LUCAS BRAGANÇA DA SILVA

# FERRAMENTAS E NOVAS ARQUITETURAS PARA ACELERADORES RECONFIGURÁVEIS EM PLATAFORMAS HETEROGÊNEAS CPU-FPGA COM MEMÓRIA COMPARTILHADA

Dissertação apresentada a Universidade Federal de Viçosa, como parte das exigências do Programa de Pós-Graduação em Ciência da Computação, para obtenção do título de *Magister Scientiae*.

VIÇOSA
MINAS GERAIS - BRASIL
2019

**Ficha catalográfica preparada pela Biblioteca Central da Universidade Federal de Viçosa - Câmpus Viçosa**

T
S586f
2019

Silva, Lucas Bragança da, 1989-
Ferramentas e novas arquiteturas para aceleradores reconfiguráveis em plataformas heterogêneas CPU-FPGA com memória compartilhada : . / Lucas Bragança da Silva. – Viçosa, MG, 2019.
x, 79 f. : il. (algumas color.) ; 29 cm.

Inclui apêndices.
Orientador: José Augusto Miranda Nacif.
Dissertação (mestrado) - Universidade Federal de Viçosa.
Referências bibliográficas: f. 62-71.

1. Arquitetura de computador. 2. Computação heterogênea. I. Universidade Federal de Viçosa. Departamento de Informática. Programa de Pós-Graduação em Ciência da Computação. II. Título.

CDD 22. ed. 004.22

**LUCAS BRAGANÇA DA SILVA**

# FERRAMENTAS E NOVAS ARQUITETURAS PARA ACELERADORES RECONFIGURÁVEIS EM PLATAFORMAS HETEROGÊNEAS CPU-FPGA COM MEMÓRIA COMPARTILHADA

Dissertação apresentada à Universidade Federal de Viçosa, como parte das exigências do Programa de Pós-Graduação em Ciência da Computação, para obtenção do título de *Magister Scientiae*.

APROVADA: 25 de fevereiro de 2019.

Assentimento:

Lucas Bragança da Silva
Autor

José Augusto Miranda Nacif
Orientador

*Dedico este trabalho aos meus pais, amigos e à minha noiva Vívian.*

# Agradecimentos

Aos meus pais Veronice e Abinael pelo total apoio e carinho.
Aos meus amigos por sempre estarem ao meu lado nessa longa caminhada.
A minha noiva e futura esposa Vívian Helene pelos bons conselhos e pela paciência durante todo o período do mestrado.
À todos meus professores por me guiarem pelo caminho do conhecimento.
À CAPES pelo apoio financeiro.
À Universidade Federal de Viçosa por todas as oportunidades.
Ao Professor José Augusto Miranda Nacif pela orientação, confiança e paciência.
Ao Professor Ricardo dos Santos Ferreira pela disponibilidade, atenção e orientação.

# Sumário

# Lista de Figuras

# Lista de Tabelas

# Resumo

SILVA, Lucas Bragança, M.Sc., Universidade Federal de Viçosa, fevereiro de 2019. **Ferramentas e Novas Arquiteturas para Aceleradores Reconfiguráveis em Plataformas Heterogêneas CPU-FPGA com Memória Compartilhada**. Orientador: José Augusto Miranda Nacif. Coorientador: Ricardo dos Santos Ferreira.

O uso de arquiteturas reconfiguráveis como aceleradores em sistemas heterogêneos de alto desempenho surgiu como uma opção complementar a outras soluções, como por exemplo os processadores gráficos ou GPUs. Arquiteturas reconfiguráveis como FPGAs possuem como principal vantagem o paralelismo intrínseco de sua estrutura, que pode se adaptar à aplicação para alcançar o máximo desempenho com baixo consumo energético. Porém, faltam ferramentas para popularizar o uso dos FPGAs como aceleradores, reduzir o tempo de projeto, sem exigir dos programadores conhecimentos aprofundados no desenvolvimento de *hardware*. Nesta dissertação são apresentados três trabalhos que permitem a geração automática de código de aceleradores para sistemas heterogêneos com FPGA, além de propor uma nova arquitetura reconfigurável, que permite a execução no modelo MIMT (*Multiple Instruction Multiple Threads*) com troca de contexto em um ciclo de relógio. O primeiro trabalho apresenta uma ferramenta capaz de gerar automaticamente todo código de um acelerador para auxiliar na exploração de Redes Reguladoras de Genes, esta abordagem obteve um ganho de desempenho de duas ordens de grandeza em comparação com processador de uso geral. O segundo trabalho apresenta um arcabouço para facilitar a implantação de aceleradores em FPGA, deixando para o desenvolvedor apenas a implementação da aplicação, sem a necessidade de codificar os mecanismos de controle de comunicação entre o acelerador e a aplicação executada em *software*. Por fim, no terceiro trabalho é proposta uma nova arquitetura reconfigurável que permite a execução de múltiplas *threads*, fazendo o uso de paralelismo temporal e espacial para acelerar aplicações descritas na forma de grafos de fluxo de dados.

# Abstract

SILVA, Lucas Bragança, M.Sc., Universidade Federal de Viçosa, February, 2019. **Tools and New Architectures for Reconfigurable Accelerators in Heterogeneous Platforms CPU-FPGA with Shared Memory**. Adviser: José Augusto Miranda Nacif. Co-adivser: Ricardo dos Santos Ferreira.

The use of reconfigurable architectures as accelerators in heterogeneous high performance systems has emerged as a complementary option to other solutions such as graphics processors or GPUs. Reconfigurable architectures like FPGAs have as main advantages the intrinsic parallelism of their structure that can be used to adapt the application to reach the maximum performance with low energy consumption. However, there is a lack of tools to popularize the use of FPGAs as accelerators to reduce design time, without requiring of programmers to have in-depth knowledge of hardware development. This dissertation presents three works that allow the automatic generation of accelerator code for heterogeneous systems with FPGA. Therefore, we propose a new reconfigurable architecture that allows execution in the MIMT (Multiple Instruction Multiple Threads) model with context change in one clock cycle. The first work presents a tool to automatically generate all the code of an accelerator to assist in the exploration of Gene Regulatory Networks, this approach achieved a performance gain of two orders of magnitude compared to general-purpose processor.The second work presents a framework to facilitate the implementation of accelerators in FPGA, leaving to the developer only the implementation of the application without the need to code the communication control mechanisms between the accelerator and the application executed in software. The third work proposes a new reconfigurable architecture that allows the execution of multiple threads, using temporal and spatial parallelism to accelerate applications described in the form of data flow graphs.

# Capítulo 1

# Introdução

Recentes avanços na tecnologia de dispositivos programáveis como os FPGAs (*Field Programmable Gate Arrays*) possibilitaram o surgimento de novas formas de computação heterogênea: a Computação Heterogênea Reconfigurável (CHR). Novas plataformas apresentam os FPGAs acoplados a processadores com múltiplos núcleos por meio de barramentos de alta velocidade, permitindo assim explorar o potencial de uso de aceleradores reconfiguráveis em sistemas heterogêneos. Grandes empresas como Intel, Microsoft e Amazon estão investindo em pesquisas de sistemas heterogêneos com FPGAs [2, 3, 4]. Todas essas plataformas são concebidas para processamento de grandes volumes de dados e têm como alvo serviços *web* oferecidos na forma de servidores na nuvem.

A principal característica destas plataformas é a presença de uma CPU e um FPGA conectados por meio de um barramento de alta velocidade. Na plataforma Intel HARPv2 [2], o FPGA foi disposto no mesmo *chip* da CPU, deixando seu acesso à memória principal mais rápido (taxa de pico de 25 GB/s) ao utilizar três canais de comunicação. Além disso o FPGA é capaz acessar toda memória principal do sistema. As plataformas Catapult [3] e as instâncias F1 da Amazon [4] usam o barramento PCI para acoplar o FPGA, obtendo uma velocidade menor que o sistema da Intel. Porém para reduzir o impacto da velocidade do PCI, os FPGAs possuem memórias dedicadas, tornando o sistema semelhante as Unidades de Processamento Gráfico (GPUs) que possuem sua própria memória. Em relação aos fabricantes de FPGAs, os sistemas da Intel e Microsoft usam os FPGAs da Altera/Intel e a Amazon usa FPGAs da Xilinx [5]. A Tabela 1.1 apresenta detalhes sobre as principais plataformas disponíveis atualmente.

Estas plataformas têm sido usadas para acelerar uma grande quantidade de aplicações, sendo que em alguns casos o desempenho é superior às GPUs em pro-

**Tabela 1.1.** Principais plataformas heterogêneas CPU-FPGA

| Plataforma | HARPv2 | Catapult | Amazon F1 |
|---|---|---|---|
| CPU do sistema | Intel Xeon E5-2680 @2.4GHz | Intel Xeon E5-2697 @2.70 GHz | Intel Xeon E5-2686 v4 @2.3 GHz |
| FPGA do sistema | Altera Arria 10 @400MHz | Altera Stratix V @200MHz | Xilinx UltraScale+ VU9P @400MHz |
| Barramento de comunicação | Intel QPI, 2xPCIe x8 ∼28.8 GB/s | PCIe x8 ∼8GB/s | PCIe x16 ∼16GB/s |
| Memória acoplada FPGA | N/A | 8GB DDR3 | 64GB DDR4 |

cessamento. Em [3] é mostrado a plataforma da Microsoft Catapult com um estudo de caso, no qual, o serviço de buscas *Bing* foi acelerado usando o FPGA. Essa abordagem foi capaz de obter 95% de ganho no tempo de resposta de pesquisas com eficiência energética. Em [6] a plataforma HARP foi usada para acelerar aplicações de redes neurais convolucionais, que mesmo não utilizando totalmente a capacidade do FPGA foi possível obter ganhos de até 2x em alguns casos.

Entretanto, o modelo de programação que permite desenvolver aplicações com melhor desempenho ainda limita a adoção em larga escala destas plataformas. Como forma de atacar este problema diversos grupos têm tentado aproximar o processo de desenvolvimento de aplicações para FPGAs ao dos ambientes de programação de alto desempenho como, por exemplo, os aceleradores desenvolvidos em CUDA ou OpenCL para executar em GPUs. Atualmente o processo de desenvolvimento de GPUs é bem semelhante ao desenvolvimento para CPUs. Desse modo é possível realizar várias compilações diárias, utilizar *softwares* para depuração e localização de erros, facilitando no processo de desenvolvimento. Atualmente as principais tecnologias utilizadas na programação de GPUs são: OpenCL, Cuda, OpenACC e OpenMP [7, 8, 9, 10]. OpenCL e Cuda são linguagens nas quais o usuário descreve os código das aplicações que serão executadas na GPU, assim o desenvolvedor possui maior controle na forma que a aplicação irá utilizar os recursos da arquitetura, permitindo implementações mais otimizadas. As tecnologias OpenMP e OpenACC, por outro lado têm o intuito de facilitar o reaproveitamento de aplicações já codificadas, inserindo diretivas de compilação em partes do código onde é possível paralelizar.

Para prover uma interface de programação semelhante ao que hoje é utilizado para GPUs em FPGAs, os principais fabricantes de FPGAs estão oferecendo o uso de OpenCL para o processo de desenvolvimento de aceleradores em um FPGA. Porém, mesmo com uma interface de programação de alto nível, o processo de desenvolvimento para FPGAs ainda não é o mais adequado quando se deseja obter um ciclo de desenvolvimento parecido com o das GPUs. Isso é devido principalmente ao fato que um desenvolvedor não precisa apenas conhecer o OpenCL, mas também é preciso

saber qual a melhor forma de escrever um código OpenCL que o compilador consiga interpretar como circuito lógico. Outro ponto é o código desenvolvido em alto nível não é diretamente compilado para a arquitetura de um FPGA. Primeiramente o código é traduzido para uma linguagem intermediária, normalmente o Verilog, para depois realizar o processo de compilação para o FPGA com as ferramentas de síntese já existentes.

Tornar o uso de CHR mais acessível tem se tornado um problema cada vez mais relevante entre os pesquisadores de *hardware* reconfigurável. Dentro deste contexto, esta dissertação apresenta três trabalhos na área de CHR, cujas contribuições são sumarizadas abaixo:

1. Ferramenta para geração automática de aceleradores para o problema de bioinformática de rede reguladoras de genes a partir de descrições em alto nível.

2. Ferramenta que permite a geração automática de código para gerenciar múltiplos aceleradores de uso geral.

3. Nova arquitetura de CGRA capaz de executar múltiplas *threads*, sendo definida como modelo MIMT (Múltiplas Instruções Múltiplas *Threads*).

4. Ferramenta que permite o uso da arquitetura de CGRA por meio de grafo de fluxo de dados descritos em alto nível.

O primeiro trabalho contribui no sentido de facilitar a utilização de CHR, por desenvolvedores de *software* e pesquisadores de outras áreas. Assim no capítulo 2 é apresentado um arcabouço capaz de auxiliar nos estudos de Redes Reguladoras de Genes (GRN), dispondo das vantagens do uso FPGAs para esse tipo de problema. O arcabouço é capaz de gerar todo o código de um acelerador para FPGA da plataforma HARP de uma GRN apenas com um descrição em alto nível da rede. Esse acelerador é capaz de extrair da rede estudada informações como estado de estabilização, tamanho do transiente e atratores da rede. Os aceleradores gerados pelo arcabouço foram duas ordens de magnitude mais rápidos que as mesmas aplicações executadas em CPU e uma ordem de magnitude mais rápidas que as mesmas aplicações executadas em uma GPU, tornando seu uso uma alternativa viável para o estudo desses problemas.

Com os desafios enfrentados na elaboração do primeiro trabalho, foi observado uma dificuldade na implementação da camada de comunicação entre o acelerador específico do usuário e a interface fornecida no *Kit* de desenvolvimento da Intel. Toda a comunicação entre o acelerador *hardware* e o *software* executado no CPU

era de responsabilidade do desenvolvedor. Essa dificuldade era ainda maior quando o desenvolvedor precisava adicionar mais de um acelerador ou unidade de computação no mesmo FPGA. Assim no segundo trabalho é apresentado um arcabouço que facilita na implantação de mais de um acelerador no mesmo FPGA. Esse arcabouço reduz o tempo de desenvolvimento de aceleradores para a plataforma HARPv2 uma vez que o usuário precisa se preocupar apenas na implementação do núcleo do acelerador, ou seja, a aplicação que se deseja acelerar. O trabalho apresenta como resultados a execução de um conjunto de aplicações com até 16 aceleradores em paralelo, chegando a obter até 353.7 Gops/W de eficiência energética.

Um problema comum no desenvolvimento de aplicações para FPGAs é o tempo de compilação. Com o uso de FPGAs como aceleradores esse tempo de compilação, além de impedir técnicas como *Just-In-Time* (JIT), não permite os desenvolvedores realizar várias compilações diárias no ciclo de desenvolvimento das aplicações. Para tentar contornar esse problema a Intel disponibiliza juntamente com SDK do HARPv2, uma ferramenta capaz de simular as interações entre o *software* no CPU e o acelerador no FPGA. Porém algumas características como *clock* máximo e uso de recursos do FPGA só são possíveis de obter após a compilação completa do acelerador, o que demanda horas de processamento dependendo do tamanho da aplicação. Para contornar esse problema no terceiro trabalho é apresentado uma nova arquitetura reconfigurável como alternativa ao FPGA. Essa arquitetura possui como principal vantagem o seu tempo de compilação quando comparada aos FPGAs. Para validar esse arquitetura ela foi sintetizada sobre o FPGA da plataforma HARPv2 e demonstrou ser capaz de compilar algumas aplicações descritas na forma de grafo de fluxo de dados na ordem de milissegundos e as execuções dessas aplicações foram até 2x mais rápidas que o CPU do sistema.

Esta dissertação está estruturada conforme o formato de coletânea de artigos científicos normalizado pelo Conselho Técnico de Pós-Graduação da Universidade Federal de Viçosa [11]. Este formato é composto por três partes principais: No capítulo 1 é realizada uma introdução geral que abrange todos os temas dos artigos que compõe a dissertação. Nos capítulos 2, 3 e 4 são apresentados os artigos produzidos durante o mestrado e, por fim, no capítulo 5 é apresentada uma conclusão geral do trabalho. No apêndice A são apresentados os grafos de fluxo de dados dos algoritmos utilizados nos experimentos realizados no capitulo 4. No apêndice B são apresentados outros trabalhos desenvolvidos durante o mestrado como colaborador.

# Capítulo 2

# Exploring the Dynamics of Large-Scale Gene Regulatory Networks using Hardware Acceleration on a Heterogeneous CPU-FPGA Platform

***Abstract*** - Systems Biology pose computational challenges that surpass the capabilities of current computing platforms. A particular challenge in this regard is simulating the dynamics of Gene Regulatory Networks (GRNs). This is because the number of possible network states, and thus the time required to compute them, grows exponentially with the number of network components. FPGA-based accelerators appear as promising alternatives to simulations on conventional software platforms. However, their application in the dynamic simulation of GRNs has been hindered by the inaccessibility of this technology to users without the proper expertise. Heterogeneous CPU-FPGA computing platforms, combined with high-level, flexible tools to deploy FPGA implementations, represent a powerful avenue to open the benefits of hardware acceleration to biologists. We present a high-level framework that exploits such acceleration for the simulation of GRNs, without compromising flexibility. Our tool takes as an input a biological network description in a standard format and translates it into a Verilog module. This module is subsequently incorporated into a hardware design optimized for the simulation of the network dynamics. To demonstrate the potential of our methodology, we simulated the dynamics of

two large-scale GRNs, one involved in tumor formation in Colitis-Associated Colon cancer consisting of 70 nodes, and the other describing the signaling pathways of a lymphocyte cell comprised of 188 nodes. Our approach simulates the dynamics of the networks up to two orders of magnitude faster than an optimized OpenMP implementation on a multicore CPU.

## 2.1 Introduction

Recent advances in Systems Biology and Genomics are posing new computational challenges that are beyond the capabilities of current computing platforms based on conventional CPUs. A particular challenge in this regard is the simulation of dynamic states of the Gene Regulatory Networks (GRNs) operating inside a cell, with the aim of gaining a predictive understanding of how the observable traits that uniquely define a cell emerge from the interactions between its molecular components [12]. While there is a wide range of theoretical approaches to describe the dynamics of such networks (depending on the kind of biological insights being pursued [13]), the simplest mathematical formalism for this purpose are Boolean models (explained in detail in Background). A fundamental problem with the simulation of biological network dynamics is that the number of possible states (i.e., its state space), and thus the time required to compute them, grows exponentially with the number of network components [14]. This has severely limited the scope of biological network studies, because they generally rely on software platforms running on general purpose CPUs, which restrict simulations to a maximum of twenty interacting molecular components to generate results in the timescale of hours or even days [15, 16, 17].

Nowadays, FPGA-based accelerators reappear as promising high performance, power efficient alternatives to simulations performed on CPUs and GPUs, as they allow parallelism at bit level that is not affected by unbalanced thread loads. In the past, FPGAs have already been used to simulate GRNs using a Boolean logic approach [18, 19, 20]; however, these implementations were limited to the evaluation of small-sized networks (with sizes up to 20 nodes [18]), and while some of them simulated the dynamics of larger networks, they only considered artificial ones with interconnections following random probability distributions [19, 20]. In a more recent work, a reconfigurable architecture was introduced to simulate an ensemble of networks generated from a distribution more consistent with the patterns of interconnections known to exist in biological networks [21].

Despite the potential of FPGAs as hardware accelerators, their application in the dynamic simulation of real biological networks has been hindered by the inaccessibility of the technology to users without the proper expertise to design hardware architectures for the implementation of custom algorithms. This situation could radically change with the emergence of heterogeneous CPU-FPGA computing platforms, which combined with high-level, flexible tools to deploy and validate FPGA implementations, represents a powerful avenue to open the benefits of hardware acceleration to biologists.

This work presents a high level tool that allows systems biologists to take advantage of hardware acceleration for the simulation of GRNs, without compromising flexibility. To achieve this, our approach starts from the specification of a biological network in the Systems Biology Markup Language (SBML) [22] whose format relies on a graph abstraction. In this abstraction, nodes stand for interacting elements of the network (e.g., molecules, genes, proteins, etc.), and edges depict specific molecular interactions. Our software tool takes an SBML file as an input, and translates it to a Verilog description, which is subsequently incorporated as a module into a hardware design optimized for the simulation of network dynamics. To take advantage of the logic resources available in the FPGA portion of the heterogeneous platform used in this work [2], the tool generates as many instances of the simulated network as possible, exploiting the bit-level computational capabilities of the device.

The proposed workflow can be integrated as a plug-in into SBML-compatible software frameworks used for biological network analysis and visualization, such as Cytoscape and similar programs [23, 24]. Thus, facilitating the widespread use of our tool among biologists. We demonstrate the potential of our methodology by applying it to the dynamic simulation of two GRNs. The first GRN is involved in tumor formation in Colitis-Associated Colon cancer (CAC), previously studied by Lu and collaborators [1]. This network has 70 nodes, and more than one Zetta states ($2^{70} = 10^{21}$). Due to computational limitations inherent in software approaches, the authors had to simplify this network down to 21 nodes(CAC21). The second network is a model of the signaling pathways of a kind of lymphocyte known as CD4 cell, comprised of 188 nodes.

In this work, we simulated a subset of the whole state space of the two studied networks. For the first, the unreduced network with 70 nodes, the corresponding subset contains $2^{27}$ states, which by the number of its elements is around two orders of magnitude larger than the state space associated to the reduced network with 21 nodes. For the second network, the CD4 network with 188 nodes, the corresponding subset consisted of $2^{24}$ states.With our approach, the simulations were completed

in just a few seconds, resulting in a speed up of two orders of magnitude over a 20-thread implementations for both networks. Additionally, our framework features an increase in power efficiency and performance, and can be automatically integrated as a service in the cloud.

The remainder of this paper is structured as follows. Section 2.2 introduces the biological foundations behind GRNs and provides details on how they are translated into Boolean models. Section 2.3 presents the architecture of the proposed accelerator, and describes how to integrate the solution into a high performance cloud platform. Section 2.4 depicts the experimental results. Section 2.5 compares our approach to others proposed in the literature; and finally, the main conclusions of this project, as well as possibilities for future work are described in section 2.6.

## 2.2 Background

The totality of the DNA content in a cell, which constitutes its genome, contains short regions known as genes. Through a process known as gene expression, these regions act as templates for the production of molecules called proteins necessary for a cell to adapt to changes in its environment. Transcription Factors (TFs) are a particular kind of proteins that act as molecular regulators that physically bind to specific DNA regions in the neighborhood of genes and can modulate the frequency with which genes are expressed. Some TFs regulate the expression of genes that code for other TFs, consequently establishing intertwined networks of interactions between genes and TFs, known as Gene Regulatory Networks (GRNs).

A gene regulatory network can be modeled as a graph, where each node $v_i$ represents a gene. The state of each gene is described by a logic value that represents whether the gene is active or inactive. Each node is connected to $k_i$ other nodes, where $k_i \geq 0$. Fig. 2.1(a) shows a simple network with 4 nodes as an example. The state of each node is updated in discrete steps according to its specific rule. At each step $t$, all node states $v_i$ are updated according to their Boolean function $f_i$. The new value of node $v_i$ will be $v_i(t+1) = f_i(v_{i_1}(t), \ldots v_{i_k}(t))$, where $k$ is the number of adjacent nodes of $v_i$. For the example shown in Fig. 2.1(a), the updated Boolean functions are displayed inside the nodes.

The network state is a vector of all the node states. Let us consider the initial network state $S_i = v_1v_2v_3v_4 = 0010$ for the 4-node GRN depicted in Fig. 2.1(a), where the node states are: $v_1 = 0, v_2 = 0, v_3 = 1$, and $v_4 = 0$. As the simulation progresses, the network state will converge to a set of stable states or a state cycle.

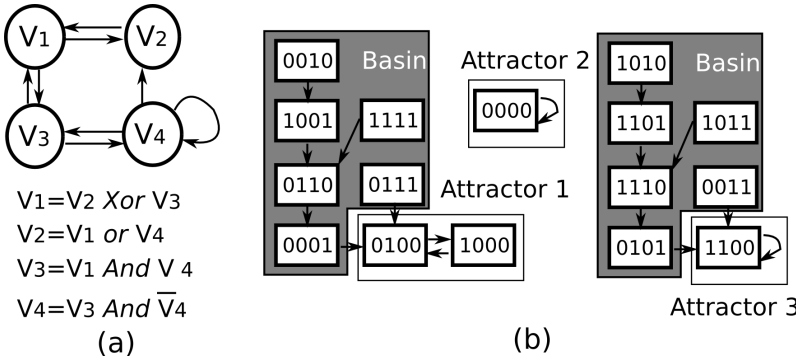



**Figura 2.1.** (a) A 4-node GRN example; (b) The state diagram ($2^4$ states).

Starting from the state 0010, the next states will be $S = 0010 \rightarrow 1001 \rightarrow 0110 \rightarrow 0001$, and then it converges to a set of stable steady states: $S = 0100 \rightleftharpoons 1000$, as shown in Fig. 2.1(b). A set of stable steady states or state cycle is called attractor, and the number of states in an attractor defines its length. The dynamics of a network can have one or more attractors, where all possible states converge to an attractor.

