UNIVERSIDADE FEDERAL DE VIÇOSA

Geração Automática de Aceleradores de Domínio Específico em FPGA

Lucas Bragança da Silva
Doctor Scientiae

VIÇOSA - MINAS GERAIS
2024

LUCAS BRAGANÇA DA SILVA

Geração Automática de Aceleradores de Domínio Específico em FPGA

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

Orientador: Jose Augusto Miranda Nacif

Coorientador: Ricardo dos S. Ferreira

VIÇOSA - MINAS GERAIS
2024

Ficha catalográfica elaborada pela Biblioteca Central da Universidade Federal de Viçosa - Campus Viçosa

T
S586f
2024

Silva, Lucas Bragança da, 1989-
Geração automática de aceleradores de domínio específico em FPGA / Lucas Bragança da Silva. – Viçosa, MG, 2024.
1 tese eletrônica (123 f.): il. (algumas color.).

Orientador: José Augusto Miranda Nacif.
Tese (doutorado) - Universidade Federal de Viçosa, Departamento de Informática, 2024.
Referências bibliográficas: f. 112-123.
DOI: https://doi.org/10.47328/ufvbbt.2025.034
Modo de acesso: World Wide Web.

1. Programação (Computadores). 2. Arranjo de lógica programável em campo. 3. Hardware. 4. Processamento eletrônico de dados. I. Nacif, José Augusto Miranda, 1978-. II. Universidade Federal de Viçosa. Departamento de Informática. Programa de Pós-Graduação em Ciência da Computação. III. Título.

CDD 22. ed. 005.1

Bibliotecário(a) responsável: Euzébio Luiz Pinto CRB-6/3317

LUCAS BRAGANÇA DA SILVA

Geração Automática de Aceleradores de Domínio Específico em FPGA

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

APROVADA: 3 de maio de 2024.

Assentimento:

_____

Lucas Bragança da Silva
Autor

_____

Jose Augusto Miranda Nacif
Orientador

AGRADECIMENTOS

À Vivian, minha esposa e melhor amiga e que nos momentos de cansaço me ajudou a seguir em frente. Seu amor e sua paciência e sua foram a base sobre o qual pude me apoiar. Sem você, este caminho teria sido mais difícil; com você, ele se tornou possível.

Ao meu filho, Daniel, cuja existência ilumina minha vida e me lembra diariamente o verdadeiro significado da dedicação e da perseverança. Seu sorriso e sua alegria renovam minha energia e me inspiram a ser um exemplo de determinação e esforço.

Aos meus pais, Vera e Abinael, que desde sempre me ensinaram o valor do conhecimento e da dedicação.

Ao meu orientador, José Augusto Miranda Nacif, por sua orientação que foi essencial para que este trabalho se concretizasse.

Ao meu coorientador, Ricardo dos Santos Ferreira, por sua dedicação e valiosas contribuições que foi fundamental para a construção deste trabalho.

Ao meu irmão, João Victor, e aos meus grandes amigos Jerônimo, Michael, Racyus e Dilson, que, de diversas formas, estiveram ao meu lado, oferecendo palavras de incentivo, compartilhando momentos de descontração e lembrando-me de que nenhuma caminhada precisa ser solitária.

Cada um de vocês fez parte desta jornada, e esta conquista é também um reflexo do apoio e inspiração que recebi. Meu sincero e eterno agradecimento!

O presente trabalho foi realizado com apoio da Coordenação de Aperfeiçoamento de Pessoal de Nível Superior – Brasil (CAPES) – Código de Financiamento 001.

"Sonho que se sonha só é só um sonho que se sonha só, mas sonho que se sonha junto é realidade". (Raul Seixas)

RESUMO

SILVA, Lucas Bragança da, D.Sc., Universidade Federal de Viçosa, maio de 2024. **Geração Automática de Aceleradores de Domínio Específico em FPGA**. Orientador: Jose Augusto Miranda Nacif. Coorientador: Ricardo dos Santos Ferreira.

