga systems. In *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '16, pages 264–273. ACM, 2016.

[36] Sandra Wienke, Paul L. Springer, Christian Terboven, and Dieter an Mey. OpenACC - first experiences with real-world applications. In *Euro-Par*, pages 859–870, New York, NY, USA, 2012. Springer.

[37] C. Zhang, R. Chen, and V. Prasanna. High throughput large scale sorting on a cpu-fpga heterogeneous platform. In *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 148–155, May 2016.

[38] Chi Zhang and Viktor Prasanna. Frequency domain acceleration of convolutional neural networks on cpu-fpga shared memory system. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '17, pages 35–44. ACM, 2017.

Apêndice A

Código e Representações Intermediárias

A.1 SubZip

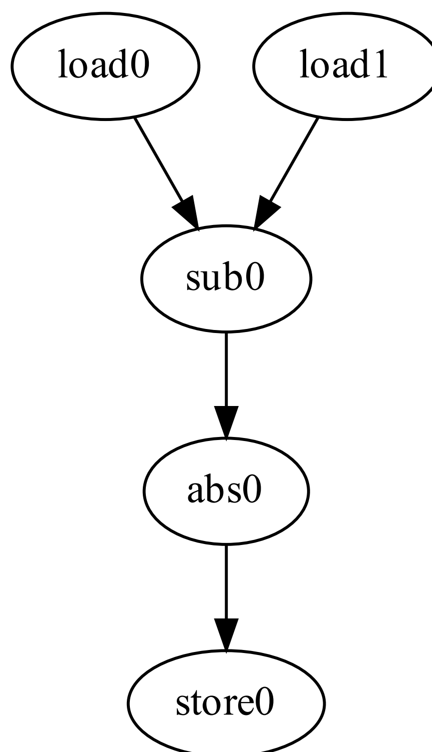
O caso de teste SubZip faz a subtração par a par entre valores de dois vetores e retorna um vetor com o valor absoluto da subtração. Podemos ver o código original, utilizando a biblioteca ParallelME em A.1. Segue também a representação intermediária original A.1 e a representação intermediária replicada A.2 para criar paralelismo. Mostramos nessa seção também o código gerado pelo nosso compilador. Pode-se verificar o código que cria os componentes ADD A.2, que são utilizados tanto para simulação quanto para gerar o código final em Verilog através do Veriloggen. Além da representação no simulador Hades é gerado também o código que faz a transferência dos dados do usuário da JVM para o Acelerador. Logo em seguida também temos efetivamente a chamada do Acelerador. Quando dizemos aqui genericamente Acelerador é porque ele pode ser simulado ou não. Podemos ver o código de chamada de um Acelerador simulado em A.3. Para trocar de um Acelerador simulado para um Acelerador real utilizando FPGA basta alterar a classe AfuManagerSimul para AfuManager. Essa abstração acontece devido a biblioteca ADD.

Código A.1: Código ParallelME SubZip

```
GenArray<Integer> a = new GenArray<Integer>(primitive_a);
GenArray<Integer> b = new GenArray<Integer>(primitive_b);

GenArray<Pair> zip;

zip = a.zip(b);

GenArray<Integer> subArray = zip.map(new Map<Integer, Pair>() {
        public Integer function(Pair element) {
                int retorno, abs;
                retorno = (Integer) element.getE2() - (Integer) element.
```

```
            ↪ getE1();
        abs = Math.abs(retorno);
        return abs;
    }
});
```

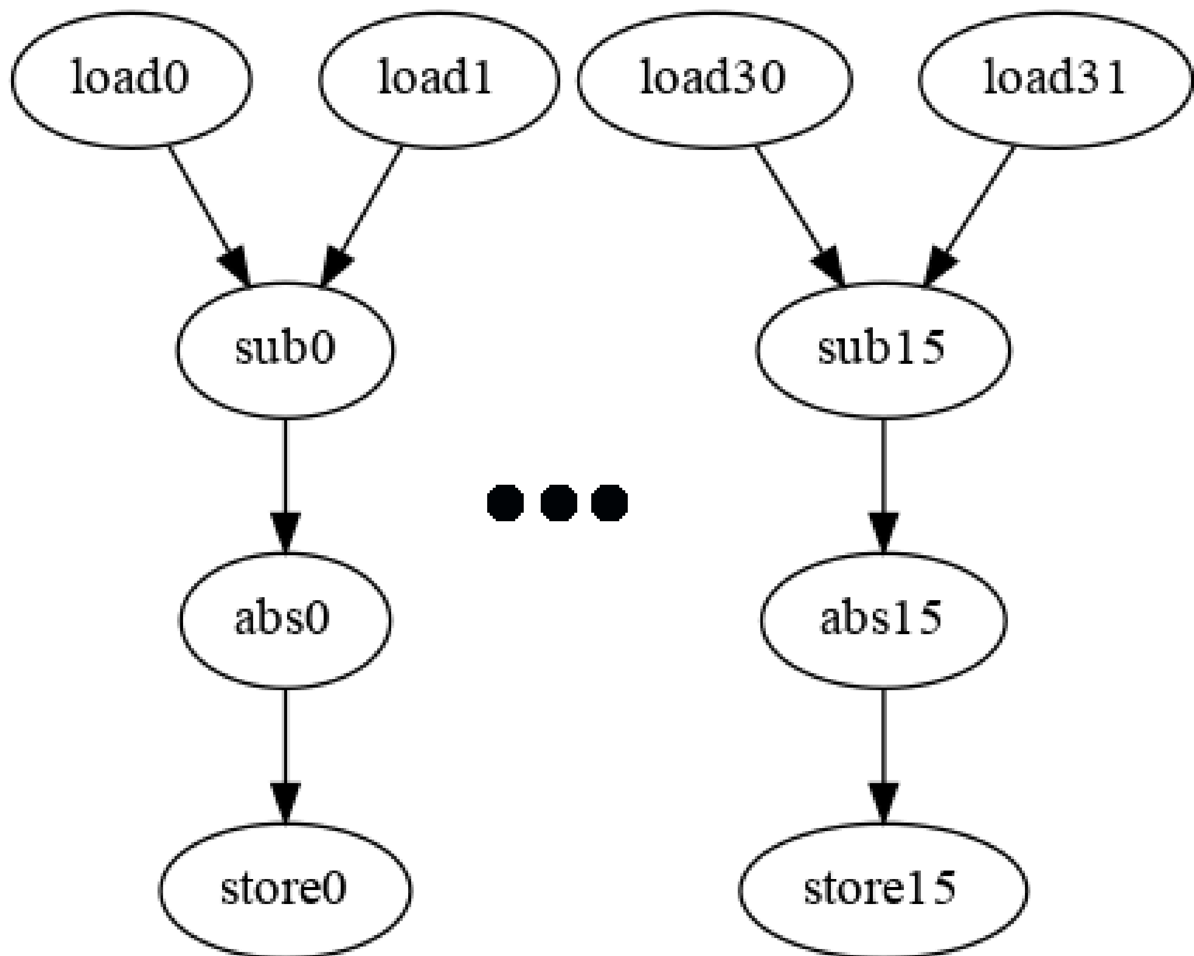
Figura A.1: Grafo de Representação Intermediária para o Benchmark SubZip