If a network is updated synchronously, then its state evolution is deterministic. Therefore, its state diagram can be partitioned into a set of disjointed graphs. Each graph of connected components correspond to an attractor and the group of states that converge into it, known as its basin. The network depicted in Fig. 2.1(a) has three attractors, as depicted in Fig. 2.1(b). Attractor 1 has two states, $S = 0100 \rightleftharpoons 1000$, and its basin is formed by six states. Attractor 2 is formed by only one state $S = 0000$, and the basin of the attractor is the point attractor itself. Finally, attractor 3 is formed by the state $S = 1100$ and has a basin of six states.

The framework proposed in this work aims at computing in a transparent manner the basin histograms for the dynamics of a network, the length of all attractors, and the average number of steps to reach an attractor from a random initial state. Unfortunately, since the state space of a network grows exponentially with the number of nodes, the solution space is too large, even for small values of $n$. Moreover, the attractor length computation is an NP-hard problem [25, 14].

## 2.3 Hardware Accelerator

Our design is automatically generated for a given network. It consists of an interface unit, a thread control unit, and a set of processing elements (PEs). Each PE has a control unit and two functional units. First, we describe the functional unit that

has a one-to-one correspondence with the GRN network to be simulated. Then, section 2.3.2 presents the architecture of a PE and the algorithm for the computation of attractors and their transients. Section 2.3.3 subsequently introduces the thread control unit to replicate parallel algorithm instances, and a thread load balancing manager to maximize the occupancy of each PE. Finally, section 2.3.4 presents the high performance interface we used and details how the proposed accelerator is coupled to a software API implemented in the heterogeneous Intel architecture [2]. The latter consists of a Xeon 10-core processor tightly coupled by shared memory to a Stratix FPGA.

### 2.3.1 Functional Unit

Each GRN node can be directly implemented by using its Boolean function and one-bit register to compute and store the node state. For the example shown in Fig. 2.1(a), the next state for all nodes, that is the global GRN state, can be computed in parallel by means of a functional unit, as shown in Fig. 2.2(a). The massive parallelism inherent in the GRN hardware unit allows all node states to be computed in only one clock cycle, generally in a way that is independent of the GRN size. This constitutes the basic step for most of the GRN simulation algorithms. A CPU-based implementation would require few assembly instructions to recompute each single node state. Therefore, for a medium size network (50 nodes), a CPU would spend a few hundreds of cycles in contrast to the one clock cycle used by the FPGA implementation.

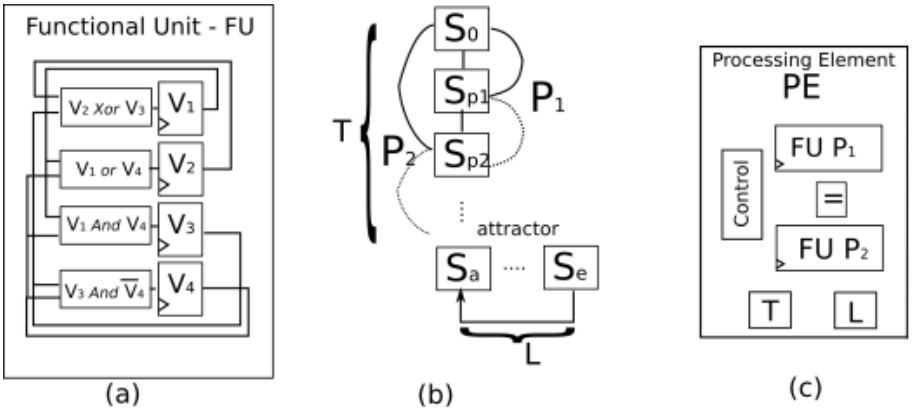



**Figura 2.2.** (a) Functional Unit (b) Transient and attractor Length; (c) Processing Element (PE);

### 2.3.2 Processing Element

This section introduces the transient and attractor computation based on two Functional Units (FUs) to build a PE. First, we describe the basic algorithm, and then its implementation.

Assume the state of the nodes in a network are updated in discrete time steps. Starting from any network state $S_i$ at time 0, the network state will converge to an attractor after $T_i$ steps, where $T_i$ is the size of the transient associated to $S_i$. Let us consider the state $S_a$ as the first state inside a cyclic attractor. The network will return to the same state $S_a$ after $L$ steps, where $L$ is the length of this attractor. A simple solution to discover this attractor is to keep all visited states in memory [26]; however, this approach becomes impractical since there are $2^n$ states. Moreover, memory operations can have a detrimental effect on performance as they are one of the major bottlenecks in parallel computation. To circumvent this problem, we propose to implement an algorithm based on the work presented by [27], which does not demand large amounts of memory. Starting from any state $S_i$ at time 0, two copies $P_1$ and $P_2$ of the network state are simulated in parallel. Every time the copy $P_1$ performs one simulation step (one clock cycle), the copy $P_2$ performs two steps, as shown in Fig. 2.2(b). An attractor is found once the condition $S_{P_1} = S_{P_2}$ is met. This approach has a complexity $O(1)$ in memory usage, and only requires to keep track of two states $S_{P_1}$ and $S_{P_2}$, even if there are attractors with large lengths.

The proposed PE implementation consists of two FUs that executes in parallel as shown in Fig. 2.2(c). The $FU_{P_1}$ performs one clock cycle and the $FU_{P_2}$ performs two clock cycles. The control unit is implemented by a finite state machine. If $T$ is the transient time, the algorithm needs at least $T$ clock cycles to compute the transient. The attractor length is computed by keeping $P_2$ stopped, while $P_1$ performs $L$ clock cycles until $P_1$ is equal to $P_2$ again. Two registers $T$ and $L$ are used to store the transient and the attractor lengths.

### 2.3.3 Thread Control Unit

Each PE receives a state as input, and in a few cycles computes the transient and the attractor length. Since the transients have variable sizes, the execution time for SIMD accelerators, such as GPUs, is determined by the worst-case (larger transient) at warp level and at the block level, as shown in Fig. 2.3(a). Moreover, since a GPU thread block should finish all threads, the block execution time will be bound by the worst case. Next GPU generation could provide a better performance since CUDA 9.0 enables a configurable warp size. However, it is not yet available in the market.

Our design could manage light, irregular thread loads by using an asynchronous set of FIFOs and PEs. The thread control unit (TCU) feeds the FIFO buffers with a sequence of states to be computed, as illustrated in Fig. 2.3(b). The PE get a state from the input FIFO, compute the transient and the attractor length, and put the computed values back in the output FIFO. Since the PEs are independent, they could execute the transient or the attractor computation with the maximum throughput (see Fig. 2.3(c)). As soon as a thread finishes its computation, the next thread can be scheduled. In addition, the following three tasks are overlapped in time: (a) insert a thread in the input FIFO; (b) schedule a thread in a PE; and (c) insert the results in the output FIFO.

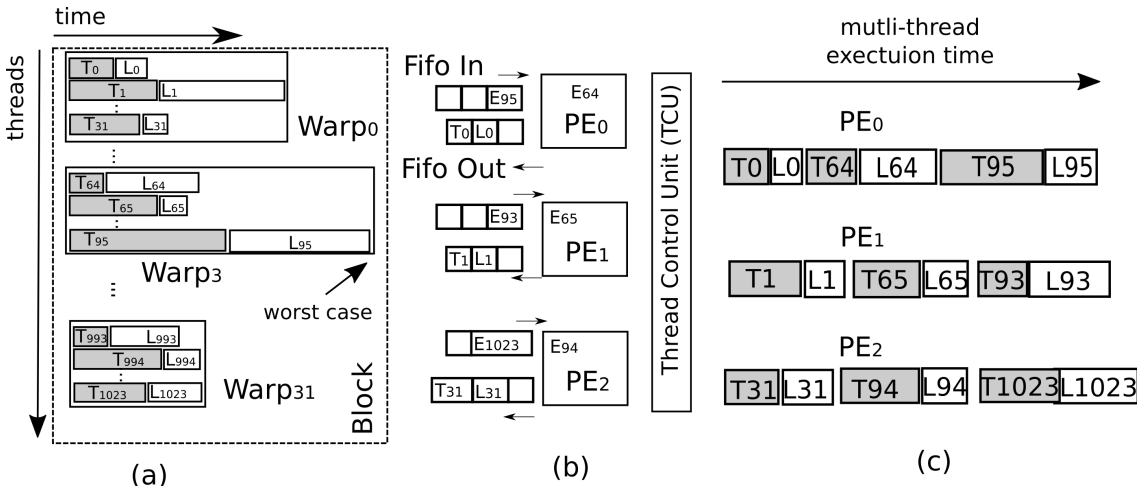



**Figura 2.3.** (a) Worst-case for SIMD GPU warps and blocks (b) Thread Control Unit (TCU), FIFOs and PEs; (c) Parallel execution asynchronous example.

### 2.3.4 Intel Heterogeneous CPU-FPGA Platform

The proposed GRN accelerator could be synthesized in any FPGA board. Nevertheless, our goal is to provide a high level abstraction implemented over a high performance accelerator platform. Therefore, we also provide an interface to integrate our accelerator into an Intel/Altera prototyping board with a 10-core general-purpose processor connected to a Stratix V FPGA [2]. The proposed framework could be extended to other cloud FPGA platforms such as Catapult [3], Amazon EC2 F1 [4], the new Intel Broadwell CPU coupled to an Arria 10 GX FPGA [2], and also to FPGAs PCIe boards by using Altera/Xilinx OpenCL compilers.

The Intel/Altera platform incorporates an Intel's Accelerator Abstraction Layer Software to provide software-hardware integration. The CPU-FPGA com-

munication is done via the Intel QuickPath Interconnect (QPI), and the read and write bandwidths are 7.0GB/s and 4.9GB/s, respectively [28]. This platform also provides a hardware layer inside the FPGA, which contains a cache of 64KB for coherency purposes.

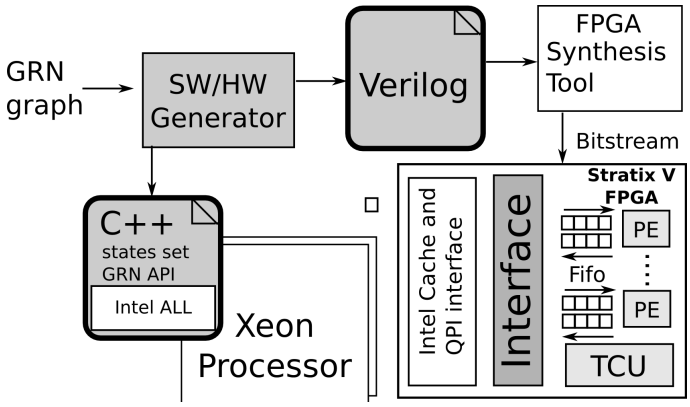



**Figura 2.4.** Accelerator Framework by using an Intel/Altera Xeon-FPGA

Fig 2.4 shows the proposed framework. The GRN graph is translated into a Verilog design. The design is synthesized using the Quartus Intel/Altera tool. Our design is parameterized and gives the user the ability to choose the number of processing elements to maximize the parallel computation. The goal is to use as much as possible the resources available in the entire FPGA. Once the bitstream is downloaded to the FPGA, a C++ code is executed to write a set of states in the shared memory; after that, the accelerator can be started. The TCU unit get the set of state requests from the cache, and send it to the PE FIFOs. As soon as a PE finishes the computation of the transient, the attractor length, and the attractor state ID, these data are written in the output FIFO.

## 2.4 Experimental Results

We evaluate our design by simulating the dynamics of two Boolean GRNs that have been studied *in silico* and validated by *in vitro* experiments [1, 29].The first network represents the signaling pathways involved in the growth and survival of tumor cells in Colitis-Associated Colon cancer (CAC). This network consists of 70 nodes and 153 edges, organized in the topology depicted in Fig. 2.5. As remarked in [1], because the size of the state space is exponentially dependent on the number number of node ($2^{70} = 10^{21}$ for CAC network), attractor identification is a strong NP-complete problem, and therefore tracking all of the attractors is a highly demanding

computational task. To study this network, the authors of [1] applied a network simplification algorithm [30] to reduce the node number down to 21 nodes. To evaluate the potential of our approach to explore the dynamics of larger networks, we explored a subset of the state space of the entire CAC network having 70 nodes, consisting of $2^{27}$ states, which is about two orders of magnitude larger than the state space of the reduced 21 node network from [1]. The second network that we simulated is a model of the signaling pathways of CD4 cells. The model includes signals relevant to the process of immune regulation [29]. This network consists of 188 nodes and 351 edges. As a proof of concept, we explored a subset of the whole state space consisting of $2^{24}$ states.

To benchmark the performance of our results, we implemented a corresponding C++ and CUDA software version of the models and simulated them using OpenMP and GPU approach respectively, thus extending our evaluation in the domain of multi-core parallelism. We simulated the entire network CAC [1], its reduced version [1], and the complete network CD4 described previously [29]. Table 2.1 depicts the results of the exploration of the state space by software running on an Intel Xeon CPU (10 cores, 20 threads) and on a GPU NVidia Tesla K40. The two last columns shows the throughput, which is defined as the number of states processed per second by the multithreaded implementation. Particularly, processing an initial state refers to the entire process of computing the transient, and the attractor length starting from such state. Although the network CD4 has 3× more nodes, the throughput is almost the same for the 20 thread OpenMP implementation, since the average transient length for the network CD4 is 3× smaller.

**Tabela 2.1.** CAC21, CAC and CD4 GRNs runned in Xeon E5-2680v2, 2.80GHz and GPU NVidia Tesla K40

| **GRN** | **State Space** | **Exec. Time Seconds** | | | | **Xeon States per Second** | **NVidia Tesla K40 States per Second** |
|---|---|---|---|---|---|---|---|
| | | **Xeon Threads** | | | **NVidia Tesla K40 Threads** | | |
| | | 1 | 10 | 20 | 65536 | | |
| **CAC21** | 2,097,152 | 4.6 | 0.63 | 0.53 | 0.056 | 3,956 K | 37,730 K |
| **CAC** | 134,217,728 | 3,616 | 515.7 | 283.0 | 10.377 | 474 K | 12,933 K |
| **CD4** | 16,777,216 | 536.2 | 50.45 | 33.4 | 8.487 | 502 K | 1,976 K |

Table 2.2 shows the results for the exploration of a $2^{27}$ subset of the state space for the network CAC with 70 nodes and a subset of $2^{24}$ state space for network CD4 with 188 nodes. Both networks have been evaluated in the proposed accelerator framework. As a proof of concept, we evaluated four design instances depicted in the first column of table 2.2; two for the network CAC with 32 and 64 Processing Elements (PEs), and two for the network CD4 with a single PE and with 32 PEs.

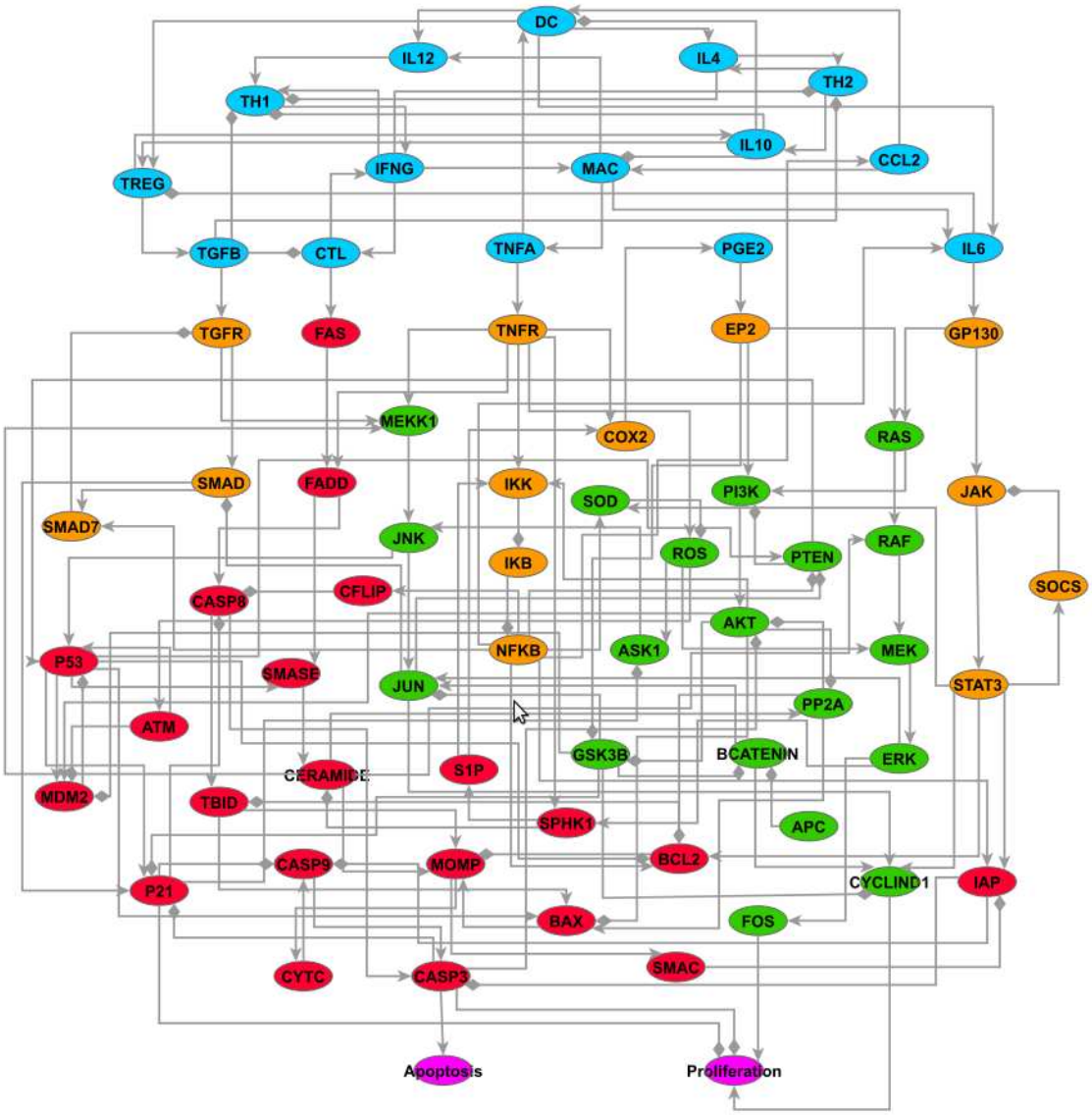



**Figura 2.5.** Boolean network - The growth and survival of preneoplastic epithelial cells in an inflammatory microenvironment - the molecular mechanisms underlying the development of colitis-associated colon cancer (CAC) [1].

**Tabela 2.2.** CAC and CD4 GRNs runned in FPGA Stratix V + Xeon E5-2680v2

| GRN | PEs | State Space | Clock Cycles | Time(s) | States per Sec. | Speedup over Xeon | Speedup over NVidia Tesla K40 |
|---|---|---|---|---|---|---|---|
| CAC | 32 | 134,217,728 | 325,529,465 | 1.62 | 82,851 K | 174.7 | 6.4 |
| | 64 | 134,217,728 | 162,765,974 | 0.81 | 165,701 K | 349.4 | 12.8 |
| CD4 | 1 | 16,777,216 | 535,607,845 | 2.67 | 6,284 K | 12.5 | 3.1 |
| | 32 | 16,777,216 | 67,140,932 | 0.34 | 49,345 K | 98.2 | 24.9 |

For the network CAC, we can observe that the execution time is reduced by a half

when the PE number is doubled. Additionally, the accelerator spends on average only 1.2 clock cycles per transient/attractor computation with 64 PEs. These results can be explained by the fact that the average transient and attractor length for the network CAC is around 60. The last column of table 2.2 depicts an FPGA speed up factor of two orders of magnitude in comparison to a software implementation with 20 threads running on a 10 core CPU and one order of magnitude in comparison to a GPU Tesla K40 with 65536 threads. It is important to highlight that the clock cycles were measured using internal FPGA counters during the execution time. Moreover, each PE running a simulation of the network CAC requires on average 77.6 cycles. This takes into account the accelerator setup, TCU manager, FIFO operation, FU transient and attractor length computation, and CPU-FPGA cache read and write operations. The exploration of the state space for the whole network CAC produced 131 steady states with 6 different attractor lengths, in approximately one second. The approach followed in this work is able to find more attractors (with perhaps relevant biological interpretations) than those originally found in a previous study based on network reduction strategies implemented in software, pointing to a reconsideration of the results published by [1].

For the network CD4, even if we use only one PE, the speed up factor is still two orders of magnitude in comparison to single-threaded software implementation on CPU, as shown in the last line of Table 2.2. Furthermore, by simulating this network using 32 PEs, the speed up increases by a factor of 8× in comparison to the single PE implementation. Since the average transient size for the network CD4 is short (around 12.9 cycles) the speed up is bound by the CPU-FPGA communication overhead.

Fig. 2.6 depicts the CAC transient histogram. The computational time for the simulation of these networks is directly related to the transient length. The FPGA implementation is bounded by the average case, even in presence of an unbalanced thread load at fine grained.

The operating system Ubuntu Linux 14.04 runs on top of the Intel Xeon CPU and executes the software component of the accelerator, including managing communication with the FPGA and measuring of execution time. The total time of the simulation is less than 4 seconds, which includes the sum of system and user components processing time obtained from the Linux command *time*. This computation time is minuscule in comparison to the more than one thousand seconds spent by the software execution, presented in Table 2.1. Hence, even when we consider the operating system overhead, the heterogeneous platform is a promising alternative to GRN computation on general purpose CPUs. Moreover, the FPGA used in this

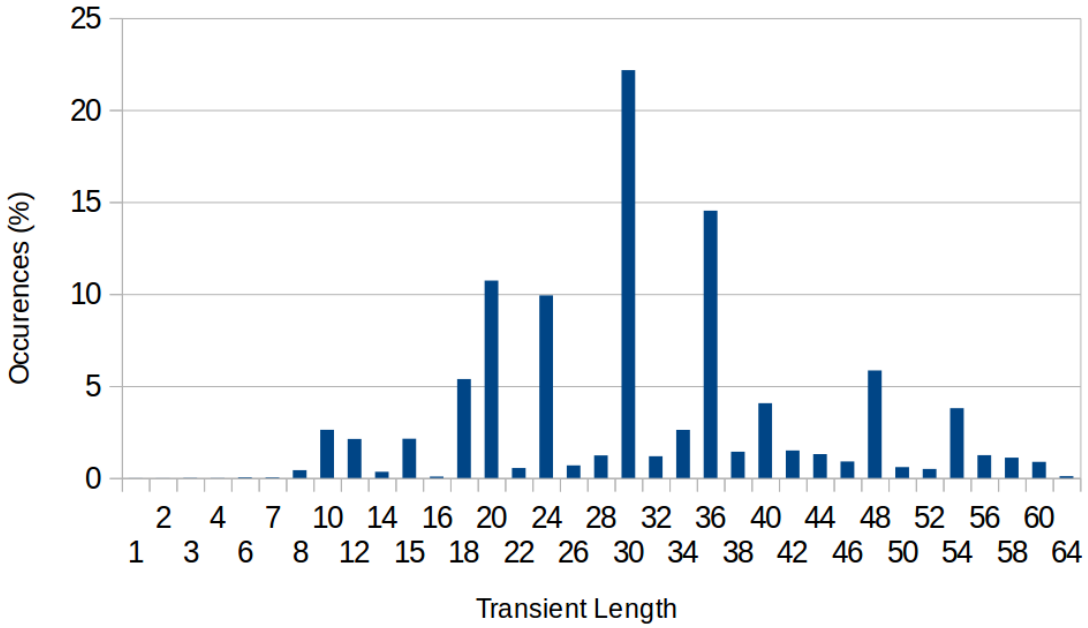



**Figura 2.6.** CAC network - Transient Length Histogram.

work consumes one order of magnitude less power than a multicore processor and a NVidia GPU Tesla K40. In addition, the FPGA used in this work consumes 25 watts of power an order of magnitude lower than that of an Intel Xeon processor and an NVidia Tesla K40 GPU that consume respectively 250 and 235 Watts. This means that if we add up the CPU+FPGA consumption our approach consumes 275 Watts total, with a performance of up to three orders of magnitude over a solution with only processors. For a GPU CPU solution the total power consumption would be 485 Watts, that is, almost double consumption and with a performance below an order of magnitude over our approach with CPU+FPGA.