Devido à sua eficiência energética e alta flexibilidade, os FPGAs (*Field Programmable Gate Arrays*), ou Arranjos de Portas Programáveis em Campo, têm desempenhado um papel relevante na computação, atuando como aceleradores de *hardware* especializados. No entanto, a programação e implantação de aceleradores em plataformas com FPGA ainda representam um desafio. Esta tese apresenta uma metodologia que visa automatizar o projeto de aceleradores em FPGAs por meio de um conjunto de ferramentas para geração de arquiteturas com um fluxo de síntese programável e flexível. A metodologia proposta foi desenvolvida em três etapas. Na primeira etapa, o objetivo foi entender os principais desafios relacionados ao projeto de aceleradores em FPGAs. Para isso, foram desenvolvidas duas abordagens que tratam de geradores de aceleradores para duas aplicações específicas. Na primeira abordagem, apresentamos um gerador de aceleradores em FPGA para o algoritmo *K-means*. O f*ramework* proposto é altamente parametrizável e gera código HDL (*Hardware Description Language*), ou Linguagem de Descrição de *Hardware*, para o projeto completo, pronto para ser implantado em uma plataforma de FPGA na nuvem. Como resultado, o acelerador proposto apresentou ganhos de tempo de execução de 1,98 vezes quando comparado com outro acelerador em uma plataforma com FPGA altamente acoplada. Na segunda abordagem, propomos um gerador de aceleradores para simulações de GRNs (*Genes Regulator Networks*), ou Redes Reguladoras de Genes, usando FPGAs na nuvem. Este gerador cria automaticamente todo o conjunto de *hardware*/*software* a partir de uma descrição em alto nível de um modelo de GRN. Utilizamos seis modelos GRN propostos na literatura para comparar o desempenho e o custo com implementações em CPU (*Central Processing Unit*), ou Unidade Central de Processamento, GPU (*Graphics Processing Unit*), ou Unidade de Processamento Gráfico, e FPGA. Como resultado, o acelerador em FPGA é pelo menos 12 vezes mais rápido do que o melhor acelerador em GPU avaliado, e oferece o melhor desempenho por dólar em serviços em nuvem, pelo menos 5 vezes melhor do que o melhor acelerador em GPU. Na segunda etapa desta tese, apresentamos um *framework* chamado *REconfigurable Accelerator DeploY* (READY), o primeiro *framework* a oferecer suporte ao mapeamento de aplicações em grafo de fluxo de

dados em plataformas CPU-FPGA de alto desempenho em tempo de execução. READY introduz um mapeamento eficiente em uma arquitetura de CGRA (*Coarse Grained Reconfigurable Architecture*), ou Arquitetura Reconfigurável de Granularidade Grossa, que oculta a latência da rede de interconexão global executando múltiplas tarefas. Além disso, demonstramos como esse sistema contribui para resolver alguns dos desafios relacionados à adoção de aceleradores de domínio específico em FPGAs. Os resultados dessa abordagem mostraram que é possível mapear e executar aplicações de grafo de fluxo de dados de forma eficiente, e o desempenho do sistema resulta em uma aceleração de até 2 vezes em comparação com uma CPU moderna. E, por fim, a terceira e última etapa desta tese apresenta o *High Performance Coarse Grained Reconfigurable Accelerator* (HPCGRA), uma ferramenta projetada para gerar aceleradores de domínio específico com interconexões multidimensionais de forma flexível. O CGRA é gerado utilizando os principais blocos de construção, como unidades funcionais, padrões de interconexão, roteamento e recursos de filas elásticas, palavras de configuração e memórias. O HPCGRA otimiza e simplifica o processo de criação de arquiteturas por meio de uma descrição portátil, gerando um código Verilog RTL genérico e eficiente. Os resultados da síntese mostram que, mesmo aumentando o número de PEs (*Processing Elements*), ou Elementos de Processamento, das arquiteturas geradas pelo HPCGRA, a frequência de *clock* e a relação do uso de recursos por PE permanecem estáveis. Foi possível sintetizar arquiteturas de CGRAs com 3036 PEs, com desempenho teórico de 1,2 TOPS.

Palavras-chave: fpga; cgra; dsa; rdsa; hpcgra

ABSTRACT

SILVA, Lucas Bragança da, D.Sc., Universidade Federal de Viçosa, May, 2024. **Automatic Generation of Domain-Specific Accelerators on FPGA**. Adviser: Jose Augusto Miranda Nacif. Co-adviser: Ricardo dos Santos Ferreira.