Fonte: Elaborado pelo autor(a).

```
private static Editor createDesign() {
    Add add = new Add(false);
    Editor editor = add.getMainEditor();
    SyncIn LOAD_load0 = new SyncIn();
    LOAD_load0.setName("load0");
    LOAD_load0.setAfuId(0);
    Util.createComponent(LOAD_load0);
    SyncSub SUB_sub0 = new SyncSub();
    SUB_sub0.setName("sub0");
```

Figura A.2: Grafo Final de Representação Intermediária para o caso de teste SubZip



Fonte: Elaborado pelo autor(a).

```
SUB_sub0.setAfuId(0);
Util.createComponent(SUB_sub0);
SyncIn LOAD_load1 = new SyncIn();
LOAD_load1.setName("load1");
LOAD_load1.setAfuId(0);
Util.createComponent(LOAD_load1);
SyncAbs ABS_abs0 = new SyncAbs();
ABS_abs0.setName("abs0");
ABS_abs0.setAfuId(0);
Util.createComponent(ABS_abs0);
SyncOut STORE_store0 = new SyncOut();
STORE_store0.setName("store0");
STORE_store0.setAfuId(0);
Util.createComponent(STORE_store0);
LOAD_load0.connectTo(SUB_sub0,"din0");
```

```
        SUB_sub0.connectTo(ABS_abs0,"din0");
        LOAD_load1.connectTo(SUB_sub0,"din1");
        ABS_abs0.connectTo(STORE_store0,"din0");
        Clock clk = Util.checkClkComponent();
        clk.setPeriod(0.5);
        Util.connectClkWire();
        Util.editorRedraw();
        return editor;
}
```

Código A.2: SubZip código que cria os componentes no Simulador. Esse código foi gerado pela nossa solução e não está replicado por simplicidade.

```
        private static GenArray<Integer> zipMap(GenArray<Pair> input){
                AfuManagerSimul afuManager = new AfuManagerSimul(
                    ↪ createDesign(), true);
            afuManager.getMainEditor().doZoomFit();

            AFUSimul afu0 = afuManager.getAFU(0);

            Pair<Integer, Integer>[] aux = new Pair[input.getLength()];
            input.toJavaArray(aux);

            Integer[] input1 = new Integer[aux.length];
            Integer[] input2 = new Integer[aux.length];

            for(int i = 0; i < aux.length; i++) {
                input1[i] = aux[i].getE1();
                input2[i] = aux[i].getE2();
            }

            int[] primitiveArray_input1 = new int[aux.length];
            int[] primitiveArray_input2 = new int[aux.length];

            int qtdeDataIn = aux.length;
            int qtdeDataOut = 2;

            int qtdeIn = afu0.getNumInputBuffer();
            int qtdeOut = afu0.getNumOutputBuffer();
```

```
        for (int i = 0; i < aux.length; i++) {
            primitiveArray_input1[i] = input1[i];
            primitiveArray_input2[i] = input2[i];
        }

        afu0.createInputBufferSW(0, qtdeDataIn, primitiveArray_input1)
            ↪ ;
        afu0.createInputBufferSW(1, qtdeDataIn, primitiveArray_input2)
            ↪ ;

        for (int j = 0; j < qtdeOut; j++) afu0.createOutputBufferSW(j,
            ↪  qtdeDataOut);
        afu0.start();
        afu0.waitDone(40000);

    return new GenArray<Integer>(IntStream.of( afu0.getOutputBuffer(0)
        ↪  ).boxed().toArray( Integer[]::new ));
    }
```

Código A.3: SubZip código que faz a transferência de dados e a execução através do Acelerador. Nesse exemplo é uma chamada para o simulador.

A.2 SimSearch

O segundo exemplo será o SimSearch. Dado um vetor de números inteiros, o algoritmo irá buscar por todos os números mais próximos de uma requisição dado uma distância máxima. Por exemplo, dado o vetor [0,1,2,3,4,5], e uma requisição por 3 e distância 1 o resultado será [2,3,4].

O algoritmo irá calcular a distância de todos os números utilizando a função *Map*. Logo em seguida são filtradas, utilizando *Filter*, todas as distâncias menores ou iguais a distância máxima. Os índices que foram selecionados pelo filtro constroem o vetor de saída do nosso algoritmo.