Finally, Table 2.3 summarizes the FPGA resources required for Intel API, and for design comprised of the 64 generated PEs for the network CAC. The Intel API (cache+protocol) is required to build the interface between the accelerator unit and the Xeon processor consumes around 30% of the FPGA resources (Stratix V). Column ALM depicts the number of Adaptive Logic Modules (ALMs) used by the design. Each ALM has has an eight input LUT, four registers and two full adders. Columns Registers and BRAMs show the total number of registers and embedded RAM modules, respectively.

**Tabela 2.3.** FPGA resources

| Design | ALMs | Registers | BRAMs |
|---|---|---|---|
| Intel API | 78915 | 73036 | 139 |
| 64 PEs | 58679 | 84076 | 725 |

## 2.5 Related Work

Previous works have addressed the simulation of real networks using FPGAs [31, 32]; however, their I/O interface with the user was very cumbersome, requiring the manual configuration of simulations using DIP switches and relying on the observation of LEDs to obtain the results. In addition, the corresponding control units only allowed for the simulation of a fixed number of states, starting an initial network state, but were unable to perform automatic search of attractors. To simulate larger networks (with more than 20 nodes) using a software approach, it has become necessary to use heuristic methods, even at the cost of losing the ability to calculate all attractors or introducing spurious ones. These heuristic methods are based on different kinds of algorithms, that will be described in the rest of this section.

A group of algorithms relies on Reduced Ordered Binary Decision Diagrams (ROBDDs) to represent the system of Boolean functions describing the studied network [33]. These algorithms can compute all steady states of networks with sizes in the order of a thousand nodes. However, the ROBDD approach is limited to the computation of attractor lengths. In contrast, an exhaustive approach can additionally compute attractor transients and basins, which can be used for the study of network robustness by means of Derrida plots [16].

Another group of algorithms, based on network reduction, eliminates nodes from the network in a way that preserves as much of their dynamic properties as possible. These algorithms can simulate the dynamics large networks; however, for the most part, they are only able to find point attractors [34]. Some of them can find cyclic attractors [17], but due to high demanding RAM requirements, the cycle attractor detection in networks with more than 60 nodes becomes prohibitive.

In a different group of algorithms based on a SAT approach, the system of equations that describes the dynamics of the network is rewritten as a single expression $G(x)$, in such a way that finding the conditions under which it is true turns out to be a surrogate for the problem of finding attractors [35]. These algorithms can simulate in a few seconds the dynamics of networks with several thousand nodes [36]; nevertheless, they can only be applied to networks with low average

interconnectivity (i.e., equal or lower than two edges per node). This represents a severe constraint given recent findings in the biological domain pointing to the fact that regulatory networks are much more interconnected than previously thought [37]. Considering all these issues, the use of hardware accelerators as simulators of network dynamics results appealing, because they can be scaled to account for large networks with a high degree of interconnectivity. Therefore, it is very likely that heterogeneous CPU-FPGA platforms will become increasingly useful for the study of real biological networks in the years to come.

## 2.6 Conclusion and Future Work

This work presents a high performance and high level framework to perform dynamic simulations and explore the state space of GRNs (described in terms of Boolean models) on a CPU-FPGA accelerator. As a case study, we simulated a regulatory network associated to CAC, incorporating 70 nodes [1]. Given the Boolean network model of the CAC, our proposed framework automatically generated a design in a synthesizable Verilog hardware description language code. The design is parameterized to replicate the maximum number of parallel processing elements that is allowed by the resources of the FPGA used in this work. Moreover, the design can easily communicate with the CPU via a FIFO-based interface from which it can receive the set of initial states for the simulations, as well as send the results back to the CPU. Our framework was validated by using the Intel/Altera prototyping system based on a Xeon CPU tightly coupled to a Stratix V 200 MHz FPGA [2], and it could be easily incorporated in Microsoft FPGA Catapult Cloud [3], Amazon FPGA Cloud EC2 system, or even in a low cost didactic board, where the data is sent/received through a JTAG interface. A direct extension is to implement our framework on the platform consisting of an Intel/Altera Broadwell CPU coupled to an Arria 10 GX FPGA with a clock of 400 MHz, and an FPGA fabric with more reconfigurable logic, and more communication bandwidth (27 GB/s) than the Stratix V.

The experimental results demonstrated a speed up of two orders of magnitude in comparison to a software implementation on a 10-core 20 thread CPU and an order of magnitude in comparison over a GPU NVidia Tesla K40. Future work could include more extensive functionalities like analysis of network robustness [38], which could be done by verifying if a network sustains its attractors when perturbations that affect some nodes are applied. In addition, we will add an option to

explore a subset of the state space when the state of some genes are fixed to constant initial values (active or inactive). For instance, a subset of genes from the CAC network model could be fixed to an inactive state to represent the absence of inflammatory factors (cyan nodes in Fig. 2.5), and the APC gene could be fixed to an active state. These conditions would reproduce the state of the CAC network that was experimentally observed on intestinal epithelial cells in a normal immune microenvironment [1]. Furthermore, for the simulation of the network composed of 188 nodes that simulates the signaling pathways of CD4 T-cells, our coverage of the network state space was almost four orders of magnitude larger than the simulation performed in the original biological work, in which only 2000 simulations were performed. This demonstrates the potential of our proposed platform for the simulation of large-scale biological networks.

## Acknowledgments

We would like to thank the Brazilian institutions: UFV and the research agencies FAPEMIG, CAPES, CNPq for their financial support, and the Center for Research and Advanced Studies of the National Polytechnic Institute Campus Irapuato, Mexico.

# Capítulo 3

# Simplifying HW/SW Integration to Deploy Multiple Accelerators for CPU-FPGA Heterogeneous Platforms

***Abstract*** - FPGAS became an interesting option for developing hardware accelerators due to their energy efficiency and recent improvements in CPU-FPGA communication speeds. In order to accelerate the development cycle, FPGA high-level synthesis tools have been developed such as Intel HLS, OpenCL, and OpenSPL. These tools aim to free the designer from having to know all the FPGA low-level details. However, in order to achieve high performance processing, the developer should still understand the details of the deeper system layers. Moreover, OpenCL usually consumes more resources/compile time than a design developed directly in RTL. In this work we propose a novel framework to automatically integrate hardware accelerator cores in a final architecture by generating a HW/SW interface for an Intel CPU-FPGA modern platform. Our goal is to simplify the Intel Open Programmable Acceleration Engine (OPAE) by introducing a novel abstraction layer with a simple stream protocol channel. The experimental results for a set of dataflow benchmarks show a performance of up to 131.7 Gops/s, and a power efficiency of up to 353.7 Gops/W even when we bound the memory bandwidth to 12 GB/s.

## 3.1 Introduction

Due to the recent poor CPU performance improvements and the lack of advances in the development of more efficient sequential algorithms, hardware accelerators became an interesting option thanks to their ability to speedup different parallel-inherent applications such as: matrix multiplication [39], neural networks [6], and data center's applications [40, 3, 41, 42]. Nowadays, a great number of high performance platforms are heterogeneous systems consisting of general-purpose processors and different types of hardware accelerators: ASICs, GPUs, and FPGAs. Application Specific Integrated Circuits (ASICs) achieve the highest level of performance-power/price ratio for a specific problem at the cost of long development time, high manufacture cost and inflexibility. GPUs are flexible, being able to deliver a peak performance in the order of Teraflops. However, GPUs do not perform well when the target application is not suitable for SIMD parallelism. Moreover, GPUs are power-hungry devices in comparison to ASICs and FPGAs.

The combination of flexibility, potential for parallelism and the energy efficiency makes FPGAs a suitable option for many cloud applications [40, 42, 43, 6, 39]. In the past, only standalone FPGAs or FPGAs with embedded processors, such as an ARM hardcore or synthesizable softcore processors, were used in order to develop hardware accelerators [39]. Due to the bandwidth limitations, these platforms were used more as proof of concept than as a real acceleration system. In addition, FPGA programming requires digital-design expertise, complex design tools and knowledge of low-level device details. Recently, Microsoft [3], Intel [2], Amazon [4] and IBM released CPU-FPGA platforms [44, 45, 42] and high-level programming models like OpenCL [46] and OpenSPL [41] have been developed.

More specifically, Intel released in 2016 a research platform (software/hardware) composed by an Intel Xeon integrated with an Arria 10 FPGA [6]. However, few frameworks and programming layers have been created to facilitate the CPU-FPGA communication on this platform. The Intel Open Programmable Acceleration Engine (OPAE) [47] is an API used to interface the CPU and the FPGA. The Intel OpenCL for FPGAs [46] is a framework which allows the user to develop an accelerator in a high-level language. OPAE does not natively offer the option to create multiple acceleration units in the same design, and Intel/OpenCL is less efficient in terms of resource utilization when compared to native RTL implementations.

In this work, we present FAM[1] (Framework for Accelerator Management), a

[1]Download available at *github.com/ComputerArchitectureUFV/FDAM*.

novel SW/HW framework to extend OPAE to abstract and support multiple parallel accelerators. FAM provides a software and hardware API where host-accelerator communication code is automatically generated. Our contributions are twofold: (1) Simplification of multiple accelerator development for Intel CPU-FPGA platform; (2) Performance improvement in terms of throughput and energy efficiency by instantiating multiple copies of a single accelerator. We have designed our framework to be used by software developers with no previous specific hardware skills. Similarly to the CUDA GPU approach, the SW programmers should only instantiate the input/output buffers, send/receive data, and make synchronous/Asynchronous calls to HW ACC units.

At the hardware side, our framework is parameterizable, automatically generating the RTL source code for communication interfaces. The hardware developer should only specify the set of homogeneous/heterogeneous ACC units, and their respective input/output queues. We generate all hardware communication interfaces to manage the input/output streams, thus encapsulating OPAE to exchange data through the coherent-shared memory. To create a dynamic development process it is possible to use the simulation infrastructure present in the OPAE framework called ASE (AFU simulation environment), to facilitate the configuration of the OPAE simulation environment, our solution creates all the necessary files to perform the simulation for two types of commercial hardware simulators. Furthermore, our framework reduces compile time by a factor of 2 times in comparison to OpenCL.

As a proof of concept, we evaluate our framework using four benchmarks: a finite impulse response (FIR), a Paeth accelerator for compression of PNG images, a reduction operator, and the interpolation method gouraud. All these accelerators have been designed as dataflows and integrated in Intel CPU-FPGA cloud server, where we evaluate bandwidth, design replication, performance, energy efficiency, compile time, and FPGA resource utilization. Our performance results achieve 131.7 Gops/s with 353.7 Gops/W energy efficiency. These results are bounded by physical memory bandwidth or FPGA DSP resources.

The remainder of this paper is organized as follows: In section 3.2 we present related work. In section 3.3 we describe both hardware and software framework components, while in section 3.4 we evaluated the proposed framework with experimental data. Finally, in section 3.5, we draw our conclusions and propose future work.

## 3.2 Related Work

This section presents other works in the field of high performance computing (HPC) FPGA acceleration in which we highlight the main challenges and potential solutions.

The Open Programmable Acceleration Engine (OPAE) is an open source programming layer created by Intel to facilitate the integration of FPGA accelerators with software applications and environments [47] at register transfer level (RTL). It frees the developer from the responsibility of knowing the details of the FPGA driver and interconnection interfaces by providing a layered software access model which allows enumerating, accessing, and managing FPGA resources through an API. Although OPAE offers a more abstract way to interface with a bi-directional communication channel on the Intel CPU-FPGA platform, the developer still needs extensive knowledge of the signals from all the communication channels which makes the creation of an ACC a cumbersome process.

A common strategy to improve the performance is to replicate the accelerator kernels to fulfill the entire FPGA. Since the FPGA hardware has a fixed cost it is usually interesting to maximize their resource occupation. A simple application is a signal processing filter (FIR) for large stream data. However, in native Intel OPAE, the developer should manually instantiate the set of replicated ACCs. Moreover, in order achieve higher bandwidth, the developer should manage the creation of communication channel arbiters from scratch. This is usually a very time-consuming process that once simplified, could free the developer to focus on the ACC design.

RTL is still the most popular design abstraction althought high-level languages are rising as an alternative to develop hardware accelerators. The Intel FPGA SDK for OpenCL [46] highly improves development efficiency since it uses the C++ high-level language to develop FPGA kernels for heterogeneous CPU-FPGA platforms. OpenCL is also portable among multiple platforms. Although OpenCL can potentially make FPGAs transparent at programming description level, the compilation phase usually still consumes several hours or even days of CPU time. In this way, our approach which directly uses RTL, presents greater flexibility to the hardware developer, allowing the use of coding strategies that can reduce FPGA resource consumption and compilation time.

The first Intel/Altera CPU-FPGA platform prototype has been released in 2015, when some specific accelerators [48, 49, 50, 43, 51, 52] and performance evaluations [28, 53] have been published. This platform is composed by a 20-core Intel processor connected by a Quickpath Interconnect (QPI) to a Stratix V FPGA and

provides a coherent shared memory. Intel released a new version of this platform in 2017 [2].

Colangelo *et al.* [6] explore binary weighted networks (BWN) acceleration using this platform. In this work, the authors developted an individual convolution layer FPGA accelerator using HDL and the OPAE API [47]. Moss *et al.* [39] implement a configurable matrix multiplication (GEMM). They provide support for several blocking strategies and data types. The configurable accelerator has been implemented using a specific C++ API and is limited to accelerate matrix multiplication in hardware. Our framework is generic and can encapsulate a wide range of HDL accelerators – the only restriction is the input/output queues. Those applications are good examples of accelerators that can be easily ported and executed in our framework.

Although Intel OpenCL claims to provide a higher abstraction level to write FPGA accelerators, several problems have been reported in literature [54]. To mitigate these issues, the developer should have deeper knowledge of FPGA programming. For example, most of best-performing kernels either failed to fit or route with the default placement and routing flow, or exhibited noticeably lower operating frequency. The compile time is also a limiting issue in the Intel OpenCL environment. Large designs may need hours or even days to compile in high-end computers.

A well known strategy to address the compile time problem is the use of simulators [40]. In this work, the authors provide PCIe and NIC communication channels between virtual machines and simulators. The idea is to simulate the behaviour of several accelerators distributed in different physical systems as, for example, data centers. The work presented in [55] also aims to improve the development cycle, where an overlay has been created to decrease the compilation time for projects with multiple design versions. The authors parse all design versions creating dataflow graphs that are then merged to optimize resource utilization, thus decreasing the compile time. Our framework uses a similar strategy since it provides support for multiple ACCs in the same design. Stitt *et al.* propose a framework that creates multiple processing units in the form of pipelines and uses FIFOs for 2D and 3D patterns. Again, this work focuses on specific applications and homogeneous pipeline replication while our framework provides support for generic FIFO-based heterogeneous accelerators.

## 3.3 FAM: Framework for Accelerator Management

The Open Programmable Acceleration Engine (OPAE) [47] has been introduced as an open community project started by Intel to simplify the integration of FPGA accelerators into software applications and environments. Although providing several important abstractions, OPAE does not support multiple accelerators. If we consider, for example, an application with five different internal functions to be accelerated, we have two approaches to perform the acceleration. The first approach could be integrate all five functions in a single RTL accelerator, which allows the execution of the five accelerators in parallel. However, the final design can become complex since it should also couple to the SW/HW APIs on a CPU-FPGA platform. The developer should perform three tasks: (a) Design each accelerator; (b) Design the OPAE SW/HW interface; (c) Design the specific manager to provide the interface between OPAE and the accelerator units. Therefore, in addition to design efficient RTL accelerators, the developer should know OPAE details (cache lines, protocols, signals) to design the CPU-FPGA interface. Furthermore, an efficient unit should be also designed in such a way that it can concurrently manage all function calls and the data transfers to/from the five accelerators.

The second approach is to separately design each accelerator, leveraging the complexity on the implementation. In this case, the developer should only couple one unit to an OPAE interface. However, this approach prevents the use of two or more accelerators in parallel and also degradates performance due to the FPGA reconfiguration time to swap the accelerator bitstreams. Moreover, the unused resources on the FPGA are lost and the final compile time is the sum of each single acceleration compile time.

In this work, we present FAM (Framework for Accelerator Management), a novel SW/HW framework to extend OPAE to abstract and support multiple parallel accelerators. FAM provides a software and hardware API where host-accelerator communication code is automatically generated. Moreover, the developers can treat each accelerator as a single design with a straightforward input/output queue interface. At the software interface, the developers only instantiate and launch a set of homogeneous/heterogeneous accelerator tasks. In addition to encapsulate OPAE, FAM uses the Intel-FPGA Basic Building Blocks [47] to transparently build the communication interface. It is important to note that the developer can use our framework to encapsulate a publicly available IP core as an OPAE accelerator.

### 3.3.1 Software API

The FAM API consists of two main parts: (1) the host source code where the developer can instantiate the accelerators which are called as asynchronous functions; and (2) the hardware interface to control all communication data flows between the FAM API and the hardware accelerators.

#### 3.3.1.1 Software API - host code

We have developed a software API in C++ to be used similarly to CUDA language in high performance GPU accelerators. The API is compiled as a shared library, and it can be further extended to languages such as Java and Python. FAM software API provides two C++ classes: *AccManagement* and *Accelerator*. The *AccManagement* object has a list of *Accelerator* objects. Once FAM creates one *AccManagement* object at runtime, the accelerators available on the FPGA are read from the hardware manager to dynamically build the current list of available *Accelerator* objects. The *AccManagement* class also provides methods to initialize each accelerator individually or as a group. Moreover, it is possible to verify if each accelerator is still running or has already finished execution. The *Accelerator* class provides methods to fulfill the input/output FIFOs, to start and to restart the accelerator tasks, to query if the result is already ready (non-blocking), and wait until the task is finished (blocking).

As an example of our software API, Figure 3.1 depicts a piece of code where we request a single accelerator. This accelerator has one input and one output FIFO. First, in line 7, we instantiate the *AccManagement* object. The class constructor method transparently reads the HW manager data though the OPAE's API. If the HW manager is not found, FAM throws an exception. In line 8, we copy the first accelerator ($id = 0$) to the *acc* object. Then, we allocate the input/output queues and send the host data to the accelerator. We also start the data transfer without blocking the host code, and then, at line 12, we launch the accelerator kernel. The data transfer (both input and output) and the kernel execution are overlapped. In this example, we block the host code by using the *waitDone* method in a synchronous mode. However, the host can just check if the accelerator has already finished while performing a local computation in parallel to the accelerator task. Finally, in line 14, we copy the data back to the host vector. Although this example presents only how to instantiate and execute a single accelerator, FAM also supports multiple accelerators with a very similar code construction.

```
#include <AccManagement.h>

int main(int argc, char *argv[]){
   int DIn[4]= {1,2,3,4};
   int DOut[4]= {0,0,0,0};
   int nBytes= sizeof(int)*4;
   int acc_id= 0;

   auto accMgt= new AccManagement();

   Accelerator *acc= accMgt->getAcc(acc_id);
   acc->createInputQueue(0, nBytes);
   acc->createOutputQueue(0, nBytes);
   acc->copyToInputQueue(0,DIn,nBytes);
   acc->start();
   acc->waitDone(maxTimeWait);
   acc->copyFromOutputQueue(0,DOut,nBytes);

   delete accMgt;

   return 0;
}
```

**Figura 3.1.** C++ sample code using FAM Software API.

#### 3.3.1.2 Interface Manager Unit for Multiple Accelerators

To simplify the integration of multiple accelerators to send/receive data to/from CPU, we have developed an interface management unit (ACC manager), as depicted in Figure 3.2.For this a Verilog code generator was developed to allow the generation of the ACC manager capable of managing multiple accelerators, being able to mutliplexar the only communication port between several accelerators. For the generation of automatic code we use the framework in Python Veriloggen [56].

We have developed the current version of FAM targeting stream-based data flow accelerators, where the data transfer is performed by a set of input/output queues. Figure 3.2 details the interface manager unit, which provides an input and an output FIFO manager, connecting the complete set of available accelerators ($ACC_i$). Each $ACC$ unit can have an arbitrary number of inputs or outputs. The maximum number is bounded by the resources available in the target FPGA. Our experiments have scaled up to 16 input and output queues at 200 MHz for Arria 10 FPGA.

The interface management unit is specified by a single file with few Python

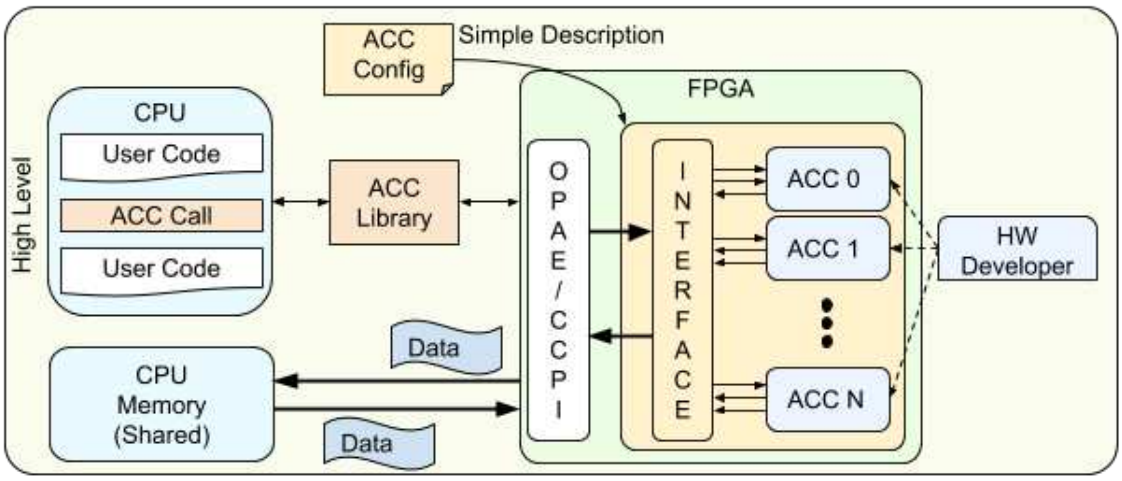


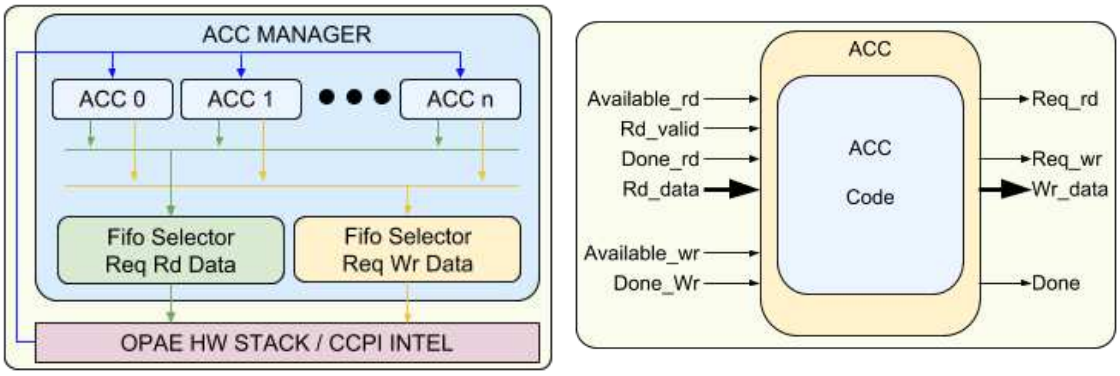

**Figura 3.2.** FAM hardware architecture.

lines. For easy of explanation, Fig. 3.3(c) depicts the code snippet to generate a ACC manager unit for a set of four accelerators. Each accelerator has a different number of input and output queues. In this example, the $ACC_0$ has 1 input and 2 outputs, $ACC_1$ has 1 input and 1 output, $ACC_2$ has 6 inputs and 8 outputs, and finally, $ACC_3$ has 12 inputs and 12 outputs. Furthermore, a template code for each accelerator local interface is also generated as shown in Fig. 3.3(b). The start, reset, and done signals are written and read directly from the host application. The host code can initialize, stop, restart, and query the accelerators. Fig. 3.4 shows an example of the execution with two concurrent accelerators. The data transfer and the kernel execution are overlapped. Since there is a single channel between the CPU-FPGA, the proposed framework automatically generated the interface unit (see Fig. 3.3(a)) to control the traffic in the shared channel.

### 3.3.2 Hardware API

The FAM hardware API, although automatically generated, is composed of small blocks formed by parameterized modules. This way you can increase the number of accelerators and the number of their input/output queues without decreasing the clock frequency. Figure 3.2 shows the general architecture of an accelerator manager. Each accelerator consists of $N$ input/output controllers and modules to control the accelerator state and include the developer accelerator code.