Due to their energy efficiency and high flexibility, Field Programmable Gate Arrays (FPGAs) have played a significant role in computing, serving as specialized hardware accelerators. However, programming and deploying accelerators on FPGA platforms still pose a challenge. This thesis presents a methodology that aims to automate the design of accelerators in FPGAs through a set of tools for generating architectures with a programmable and flexible synthesis flow. The proposed methodology was developed in three stages. In the first stage, the aim was to understand the main challenges related to accelerator design on FPGAs. For this, two approaches were developed that address accelerator generators for two specific applications. In the first approach, we present an FPGA accelerator generator for the K-means algorithm. The proposed framework is highly parameterizable and generates HDL (Hardware Description Language) code for the complete project, ready to be deployed on a cloud FPGA platform. As a result, the proposed accelerator showed runtime gains of 1.98 times when compared to another accelerator on a highly coupled FPGA platform. In the second approach, we propose an accelerator generator for simulations of Gene Regulator Networks (GRNs) using FPGAs in the cloud. This generator automatically creates the entire hardware/software set from a high-level description of a GRN model. We used six GRN models proposed in the literature to compare performance and cost with implementations on CPU (Central Processing Unit), GPU (Graphics Processing Unit), and FPGA. As a result, the FPGA accelerator is at least 12 times faster than the best evaluated GPU accelerator and offers the best performance per dollar in cloud services, at least 5 times better than the best GPU accelerator. In the second stage of this thesis, we present a framework called REconfigurable Accelerator DeploY (READY), the first framework to support mapping applications to high-performance CPU-FPGA platforms at runtime. READY introduces efficient mapping onto a Coarse Grained Reconfigurable Architecture (CGRA), which hides the latency of the global interconnection network by executing multiple tasks. Additionally, we demonstrate how this system contributes to solving some of the challenges related to the adoption of domain-specific accelerators on FPGAs. The results of this approach show that it is possible to map and execute dataflow graph

applications efficiently, and the system's performance results in acceleration of up to 2 times compared to a modern CPU. And finally, the third and final stage of this thesis presents the High Performance Coarse Grained Reconfigurable Accelerator (HPCGRA), a tool designed to generate domain-specific accelerators with flexible multidimensional interconnections. The CGRA is generated using key building blocks such as functional units, interconnection patterns, routing and elastic queue resources, configuration words, and memories. The HPCGRA optimizes and simplifies the architecture creation process through a portable description, generating generic and efficient Verilog RTL code. Synthesis results show that, even with an increased number of Processing Elements (PEs) in the architectures generated by HPCGRA, the clock frequency and the resource usage per PE remain stable. It was possible to synthesize CGRA architectures with 3036 PEs, with a theoretical performance of 1.2 TOPS.

Keywords: fpga; cgra; dsa; rdsa; hpcgra

LISTA DE FIGURAS

LISTA DE TABELAS

SUMÁRIO

Capítulo 1

Introdução

A eficiência energética em centros de processamento de dados e dispositivos móveis tem se tornado uma preocupação cada vez maior. Esse cenário se intensifica devido ao grande volume de dados gerados diariamente. Diante disso, é suma importância investigar novas estratégias que permitam realizar o processamento de dados de maneira mais eficaz[20].

A busca por melhorias no desempenho dos sistemas computacionais enfrenta diversos desafios, como a dissipação de calor e a subutilização de área [38]. Com a quebra da Lei de Escala de Dennard [30, 97] e à medida que os processos de fabricação avançam para escalas nanométricas, a indústria de semicondutores tem enfrentado dificuldades para manter válida a Lei de Moore [36]. Essa limitação tem resultado em uma redução na taxa de aumento do desempenho dos novos processadores. Uma possível abordagem para mitigar esse problema é a adoção de novas arquiteturas e linguagens projetadas para solucionar problemas específicos [89, 20].

A arquitetura de Von Neumann [53] é atualmente a arquitetura de processador dominante, na qual programas e dados são armazenados na memória principal do sistema, e uma CPU (*Central Processing Unit*), ou Unidade Central de Processamento, realiza a busca e execução de instruções e dados. Esta arquitetura evoluiu ao longo das décadas, resultando no desenvolvimento de CPUs avançadas com melhorias significativas de desempenho. Entre as melhorias mais notáveis estão o aumento da frequência do *clock* e a implementação de uma estrutura hierárquica de memória *cache*. O aumento da frequência do *clock* tem um impacto direto no tempo de execução, tornando as operações mais rápidas, enquanto uma hierarquia de memória aumenta a localidade de acesso e reuso dos dados, reduzindo a latência de acesso à memória principal mais lenta.