Código A.4: Código SimSearch

```
public static GenArray<Integer> Search(GenArray<Integer> d, int Q, int
    ↪ q_r) {
```

```
        // Map used to calculate the distance between each
          ↪ element in d and
        // the query q:
        GenArray<Integer> d_map = d.map(new Map<Integer, Integer
          ↪ >() {
          @Override
          public Integer function(Integer element) {
            int Q_less_element, abs;
            Q_less_element = Q - element;
            abs = Math.abs(Q_less_element);
            return abs; //need to apply absolute value.
          }
        }).filter(new Filter<Integer>() {
                  @Override
                  public boolean function(Integer element) {
                    boolean compare_return;
                    compare_return = element < q_r;
                    return compare_return;
                  }
        });

        return d_map;
}
```

Nesse método Search podemos ver o GenArray<Integer> que é o vetor onde irá ocorrer a busca. A requisição Q e a distância máxima q_r.

O diferencial dessa função é a demonstração do método *Filter*. Outro fator importante nesse código é a sobreposição das funções *Map* e *Filter*. Quando é escrito dessa maneira, caso seja possível, o compilador irá transformar as duas funções em um único dataflow, evitando uma transferência de dados desnecessária entre a FPGA e o Host. Nesse caso, a saída do dataflow de Map é enviada direto para o dataflow de Filter, dentro da FGPA.

Abaixo temos a representação intermediária final, após a união dos dois dataflows. O leitor pode perceber que a função Map termina exatamente após o componente abs0, que calcula o valor absoluto da entrada. Logo em seguida temos a operação de Filter. A nossa função Filter, dentro do dataflow consistem em um multiplexador final que decide qual será a saída. No nosso caso a saída pode ser o valor absoluto calculado pela função Map ou pode ser -1. A função controle do multiplexador é o caminho que passa pelo componente Set Less Than (slt0) e Branch Equals (beq0). O componente slt0 define

sua saída como 1 caso o seu valor de entrada for menor que o valor definido por q_r. O outro segmento do dataflow, que passa pelos eegistradores apenas replica o valor até o multiplexador. O segmento do dataflow, que passa pelos dois subtratores serve apenas para gerar o valor -1. Perceba que o primeiro subtrator faz uma subtração de dois valores iguais, gerando 0. O subtrator sequente decrementa em 1 esse valor gerando o -1.

Nesse caso de teste também é possível fazer a etapa de *Parallelization*. Podemos multiplicar esse dataflow por 32x. Basta notar que existe exatamente uma única entrada para uma única saída. Lembrando que podemos transferir 128 bytes por ciclo, que são exatamente 32 entradas.

Por fim iremos mostrar o código final gerado, após a representação intermediária. Assim como no SubZip o código é gerado, primeiramente, para o simulador Hades para testes e depuração antes de gerar código para a FPGA.

A implementação do algoritmo para o Simulador Hades para o SimSearch é feita dinamicamente através do código A.5. Note que, por simplicidade e melhor exibição, foi exibido código gerado sem a etapa de *Parallelization*.

```
private static Editor createDesign(int Q, int q_r) {
        Add add = new Add(false);
                Editor editor = add.getMainEditor();
        SyncIn LOAD_load0 = new SyncIn();
        LOAD_load0.setName("load0");
        LOAD_load0.setAfuId(0);
        Util.createComponent(LOAD_load0);
        SyncSubI SUB_sub0 = new SyncSubI();
        SUB_sub0.setName("sub0");
        SUB_sub0.setAfuId(0);
        SUB_sub0.setImmediate(Q);
        Util.createComponent(SUB_sub0);
        SyncAbs ABS_abs0 = new SyncAbs();
        ABS_abs0.setName("abs0");
        ABS_abs0.setAfuId(0);
        Util.createComponent(ABS_abs0);
        SyncBeqI BEQ_beq0 = new SyncBeqI();
        BEQ_beq0.setName("beq0");
        BEQ_beq0.setAfuId(0);
        BEQ_beq0.setImmediate(1);
        Util.createComponent(BEQ_beq0);
        SyncSltI SLT_slt0 = new SyncSltI();
        SLT_slt0.setName("slt0");
        SLT_slt0.setAfuId(0);
```

```
SLT_slt0.setImmediate(q_r);
Util.createComponent(SLT_slt0);
SyncOut STORE_store1 = new SyncOut();
STORE_store1.setName("store1");
STORE_store1.setAfuId(0);
Util.createComponent(STORE_store1);
SyncMux MUX_mux0 = new SyncMux();
MUX_mux0.setName("mux0");
MUX_mux0.setAfuId(0);
Util.createComponent(MUX_mux0);
SyncRegister REG_register1 = new SyncRegister();
REG_register1.setName("register1");
REG_register1.setAfuId(0);
Util.createComponent(REG_register1);
SyncRegister REG_register0 = new SyncRegister();
REG_register0.setName("register0");
REG_register0.setAfuId(0);
Util.createComponent(REG_register0);
SyncSubI SUB_sub2 = new SyncSubI();
SUB_sub2.setName("sub2");
SUB_sub2.setAfuId(0);
SUB_sub2.setImmediate(1);
Util.createComponent(SUB_sub2);
SyncSub SUB_sub1 = new SyncSub();
SUB_sub1.setName("sub1");
SUB_sub1.setAfuId(0);
Util.createComponent(SUB_sub1);
LOAD_load0.connectTo(SUB_sub0,"din0");
SUB_sub0.connectTo(ABS_abs0,"din0");
ABS_abs0.connectTo(SLT_slt0,"din0");
ABS_abs0.connectTo(REG_register0,"din0");
ABS_abs0.connectTo(SUB_sub1,"din0");
ABS_abs0.connectTo(SUB_sub1,"din1");
SLT_slt0.connectTo(BEQ_beq0,"din0");
BEQ_beq0.connectTo(MUX_mux0,"bin0");
MUX_mux0.connectTo(STORE_store1,"din0");
REG_register1.connectTo(MUX_mux0,"din0");
REG_register0.connectTo(REG_register1,"din0");
SUB_sub2.connectTo(MUX_mux0,"din1");
```

```
        SUB_sub1.connectTo(SUB_sub2,"din0");
        Clock clk = Util.checkClkComponent();
        clk.setPeriod(0.5);

        Util.connectClkWire();
        Util.editorRedraw();

        return editor;
}
```

Código A.5: SimSearch código que cria os componentes no Simulador. Esse código foi gerado pela nossa solução e não está replicado por simplicidade.