Since FAM creates a layer over OPAE [47] hardware API, we are also restricted to use one channel between the accelerator and the CPU. The accelerators input/output queues share this single channel access through a tree interconnection,




(a) (b)

```
from make_acc_management import *

accs= [(1, 2),(1, 1),(6, 8),(12, 12)]

accMgt= make_acc_management(accs)
pathForRtl = "<path>"
accMgt.to_verilog(pathForRtl)
```

(c)

**Figura 3.3.** Acceleration manager: (a) Interface; (b) Template; (c) Generation code.

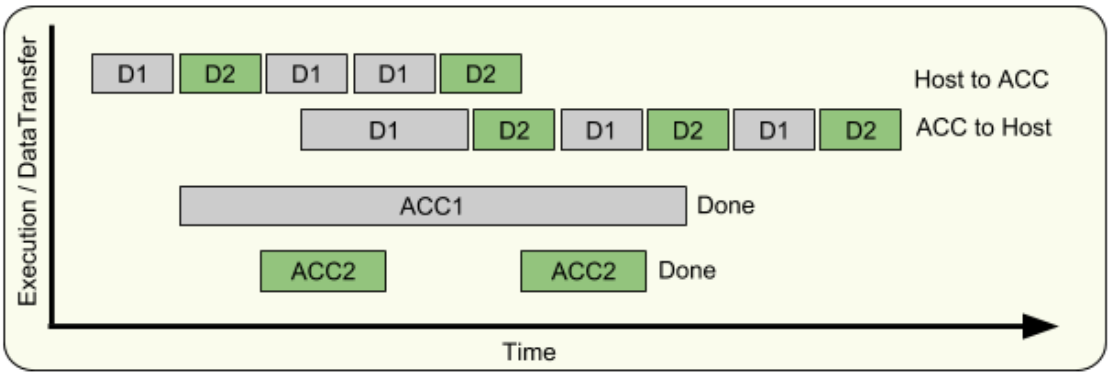



**Figura 3.4.** Parallel execution example.

which is automatically generated according to the number of input/output queues for each accelerator. In addition, we use a pipelined hierarchical sub-tree topology, thus maximizing the clock frequency and the throughput. We limit each local tree to have four rows while we connect the tree root to the OPAE hardware API request queue. Figure 3.5 shows our tree architecture for an example of twelve FIFOs.

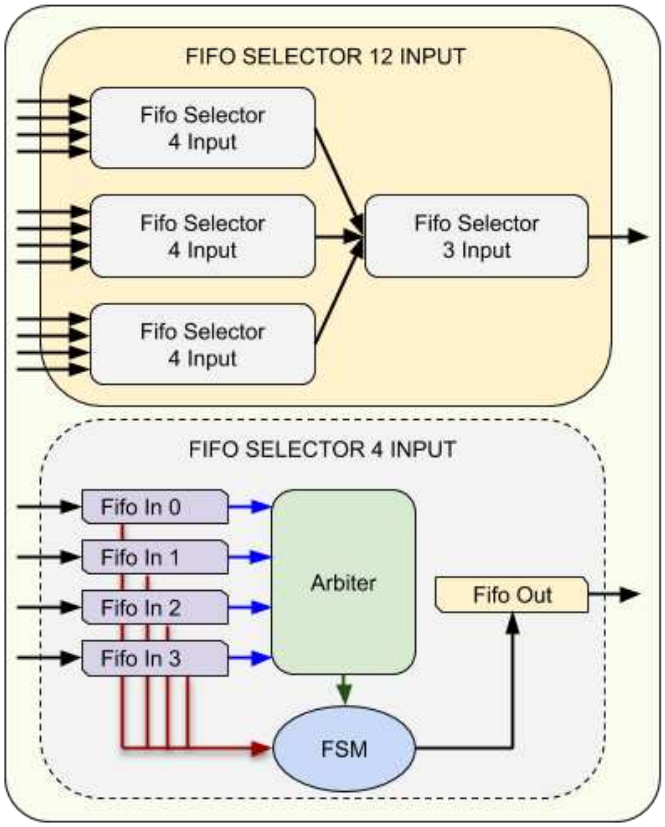



**Figura 3.5.** Internal structure for FIFO selector with 12 inputs.

## 3.4 Results

Since the current the state-of-art CPU-FPGA platforms have limited memory bandwidth, the common approach to provide high-performance designs uses deep pipelines [57]. Furthermore, the HPC FPGA approaches also apply pipeline replication, loop unrolling, and overlaps the computation with the memory access. As a proof of concept for our approach, four benchmarks have been selected as case study where we show how to exploit the common HPC FPGA strategies. Our framework helps the developers to apply replication and, at same time, provides transparent memory/computation overlapping. Moreover, we have selected the benchmarks to show the impact of pipeline depth, data reuse, and FPGA resource utilization. The benchmarks are stream dataflow algorithms implemented in RTL (Verilog).

The first experiment evaluates the replication and the maximum throughput. The number of inputs and operations for each benchmark is depicted in Table 3.1. The replication was implemented at two levels as shown in Figure 3.6. Each unit receives an OPAE packet, which has a fixed size (512 bits or 64 bytes = 1 cache

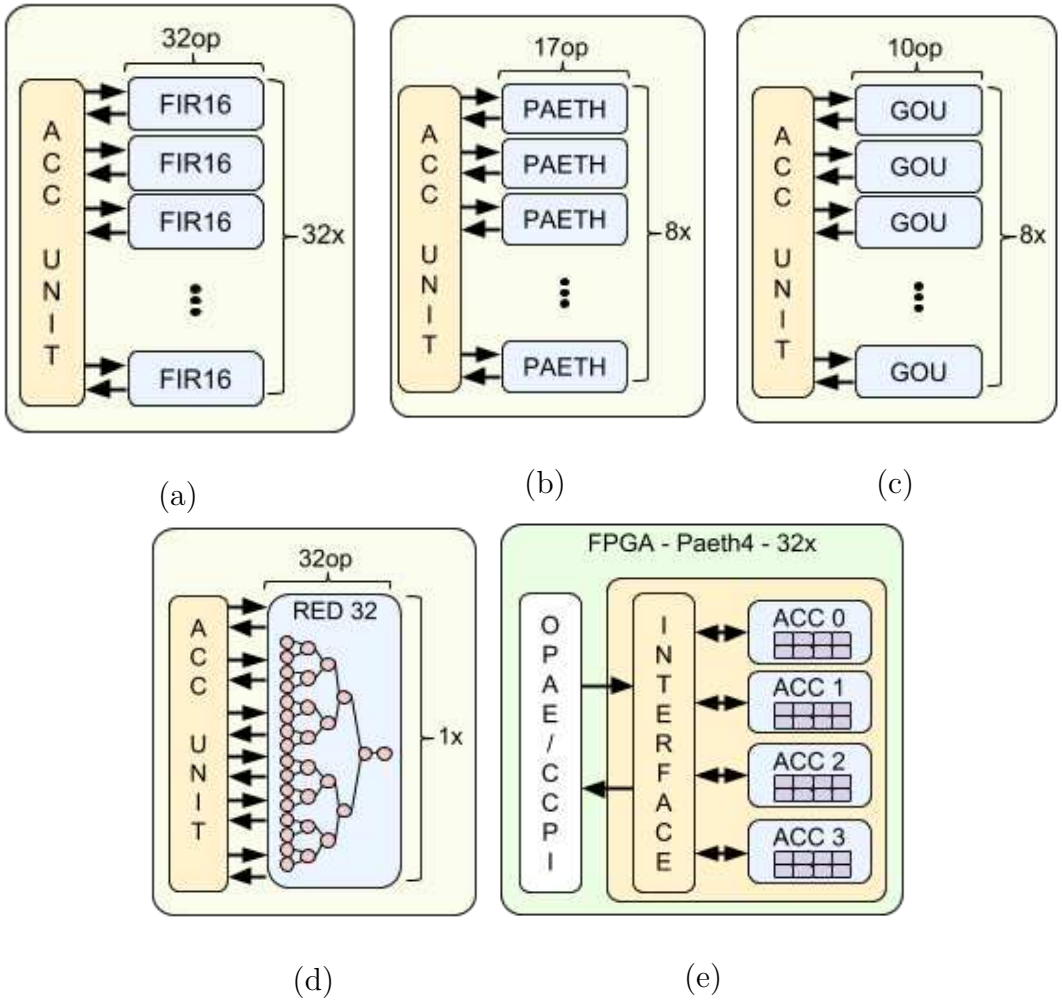



**Figura 3.6.** Replication inside ACC units: (a) Fir; (b) Paeth; (c) Gouraud; (d) Reduce, and multiple ACC with 4 Paeth Units.

line). If a kernel requires $n$ input bytes, and $n < 64$, we can replicate the kernel to maximize the throughput as shown in Fig. 3.6(a-d). For instance, the *paeth* has 4 input queues (16 bits) or 8 bytes input. Therefore, 8 copies can be generated per ACC unit as depicted in Figure 3.6(b). Since the paeth ACC unit uses few resources, we also replicate the units by instantiating multiple ACC as shown in Figure 3.6(e), where the paeth was replicated 8 times inside the ACC unit. For our experiment, each benchmark has executed with 1, 4, 8, and 16 homogeneous ACC units. The number of ACC units for each benchmark is displayed in Figure 3.6.

The maximum throughput is bounded by the data reuse and the memory bandwidth. The *reduce* benchmark is the worst case size, where for each cache line (64 bytes) only 32 operations are performed. Therefore, $\frac{64}{32} = \frac{1}{2}$ operations per byte. If the a memory bandwidth is 12 GB/s, the performance is bounded to $12 * \frac{1}{2} = 6$

**Tabela 3.1.** Number of operations and stream inputs per benchmark.

| Benchmark | OPs | IN |
|---|---|---|
| PAETH | 17 | 4 |
| FIR16 | 31 | 1 |
| Gouraud | 10 | 4 |
| Reduce_Sum | 32 | 32 |

G ops/s as depicted in Figure 3.7. The Gouraud executes 10 operations per kernel, and each ACC has 8 Gouraud copies inside. Therefore, 80 operations per 64 bytes, or 1.25 bytes per op, and the maximum of 15 Gops/s. The Paeth has 17 operations per kernel and 8 copies, or 136 operations per line, and it can reach 25.5 Gops/s. These theoretical bounds have been validated in the Intel CPU-FPGA platform [2] as shown in Figure 3.7.

One exception is the FIR benchmark which has 16 taps with 16 multiplies and 15 additions, and only one stream input. Therefore, the FIR ACC unit can have 32 FIR copies. However, it requires 512 DSP for the multiplies (in case of reconfigurable coefficients). Therefore, the FIR can be only evaluated with one/two ACC units since it is bounded by the maximum number of FPGA DSP units. Since the FIR has the highest data reuse rate, and even considering a modest 12 GB/s bandwidth, the theoretical performance bound is 186 Gops/s. While the three previous benchmarks were bounded by the CPU-FPGA bandwidth, the FIR achieves 131.7 Gops/s as shown in Table 3.2.

In order to analyze the power consumption we used a Python script provided by OPAE which returns the current consumption in milliWatts/h, the number of Gops/W is shown in Figure 3.8 and Table 3.2.

**Tabela 3.2.** FIR 16-Tap performance and power efficiency.

| Accelerator | Gops/s | Gops/W |
|---|---|---|
| fir16_1 | 65.9 | 177.0 |
| fir16_2 | 131.7 | 353.7 |

Regarding the FPGA resource utilization for each multiple homogeneous accelerator depicted in Table 3.3, since most of them (except the FIR example) consumes less than the half of FPGA resources, one could ask why not increase the number of copies. However, as showed in Figure 3.7, four units are enough to reach the maximum performance which is bounded by the CPU-FPGA bandwidth. Although the previous HW accelerator frameworks for the same Intel platform targets a specific application [57, 6, 39, 58], our framework helps the HW developers to design

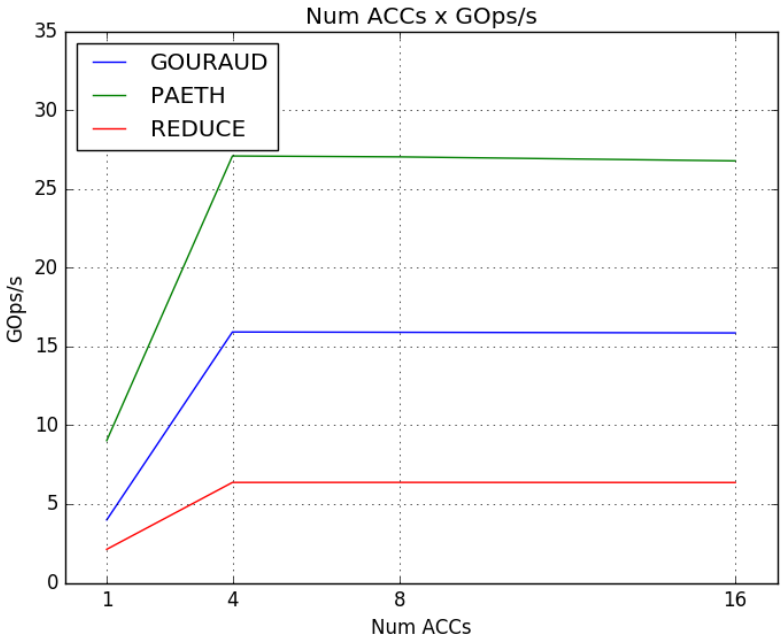



**Figura 3.7.** Performance in Gops/s for different ACC homogeneous sets executed in Broadwell Xeon Processor coupled to an Arria 10 FPGA.

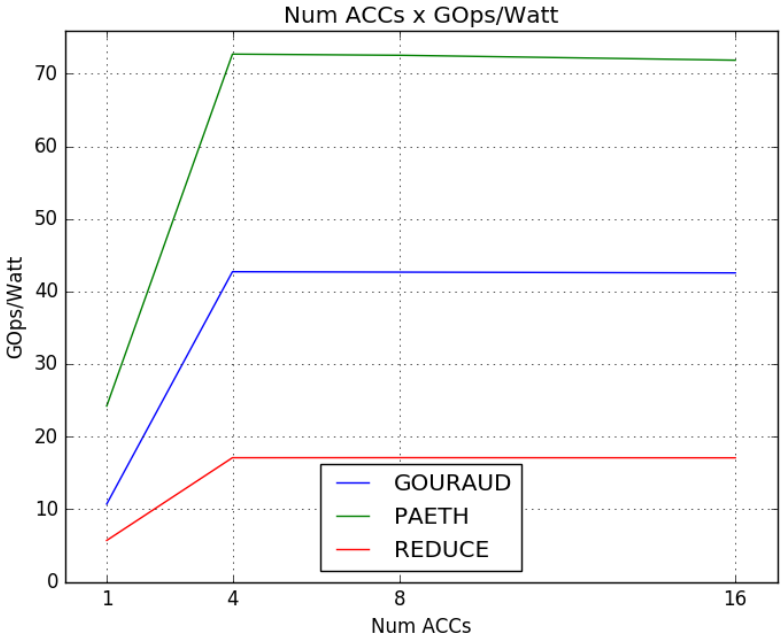



**Figura 3.8.** Energy efficiency in Gops/W for different ACC homogeneous sets executed in Broadwell Xeon Processor coupled to an Arria 10 FPGA.

heterogeneous multiple accelerators, where more than one kind of accelerator can be instantiated together. In addition, thanks to the high-level software API, the hete-

rogeneous accelerator computations can overlap in time. It is important to highlight that the OPAE HW API itself consumes 15-18% of the FPGA ALMs and 13% of FPGA RAM blocks.

**Tabela 3.3.** FPGA resource utilization for each ACC design in an Arria 10 FPGA.

| ACC | ALMs | Memory bits(KB) | RAM Blocks | DSP Blocks |
|---|---|---|---|---|
| fir16_1 | 91.166 (21%) | 408 (6%) | 439 (16%) | 512 (34%) |
| fir16_2 | 102.386 (24%) | 428 (6%) | 508 (19%) | 1.024 (67%) |
| gouraud_1 | 84.402 (20%) | 408 (6%) | 439 (16%) | 0 (0%) |
| gouraud_4 | 98.899 (23%) | 468 (7%) | 646 (24%) | 0 (0%) |
| gouraud_8 | 120.693 (28%) | 554 (8%) | 990 (36%) | 0 (0%) |
| gouraud_16 | 162.475 (38%) | 721 (11%) | 1.610 (59%) | 0 (0%) |
| paeth_1 | 84.699 (20%) | 408 (6%) | 439 (16%) | 8 (<1%) |
| paeth_4 | 99.814 (23%) | 468 (7%) | 646 (24%) | 32 (2%) |
| paeth_8 | 123.319 (29%) | 554 (8%) | 990 (36%) | 64 (4%) |
| paeth_16 | 167.783 (39%) | 721 (11%) | 1.610 (59%) | 128 (8%) |
| reduce_1 | 83.117 (19%) | 408 (6%) | 439 (16%) | 0 (0%) |
| reduce_4 | 93.086 (22%) | 468 (7%) | 646 (24%) | 0 (0%) |
| reduce_8 | 110.245 (26%) | 554 (8%) | 990 (36%) | 0 (0%) |
| reduce_16 | 141.610 (33%) | 721 (11%) | 1.610 (59%) | 0 (0%) |
| mixed_4 | 104.477 (24%) | 468 (7%) | 646 (24%) | 520 (34%) |

As a proof of concept, a mixed design with one copy of each accelerator was created as shown in the last line of Table 3.3. Regarding the performance, the mixed design was evaluated with a homogeneous workload for the 4 applications, and therefore, the performance was bound by the worst case, the reduce benchmark, reaching only 4.4 Gops/s. Althgough presenting poor performance, using heterogeneous accelerators can be very interesting in some application scenarios: (a) One or few accelerators copies with high data reuse rates; (b) No FPGA partial reconfiguration is required to swap the accelerator bitstream at run-time; (c) Once one accelerator computation finishes, in few cycles the user can run another accelerator; (d) Testing multiple ACCs since it saves compilation time and it also has the potential for improving the FPGA resource usage since idle resources can be used to implement more ACCs.

We have also implemented a simple loopback design using our framework to compare to the automatically code generated by Intel/OpenCL for FPGA. Our compilation time is less than the half of the OpenCL, where our design needs $3h21m$, and the OpenCL design spends $7h52m$. Both designs were compiled using the same

machine, a 2.8 GHz Xeon E5-2680 v2 processor with 96GB of RAM. We have also compared our design in terms of throughput. Our framework was able to reach 28 GB/s while OpenCL reached 27 GB/s considering that the send/receive data are overlapped. These results show that our is competitive with the current available tools like OpenCL.

## 3.5 Conclusions and Future Work

We have developed framework capable of automatically generating code for the hardware developers, who needs only to worry about integrating RTL accelerator to a simple interface. The framework is capable of creating multiple ACCs interfaces with just a few lines of Python code for the Intel CPU-FPGA research platform [2]. In addition, our framework saves compile time since there is resource sharing among the accelerators enabling the developers to easily replicate designs in order to achieve a better resource utilization.

Our results show that replicating homogeneous accelerators into multiple instances drastically improves both energy efficiency and throughput, when the kernel has a high reuse data rate. Moreover, our framework beats well stabilised high-level industrial tools in terms of throughput for a simple application. Althought currently presenting poor performance in our examples, the use of heterogeneous accelerators can be interesting to save compile time. We believe that it is possible to improve the heterogeneous accelerator performance by increasing data reuse among the accelerators.

We plan to extend our framework to other cloud platforms as [3, 4] and FPGA HPC boards with more memory bandwidth. In addition, we also plan to use our platform in scenarios which demand an heterogeneous set of accelerators for stream based application with high reuse data rate.

# Capítulo 4

# MIMT: Uma arquitetura reconfigurável com Múltiplas Instruções Múltiplas *Threads*

***Resumo*** - As arquiteturas reconfiguráveis são flexíveis e podem prover desempenho para aplicações específicas, sendo uma solução promissora em sistemas heterogêneos. Dentro desta categoria de arquiteturas, podemos destacar os FPGAs. Embora os FPGAs permitam soluções com eficiência energética que exploram o paralelismo do problema, o preço a ser pago é sua complexidade gerada pela reconfiguração a nível de *bit* e os tempos de compilação da ordem de horas. Outra grande barreira é a dificuldade de especificar o problema, pois a maioria das soluções eficientes são descritas na forma de circuito lógico, demandando também conhecimentos de linguagens de descrição de *hardware* como Verilog e VHDL do programador. Para reduzir a complexidade, os CGRAs foram propostos com reconfiguração a nível de palavra e programados por meio de mapeamento de grafos de fluxo de dados, reduzindo o tempo de compilação para ordem de segundos. Este trabalho apresenta um novo CGRA ao propor uma arquitetura reconfigurável capaz de executar múltiplas *threads* com registradores embarcados na rede de interconexões. A arquitetura proposta complementa as GPUs que possuem um modelo SIMT (*Single Instruction Multiple Thread*), introduzindo o modelo MIMT (*Multiple Instruction Multiple Thread*). Para validar a nova arquitetura, um conjunto de ferramentas de *software* e *hardware* foram desenvolvidos. Para realizar a programação do CGRA, primeiramente, o algoritmo deve ser descrito na forma de grafo de fluxo de dados na linguagem C++. Todo o processo de mapeamento e roteamento do grafo de fluxo de dados é realizado de forma transparente para o CGRA. O CGRA foi implementado como uma

camada virtual sobre o FPGA da plataforma de computação heterogênea da Intel. Todo o processo de compilação e geração da configuração do CGRA é realizado em milissegundos. Apesar de ser uma prototipação, o CGRA foi em média 2 vezes mais rápido quando comparado a um processador Intel Xeon com múltiplos núcleos. Portanto, obteve-se uma redução de ordens de grandeza no tempo de compilação, simplificação do uso de uma arquitetura reconfigurável e desempenho mesmo sendo uma prototipação.

## 4.1 Introdução

O uso de aceleradores em sistemas de computação de alto desempenho (HPC) comprova que os processadores superescalares não oferecem mais o desempenho demandado. De acordo com a lista dos 500 sistemas mais rápidos do mundo, TOP500, dos dez primeiros sistemas, cinco usam GPUs da NVidia como aceleradores externos [59]. Além dos GPUs, outras arquiteturas têm sido avaliadas, onde podemos destacar os FPGAs que podem acelerar problemas, onde, o modelo de execução *Single Instruction Multiple Data* (SIMT) das GPUs não são bem sucedidos.Outro fator que torna os FPGAs boas alternativas quando comparadas as GPUs é o seu baixo consumo energético. Grandes empresas como *Microsoft*, *Intel* e *Amazon* têm investido em sistemas que usam FPGAs como aceleradores [3][2][4]. Segundo os pesquisadores John Hennessy e David Patterson [60], que receberam o prêmio Turing em 2017, o uso de linguagens e arquiteturas específicas tem acelerado o progresso das abordagens baseadas em aprendizado de máquina e existe uma demanda por novas ferramentas e arquiteturas.

Os FPGAs são arquiteturas capazes de ser customizadas para resolução de problemas específicos, porém, o seu processo de desenvolvimento é complexo quando comparado aos processos de desenvolvimento para aceleradores comuns como as GPUs. O principal motivo da complexidade no uso de FPGAs deve-se ao fato que seus programas são, em geral, descritos na forma de circuitos lógicos, com linguagens como VHDL e Verilog. Outro ponto importante é o fato dos FPGAs serem uma arquitetura reconfigurável de grão fino, ou seja, programados a nível de *bit*, demandando um alto tempo de compilação, devido aos processos de mapeamento, posicionamento e roteamento das estruturas internas. Para simplificar estas etapas, arquiteturas reconfiguráveis a nível de palavra ou arquiteturas de grão grosso conhecidas pelo termo CGRA (*Coarse-Grained Reconfigurable Architectures*) foram propostas e possuem um tempo de compilação rápido quando comparados com os

FPGAs [61]. Apesar de serem promissores, os CGRAs ainda não foram bem explorados comercialmente. Faltam ferramentas como compiladores para essas arquiteturas e a maioria dos CGRAs são projetos acadêmicos avaliados para problemas específicos em etapas de projeto, sem serem fabricados e avaliados em campo.