Os ganhos de desempenho da arquitetura Von Neumann persistiu por muitas décadas. No entanto, por volta do início do século 21, ficou claro que havia um limite físico para a frequência de *clock* de uma CPU devido à geração de calor e ao aumento resultante da densidade de potência (Watt/mm^2) que o *die* de uma CPU conseguiria suportar. Atingido o limite no *clock* de uma única CPU e ainda havendo

a possibilidade de aumentar o número de transistores por *die* devido à Lei de Moore, ficou claro para os arquitetos de CPUs que o caminho para fornecer desempenho crescente seria incluir mais núcleos em um único *die*. Deste modo, com o lançamento do IBM POWER4 [33] no início dos anos 2000, a primeira arquitetura *multicore*, iniciou-se uma nova era em arquitetura de CPUs.

Apesar do surgimento de arquiteturas *multicore*, os arquitetos continuaram a pesquisar mecanismos para aumentar a eficiência energética das CPUs, como *clock gating*, *frequency scaling*, etc. Estes mecanismos têm contribuído muito para redução do consumo energético das CPUs, mas não ao ponto de permitir retomar o aumento na frequência de *clock* de cada núcleo individualmente. Por outro lado, todo um esforço foi feito no desenvolvimento de *software* que permitisse explorar o paralelismo de arquiteturas *multicores*. No entanto, está ficando claro também que existe um limite na capacidade dos programas explorarem este paralelismo [74]. Deste modo, os últimos anos vem presenciando um crescente consenso na indústria de que estamos entrando na Era dos Aceleradores [29].

Dentre os dez primeiros sistemas na lista do TOP500 de novembro de 2023, sete usam algum tipo de acelerador [111]. As principais categorias de aceleradores são: ASICs (*Application Specific Integrated Circuits*), GPUs (*Graphics Processing Units*) e FPGAs (*Field Programmable Gate Arrays*).

Os ASICs possuem alto desempenho e baixo consumo energético quando comparados às CPUs; porém, não são flexíveis e normalmente se destinam a problemas específicos, possuindo um alto custo de desenvolvimento.

As GPUs são flexíveis, conseguindo fornecer um desempenho na ordem de 10^{12} operações de ponto flutuante por segundo (teraFLOPS). No entanto, as GPUs não apresentam um bom desempenho para aplicações que não possuem paralelismo do tipo SIMD (*Single Instruction Multiple Data*), ou Única Instrução Múltiplos Dados. Além disso, as GPUs são dispositivos com alto consumo de energia quando comparadas aos ASICs e FPGAs [25].

Os FPGAs, quando comparados aos CPUs e GPUs, possuem um baixo consumo energético [25], além de serem flexíveis e permitirem a exploração de todos os tipos de paralelismo. Outra característica dos FPGAs é a possibilidade de trabalhar com larguras de *bits* personalizadas, o que pode superar o desempenho de GPUs em aplicações que permitem este tipo de otimização [79].

Porém, devido à sua alta reconfigurabilidade, os FPGAs não possuem uma alta frequência de *clock* como os CPUs e GPUs. Este fator, que chega a ser uma ordem de magnitude menor, torna os FPGAs dispositivos lentos. Por outro lado, FPGAs modernos trazem novos recursos de *hardware* embarcados que permitem a realização de operações com maior granularidade. Estes recursos são conhecidos como DSPs (*Digital Signal Processing*), ou Processadores Digitais de Sinais, e podem realizar

diversas operações, como multiplicação de escalares, de vetores e de matrizes, com o uso de núcleos de tensores semelhantes aos presentes em GPUs modernas. Um exemplo de FPGA com recursos de blocos de DSPs para otimizar operações de matrizes é avaliado em [12].

O uso de FPGAs modernos em sistemas heterogêneos CPU-FPGA, nos quais o FPGA atua como um acelerador de funções do processador host, enfrenta desafios como a baixa frequência de clock e a movimentação de dados entre a memória do host e o FPGA. Esse gargalo pode ter um impacto significativo no tempo total de execução de uma aplicação, um problema também observado em GPUs [25]. No entanto, FPGAs oferecem excelente desempenho para aplicações altamente paralelizáveis.

Grandes empresas como a Microsoft, Intel e Amazon vêm equipando seus centros de dados com FPGAs [77, 48, 6]. Contudo, os FPGAs apresentam desvantagens, como uma complexidade maior para os programadores e longos tempos de compilação. Estas desvantagens criam grandes desafios para a adoção do FPGA como um acelerador em sistemas computacionais de alto desempenho.