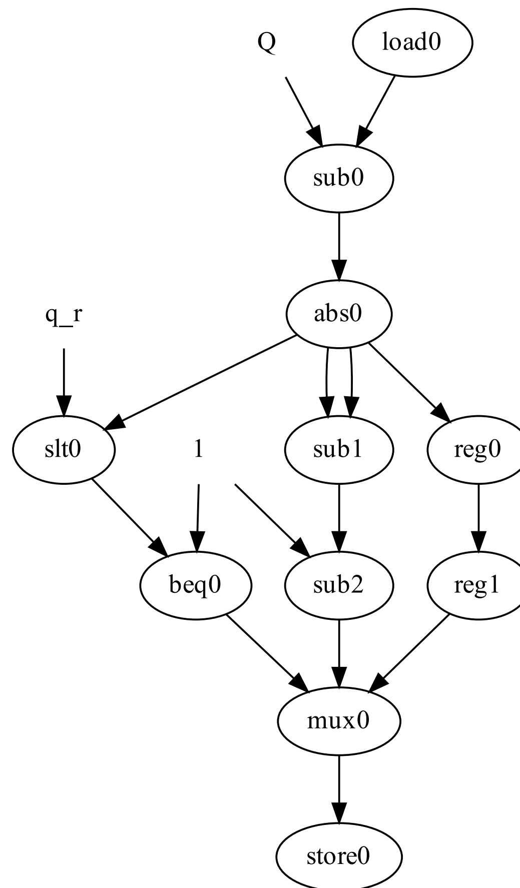
Além da representação no simulador Hades é gerado também o código que faz a transferência dos dados do usuário da JVM para o Acelerador. Logo em seguida também temos efetivamente a chamada do Acelerador. Quando dizemos aqui genericamente Acelerador é porque ele pode ser simulado ou não. Podemos ver o código de chamada de um Acelerador simulado em A.6. Para trocar de um Acelerador simulado para um Acelerador real utilizando FPGA basta alterar a classe AfuManagerSimul para AfuManager. Essa abstração acontece devido a biblioteca ADD.

```
private static GenArray<Integer> d_everything(GenArray<Integer> input,
    ↪ int Q, int q_r){
        AfuManagerSimul afuManager = new AfuManagerSimul(createDesign(Q,
            ↪ q_r), true);
        afuManager.getMainEditor().doZoomFit();

        AFUSimul afu0 = afuManager.getAFU(0);

        Integer[] aux = new Integer[input.getLength()];
        input.toJavaArray(aux);

        int[] primitiveArray = new int[aux.length];
        int qtdeDataIn = primitiveArray.length;
        int qtdeDataOut = qtdeDataIn;
        int qtdeIn = afu0.getNumInputBuffer();
        int qtdeOut = afu0.getNumOutputBuffer();
        for (int i = 0; i < aux.length; i++) primitiveArray[i] = aux[i];
                afu0.createInputBufferSW(0, qtdeDataIn, primitiveArray);
                for (int j = 0; j < qtdeOut; j++) afu0.createOutputBufferSW
                    ↪ (j, qtdeDataOut);
```

```
            afu0.start();
        afu0.waitDone(40000);
        return new GenArray<Integer>(IntStream.of( afu0.getOutputBuffer(0)
            ↪  ).boxed().toArray( Integer[]::new ));
}
```

Código A.6: SimSearch código que faz a transferência de dados e a execução através do Acelerador. Nesse exemplo é uma chamada para o simulador.

Figura A.3: Grafo de Representação Intermediária para o caso de teste SimSearch



Fonte: Elaborado pelo autor(a).

A.3 StringHash

Esse caso de teste calcula o valor Hash de uma String exatamente do mesmo jeito que o método padrão implementado pela classe String da JVM. Basicamente consiste em multiplicar cada caracter, da esquerda para a direita, pelas respectivas potências de 31 e somar todas as multiplicações ao final.

Código A.7: Código String Hash

```
public int hashCode() {
		if(this.hash == 0) {
			Integer[] multipliers = new Integer[this.value.
			   ↪ getLength()];
			int multiplier_base = 1;
			for(int n = multipliers.length; n >= 1; n--) {
				multipliers[n-1] = multiplier_base;
				multiplier_base *= 31;
			}

			GenArray<Integer> genArray_multipliers = new
			   ↪ GenArray<Integer>(multipliers);
			GenArray<Pair> zipped = this.value.zip(
			   ↪ genArray_multipliers);

			this.hash = zipped.map(new Map<Integer, Pair>() {
			   ↪
				@Override
				public Integer function(Pair element) {
					int retorno;
					retorno = (Integer) element.getE1() *
					   ↪  (Integer) element.getE2();
					return retorno;
				}

			}).reduce(new Reduce<Integer>() {
				@Override
				public Integer function(Integer element1,
				   ↪ Integer element2) {
					element1 = element1 + element2;
```

```
                                        return element1;
                                }

                        });
                }
                return this.hash;
        }
```

No código acima temos a função que calcula o hash da String utilizando a nossa solução escrita em ParallelME.

As potências de 31 são calculadas previamente e sequencialmente pela máquina hospedeira, como podemos ver no código.

É enviado para a FPGA um par com o valor do caracter como inteiro e sua respectiva potência de 31 para ser multiplicado. Vemos uma função Map que faz a multiplicação par a par.