Embora não existam sistemas comerciais que usam CGRAs como acelerador, vários trabalhos de pesquisa têm usado CGRAs virtualizados em FPGAs para acelerar aplicações. Desse modo é possível obter as vantagens dos CGRAs sem a necessidade da existência do circuito em silício [62][63][64][65]. Esses tipos arquiteturas são conhecidos na literatura como *Overlays* ou OCGRAs. A maioria dos CGRAs possuem uma arquitetura com elementos de processamento dispostos com uma estrutura bi-dimensional em malha. A principal vantagem é a escalabilidade da rede de interconexão local. Entretanto, o mapeamento do grafo de fluxo de dados demanda algoritmos de posicionamento e roteamento com longos tempos de execução. Para simplificar estes passos, existem outras topologias como os CGRAs em linhas e os CGRAs com interconexão global. Porém o custo da interconexão e a latência podem ser tomar proibitivos. Este trabalho propõe uma nova arquitetura de OCGRA que complementa as GPUs com um novo modelo de execução para Múltiplas Instruções Múltiplas *Threads* (MIMT), que possui as vantagens do tempo de compilação de um CGRA. O modelo de interconexão é global, que simplifica o mapeamento, onde uma rede multiestágio com bancos de registradores internos reduz também a latência das interconexões que é mascarada com o uso dos múltiplos *threads*. Para validação do OCGRA como um acelerador de um sistema heterogêneo. um conjunto de ferramentas de *software* e *hardware* foi desenvolvido e validado na plataforma de pesquisa de arquiteturas heterogêneas (HARP) da *Intel* [2]. As principais contribuições deste trabalho são:

1. Uma nova arquitetura CGRA para execução de múltiplas *threads* heterogêneas com bancos de registradores embarcados na rede de interconexão para redução da latência.

2. Um conjunto de ferramentas de *software* e *hardware* para geração de código Verilog para implantação de aceleradores com OCGRAs em uma plataforma comercial de HPC com FPGA em nuvem;

3. Uma API C++ para mapeamento, roteamento e execução de programas descritos na forma de grafos de fluxo de dados na arquitetura de CGRA proposta.

O restante do trabalho está divido da seguinte forma: Na Seção 4.2 são apresentados trabalhos relacionados ao uso de CGRAs virtualizados em FPGAs como

aceleradores de sistemas heterogêneos, a Seção 4.3 descreve a nova arquitetura proposta e as ferramentas desenvolvidas para prototipação na plataforma HARP, na Seção 4.5 são apresentados os resultados para validação da proposta, por fim, a Seção 4.6 apresenta as principais conclusões e perspectivas de trabalhos futuros.

## 4.2 Trabalhos Relacionados

Com o interesse em simplificar o uso dos FPGAs nos últimos anos, vários trabalhos [65, 64, 61, 62, 63] apresentaram propostas de CGRAs projetados para funcionar como uma camada de abstração sobre os FPGAs. Essas arquiteturas são denominadas na literatura como CGRA virtualizados em FPGA e mais recentemente pelo termo *Overlays* ou OCGRAs.

Um OCGRA que utiliza os blocos de DSPs do FPGA como elementos de processamento (PEs) interconectados com uma topologia 2D em malha foi proposto em [65]. O OCGRA foi modelado para ser mapeado especificamente para o FPGA Xilinx Zynq XC7Z020, podendo não ser portável para FPGAs de outros fabricantes. Devido a topologia 2D e os DSPs exigir menor recursos do FPGA, foi possível sintetizar CGRA com até 400 PEs, porém dados de tempo de compilação de aplicações para o CGRA não foram mencionados no artigo. Diferente da abordagem adotada em [65], o projeto proposto neste trabalho foi implementado de forma a gerar código em Verilog puro, o que permite que a arquitetura projetada seja portável para plataformas de diferentes fabricantes.

Um compilador para extrair grafos de fluxos de dados de laços escritos em C/C++ (anotados com diretivas) e mapeá-los em um OCGRA sintetizado sobre um FPGA foi apresentado em [64]. O principal objetivo deste trabalho é diminuir o tempo de implementação e depuração de aceleradores de *hardware*. Uma biblioteca de OCGRAs foi construída, onde, dependendo da aplicação o compilador usa a arquitetura mais adequada, caso contrário, uma nova arquitetura é gerada e precisa ser sintetizada para o FPGA. Os OCGRAs gerados utilizam uma topologia de interconexão em malha com *torus*, onde os PEs das extremidades estão conectados aos PES da outra extremidade. A estratégia de escolher o melhor CGRA em um banco de dados de CGRAs é uma boa alternativa quando os CGRAs serão apenas virtualizados sobre o FPGA, porém em uma abordagem com o CGRA fabricado diretamente em um circuito essa abordagem é inviável. Com o objetivo de construir o CGRA com *threads* diretamente em silício nossa abordagem de uma arquitetura genérica para todos problemas é viável, uma vez que é possível obter um maior

desempenho em relação a um CGRA virtualizado sobre um FPGA.

Com a redução do tempo de compilação, o uso do CGRAs possibilita o uso de compiladores *Just-in-Time* ou JIT, um exemplo foi apresentado em [61] onde o *kernel* da aplicação é descrito em OpenCL [7] e mapeado em uma arquitetura específica de OCGRA. Desse modo é possível obter um ciclo de desenvolvimento semelhante ao usado em CPUs e GPUs. Como resultados foi comparado o tempo de compilação de aplicações desenvolvidas usando Vivado HLS[66] uma ferramenta para desenvolver para FPGA usando linguagem de alto nível da Xilinx.

Nas últimas duas décadas diversos CGRA vem sendo propostos onde a maioria deles utiliza a topologia em malha com ligações locais entre os PEs vizinhos como a arquitetura ADRES [67]. A vantagem é a escalabilidade da rede de interconexões. Porém o posicionamento e roteamento é NP-Completo e heurísticas devem ser usadas [68]. Entretanto, vários caminhos internos podem ficar desbalanceados e *buffers* devem ser inseridos para o correto funcionamento do *pipeline*. Uma alternativa para simplificar o mapeamento são as arquiteturas em tiras ou linhas (*Strip*) [69]. O preço a ser pago é o custo de rede de interconexão entre as linhas, o custo da memória de configuração e a baixa taxa de utilização de várias linhas. Redes multiestágio podem ser usadas para amortizar o custo das interconexões [70], técnicas podem ser usadas para reduzir o custo da memória de configuração [71] e reconfiguração e geração customizada podem melhorar a ocupação das linhas [72, 63, 62]. Outra alternativa é uma arquitetura com rede totalmente conectada, entretanto o custo da interconexão é proibitivo para redes *crossbar*, limitando esta abordagem a 16-32 elementos de processamento. O uso de redes multiestágio [73, 74] pode reduzir custo $O(n^2)$ da *crossbar* para $O(n \log n)$. Entretanto a latência capacidade de roteamento da rede restringe sua escalabilidade.

A arquitetura proposta neste trabalho reduz a complexidade do posicionamento e roteamento ao usar uma rede multiestágio global e ao mesmo tempo provê escalabilidade mascarando a latência com registradores embarcados na rede, onde múltiplos *threads* heterogêneos podem executar simultaneamente no modelo MIMT. A rede global resolve também o problema de ocupação das linhas das arquiteturas em tiras. Ademais, as ferramentas de *software* e *hardware* produzidas estão integradas a uma API C++ permitindo o uso transparente da arquitetura mapeada como um *overlay* na plataforma HARP da Intel. Estas ferramentas tem tempo de execução da ordem de milisegundos e podem ser integradas à compiladores JIT.

## 4.3 A Nova Arquitetura CGRA MIMT

Os processadores com arquiteturas superescalares exploram o paralelismo a nível de instruções, buscando várias instruções a cada ciclo de relógio, gerenciando as dependências de dados com técnicas de escalonamento dinâmico e execução fora de ordem. Porém, o consumo de energia e a complexidade do processador aumentam. Os processadores VLIW transferem para o compilador várias tarefas, simplificando o *hardware* e executando $n$ instruções por ciclo sem verificação de dependências. Entretanto em muitos trechos de código, o compilador não consegue escalonar $n$ instruções. Além disso, a nível de *hardware*, o custo do banco de registradores com $2n$ leituras e $n$ escritas limita a escalabilidade, sendo que o valor de $n$ em geral é menor ou igual a 8.

As GPUs introduziram um novo modelo com milhares (ou até milhões) de *Threads*. Cada *thread* armazena seu estado em um conjunto privado de registradores que deve permanecer vivo durante toda a execução do *Thread*. Os *threads* são agrupados em *warp* de 32 *threads* e executam a mesma instrução, simplificando a parte de controle do processador. Apesar de uma instrução demorar 10 ciclos para executar, se a próxima instrução não possuir dependências será executada em *pipeline* logo em seguida. Se a instrução possui dependência de dados, o *warp* sai de execução e no próximo ciclo outro *warp* irá executar uma instrução que estiver pronta. Portanto, ao disparar vários *threads* na GPU a latência da unidade de execução é mascarada fazendo com que ela sempre esteja ocupada, fazendo o uso da maior parte das suas unidades de processamento. Além disso, ao disparar milhares de *threads*, a GPU consegue mascarar também a longa latência das instruções de memória (*load/store*) que são aproximadamente 800 ciclos de relógio.

A Figura 4.1 mostra de forma simplificada o modelo das GPUs. Neste exemplo são ilustrados apenas 3 *warps* totalizando 96 *threads* em execução. A cada ciclo uma instrução de um *warp* é disparada, onde os 32 *threads* em paralelo irão ler seus registros da sua fatia do banco de registradores e enviar para uma unidade de execução que demora 10 ciclos em *pipeline*. No exemplo, o $warp_0$ está lendo os registradores da instrução $a = b + c$. No *pipeline* das unidades de execução estão ilustrados simbolicamente trechos dos $warp_0$, $warp_1$ e $warp_2$ executando as instruções $x = w * y$, $e = 2 + f$ e $a = b + c$, respectivamente. No próximo ciclo, o $warp_0$ não pode executar a instrução $d = a - 3$ pois tem dependência com a instrução $a = b + c$ que começou agora sua execução. Portanto, durante os próximos 10 ciclos somente os $warp_1$ e $warp_2$ poderão escalonar instruções. Em alguns casos, são necessários muitos *warps*. O $warp_2$ poderá executar $d = a - 3$ pois a sua

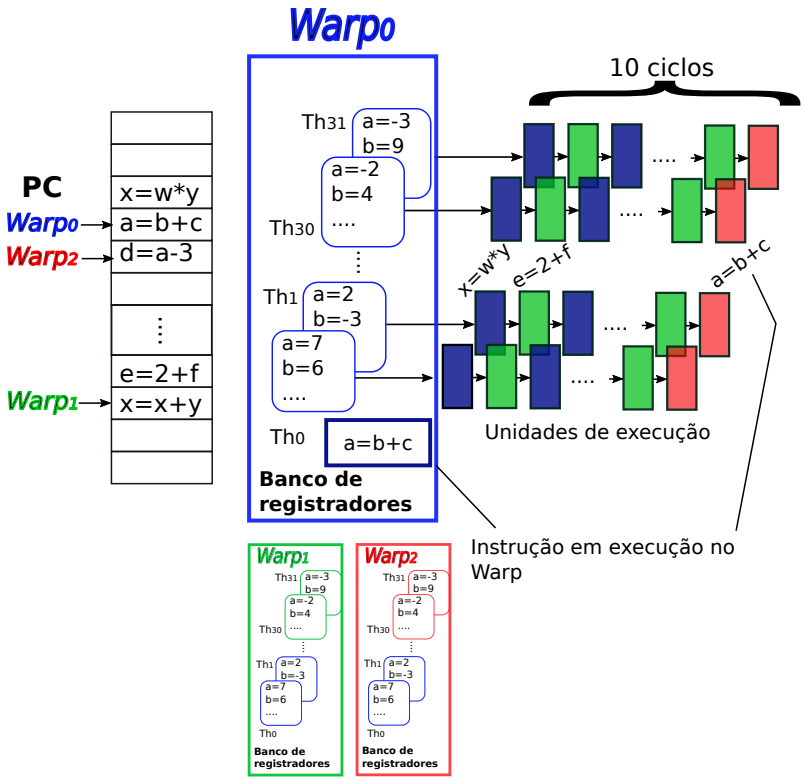



**Figura 4.1.** Exemplo de execução com 3 Warps e 96 threads em uma GPU com 10 estágio de pipeline na unidade de execução.

instrução $a = b + c$ já está terminando a execução e pode ser escalonada no próximo ciclo. O $warp_1$ também pode executar $x = x + y$. A vantagem da GPU é sua a rápida troca de contexto dos *warps* em um ciclo de relógio, porém precisamos alocar milhares registradores e centenas de *threads* para manter a GPU em execução. Neste exemplo ilustrativo com poucos *threads*, se cada *thread* tem 20 registradores, estarão alocados $96 * 20 = 1920$ registradores para apenas 3 *warps* em execução, ou seja, manter os *warps* vivos gera impacto no *hardware*. Os registradores consome uma parte significativa de energia nas GPUs, pelo menos 15% da energia [75].

Uma GPU moderna tem de 10 à 20 milhões de registradores e 2000 a 5000 unidades de execução. Porém a GPU fica limitada ao paralelismo SIMT com um demanda de milhares de cálculo simultâneos com mesma operação, onde a GPU tira proveito da sua estrutura de *hardware* como acelerador para mascarar a latência e gerar desempenho. O sucesso das GPU é explicado pela simplicidade de programação (modelo CUDA) e várias aplicações que podem ser facilmente mapeadas no modelo SIMT com milhares de *threads* como processamento de imagens, vídeos, criptografia, dentre outras. Porém algoritmos com o paradigma *Map-Reduce* tendem a

deixar unidades de processamento das GPUs ociosas, o que causa uma diminuição do desempenho. Outro problema das GPUs é a presença de desvios condicionais causando divergência e perda de desempenho dentro de cada *Warp*. A arquitetura proposta possui operadores de controle que implementa comandos condicionais sem perda de desempenho e problemas de divergências.

Neste trabalho introduzimos uma nova arquitetura, que complementa as GPUs e outras arquitetura abrindo novas oportunidades para serem incluídas em sistemas heterogêneos. Diferente das GPUs que precisam de centenas ou milhares de *Threads* para mascarar as latências das dependências de dados, nossa proposta precisa precisa apenas de poucos *threads* e poucos registradores. A Figura 4.2 ilustra de forma simplificada a nova arquitetura *multi-thread* proposta. O exemplo tem 128 unidades de execução ou elementos de processamento (PEs). A saída do PE está ligada a registrador. O *thread* estará mapeado nos PEs e seus registradores serão armazenados nas saídas. A cada ciclo um *thread* irá sair da rede de interconexão e ir para as unidades de execução, enquanto o *thread* atual irá entrar na rede de interconexão para percorrê-la. No exemplo, a rede tem apenas 2 estágios. Portanto, o modelo requer 3 *threads* apenas para manter o desempenho. Em contraste com as GPUs, cada *thread* pode executar uma aplicação diferente. No exemplo, três aplicações com grafo de fluxo de dados diferentes estão sendo executadas, caracterizando o modelo MIMT.

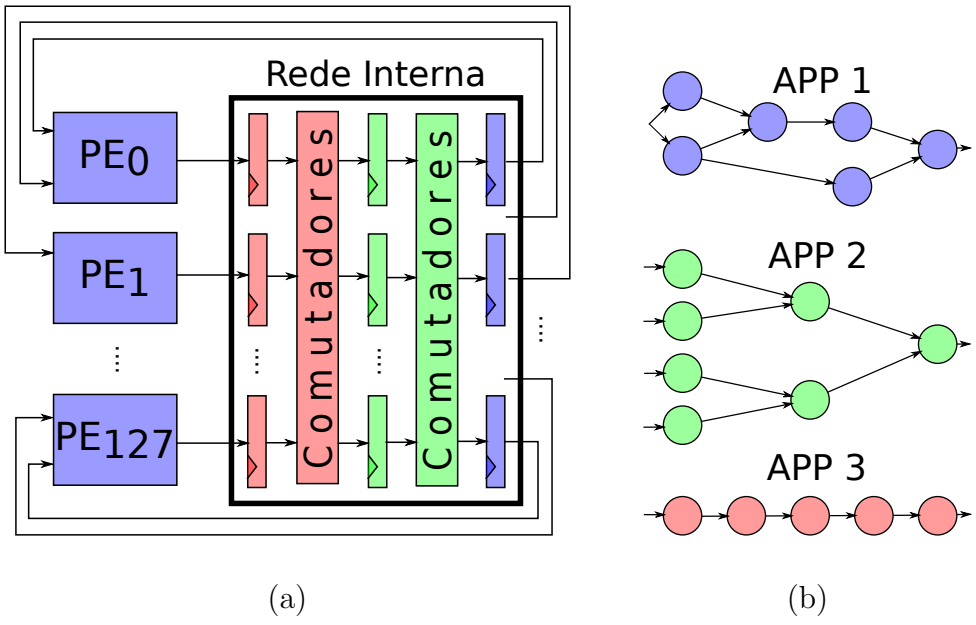
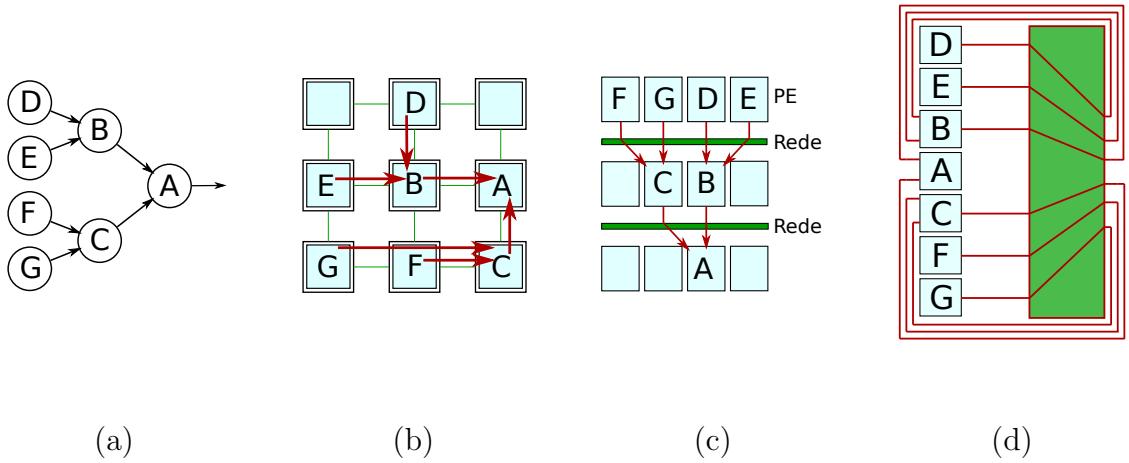



**Figura 4.2.** (a) Execução de três threads no modelo MIMT; (b) Três aplicações com grafos de fluxo de dados perfis diferenciados.

A arquitetura é genérica, as unidades de processamento podem, como as GPUs, ter uma latência de 10 ciclos em *pipeline*, mas neste caso precisamos ter 12 *threads* para mascarar a latência. A rede também pode ter latência maior aumentando assim o número de PEs. Outro ponto importante a destacar, é que o banco de registradores de cada *thread* irá caminhar pela rede, tendo acesso paralelo de escrita e leitura, no exemplo 128 acessos que é significativamente maior que um processador VLIW com 8-16 acessos. Outro aspecto é o modelo de grafo de fluxo de dados que está mapeado na arquitetura. O grafo pode ter qualquer topologia como ilustrado na Figura 4.2, onde a aplicação 2 é um grafo estilo redução e a aplicação 3 é um grafo sequencial que terá o paralelismo temporal explorado. Estes exemplos de grafos não são apropriados para a execução em arquiteturas de uma GPU. O grafo pode também ser um misto de paralelismo temporal e espacial como da aplicação 1. Na Figura 4.2(a), a aplicação 1 está executando e as aplicações 2 e 3 estão na fase de roteamento enviando os dados para a execução do próximo ciclo, onde a aplicação 2 irá executar nos PEs, enquanto as aplicações 3 e 1 estarão na fase de roteamento dos dados. A rede de interconexão global simplifica o posicionamento e roteamento. A arquitetura permite também mapear duas ou mais dentro de um mesmo "*slot de thread*"ou mapear um grafo com mais operadores que a arquitetura usando a técnica de *module scheduling* para escalonar [67]. Mais uma vantagem é o fato do modelo de grafo de fluxo de dados eliminar a busca e decodificação das instruções a cada ciclo, reduzindo o consumo de energia gerado por estas etapas em todo ciclo, pois o grafo é carregado uma única vez na memória de configuração da arquitetura e permanecerá no estágio de execução até finalizar.

Outros aspectos da abordagem de CGRA com uso de rede global devem ser destacados. Comparando com as arquiteturas de CGRAs em malha e em tiras. A Figura 4.3 ilustra um grafo de fluxo de dados mapeado nos três modelos: malha, tiras e a global. Primeiro, o modelo em malha requer mais tempo para encontrar um mapeamento otimizado do grafo que está ilustrado na Figura 4.3(b), onde apenas a aresta $G \rightarrow C$ foi mapeada com distância de 2 PEs. Entretanto, o grafo ficará desbalanceado, sendo necessário inserir registradores auxiliares para realizar o balanceamento. O primeiro na aresta $F \rightarrow C$ para balancear o caminho até $C$ e o segundo na aresta $B \rightarrow A$ para balancear o caminho do sub-grafo da parte de cima da redução até $A$. Além disso, o modelo em malha dificulta uma execução *multi-threads*, caso seja necessário esconder a latência. O modelo em tiras, ilustrado na Figura 4.3(c) já possibilita uma implementação *multi-threads*, simplifica o mapeamento e todas as conexões passam a ter atraso de 1 ciclo, eliminando a inserção de registradores adicionais. Porém o custo da rede e a baixa ocupação de algumas




(a) (b) (c) (d)

**Figura 4.3.** (a) Grafo de Fluxo de Dados com uma Redução;(b) Mapeamento do Grafo em uma arquitetura CGRA em Malha; (c) Mapeamento em um CGRA em Tiras; (d) Mapeamento no CGRA Proposto com Rede Global de Interconexão.

linhas é a principal desvantagem deste modelo. O modelo proposto aqui, ilustrado na Figura 4.3(d), maximiza a ocupação dos PEs além das vantagens enumeradas para modelo de tiras. Além disso, a rede é compartilhada, reduzindo o custo global da arquitetura.

### 4.3.1 Arquitetura da rede de Interconexão

Nesta seção é dado o foco em detalhes da rede global de interconexões adotada. Uma alternativa seria uma rede *crossbar* [76], onde o posicionamento e roteamento tem complexidade $O(1)$. Entretanto, o custo da rede é $O(n^2)$ que a torna proibitiva para quantidades de PEs acima 32. O custo pode ser reduzido para $O(n \log n)$ usando as redes multiestágios como a rede *omega* que pode ser eficientemente mapeada em FPGA como avaliado em [77], onde os recursos e o ciclo máximo de relógio de vários tamanhos de OCGRAs são estudados. Porém a latência cresce em função do tamanho da rede o diminui o número de operações realizadas nos PEs. A nova arquitetura proposta soluciona esta questão com o uso do *multi-thread*, como mencionando anteriormente.

A rede *omega* desenvolvida possui $O(\log n)$ estágios de comutadores e sua geração foi parametrizada em função do *radix* do comutador. Estágios extras podem ser adicionados a rede para melhorar sua capacidade de roteamento, uma vez que uma rede com apenas $\log n$ estágio é bloqueante e pode não suportar o roteamento demandado. Na literatura, a maioria das rede multiestágio são projetadas com *radix* 2. Redes com *radix* 4 são mapeadas de forma mais eficiente em FPGA e

reduzem os conflitos de roteamento [78]. A implementação proposta aqui avaliou as estruturas com *radix* 2,4 e 8. O *radix* 8 mostrou o melhor custo benefício em área e em capacidade de roteamento.

Diferente das redes *omega* implementadas em [77, 78], onde uma memória de configuração centralizada armazenava as configurações de todos comutadores, a implementação proposta aqui é distribuída para facilitar a configuração do modelo *multi-thread*. Para realizar a troca de configuração do comutador e a escrita das configurações na memória, cada comutador possui um controle individual, como mostrado na Figura 4.4. As configurações são recebidas por meio de um barramento que interliga todos os controles, cada comutador possui um identificador único, e por meio desse identificador o controle realiza leitura dos dados de configuração do barramento e os escreve em sua memória.