Por fim, o resultado das multiplicações é somado utilizando o método Reduce.

Nesse caso de teste também ocorre a fusão das duas funções em único dataflow.

O diferencial desse caso de teste é a replicação de dataflows que contém a função Reduce.

O dataflow é replicado para maximizar a entrada de dados, portando é multiplicado 16 vezes para obtermos a entrada máxima por ciclo.

Porém, na etapa de replicação, iremos adicionar uma camada de somadores para cada dupla de multiplicações para fazer as reduções, como podemos ver na imagem.

Como resultado final teremos uma quantidade de camadas definida pelo resultado do logaritmo na base 2 das entradas, ou seja, 4 camadas de somadores até o somador acumulador final.

Tendo a Representação Intermediária final iremos finalmente gerar código. Em primeiro lugar é gerado código para o simulador Hades, onde o programador pode testar e depurar a corretude da sua solução e do código gerado pelo compilador.

A implementação do algoritmo para o Simulador Hades para o String Hash é feita dinamicamente através do código A.8. Note que, por simplicidade e melhorar a exibição, foi exibido código gerado sem a etapa de paralelização.

```
        private static Editor createDesign() {

                Add add = new Add(true);
                Editor editor = add.getMainEditor();
                SyncIn LOAD_load0 = new SyncIn();
                LOAD_load0.setName("load0");
                LOAD_load0.setAfuId(0);
```

```
                Util.createComponent(LOAD_load0);
                SyncMul MUL_mul0 = new SyncMul();
                MUL_mul0.setName("mul0");
                MUL_mul0.setAfuId(0);
                Util.createComponent(MUL_mul0);
                SyncIn LOAD_load1 = new SyncIn();
                LOAD_load1.setName("load1");
                LOAD_load1.setAfuId(0);
                Util.createComponent(LOAD_load1);
                SyncOut STORE_store1 = new SyncOut();
                STORE_store1.setName("store1");
                STORE_store1.setAfuId(0);
                Util.createComponent(STORE_store1);
                SyncAccAdd ACC_acc0 = new SyncAccAdd();
                ACC_acc0.setName("acc0");
                ACC_acc0.setAfuId(0);
                ACC_acc0.setImmediate(-1);
                Util.createComponent(ACC_acc0);
                LOAD_load0.connectTo(MUL_mul0,"din0");
                MUL_mul0.connectTo(ACC_acc0,"din0");
                LOAD_load1.connectTo(MUL_mul0,"din1");
                ACC_acc0.connectTo(STORE_store1,"din0");
                Clock clk = Util.checkClkComponent();
                clk.setPeriod(0.5);

                Util.connectClkWire();
                Util.editorRedraw();

                return editor;
```

Código A.8: String Hash código que cria os componentes no Simulador. Esse código foi gerado pela nossa solução e não está replicado por simplicidade.

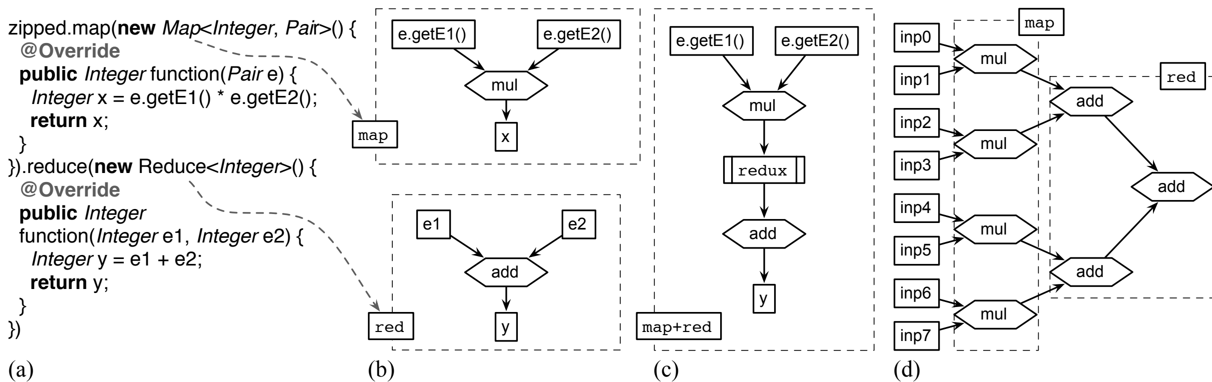
Além da representação no simulador Hades é gerado também o código que faz a transferência dos dados do usuário da JVM para o Acelerador. Logo em seguida também temos efetivamente a chamada do Acelerador. Quando dizemos aqui genericamente Acelerador é porque ele pode ser simulado ou não. Podemos ver o código de chamada de um Acelerador simulado em A.9. Para trocar de um Acelerador simulado para um Acelerador real utilizando FPGA basta alterar a classe AfuManagerSimul para AfuManager. Essa abstração acontece devido a biblioteca ADD.

```
private static Integer zippedMapFilter(GenArray<Pair> input){
        AfuManagerSimul afuManager = new AfuManagerSimul(createDesign(),
            ↪ true);
        afuManager.getMainEditor().doZoomFit();

        AFUSimul afu0 = afuManager.getAFU(0);

        Pair<Integer, Integer>[] aux = new Pair[input.getLength()];
        input.toJavaArray(aux);

        Integer[] input1 = new Integer[aux.length];
        Integer[] input2 = new Integer[aux.length];

        for(int i = 0; i < aux.length; i++) {
                input1[i] = aux[i].getE1();
                input2[i] = aux[i].getE2();
        }

        int[] primitiveArray_input1 = new int[aux.length];
        int[] primitiveArray_input2 = new int[aux.length];

        int qtdeDataIn = aux.length;
        int qtdeDataOut = 1;

        int qtdeIn = afu0.getNumInputBuffer();
        int qtdeOut = afu0.getNumOutputBuffer();

        for (int i = 0; i < aux.length; i++) {
                primitiveArray_input1[i] = input1[i];
                primitiveArray_input2[i] = input2[i];
        }

        afu0.createInputBufferSW(0, qtdeDataIn, primitiveArray_input1);
        afu0.createInputBufferSW(1, qtdeDataIn, primitiveArray_input2);

        for (int j = 0; j < qtdeOut; j++) afu0.createOutputBufferSW(j,
            ↪ qtdeDataOut);
        afu0.start();
```

```
        afu0.waitDone(40000);

        return afu0.getOutputBuffer(0)[0];
}
```

Código A.9: String Hash código que faz a transferência de dados e a execução através do Acelerador. Nesse exemplo é uma chamada para o simulador.

Figura A.4: (a) Código do StringHash (b) Dataflow para cada função (c) Super grafo contendo a fusão das duas funções (d) Dataflow replicado



Fonte: Elaborado pelo autor(a).

A.4 RGB2YUV

Esse caso de teste transforma uma imagem que está no formato RGB para o formato YUV. Constitui apenas de uma função Map que faz transformações nas cores de cada pixel. Apesar de utilizar apenas a função Map esse é o caso de teste que mais possui componentes e que reutiliza operações.

Código A.10: Código RGB2YUV

```
                GenArray<Triple> image_array = new GenArray<Triple>(
                    ↪ primitive_image);
                image_array = image_array.map(new Map<Triple, Triple
                    ↪ >(){

                        @Override
                        public Triple function(Triple element) {
```

```
int Y, U, V, y1, y2, y3, u1, u2, u3,
    ↪ v1, v2, v3;

y1 = 9798 * (Integer) element.getE1()
    ↪ ;
u1 = 21208 * (Integer) element.getE1
    ↪ ();
v1 = -4784 * (Integer) element.getE1
    ↪ ();

y2 = 19235 * (Integer) element.getE2
    ↪ ();
u2 = 16941 * (Integer) element.getE2
    ↪ ();
v2 = 9437 * (Integer) element.getE2()
    ↪ ;

y3 = 3736 * (Integer) element.getE3()
    ↪ ;
u3 = 3277 * (Integer) element.getE3()
    ↪ ;
v3 = 4221 * (Integer) element.getE3()
    ↪ ;

Y = y1 + y2;
Y = Y + y3;
Y = Y >> 15;

U = u1 + u2;
U = U + u3;
U = U >> 15;
U = U + 128;

V = v1 + v2;
V = V + v3;
V = V >> 15;
V = V + 128;

return new Triple(Y,U,V);
```

```
                        }

                 });
```

No código acima temos a função que faz as transformações necessárias para converter um pixel em RGB para YUV. Por simplicidade, o código que faz a leitura da arquivo imagem em RGB foi omitido. Outro diferencial desse caso de teste é que utilizamos GenArray<Triple> para salvar os valores, seja RGB ou YUV do Pixel.

O dataflow é replicado para maximizar a entrada de dados, portanto é multiplicado 10 vezes. Não é possível alcançar o valor máximo de 32 entradas pela característica do problema de receber 3 entradas por pixel.

Tendo a Representação Intermediária final iremos finalmente gerar código. Em primeiro lugar é gerado código para o simulador Hades, onde o programador pode testar e depurar a corretude da sua solução e do código gerado pelo compilador.