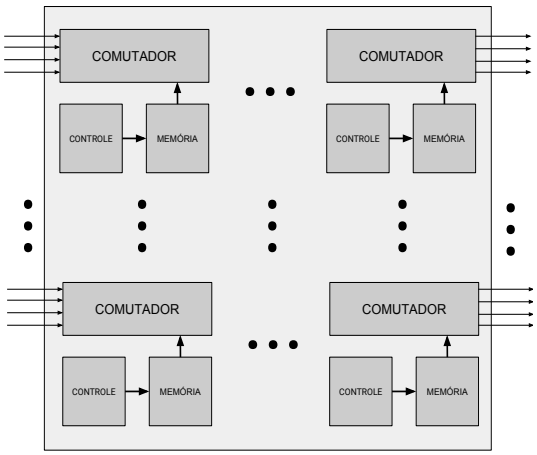



**Figura 4.4.** Arquitetura rede *omega*.

### 4.3.2 Elemento de processamento (PE)

Para implementar a nova arquitetura é necessário definir a funcionalidade dos elementos de processamento (PEs). Neste trabalho para validação inicial da proposta, optamos por um conjunto homogêneo de PEs. A definição do PE é independente da arquitetura e da rede, e pode ser facilmente modificado. Como primeira proposta, cada PE possui duas entradas e duas saídas de dados e é capaz realizar operações lógicas e aritméticas. Cada PE possui sua memória de configuração local, denominada memória de instrução e o mecanismo adotado para realizar sua configuração é o mesmo dos comutadores, por meio de um barramento, onde cada PE possui um identificador próprio e um controlador responsável por receber as instruções do

barramento e escrevê-las na memória. Como iremos validar a arquitetura em um FPGA, para que a latência do PE não aumente o ciclo de relógio, o PE possui *pipeline* interno.

Cada PE possui um banco de registradores onde cada *thread* pode alocar 16 registradores. Os registradores armazenam dados da ALU ou podem ser inicializados na etapa de configuração que funcionam como valores imediatos ou constantes. Como já mencionado os PEs são homogêneos, porém operações de entrada e saída são restritas a um subconjunto dos PEs. Os PEs são classificados de três formas: PEs de entrada, PEs de saída e PEs básicos. A Figura 4.5 mostra o caminho de dados do PE onde as partes tracejadas são partes pertencentes apenas aos PEs que realizam entrada e saída de dados sendo uma parte tracejada de vermelho e outra de azul para entrada e saída respectivamente.

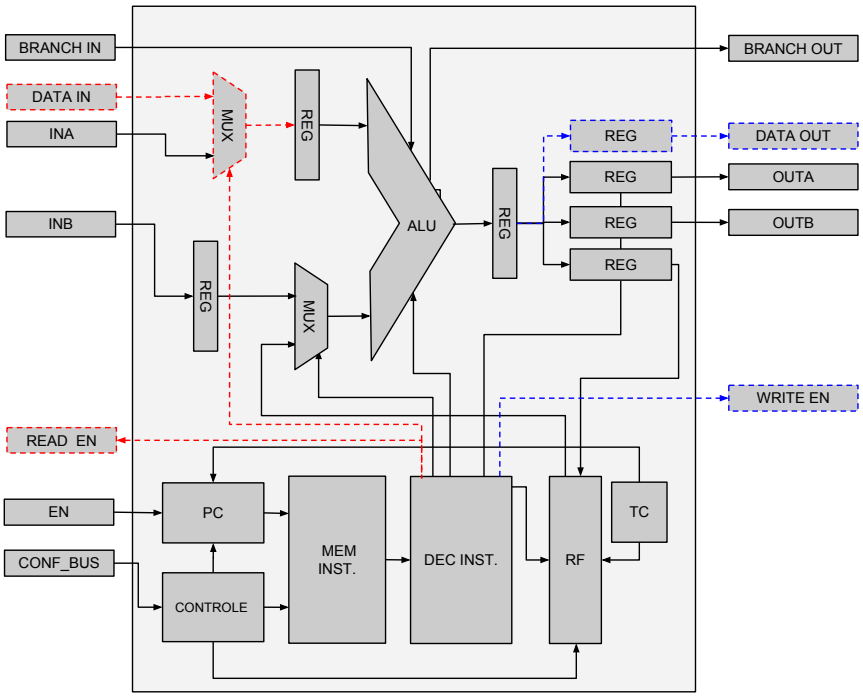



**Figura 4.5.** Caminho de dados dos elementos de processamento.

Durante a execução cada PE pode realizar a computação de múltiplos operadores do grafo por meio da técnica de *module scheduling*, para isso um contador de programa (PC) é incrementado para o próximo endereço da memória contendo a instrução do operador do grafo no qual foi previamente mapeado para o PE a cada ciclo de *clock*. Como o PE executa mais de uma *thread* ou seja mais de um DGF em *pipeline* cada *thread* possui seu próprio PC. Um contador de *thread* (TC) também

foi usado para simplificar a implementação, assim todas as *threads* sempre são executadas sequencialmente, mesmo que não tenha nenhum DFG alocado nesta *thread*. Ou seja o TC é incrementado a cada ciclo de *clock* trocando a execução da *thread* atual para a próxima *thread*. Os PEs são capazes de realizar operações condicionais, neste caso a saída BRANCH OUT é utilizada para propagar a resposta da operação para outro PE por meio de uma rede de interconexão específica para os sinais de controle de desvio. Essas operações possibilitam a execução de grafos com fluxo de controle. A Tabela 4.1 apresenta todas as operações que são possíveis realizar nos PEs sobre suas entradas.

**Tabela 4.1.** Operações realizadas pelos PEs

| Classe | Nome | Operação |
|---|---|---|
| Aritmética | ABS | \|a\| |
| | ADD | a + b |
| | MULT | a * b |
| | SUB | a - b |
| Lógica | AND | a & b |
| | XOR | a $\oplus$ b |
| | OR | a \| b |
| | NOT | !a |
| Controle de fluxo | BEQ | a == b ? 1 : 0 |
| | BNE | a != b ? 1 : 0 |
| | SLT | a < b ? 1 : 0 |
| | SGT | a > b ? 1 : 0 |
| | MUX | BRANCH IN ? a : b |
| Comparação | MIN | a < b ? a : b |
| | MAX | a > b ? a : b |
| Deslocamento de *bits* | SHL | a << b |
| | SHR | a >> b |
| Outras | PASS A | a |
| | PASS B | b |

As operações listadas na Tabela 4.1 podem ser realizadas sobre as entradas e sobre os registradores de cada PE, o resultado das operações podem ser enviadas para outra PE pelas saídas OUTA ou OUTB ou podem ser escritas em seu próprio banco de registradores. Os resultados das operações de desvios condicionais sempre usam como saída a porta BRANCH OUT. A operação MUX usa a porta BRANCH IN para realizar a escolha de umas das entradas, permitindo a execução grafos com desvios condicionais.

As instruções de cada PE é formada por uma palavra de 16 *bits*, como pode ser observado na Tabela 4.2, a palavra possui 4 campos de 4 *bits* cada, sendo o primeiro campo relativo às operações da ALU, é importante ressaltar que é possível realizar mais que 16 operações na ALU embora apenas 4 *bits* são dedicados a sua configuração, isto se dá devido o fato das operações que envolvem desvio condicional como as operações: BEQ, BNE, SLT e SGT que usam a saída BRANCH OUT, possuírem o mesmo código que as operações PASS A, PASS B, MIN e MAX respectivamente. O segundo campo de 4 *bits* é responsável por selecionar as entradas e saídas da ALU. Em PEs que possuem entrada de dados externa, é possível selecionar como entrada da ALU as portas DATA IN, INA, INB e um registrador. Para cada combinação dessas entradas é possível enviá-las para a saída OUTA ou OUTB ou um registrador. Para os PEs que possuem saída de dados externa também é possível enviar a saída da ALU para a porta DATA OUT. Os demais 8 *bits* da palavra são responsáveis pelos endereços de leitura e escrita de registradores no banco de registrador local do PE sendo 4 *bits* de endereço de leitura e 4 *bits* para escrita.

**Tabela 4.2.** Palavra de instrução dos PEs

| ALU | | CONTROL | | RADDR | | WADDR | |
|---|---|---|---|---|---|---|---|
| 0 | 3 | 4 | 7 | 8 | 11 | 12 | 15 |

### 4.3.3 Ferramenta para geração de código Verilog

A arquitetura proposta pode ser usada sobre FPGAs comerciais que possuem recursos limitados. Para isso uma ferramenta para a geração de código em Verilog da arquitetura foi implementada utilizando a linguagem Python juntamente com uma biblioteca que permite a geração automática de código Verilog [56] de todo o OCGRA. A ferramenta permite ao usuário personalizar todos detalhes do OCGRA a ser gerado, como: Quantidade de PEs totais, quantidades de PEs com entrada externa, quantidades de PEs com saída externa, tamanho da palavra de processamento em *bits*, tamanho das memórias de configuração, *radix* da rede de interconexão e quantidade de estágios extras da rede. A Figura 4.6 apresenta um diagrama de blocos da ferramenta. A interface depende da plataforma alvo.

A ferramenta desenvolvida está integrada com o FDAM (*Framework for Dataflow Accelerator Management*). O FDAM tem o objetivo de facilitar a implantação de vários aceleradores na plataforma de pesquisa de arquiteturas heterogêneas da

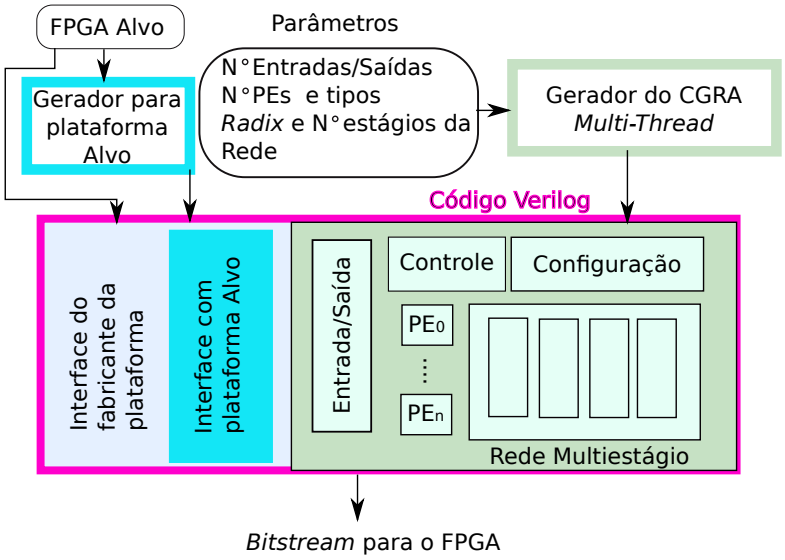



**Figura 4.6.** Diagrama de blocos dos Geradores desenvolvidas para a implementação do OCGRA em FPGAs comerciais.

Intel HARP [79]. O HARP é um sistema heterogêneo que possui um FPGA acoplado com o CPU por meio de barramento de alta velocidade, mais detalhes do sistema HARP pode ser visto em [2]. Assim é possível gerar mais de um acelerador com OCGRAs de diferentes tamanhos, ou até outros aceleradores para computações específicas e usá-los diretamente no sistema HARP, usando a API em C++ do FDAM que permite o gerenciamento dos aceleradores.

## 4.4 Modelo de Programação do OCGRA

Semelhante ao modelo das APIs do CUDA adotado pela NVIDIA que favoreceu a rápida difusão da programação em aceleradores com GPU, nesta seção apresentamos uma API para programação com grafos de fluxo de dados mapeados em OCGRA. O modelo tem portabilidade e pode ser implementado para outras arquiteturas de OCGRA. O programador usa classes C++ para instanciar o grafo de fluxo de dados, mapeá-lo no OCGRA, conectar as entradas e saídas em estrutura de dados do código *host* do processador e executar a aplicação mapeada no OCGRA virtualizado no FPGA de alto desempenho. A Figura 4.7 ilustra um esboço da proposta. O programador descreve a aplicação no formato de um grafo, que é mapeado para um objeto CGRA. Vários grafos podem ser instanciados, o exemplo da Figura 4.7 ilustra a execução com quatro *threads* para a mesma aplicação que foram mapeadas no CGRA com o método *addDataflow*. Trabalhos futuros podem incluir um compilador

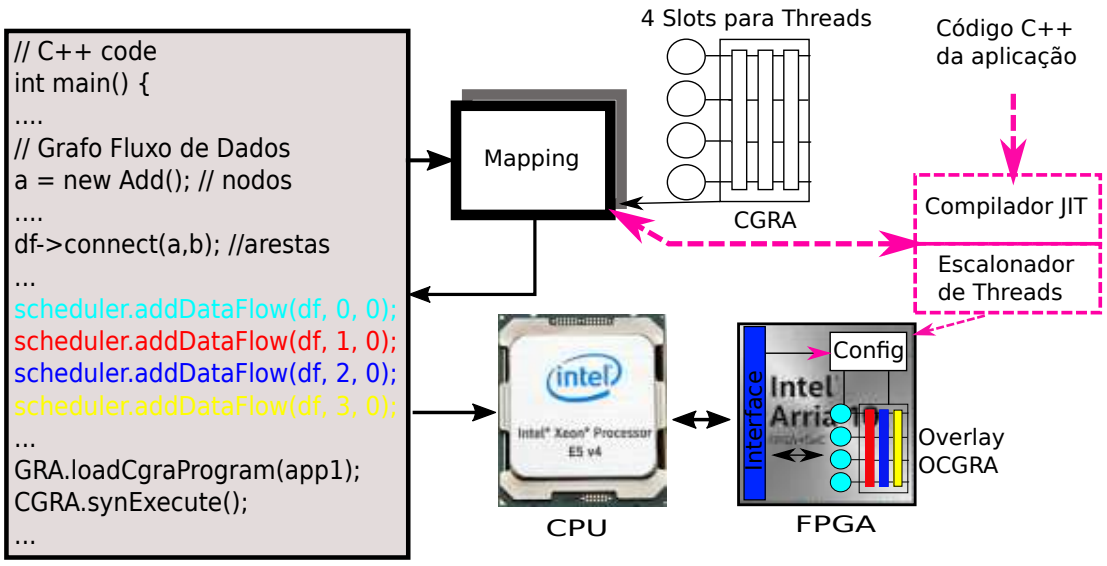



**Figura 4.7.** Exemplo de mapeamento com a API proposta

JIT para extração do grafo e escalonamento em tempo de execução (ilustrados de forma tracejada na Figura 4.7). Além da API, foram desenvolvidas ferramentas para implementar as partes de *software* e *hardware* que possibilitam o uso do OCGRA na plataforma HARP da Intel onde um processador Xeon com múltiplos núcleos é fortemente acoplado a um FPGA Arria 10.

### 4.4.1 Controle de configuração e execução da arquitetura

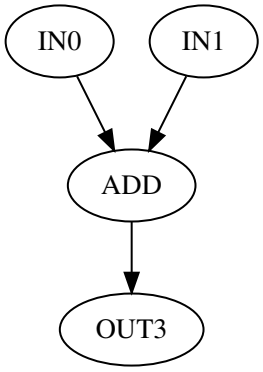
Todo o controle de execução da arquitetura é realizado por entidades externas. São dois módulos principais que realizam esse controle, o primeiro é o controle de configuração, este módulo é responsável por receber as configurações e enviá-las para o barramento de comunicação de configuração do OCGRA. Após todas as configurações serem enviadas o modulo de execução é acionado, este módulo é responsável por gerenciar as filas de dados de entrada e saída, para que caso alguma fila pare por falta de dados no caso das filas de entrada ou por estar cheias no caso das filas de saída, um sinal para desabilitar todo o OCGRA é ativado até a normalização das filas.

Todas as mensagens de configurações enviadas para o OCGRA pelo barramento possuem o mesmo tamanho de 64 *bits*. Devido o fato de existir diferentes tipos de configurações esse pacote de dados possui campos distintos para cada configuração, sendo que apenas um cabeçalho é igual para todos. O cabeçalho é definido

pelos 24 primeiros *bits* do pacote que possui dois campos um de 8 *bits* para tipo de configuração e outro de 16 *bits* para um identificador único de cada entidade que pode ser configurada. As três entidades que recebem configurações são os PEs e os comutadores da rede de dados entre os PEs e os comutadores da rede de sinais de desvio entres os PEs.

### 4.4.2 Modelo de Programação e Execução

Cada aplicação a ser mapeada na arquitetura do OCGRA é descrita na forma de grafo de fluxo de dados (GFDs). Primeiro o grafo é instanciado. O segundo passo é o mapeamento, que inclui a alocação dos PEs e roteamento das ligações no OCGRA. A API C++ desenvolvida oferece uma interface em alto nível para o programador. A Figura 4.8 mostra um exemplo de GFD e sua representação em C++ usada pela classe *Scheduler* para gerar as configurações para o OCGRA. O exemplo inclui dois *streams* de entrada, um operador de soma e um *stream* de saída. Os operadores e ligações devem ser instanciados. A quantidade de DFGs que podem ser executadas ao mesmo tempo dependem da quantidade de *slots de threads* que a arquitetura de OCGRA possui.




(a)

```
...
auto df = new DataFlow(0,"soma_vetorial");
auto in1 = new InputStream(0);
auto in2 = new InputStream(1);
auto add = new Add(2);
auto out = new OutputStream(3);

df->connect(in1,add,add->getPortA());
df->connect(in2,add,add->getPortB());
df->connect(add,out,out->getPortA());
...
```

(b)

**Figura 4.8.** API para descrição de grafos de fluxo de dados: (a) Grafo de fluxo de dados de uma soma vetorial; (b) Representação do grafo da soma vetorial em C++.

### 4.4.3 API para uso na plataforma HARP

Como mencionado na Seção 4.3.3, é possível realizar a implantação de vários OCGRAs como aceleradores na plataforma HARP utilizando o *framework* FDAM.

Deste modo uma API em C++ foi desenvolvida para permitir o usuário a usar o OCGRA. Está API possui métodos que permite o usuário a carregar um programa para o OCGRA, configurar os ponteiros dos dados de entrada e saída dos GFDs contidos no programa e então iniciar a computação dos dados. A Figura 4.9 apresenta um exemplo de código execução na plataforma HARP. Primeiro é preciso instanciar um objeto da classe *Cgra* e realizar a alocação dos vetores de dados que serão enviados para processamento. Caso os DFGs criados tenham sido previamente compilados para o OCGRA e salvo em arquivo, então é preciso apenas o carregamento desse arquivo para o objeto Cgra por meio do método *loadProgram* e então realizar a passagem de argumentos para os GFDs por meio dos métodos *setCgraProgramInputStreamByID* ou *setCgraProgramInputStreamByName* onde devem ser passados os parâmetros: identificador do GFD ou o nome, o identificador do operador que vai realizar a entrada dos dados, o ponteiro para os dados e a quantidade de *bytes* a serem processados. O mesmo procedimento é realizado para realizar a passagem dos argumentos de saída dos GFDs porém devem ser usados os métodos *setCgraProgramOutputStreamByID* ou *setCgraProgramOutputStreamByName*. Após a passagem dos argumentos para os GFDs é possível iniciar a computação por meio dos métodos *syncExecute* e *asyncExecute*, sendo o primeiro bloqueante e o segundo não bloqueante. Caso os GFDs descritos em C++ ainda não tenham sido previamente compilados para o OCGRA e salvos em um arquivos é possível também utilizar os objetos das classes *CgraArch* e *Scheduler* para realizar o agendamento dos DFGs então obter o programa da classe *CgraArch* usando o método *getCgraProgram* e realizar seu carregamento na classe Cgra utilizando uma sobrecarga do método *loadCgraProgram*.

## 4.5 Resultados

Para demonstrar o potencial da nova arquitetura proposta e as facilidades proporcionadas pelo conjunto de ferramentas desenvolvidas, um conjunto de 4 algoritmos foram implementados na forma de grafo de fluxo de dados. A Tabela 4.3 apresenta as características dos algoritmos implementados. Todos os foram descritos utilizando a API apresentada na Seção 4.4.2. O FIR é filtro de impulso que implementa uma convolução 1D. O FIR utilizado possui 63 coeficientes. O segundo algoritmo é o *K-means* , um algoritmo de aprendizado de máquina que realiza agrupamento de dados. A versão implementada possui $K = 8$ agrupamentos sobre dados com 5 dimensões. Este algoritmo utiliza comandos condicionais para selecionar em qual

```
#include <fdam/cgra/cgra.h>

using namespace std;

int main(int argc, char *argv[]) {
  Cgra cgra;
  short ina[1024];
  short inb[1024];
  short out[1024];
  size_t len = 1024 * sizeof(short);
  for (short i = 0; i < 1024; ++i) {
    ina[i] = (short) (random() % 256);
    inb[i] = (short) (random() % 256);
    out[i] = 0;
  }
  cgra.loadCgraProgram("soma_vetorial.cgra");
  cgra.setCgraProgramInputStreamByID(0,0,ina,len);
  cgra.setCgraProgramInputStreamByID(0,1,inb,len);
  cgra.setCgraProgramOutputStreamByID(0,3,out,len);
  cgra.syncExecute(0);
  for (short i : out) {
    printf("%d ", i);
  }
  return 0;
}
```

**Figura 4.9.** Exemplo de código para execução na plataforma HARP.

agrupamento o ponto será classificado. O terceiro exemplo é o algoritmo Paeth para compressão de imagem no formato PNG. Devido a fato do *Paeth* possuir poucas operações, um grafo com duas cópias do algoritmo foi descrito para maximizar o uso dos PEs da arquitetura. O quarto exemplo é o algoritmo Sobel que realiza a detecção de bordas em imagens por meio de convoluções utilizando máscaras 3x3 previamente calculadas. O conjunto de algoritmos implementados neste trabalho são melhores descritos no Apêndice A.

A Tabela 4.3 apresenta o tempo de compilação dos grafos de fluxo de dados para mapeamento no OCGRA. Importante destacar que este tempo se refere a todo processo de mapeamento e roteamento gerando como saída todas informações necessárias para a execução do algoritmo no OCGRA. Este passo é semelhante a compilação com geração do *bitstream* de um FPGA. Enquanto a geração completa do *bitstream* do OCGRA é da ordem de milissegundos, a geração do *bitstream* para um FPGA pode demorar minutos ou até horas.

**Tabela 4.3.** Características dos DFs dos algoritmos utilizados

| Algoritmo | Nós | | | Tempo de compilação(ms) |
|---|---|---|---|---|
| | Entrada(s) | Saída(s) | Total | |
| FIR | 1 | 1 | 128 | 10,5 |
| K-means | 5 | 1 | 116 | 11,5 |
| Paeth | 6 | 2 | 108 | 8,5 |
| Sobel | 8 | 1 | 61 | 3,2 |

A arquitetura OCGRA proposta e o conjunto de ferramentas desenvolvido foi validado virtualizando o OCGRA na plataforma HARP [2], onde foi também utilizado o *framework* FDAM [79] para gerar um acelerador com um OCGRA customizado. A customização do OCGRA foi realizada de forma a obter a maior quantidade de PEs possível sem degradar a frequência máxima do acelerador e sem ultrapassar a quantidade de recursos de *hardware* da FPGA. OCGRA sintetizado para validação possui 128 PEs com 8 entradas e 8 saídas externas. O acelerador completo consumiu 58% dos recursos do FPGA, sendo 37% utilizados para o OCGRA e 21% para a *interface* de comunicação. A plataforma HARP juntamente com o *framework* OPAE[47] disponibiliza 3 opções de *clocks* 100Mhz, 200Mhz, 400Mhz para o acelerador. A versão atual foi mapeado com 200Mhz podendo ter um desempenho de pico de $128x200M = 25{,}6$ Gops/s. Futuras versões com uma análise de tempo mais detalhada é possível sintetizar OCGRAs com 400Mhz dobrando sua capacidade de processamento.

Assim como nos trabalhos [64, 65], o próprio CPU do sistema heterogêneo foi utilizado para realizar as comparações de tempo execução, diferente de outros trabalhos com FPGA que fazem comparação com processadores de baixo desempenho como microBlazer e NIOS, ou versões ARM de 600 Mhz embarcadas em FPGAs. Desse modo a comparação realizada é entre um processador com 14 núcleos da *Intel*, um *Xeon* E52680 2.4Ghz e 24 MB de cache L3 e o OCGRA proposto sintetizado sobre um FPGA com a frequência de *clock* de 200Mhz. A Figura 4.10 apresenta um gráfico com os resultados de execução em ambas as arquiteturas. Para realizar medição tempo os algoritmos foram executados na CPU e no OCGRA com a mesma quantidade de dados e foram aferidos apenas o tempo gasto de leitura dos dados na memória principal, processamento e escrita dos resultados na memória principal em ambos os casos. Vale ressaltar que o OCGRA está virtualizado sobre o FPGA onde todos os dados são lidos da memória da CPU através da interface fortemente acoplada com coerência de cache CPU/FPGA da plataforma HARP da Intel. O

tempo de execução no acelerador inclui a transferência de dados que é limitada a 25 GB/s pois utiliza dois barramentos PCI e um barramento QPI. No OCGRA, os algoritmos foram executados com 8 *threads* com diferentes dados de entrada para cada *thread*, na CPU foram executados com a mesma quantidade de *threads* por meio de diretivas OpenMP[8] e compilados com otimização de terceiro nível. Os resultados mostram que a arquietura proposta foi em média 1,9x mais rápida que a execução com o mesmo número de threads em um processador de alto desempenho, mesmo com a diferença de uma ordem de grandeza no ciclo de relógio e as sobrecargas do *overlay* em FPGA. Para o algoritmo *Sobel* uma imagem com 1333x1333 *pixels* foi passada para cada *thread* tanto no CPU quanto no OCGRA, para os demais algoritmos foram gerados randomicamente vetores com $2^{20}$ elementos de dados para cada entrada. Todos resultados de saída das execuções do OCGRA foram comparados com as saídas da CPU para confirmar que a tradução dos algoritmos para DFG foram feitas corretamente.

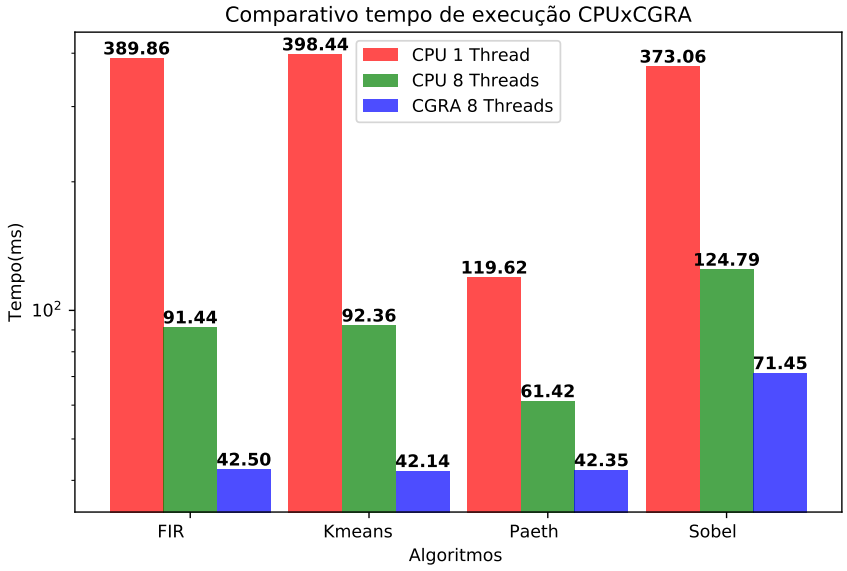



**Figura 4.10.** Comparação do tempo de execução entre processador *Xeon* E52680 2.4Ghz com um e com oito threads e a execução no acelerador proposto com 128 PEs 200Mhz e oito threads.

Um parâmetro importante a ser destacado é o tempo gasto para realizar a configuração do *bitstream* no CGRA. A configuração de cada *thread* no CGRA se dá em tempo de execução, onde os dados de configuração são enviados utilizando o mesmo barramento de dados. Para realizar essas medidas, a metodologia adotada foi realizar uma execução de um *thread* com 1 dado de entrada e saída. Como os

algoritmos possuem tamanhos diferentes de configurações os tempos aferidos dos algoritmos estão dispostos na Tabela 4.4.

**Tabela 4.4.** Tempo de configuração dos algoritmos

| **Algoritmo** | **Tamanho do *Bitstream(bytes)*** | **Tempo de configuração(*us*)** |
|---|---|---|
| FIR | 6528 | 603,808 |
| K-means | 6656 | 617,648 |
| Sobel | 4096 | 611,716 |
| Paeth | 6144 | 617,68 |

## 4.6 Conclusão

Um novo modelo de arquitetura de OCGRA com o modo de execução MIMT (*Multiple Instruction Multiple Threads*) foi proposto e validado, juntamente com um conjunto completo de ferramentas que possibilita o fluxo completo de projeto de forma transparente, deste a especificação do grafo de fluxo de dados até sua execução virtualizada em uma plataforma heterogênea de alto desempenho com CPU-FPGA na nuvem. O novo modelo permite executar múltiplas *threads* de diferentes aplicações simultaneamente fazendo uso do paralelismo temporal e espacial, complementando outros aceleradores como as GPUs. O modelo MIMT tem troca de contexto imediata em um ciclo de relógio. Apesar de ser a primeira implementação da ferramenta completa, os resultados preliminares já demonstraram um bom desempenho quando comparado a um processador de alto desempenho da *Intel*, sendo em média 1.9x mais rápido considerando a mesma quantidade de *threads*, mesmo operando de forma virtualizada sobre um FPGA com a restrição de frequência de relógio de 200MHz e sobrecarga geradas pelas interfaces de memória e virtualização sobre uma arquitetura reconfigurável de grão fino.

Outra contribuição importante é disponibilizar um sistema modular para comunidade acadêmica explorar os aceleradores em *hardware* reconfigurável. Possibilitar o uso e desenvolvimento de novos operadores para os grafos de fluxo de dados, arquiteturas homogêneas ou heterogêneas com validação em tempo de execução e compilação da ordem de milissegundos, em contraste ao uso dos FPGAs que chegam a demandar horas durante a geração de um *bitstream*

É destacado também o conjunto de ferramentas criado para geração de OCGRAs personalizados que juntamente com o FDAM facilita a implantação e o uso de aceleradores com OCGRA na plataforma *Intel* HARP, podendo ser estendidos para outras plataformas de FPGA.

Existem muitas possibilidades de trabalhos futuros tanto para o novo modelo de arquitetura quanto para as ferramentas desenvolvidas de forma modular. Podemos enumerar algumas sugestões como: (a) explorar vários algoritmos sobre diversos conjuntos de operadores para o grafo de fluxo de dados e mapeá-los facilmente em OCGRAs que serão sintetizados uma única vez; (b) Desenvolvimento de compiladores JIT para extração dos grafos de trechos de código em alto nível; (c) Desenvolvimento de escalonadores para tempo de execução para fazer o gerenciamento de múltiplos *threads* para manter os aceleradores CGRAs sempre ocupados; (d) Desenvolvimento de mapeamento de tarefas com múltiplos *threads* acoplados eliminando leituras e escritas em memória com a transferência direta de dados entre os *threads*; (e) Desenvolvimento de novos CGRA com clusterização de OCGRAs de rede global; (f) Acoplamento da nova arquitetura CGRA em plataformas com processamento em memória (PIM); (g) Prototipação de CGRA para serem sintetizados em silício; (h) Redução do tempo de mapeamento do grafo de fluxo de dados para CGRA para ordem de microssegundos; (i) Explorar arquitetura com computação aproximada e largura variável da palavra.

# Capítulo 5

# Conclusão

Nesta dissertação foram apresentados três trabalhos na área da computação heterogênea reconfigurável. Esse tipo de computação envolve o uso de CPU e uma arquitetura reconfigurável. Os principais sistemas de CHR usam os FPGAs como aceleradores dessas plataformas. Embora os FPGAs possuem várias vantagens devido ao fato de ser possível expressar algoritmos na forma de circuitos lógicos, os FPGAs apresentam dificuldades que impedem o ciclo de desenvolvimento usado atualmente para outras arquiteturas como as GPUs. Os três trabalhos apresentados nesta dissertação trazem soluções para tornar o uso de sistemas heterogêneos mais acessíveis para usuários não familiarizados com o desenvolvimento para FPGAs.

No capítulo 2 mostramos um arcabouço capaz de gerar automaticamente aceleradores de domínio específico para plataforma HARP para um algoritmo de busca de estados de estabilização em redes reguladoras de genes. O arcabouço recebe como entrada a descrição da rede de genes como um simples grafo com equações booleanas e é capaz gerar várias unidades de processamento para realizar as computações fazendo o uso de paralelismo espacial. Os resultados mostraram que o uso dos aceleradores em FPGAs foram duas ordens de magnitude mais rápidos que a mesma aplicação executada em CPU otimizadas usando OpenMP com 20 *Threads* e uma ordem de magnitude mais rápidos que a mesma aplicação executada em uma GPU da NVidia Tesla K40. Embora o esse tipo abordagem tenha demonstrado o potencial do FPGAs para aceleração de aplicações em sistemas heterogêneos, para o usuário realizar a mesma computação em uma rede diferente, todo o processo de geração de código e compilação para o FPGA deve ser realizado novamente. Devido ao fato do FPGA ser uma arquitetura de grão fino, ou seja, toda sua configuração ser realizada a nível de *bit* os tempos de compilações de suas aplicações chegam a durar horas. Apesar disso, como o espaço de estados de uma rede de genes é expo-

nencial e para explorar apenas um sub-espaço, o acelerador em FPGA pode realizar a tarefa em algumas horas ou dias, uma busca que demandaria semanas ou meses na CPU. Neste caso, o tempo de horas para compilação é irrelevante. A solução é escalável e pode ser mapeada em múltiplos FPGAs.

No capítulo 3 foi apresentado FDAM um arcabouço para facilitar na aceleração de várias aplicações no mesmo acelerador, gerando de forma automática toda a lógica de controle na transferência de dados entre a aplicação executada no CPU e as aplicações no FPGA. O FDAM também conta com uma API que permite todo o gerenciamento dos aceleradores e métodos para realizar a transferência de dados de forma simplificada. Os resultados mostraram que foi possível executar até 16 aceleradores obtendo desempenho de até 131 Gops/s.

Nestes dois trabalhos os FPGAs demonstraram que podem ser usados para processamento de grandes quantidades de dados com desempenho superior aos CPU utilizados em servidores e com uma boa eficiência energética. Porém, a alta complexidade no desenvolvimento de aplicações motiva a criação de uma nova arquitetura reconfigurável como alternativa ao FPGA, apresentada no terceiro trabalho. Diferente dos FPGAs, essa nova arquitetura é reconfigurável a nível de palavra, sendo conhecida na literatura como arquitetura reconfigurável de grão grosso (CGRA). O CGRA tem como principal vantagem o tempo de compilação na ordem de milissegundos, o que permite um ciclo de desenvolvimento mais semelhante com os usados para GPUs. Foi mostrado que essa arquitetura mesmo sendo virtualizada sobre um FPGA é capaz de obter até 2x o desempenho de um CPU. Embora o CGRA seja uma arquitetura com potencial para o uso em sistemas heterogêneos reconfiguráveis, ainda faltam ferramentas, principalmente compiladores e definição do conjunto de operações dos PEs e implementações em sílicio. Recentemente, a Intel e a Microsoft demonstraram interesse em CGRA em sílicio. A Microsoft com o E2 EDGE processor [1] que foi apresentado no ISCA 2018. A Intel com o acelerador (Configurable Spatial Accelerator) CSA [2], além da evolução do HARP com o processador Xeon Gold 6138P integrado com um Altera Arria 10 GX 1150 FPGA [3], isso mostra que pesquisas para novas ferramentas e novas arquiteturas são relevantes no cenário atual.

[1] https://arxiv.org/pdf/1803.06617.pdf

[2] https://www.nextplatform.com/2018/08/30/intels-exascale-dataflow-engine-drops-x86-and-von-neuman/

[3] https://www.anandtech.com/show/12773/intel-shows-xeon-scalable-gold-6138p-with-integrated-fpga-shipping-to-vendors

# Referências Bibliográficas


[1] J. Lu, H. Zeng, Z. Liang, L. Chen, L. Zhang, H. Zhang, H. Liu, H. Jiang, B. Shen, M. Huang, *et al.*, "Network Modelling Reveals the Mechanism Underlying Colitis-Associated Colon Cancer and Identifies Novel Combinatorial Anti-Cancer Targets," *Scientific reports*, vol. 5, 2015.

[2] P. Gupta, "Accelerating Datacenter Workloads," in *26th International Conference on Field Programmable Logic and Applications*, 2016.

[3] 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," *SIGARCH Comput. Archit. News*, vol. 42, pp. 13–24, June 2014.

[4] Amazon, "Elastic Compute Cloud - Amazon EC2 - AWS." `http://aws.amazon.com/ec2/`, 2018.

[5] Xilinx, "Xilinx - adaptable. intelligent." `https://www.xilinx.com/`, 2019. Acessado em: 2019-02-01.

[6] 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)*, pp. 1–7, IEEE, 2017.

[7] Khronos, "The open standard for parallel programming of heterogeneous systems." `https://www.khronos.org/opencl/`, 2019. Acessado em: 2019-02-01.

[8] OpenMP, "The OpenMP API Specification for Parallel Programming." `https://www.openmp.org/`, 2019. Acessado em: 2019-02-01.

[9] NVidia, "About cuda." `https://developer.nvidia.com/about-cuda`, 2019. Acessado em: 2019-02-01.

[10] OpenACC, "What is openacc?." `https://www.openacc.org/`, 2019. Acessado em: 2019-02-01.

[11] UFV, Conselho Técnico de Pós Graduação da Universidade Federal de Viçosa, "Normas de redação de teses e dissertações." `http://www.dpi.ufv.br/arquivos/ppgcc/doc/PPG-2015-normascorrigidas.pdf`, 2019. Accessed: 2019-01-14.

[12] E. Davidson and M. Levin, "Gene regulatory networks," 2005.

[13] I. Sanchez-Osorio, F. Ramos, P. Mayorga, and E. Dantan, "Foundations for Modeling the Dynamics of Gene Regulatory Networks: a Multilevel-Perspective Review," *Journal of bioinformatics and computational biology*, vol. 12, no. 01, p. 1330003, 2014.

[14] T. Akutsu, S. Kuhara, O. Maruyama, and S. Miyano, "Identification of Gene Regulatory Networks by Strategic Gene Disruptions and Gene Overexpressions," in *SODA*, vol. 98, pp. 695–702, 1998.

[15] I. Irurzun-Arana, J. M. Pastor, I. F. Trocóniz, and J. D. Gómez-Mantilla, "Advanced Boolean Modeling of Biological Networks Applied to Systems Pharmacology," *Bioinformatics*, vol. 33, no. 7, pp. 1040–1048, 2017.

[16] C. Darabos, F. Di Cunto, M. Tomassini, J. H. Moore, P. Provero, and M. Giacobini, "Additive Functions in Boolean Models of Gene Regulatory Network Modules," *PloS one*, vol. 6, no. 11, p. e25110, 2011.

[17] N. Berntenis and M. Ebeling, "Detection of Attractors of Large Boolean Networks Via Exhaustive Enumeration of Appropriate Subspaces of The State Space," *BMC bioinformatics*, vol. 14, no. 1, p. 361, 2013.

[18] I. Tagkopoulos, C. Zukowski, G. Cavelier, and D. Anastassiou, "A Custom FPGA for The Simulation of Gene Regulatory Networks," in *Proceedings of ACM Great Lakes symposium on VLSI*, pp. 132–135, 2003.

[19] M. Zerarka, J. David, and E. Aboulhamid, "High Speed Emulation of Gene Regulatory Networks Using FPGAs," in *Circuits and Systems, MWSCAS. The 47th Midwest Symposium on*, vol. 1, IEEE, 2004.

[20] I. Pournara, C.-S. Bouganis, and G. A. Constantinides, "FPGA-Accelerated Bayesian Learning for Reconstruction of Gene Regulatory Networks," in *IEEE FPL*, 2005.

[21] R. Ferreira and J. C. G. Vendramini, "FPGA-Accelerated Attractor Computation of Scale Free Gene Regulatory Networks," in *International Conference on Field Programmable Logic and Applications*, 2010.

[22] M. Hucka, A. Finney, *et al.*, "The Systems Biology Markup Language (SBML): a Medium for Representation and Exchange of Biochemical Network Models," *Bioinformatics*, vol. 19, no. 4, pp. 524–531, 2003.

[23] M. E. Smoot, K. Ono, J. Ruscheinski, P.-L. Wang, and T. Ideker, "Cytoscape 2.8: New Features for Data Integration and Network Visualization," *Bioinformatics*, vol. 27, no. 3, pp. 431–432, 2010.

[24] A. Najafi, G. Bidkhori, J. H Bozorgmehr, I. Koch, and A. Masoudi-Nejad, "Genome Scale Modeling in Systems Biology: Algorithms and Resources," *Current genomics*, vol. 15, no. 2, pp. 130–159, 2014.

[25] S.-Q. Zhang, M. Hayashida, T. Akutsu, W.-K. Ching, and M. K. Ng, "Algorithms for Finding Small Attractors in Boolean Networks," *EURASIP Journal on Bioinformatics and Systems Biology*, 2007.

[26] D. J. Irons, "Improving the Efficiency of Attractor Cycle Identification in Boolean Networks," *Physica D: Nonlinear Phenomena*, vol. 217, no. 1, pp. 7–21, 2006.

[27] A. Bhattacharjya and S. Liang, "Median Attractor and Transients in Random Boolean Nets," *Physica D: Nonlinear Phenomena*, vol. 95, no. 1, pp. 29–34, 1996.

[28] Y.-k. Choi, J. Cong, Z. Fang, Y. Hao, G. Reinman, and P. Wei, "A quantitative analysis on microarchitectures of modern cpu-fpga platforms," in *Proceedings of the 53rd Annual Design Automation Conference*, p. 109, ACM, 2016.

[29] B. D. Conroy, T. A. Herek, T. D. Shew, M. Latner, J. J. Larson, L. Allen, P. H. Davis, T. Helikar, and C. E. Cutucache, "Design, Assessment, and In Vivo Evaluation of a Computational Model Illustrating the Role of CAV1 in CD4+ T-Lymphocytes," *Frontiers in immunology*, vol. 5, 2014.

[30] A. Saadatpour, I. Albert, and R. Albert, "Attractor Analysis of Asynchronous Boolean Models of Signal Transduction Networks," *Journal of theoretical biology*, vol. 266, no. 4, pp. 641–656, 2010.

[31] N. Miskov-Zivanov, A. Bresticker, D. Krishnaswamy, S. Venkatakrishnan, D. Marculescu, and J. R. Faeder, "Emulation of Biological Networks in Reconfigurable Hardware," in *Proc. of ACM Conf. on Bioinformatics, Computational Biology and Biomedicine*, pp. 536–540, 2011.

[32] N. Miskov-Zivanov, A. Bresticker, D. Krishnaswamy, S. Venkatakrishnan, P. Kashinkunti, D. Marculescu, and J. R. Faeder, "Regulatory Network Analysis Acceleration With Reconfigurable Hardware," in *Engineering in Medicine and Biology Society, EMBC*, IEEE, 2011.

[33] A. Garg, A. Di Cara, I. Xenarios, L. Mendoza, and G. De Micheli, "Synchronous Versus Asynchronous Modeling of Gene Regulatory Networks," *Bioinformatics*, vol. 24, no. 17, pp. 1917–1925, 2008.

[34] A. Naldi, E. Remy, D. Thieffry, and C. Chaouiya, "Dynamically Consistent Reduction of Logical Regulatory Graphs," *Theoretical Computer Science*, vol. 412, no. 21, pp. 2207–2218, 2011.

[35] A. Veliz-Cuba, B. Aguilar, F. Hinkelmann, and R. Laubenbacher, "Steady State Analysis of Boolean Molecular Network Models Via Model Reduction and Computational Algebra," *BMC bioinformatics*, vol. 15, no. 1, p. 221, 2014.

[36] E. Dubrova and M. Teslenko, "A SAT-Based Algorithm for Finding Attractors in Synchronous Boolean Networks," *IEEE/ACM transactions on computational biology and bioinformatics*, vol. 8, no. 5, 2011.

[37] J. E. Galagan, K. Minch, M. Peterson, A. Lyubetskaya, E. Azizi, L. Sweet, A. Gomes, T. Rustad, G. Dolganov, I. Glotova, *et al.*, "The Mycobacterium Tuberculosis Regulatory Network and Hypoxia," *Nature*, vol. 499, no. 7457, p. 178, 2013.

[38] H.-C. Trinh, D.-H. Le, and Y.-K. Kwon, "PANET: a GPU-Based Tool for Fast Parallel Analysis of Robustness Dynamics and Feed-Forward/Feedback Loop Structures in Large-Scale Biological Networks," *PloS one*, vol. 9, no. 7, p. e103010, 2014.

[39] D. J. Moss, S. Krishnan, E. Nurvitadhi, P. Ratuszniak, C. Johnson, J. Sim, A. Mishra, D. Marr, S. Subhaschandra, and P. H. 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*, pp. 107–116, ACM, 2018.

[40] S. Cho, M. Patel, H. Chen, M. Ferdman, and P. Milder, "A full-system vm-hdl co-simulation framework for servers with pcie-connected fpgas," in *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pp. 87–96, ACM, 2018.

[41] T. Becker, O. Mencer, and G. Gaydadjiev, "Spatial programming with openspl," in *FPGAs for Software Programmers*, pp. 81–95, Springer, 2016.

[42] M. Ozdal, G.-J. Nam, and D. Marr, "Guest editors' introduction: Hardware accelerators for data centers," *IEEE Design & Test*, vol. 35, no. 1, pp. 5–6, 2018.

[43] 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)*, pp. 32–39, IEEE, 2016.

[44] L. Di Tucci, M. Rabozzi, L. Stornaiuolo, and M. D. Santambrogio, "The role of cad frameworks in heterogeneous fpga-based cloud systems," in *2017 IEEE International Conference on Computer Design (ICCD)*, pp. 423–426, IEEE, 2017.

[45] A. Iordache, G. Pierre, P. Sanders, J. G. de F Coutinho, and M. Stillwell, "High performance in the cloud with fpga groups," in *Proceedings of the 9th International Conference on Utility and Cloud Computing*, pp. 1–10, ACM, 2016.

[46] Intel, "AOCL Programming Guide." `https://software.intel.com/en-us/intel-opencl`, 2019. Acessado em: 2019-02-01.

[47] E. Luebbers, "OPAE." `https://01.org/OPAE`, 2017. Acessado em: 2017-05-01.

[48] 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)*, pp. 148–155, IEEE, 2016.

[49] T. S. Abdelrahman, "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)*, 2016.

[50] Z. István, 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)*, 2016.

[51] F. A. M. Alves, P. Jamieson, L. Bragança, R. S. Ferreira, and J. A. M. Nacif, "Designing a Collision Detection Accelerator on a Heterogeneous CPU-FPGA Platform," in *ReConFigurable Computing and FPGAs (ReConFig), 2017 International Conference on*, pp. 1–6, IEEE, 2017.

[52] L. B. da Silva, D. Almeida, J. A. M. Nacif, I. Sánchez-Osorio, C. A. Hernández-Martínez, and R. Ferreira, "Exploring the dynamics of large-scale gene regulatory networks using hardware acceleration on a heterogeneous cpu-fpga platform," in *2017 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*, pp. 1–7, IEEE, 2017.

[53] G. Weisz, J. Melber, Y. Wang, K. Fleming, E. Nurvitadhi, and J. 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*, pp. 264–273, ACM, 2016.

[54] H. R. Zohouri, A. Podobas, and S. Matsuoka, "Combined spatial and temporal blocking for high-performance stencil computation on fpgas using opencl," in *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pp. 153–162, ACM, 2018.

[55] H.-C. Ng, S. Liu, and W. Luk, "Adam: Automated design analysis and merging for speeding up fpga development," in *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pp. 189–198, ACM, 2018.

[56] S. Takamaeda-Yamazaki, "Pyverilog: A Python-Based Hardware Design Processing Toolkit for Verilog HDL," in *Applied Reconfigurable Computing*, Apr 2015.

[57] C. Zhang and V. Prasanna, "Frequency Domain Acceleration of Convolutional Neural Networks on CPU-FPGA Shared Memory System," in *International Symposium on Field-Programmable Gate Arrays*, 2017.

[58] G. Stitt, A. Gupta, M. N. Emas, D. Wilson, and A. 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*, pp. 173–182, ACM, 2018.

[59] TOP500, "Top500 list." `https://www.top500.org/lists/`, 2019. Acessado em: 2019-02-01.

[60] ACM, "John hennessy and david patterson deliver turing lecture at isca 2018." `https://www.acm.org/hennessy-patterson-turing-lecture`, 2019. Acessado em: 2019-02-01.

[61] A. K. Jain, D. L. Maskell, and S. A. Fahmy, "Resource-aware just-in-time opencl compiler for coarse-grained fpga overlays," *CoRR*, vol. abs/1705.02730, 2017.

[62] A. Kulkarni, D. Stroobandt, A. Werner, F. Fricke, and M. Hübner, "Pixie: A heterogeneous virtual coarse-grained reconfigurable array for high performance image processing applications," *CoRR*, vol. abs/1705.01738, 2017.

[63] A. K. Jain, X. Li, P. Singhai, D. L. Maskell, and S. A. Fahmy, "Deco: A dsp block based fpga accelerator overlay with low overhead interconnect," in *2016 IEEE 24th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pp. 1–8, May 2016.