A implementação do algoritmo para o Simulador Hades para o RGB2YUV é feita dinamicamente através do código A.11. Note que, por simplicidade e melhorar a exibição, foi exibido código gerado sem a etapa de paralelização.

```
private static Editor createDesign() {
       Add add = new Add(false);
       Editor editor = add.getMainEditor();
       SyncIn LOAD_load0 = new SyncIn();
       LOAD_load0.setName("load0");
       LOAD_load0.setAfuId(0);
       Util.createComponent(LOAD_load0);
       SyncMulI MUL_mul0 = new SyncMulI();
       MUL_mul0.setName("mul0");
       MUL_mul0.setAfuId(0);
       MUL_mul0.setImmediate(9798);
       Util.createComponent(MUL_mul0);
       SyncMulI MUL_mul1 = new SyncMulI();
       MUL_mul1.setName("mul1");
       MUL_mul1.setAfuId(0);
       MUL_mul1.setImmediate(21208);
       Util.createComponent(MUL_mul1);
       SyncMulI MUL_mul2 = new SyncMulI();
       MUL_mul2.setName("mul2");
       MUL_mul2.setAfuId(0);
       MUL_mul2.setImmediate(4784);
       Util.createComponent(MUL_mul2);
```

```
SyncIn LOAD_load1 = new SyncIn();
LOAD_load1.setName("load1");
LOAD_load1.setAfuId(0);
Util.createComponent(LOAD_load1);
SyncMulI MUL_mul3 = new SyncMulI();
MUL_mul3.setName("mul3");
MUL_mul3.setAfuId(0);
MUL_mul3.setImmediate(19235);
Util.createComponent(MUL_mul3);
SyncMulI MUL_mul4 = new SyncMulI();
MUL_mul4.setName("mul4");
MUL_mul4.setAfuId(0);
MUL_mul4.setImmediate(16941);
Util.createComponent(MUL_mul4);
SyncMulI MUL_mul5 = new SyncMulI();
MUL_mul5.setName("mul5");
MUL_mul5.setAfuId(0);
MUL_mul5.setImmediate(9437);
Util.createComponent(MUL_mul5);
SyncIn LOAD_load2 = new SyncIn();
LOAD_load2.setName("load2");
LOAD_load2.setAfuId(0);
Util.createComponent(LOAD_load2);
SyncMulI MUL_mul6 = new SyncMulI();
MUL_mul6.setName("mul6");
MUL_mul6.setAfuId(0);
MUL_mul6.setImmediate(3736);
Util.createComponent(MUL_mul6);
SyncMulI MUL_mul7 = new SyncMulI();
MUL_mul7.setName("mul7");
MUL_mul7.setAfuId(0);
MUL_mul7.setImmediate(3277);
Util.createComponent(MUL_mul7);
SyncMulI MUL_mul8 = new SyncMulI();
MUL_mul8.setName("mul8");
MUL_mul8.setAfuId(0);
MUL_mul8.setImmediate(4221);
Util.createComponent(MUL_mul8);
SyncAdd SUM_sum0 = new SyncAdd();
```

```
SUM_sum0.setName("sum0");
SUM_sum0.setAfuId(0);
Util.createComponent(SUM_sum0);
SyncAdd SUM_sum1 = new SyncAdd();
SUM_sum1.setName("sum1");
SUM_sum1.setAfuId(0);
Util.createComponent(SUM_sum1);
SyncShrI SHR_shr0 = new SyncShrI();
SHR_shr0.setName("shr0");
SHR_shr0.setAfuId(0);
SHR_shr0.setImmediate(15);
Util.createComponent(SHR_shr0);
SyncAdd SUM_sum2 = new SyncAdd();
SUM_sum2.setName("sum2");
SUM_sum2.setAfuId(0);
Util.createComponent(SUM_sum2);
SyncAdd SUM_sum3 = new SyncAdd();
SUM_sum3.setName("sum3");
SUM_sum3.setAfuId(0);
Util.createComponent(SUM_sum3);
SyncShrI SHR_shr1 = new SyncShrI();
SHR_shr1.setName("shr1");
SHR_shr1.setAfuId(0);
SHR_shr1.setImmediate(15);
Util.createComponent(SHR_shr1);
SyncAddI SUM_sum4 = new SyncAddI();
SUM_sum4.setName("sum4");
SUM_sum4.setAfuId(0);
SUM_sum4.setImmediate(128);
Util.createComponent(SUM_sum4);
SyncAdd SUM_sum5 = new SyncAdd();
SUM_sum5.setName("sum5");
SUM_sum5.setAfuId(0);
Util.createComponent(SUM_sum5);
SyncAdd SUM_sum6 = new SyncAdd();
SUM_sum6.setName("sum6");
SUM_sum6.setAfuId(0);
Util.createComponent(SUM_sum6);
SyncShrI SHR_shr2 = new SyncShrI();
```

```
SHR_shr2.setName("shr2");
SHR_shr2.setAfuId(0);
SHR_shr2.setImmediate(15);
Util.createComponent(SHR_shr2);
SyncAddI SUM_sum7 = new SyncAddI();
SUM_sum7.setName("sum7");
SUM_sum7.setAfuId(0);
SUM_sum7.setImmediate(128);
Util.createComponent(SUM_sum7);
SyncOut STORE_store0 = new SyncOut();
STORE_store0.setName("store0");
STORE_store0.setAfuId(0);
Util.createComponent(STORE_store0);
SyncOut STORE_store1 = new SyncOut();
STORE_store1.setName("store1");
STORE_store1.setAfuId(0);
Util.createComponent(STORE_store1);
SyncOut STORE_store2 = new SyncOut();
STORE_store2.setName("store2");
STORE_store2.setAfuId(0);
Util.createComponent(STORE_store2);
SyncRegister REG_register0 = new SyncRegister();
REG_register0.setName("register0");
REG_register0.setAfuId(0);
Util.createComponent(REG_register0);
SyncRegister REG_register1 = new SyncRegister();
REG_register1.setName("register1");
REG_register1.setAfuId(0);
Util.createComponent(REG_register1);
SyncRegister REG_register2 = new SyncRegister();
REG_register2.setName("register2");
REG_register2.setAfuId(0);
Util.createComponent(REG_register2);
LOAD_load0.connectTo(MUL_mul0,"din0");
LOAD_load0.connectTo(MUL_mul1,"din0");
LOAD_load0.connectTo(MUL_mul2,"din0");
MUL_mul0.connectTo(SUM_sum0,"din0");
MUL_mul1.connectTo(SUM_sum2,"din0");
MUL_mul2.connectTo(SUM_sum5,"din0");
```

```
        LOAD_load1.connectTo(MUL_mul3,"din0");
        LOAD_load1.connectTo(MUL_mul4,"din0");
        LOAD_load1.connectTo(MUL_mul5,"din0");
        MUL_mul3.connectTo(SUM_sum0,"din1");
        MUL_mul4.connectTo(SUM_sum2,"din1");
        MUL_mul5.connectTo(SUM_sum5,"din1");
        LOAD_load2.connectTo(MUL_mul6,"din0");
        LOAD_load2.connectTo(MUL_mul7,"din0");
        LOAD_load2.connectTo(MUL_mul8,"din0");
        MUL_mul6.connectTo(REG_register0,"din0");
        MUL_mul7.connectTo(REG_register1,"din0");
        MUL_mul8.connectTo(REG_register2,"din0");
        SUM_sum0.connectTo(SUM_sum1,"din0");
        SUM_sum1.connectTo(SHR_shr0,"din0");
        SHR_shr0.connectTo(STORE_store0,"din0");
        SUM_sum2.connectTo(SUM_sum3,"din0");
        SUM_sum3.connectTo(SHR_shr1,"din0");
        SHR_shr1.connectTo(SUM_sum4,"din0");
        SUM_sum4.connectTo(STORE_store1,"din0");
        SUM_sum5.connectTo(SUM_sum6,"din0");
        SUM_sum6.connectTo(SHR_shr2,"din0");
        SHR_shr2.connectTo(SUM_sum7,"din0");
        SUM_sum7.connectTo(STORE_store2,"din0");
        REG_register0.connectTo(SUM_sum1,"din1");
        REG_register1.connectTo(SUM_sum3,"din1");
        REG_register2.connectTo(SUM_sum6,"din1");
        Clock clk = Util.checkClkComponent();
        clk.setPeriod(0.5);

        Util.connectClkWire();
        Util.editorRedraw();

        return editor;
}
```

Código A.11: RGB2YUV código que cria os componentes no Simulador. Esse código foi gerado pela nossa solução e não está replicado por simplicidade.