[64] C. Liu, H. C. Ng, and H. K. H. So, "Quickdough: A rapid fpga loop accelerator design framework using soft cgra overlay," in *2015 International Conference on Field Programmable Technology (FPT)*, pp. 56–63, Dec 2015.

[65] A. K. Jain, D. L. Maskell, and S. A. Fahmy, "Throughput oriented fpga overlays using dsp blocks," in *2016 Design, Automation Test in Europe Conference Exhibition (DATE)*, pp. 1628–1633, March 2016.

[66] Xilinx, "Vivado high-level synthesis." `https://www.xilinx.com/products/design-tools/vivado/integration/esl-design.html`, 2019. Acessado em: 2019-02-01.

[67] B. Mei, A. Lambrechts, J.-Y. Mignolet, D. Verkest, and R. Lauwereins, "Architecture exploration for a reconfigurable architecture template," *IEEE Design & Test of Computers*, vol. 22, no. 2, pp. 90–101, 2005.

[68] R. Ferreira, A. Garcia, T. Teixeira, and J. M. P. Cardoso, "A polynomial placement algorithm for data driven coarse-grained reconfigurable architectures," in *IEEE Computer Society Annual Symposium on VLSI (ISVLSI '07)*, 2007.

[69] A. C. S. Beck, M. B. Rutzig, G. Gaydadjiev, and L. Carro, "Transparent reconfigurable acceleration for heterogeneous embedded applications," in *Design, Automation and Test in Europe, 2008. DATE'08*, pp. 1208–1213, IEEE, 2008.

[70] R. Ferreira, M. Laure, A. C. Beck, T. Lo, M. Rutzig, and L. Carro, "A low cost and adaptable routing network for reconfigurable systems," in *Reconfigurable Architecture WorkShop (RAW) in IEEE International Symposium on Parallel Distributed Processing*, 2009.

[71] T. B. Ló, A. C. S. Beck, M. B. Rutzig, and L. Carro, "A low-energy approach for context memory in reconfigurable systems," in *IEEE International Symposium on Parallel Distributed Processing, Workshops and Phd Forum (IPDPSW)*, 2010.

[72] N. M. C. Paulino, J. C. Ferreira, and J. M. P. Cardoso, "Dynamic partial reconfiguration of customized single-row accelerators," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 27, no. 1, 2019.

[73] R. Ferreira, J. G. Vendramini, L. Mucida, M. M. Pereira, and L. Carro, "An fpga-based heterogeneous coarse-grained dynamically reconfigurable architecture," in *Proceedings of the 14th international conference on Compilers, architectures and synthesis for embedded systems*, pp. 195–204, ACM, 2011.

[74] R. Ferreira, V. Duarte, W. Meireles, M. Pereira, L. Carro, and S. Wong, "A just-in-time modulo scheduling for virtual coarse-grained reconfigurable architectures," in *Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS XIII), 2013 International Conference on*, pp. 188–195, IEEE, 2013.

[75] S. Lee, K. Kim, G. Koo, H. Jeon, W. W. Ro, and M. Annavaram, "Warped-compression: enabling power efficient gpus through register compression," in *ACM SIGARCH Computer Architecture News*, vol. 43, pp. 502–514, ACM, 2015.

[76] A. B. M. Gomes, "Ômega assimétrica: Uma nova rede de interconexão para depuração pós-silício," Master's thesis, Universidade Federal de Viçosa, 2015.

[77] J. C. G. Vendramini, "Rede Ômega virtual em fpga com reconfiguração em tempo de execução. estudo de caso: Cálculo de atratores em redes reguladoras de genes," Master's thesis, Universidade Federal de Viçosa, 2012.

[78] R. Ferreira, J. Vendramini, and M. Nacif, "Dynamic reconfigurable multicast interconnections by using radix-4 multistage networks in fpga," in *Industrial Informatics (INDIN), 2011 9th IEEE International Conference on*, pp. 810–815, IEEE, 2011.

[79] L. Bragança, F. Alves, J. C. Penha, G. Coimbra, R. Ferreira, and J. A. M. Nacif, "Simplifying hw/sw integration to deploy multiple accelerators for cpu-fpga heterogeneous platforms," in *Proceedings of the 18th International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation*, SAMOS '18, (New York, NY, USA), pp. 97–104, ACM, 2018.

[80] J. Penha, L. Bragança, D. Almeida, J. Nacif, and R. Ferreira, "ADD-Uma Ferramenta de Projeto de Aceleradores com DataFlow para Alto Desempenho," *Simpósio em Sistemas Computacionais de Alto Desempenho (WSCAD)*, 2017.

[81] N. Hendrich, "A java-based framework for simulation and teaching: Hades—the hamburg design system," in *Microelectronics Education*, pp. 285–288, Springer, 2000.

[82] Terasic, "Altera DE2-115 Development and Education Board." `https://www.terasic.com.tw/cgi-bin/page/archive.pl?Language=English&No=502`, 2019. Acessado em: 2019-02-01.

[83] G. Coimbra, L. Brangança, and J. A. M. Nacif, "Investigating the use of modern heterogeneous cpu-fpga architectures on the 3-sat problem," *SFORUM*, 2018.

[84] J. C. Penha, L. Bragança, J. M. Silva, K. K. Coelho, H. P. Baranda, J. A. M. Nacif, and R. S. Ferreira, "ADD: Accelerator Design and Deploy-A Tool for FPGA High-Performance Dataflow Computing," *Concurrency and Computation: Practice and Experience*, p. e5096, 2018.

[85] P. Caldeira, J. C. Penha, L. Bragança, R. Ferreira, J. A. M. Nacif, R. Ferreira, and F. M. Q. Pereira, "From java to fpga: an experience with the intel harp system," *SBAC-PAD*, 2018.

[86] G. Andrade, W. de Carvalho, R. Utsch, P. Caldeira, A. Alburquerque, F. Ferracioli, L. Rocha, M. Frank, D. Guedes, and R. Ferreira, "ParallelME: A Parallel Mobile Engine to Explore Heterogeneity in Mobile Computing Architectures," in *European Conference on Parallel Processing*, pp. 447–459, Springer, 2016.

[87] J. C. Penha, L. Bragança, K. Coelho, M. Canesche, J. Silva, G. Comarela, J. A. M. Nacif, and R. Ferreira, "Gerador parametrizável de aceleradores para k-means em fpga e gpu," *WSCAD*, 2018.

# Apêndice A

# Grafos dos algoritmos utilizados no Capítulo 4

Neste apêndice são apresentados os grafos de fluxo de dados dos algoritmos executados no OCGRA proposto no capítulo 4. Também é realizado um detalhamento mais técnico do funcionamento desses algoritmos na forma de fluxo de dados no CGRA.

O grafo de fluxo de dados da Figura A.1 é o algoritmo de *K-means* com 8 *clusters* e 5 dimensões. Apenas uma parte do algoritmo foi traduzida para o grafo e, desse modo a segunda parte do algoritmo é executada na CPU. Cada dimensão corresponde a uma entrada, e apenas um saída é utilizada para o processamento. Cada ponto de entrada é processado e uma saída correspondente a esse ponto é dada de acordo com o índice do *cluster* mais próximo desse ponto.

O algoritmo para detecção de bordas em imagens Sobel representado na Figura A.2 foi mapeado aproveitando o paralelismo espacial intrínseco dos grafos de fluxo de dados. Assim, as convoluções para detectar as bordas verticais e horizontais são realizadas em paralelo. A entrada é composta por 8 vetores, sendo cada posição um elemento da janela de uma matriz 3x3 onde realizada a convolução da imagem. Para esse algoritmo apenas o processamento da imagem resultante é realizado no CPU.

O algoritmo Paeth é menor entre os demais, desse modo, foi possível processar duas cópias no mesmo grafo de fluxo de dados. Nenhuma otimização ou melhoria foi realizada nesse algoritmo, sendo todo o processamento realizado no CGRA.

Por fim na Figura A.4 é mostrado o grafo de fluxo de dados do algoritmo FIR. Este algoritmo realiza o melhor aproveitamento dos dados de entrada, realizando 125 operações para cada dado de entrada.

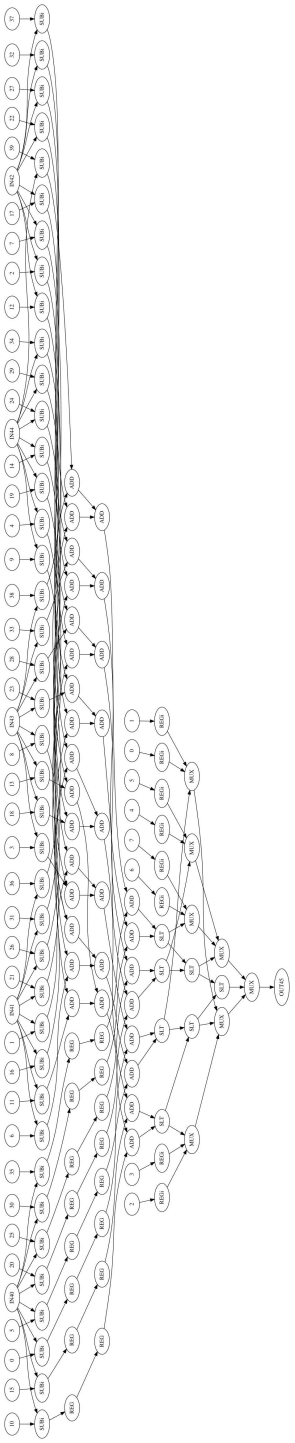


**Figura A.1.** Grafo de fluxo de dados algoritmo *K-means*.

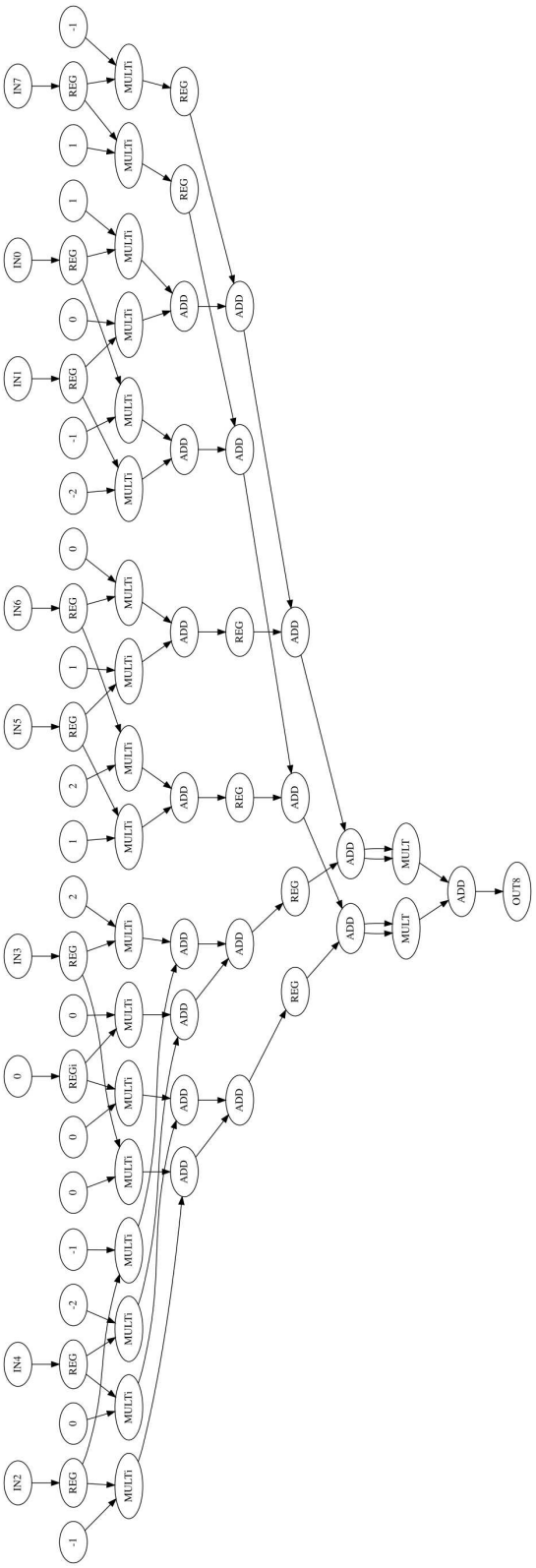



**Figura A.2.** Grafo de fluxo de dados algoritmo Sobel.

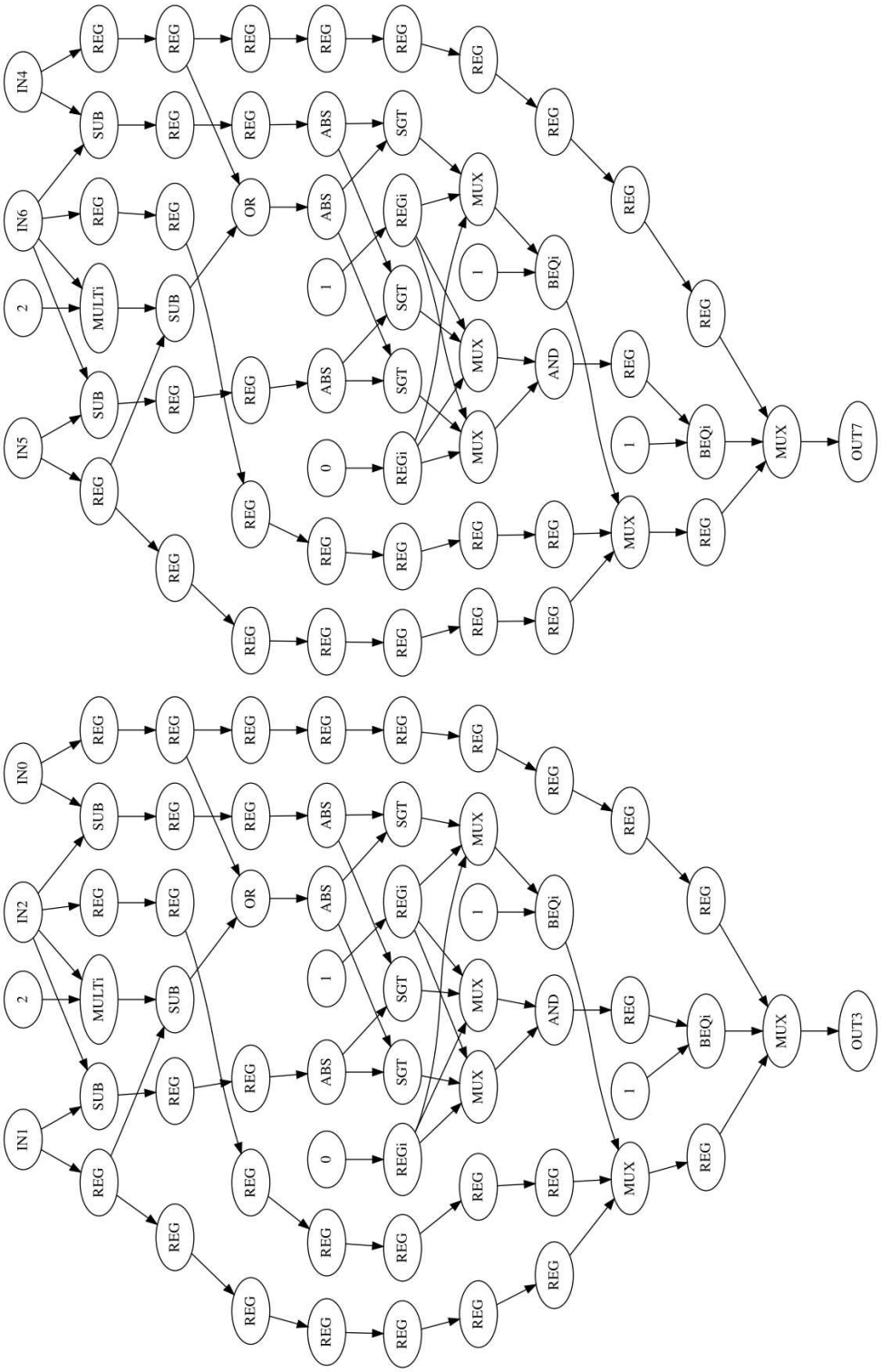



**Figura A.3.** Grafo de fluxo de dados algoritmo Paeth.

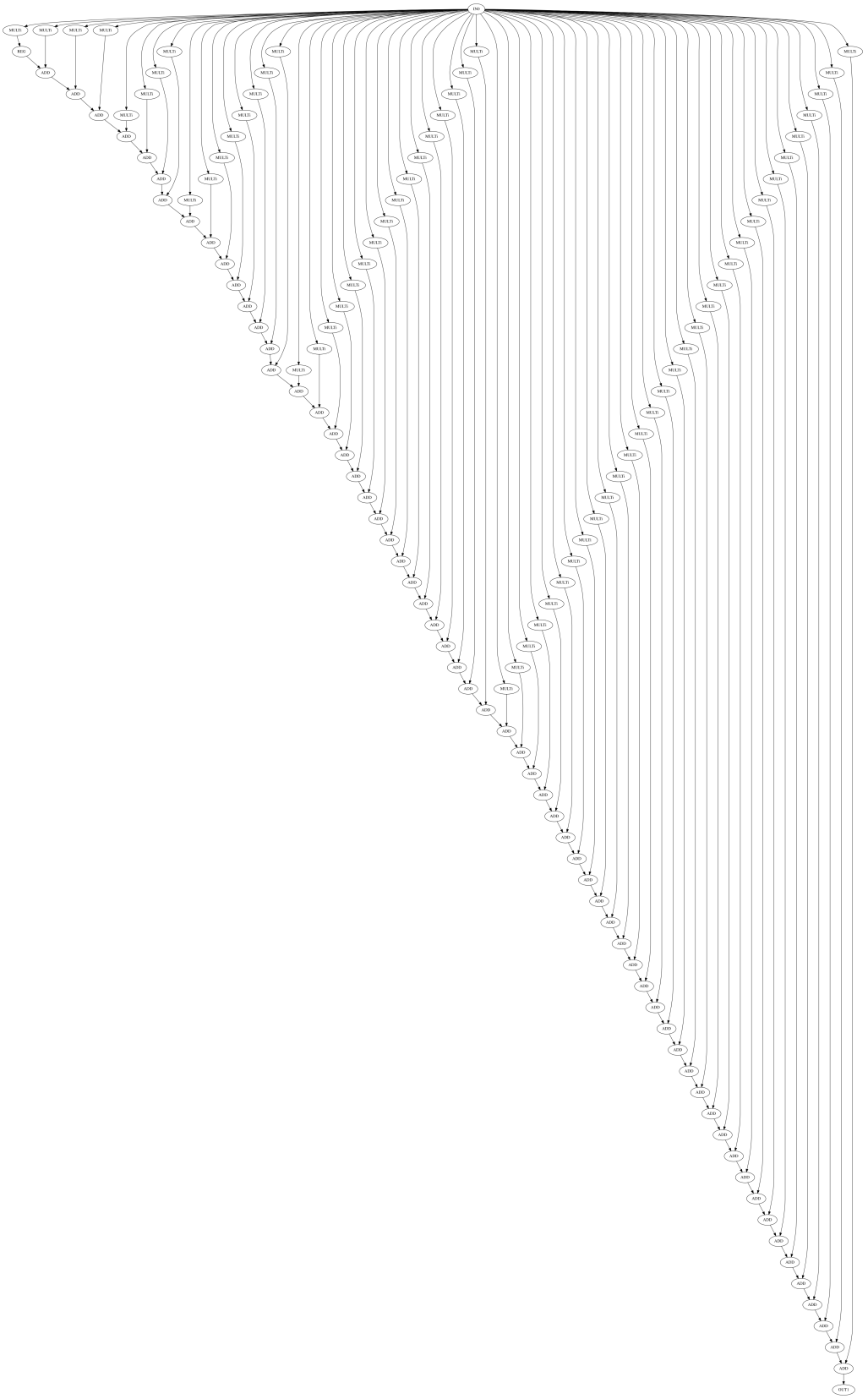



**Figura A.4.** Grafo de fluxo de dados algoritmo FIR.

# Apêndice B

# Outros trabalhos desenvolvidos

Além dos artigos como primeiro autor, outros trabalhos foram desenvolvidos durante o mestrado em cooperação com outros pesquisadores. Esses trabalhos possuem temáticas que seguem a mesma linha de pesquisa dos artigos apresentados nesta dissertação.

Em [51], uma solução em FPGA foi desenvolvida para acelerar um algoritmo de detecção de colisão de objetos virtuais. Esse tipo algoritmo normalmente é usado em sistemas de tempo real, como simulações e em jogos digitais. Para acelerar a detecção de colisão, apenas uma parte do algoritmo foi implementado no FPGA. Para isso foi utilizado a plataforma de computação heterogênea da Intel HARP [2]. Desse modo uma parte do algoritmo foi executada na CPU do sistema e outra parte foi acelerada pelo FPGA. Mesmo acelerando apenas uma pequena parte do algoritmo foi possível obter um desempenho de 14.81% em comparação com o algoritmo executado apenas em CPU.

Em [80] foi desenvolvido ADD uma ferramenta para o implementação e simulação de grafos a fluxo de dados (GFD) para FPGAs. O ADD permite a criação/edição de DFGs de forma visual ou descritiva. A principal contribuição do ADD é simplificar a simulação, o mapeamento e a execução de DFG em ambientes heterogêneos com processadores e FPGAs. A ferramenta é integrada no simulador de circuitos HADES [81] onde uma biblioteca de operadores para DFG foi adicionada. O DFG construído com o ADD pode ser simulado para depuração e correções e posteriormente exportado para linguagem Verilog. É possível exportar os GFDs para serem executados na plataforma HARP [2], ou para execução em placas de *Kits* didáticos como a DE2-115 [82].

Em [83] foi estudado um algoritmo de força bruta em FPGA para resolver problema de satisfatibilidade booleana conhecido na literatura com SAT. Para isso,

foi utilizado o *framework* de OpenCL para FPGA da Intel [46], uma estrutura que fornece uma interface de programação fácil de usar em plataformas heterogêneas. Desse modo foi implementado um *kernel* para um acelerador capaz resolver problemas SAT de até 40 variáveis de uma instância satisfatória em 14 ms. Instâncias insatisfatórias com 20 variáveis foi resolvidos em 343 ms. Embora OpenCL tenha facilitado na implementação e na execução, por outro lado foi necessário que o programador tenha conhecimento em FPGAs para escrever códigos em OpenCL de forma que o compilador da Intel fosse capaz realizar uma boa tradução para *hardware*, outro ponto negativo é alto tempo de compilação das aplicação gastando mais 2 horas para realizar todo o processo.

Em [84] foi escrita uma versão em estendida do artigo [80], onde, foi adicionado 50% de novas contribuições e foi publicado no periódico *Concurrency and Computation Practice and Experience* (CCPE). Nesta versão foi adicionado como novos recursos componentes para realizar a execução de GFDs assíncronos. GFDs assíncronos possuem vantagens de não precisarem ser balanceados, sendo assim usando menos recursos de *hardware* que os GFDs síncronos. Outra melhoria foi adicionar a exportação de código para a execução na segunda versão da plataforma de computação heterogênea da Intel HARPv2. Nessa plataforma foi possível obter maior desempenho devido o fato dela fornecer uma maior taxa de transferência entre o FPGA e a memória principal do sistema. Também foi adicionado a opção de realizar comunicação com o acelerador gerado por meio de comunicação serial RS232 que permite o acelerador ser usado em uma variedade maior de dispositivos FPGA.

Em [85] foi desenvolvido um compilador para aplicações escritas em Java com uso da biblioteca *ParallelME* [86], capaz de gerar código para execução na plataforma HARP v2 [2]. Este compilador primeiramente cria uma formato intermediário do código Java no formato de GFDs, que são traduzido em código Verilog, criando toda interface para execução diretamente no FPGA. Apenas funções da biblioteca do *ParallelME* são executadas no acelerador em FPGA. Essa abordagem foi capaz de obter um desempenho de 20x em relação ao código executado na JVM.

Em [87] é apresentado um trabalho para geração de código de domínio específico para o algoritmo *K-means*. Este gerador é capaz de fornecer código para GPUs e FPGAs. O gerador é parametrizável de forma que o usuário pode personalizar a quantidade de centróides e dimensões do algoritmo. Para GPU o gerador fornece como saída código para a execução na linguagem CUDA [9], para o FPGA foi utilizado o gerador fornece os códigos na linguagem Verilog. Outra contribuição deste trabalho é simplificação do uso de FPGAs de alto desempenho para programadores, pois não requer nenhum conhecimento de *hardware* por parte do usuário para prover

um acelerador de alto desempenho no nível de *software*. O gerador também simplifica a programação para GPU. Embora a GPU demonstrou maior desempenho que os FPGAs em relação à eficiência energética, as execuções em FPGA foram 10 vezes mais eficiente energeticamente.