Além da representação no simulador Hades é gerado também o código que faz a transferência dos dados do usuário da JVM para o Acelerador. Logo em seguida também

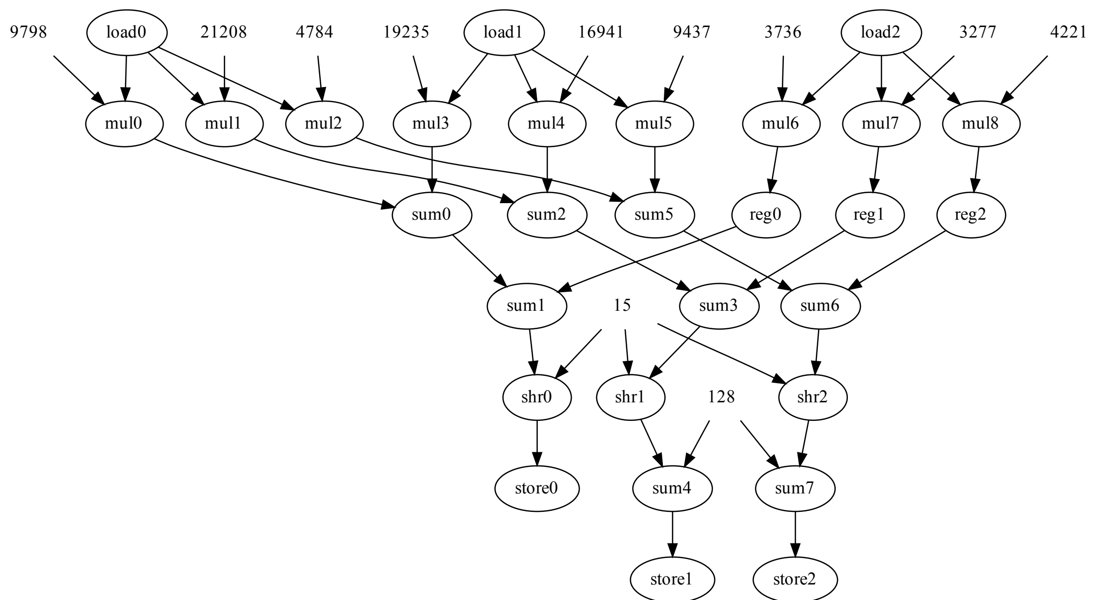
temos efetivamente a chamada do Acelerador. Quando dizemos aqui genericamente Acelerador é porque ele pode ser simulado ou não. Podemos ver o código de chamada de um Acelerador simulado em A.12. Para trocar de um Acelerador simulado para um Acelerador real utilizando FPGA basta alterar a classe AfuManagerSimul para AfuManager. Essa abstração acontece devido a biblioteca ADD.

```
private static GenArray<Triple> image_arrayMap(GenArray<Triple> input){
        AfuManagerSimul afuManager = new AfuManagerSimul(createDesign(),
           ↪ true);
        afuManager.getMainEditor().doZoomFit();

        AFUSimul afu0 = afuManager.getAFU(0);

        Triple<Integer>[] aux = new Triple[input.getLength()];
        input.toJavaArray(aux);

        Integer[] input1 = new Integer[aux.length];
        Integer[] input2 = new Integer[aux.length];
        Integer[] input3 = new Integer[aux.length];

        for(int i = 0; i < aux.length; i++) {
                input1[i] = aux[i].getE1();
                input2[i] = aux[i].getE2();
                input3[i] = aux[i].getE3();
        }

        int[] primitiveArray_input1 = new int[aux.length];
        int[] primitiveArray_input2 = new int[aux.length];
        int[] primitiveArray_input3 = new int[aux.length];



        int qtdeDataIn = aux.length;
        int qtdeDataOut = qtdeDataIn;

        int qtdeIn = afu0.getNumInputBuffer();
        int qtdeOut = afu0.getNumOutputBuffer();

        for (int i = 0; i < aux.length; i++) {
                primitiveArray_input1[i] = input1[i];
                primitiveArray_input2[i] = input2[i];
```

```
                primitiveArray_input3[i] = input3[i];
        }

        afu0.createInputBufferSW(0, qtdeDataIn, primitiveArray_input1);
        afu0.createInputBufferSW(1, qtdeDataIn, primitiveArray_input2);
        afu0.createInputBufferSW(2, qtdeDataIn, primitiveArray_input3);

        for (int j = 0; j < qtdeOut; j++) {
                afu0.createOutputBufferSW(j, qtdeDataOut);
        }

        afu0.start();
        afu0.waitDone(40000);

        Triple<Integer>[] output = new Triple[aux.length];

        Integer[] output1 = IntStream.of( afu0.getOutputBuffer(0) ).boxed
            ↪ ().toArray( Integer[]::new );
        Integer[] output2 = IntStream.of( afu0.getOutputBuffer(1) ).boxed
            ↪ ().toArray( Integer[]::new );
        Integer[] output3 = IntStream.of( afu0.getOutputBuffer(2) ).boxed
            ↪ ().toArray( Integer[]::new );

        for(int i = 0; i < aux.length; i++) {
                output[i] = new Triple(output1[i], output2[i], output3[i]);
        }

        return new GenArray<Triple>(output);
}
```

Código A.12: RGB2YUV código que faz a transferência de dados e a execução através do Acelerador. Nesse exemplo é uma chamada para o simulador.

Figura A.5: Representação Intermediária de RGB2YUV



Fonte: Elaborado pelo autor(a).