Fabricantes de FPGAs como Intel [54] e AMD Xilinx [117] investem em HLS (*High Level Synthesis*), ou Síntese de Alto Nível. Neste modelo de desenvolvimento, linguagens de alto nível são usadas para criar aceleradores de *hardware*, porém essas linguagens precisam ser traduzidas para uma HDL (*Hardware Description Language*), ou Linguagem de Descrição de *Hardware*, para só depois serem sintetizadas em circuitos no FPGA. Essa tradução pode gerar perda de desempenho e um maior uso de recursos, dado que a estrutura das linguagens de alto nível não foi desenvolvida para ser compilada diretamente para *hardware*. Outro ponto que deve ser considerado é que os FPGAs possuem restrições de recursos e de sincronização de áreas de *clock*, que, dependendo da estrutura de código do circuito, uma solução viável de síntese pode não ser encontrada, mesmo após o circuito ter passado por simulações.

Embora o uso de HLS torne o desenvolvimento de aceleradores em *hardware* mais próximo do desenvolvimento de aplicações em *software*, a compilação ainda precisa passar pelo processo de síntese para ser executada em FPGA. Esse processo é lento quando comparado à compilação para um GPP (*General Purpose Processor*), ou Processador de Propósito Geral, podendo levar horas.

Diferentemente dos FPGAs, que demandam horas de compilação, uma arquitetura amplamente estudada na academia por possuir um baixo tempo de síntese são os CGRAs (*Coarse Grained Reconfigurable Architecture*), ou Arquiteturas Reconfiguráveis de Granularidade Grossa. Um CGRA é configurado a nível de palavra, ou seja, as operações lógicas e aritméticas de uma aplicação são diretamente mapeadas para os PEs (*Processing Element*), ou Elementos de Processamento, da

arquitetura. Diferentemente do FPGA, que possui um arranjo de blocos reprogramáveis que implementam funções lógicas a nível de *bit*, os CGRAs são compostos por um arranjo de PEs, os quais possuem unidades lógicas e aritméticas que processam palavras de *N bits*. Este fato torna os CGRAs uma boa alternativa para serem usados como aceleradores em sistemas heterogêneos.

Dessa forma, metodologias inovadoras que exploram arquiteturas reconfiguráveis com granularidade alta têm o potencial de reduzir a lacuna existente no uso desses dispositivos como aceleradores em sistemas heterogêneos. No entanto, a falta de padronização e especificação das arquiteturas de CGRAs atuais dificulta sua adoção. Além disso, a inexistência de dispositivos CGRAs comercialmente disponíveis representa uma barreira adicional. Diante desse cenário, uma abordagem promissora é o uso de arquiteturas CGRA operando como uma camada virtual sobre FPGAs. Essa abordagem, conhecida como Overlay, permite que projetistas utilizem FPGAs disponíveis no mercado para a implementação de arquiteturas personalizadas. No entanto, o alto tempo de síntese dos FPGAs e a complexidade do desenvolvimento para essas plataformas exigem novas estratégias e ferramentas que facilitem seu uso.

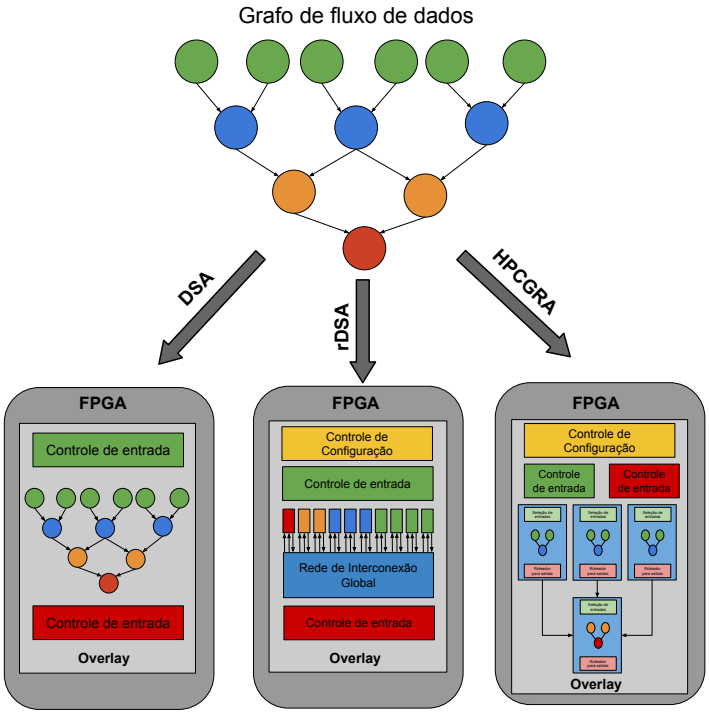


Figura 1.1: Abordagens adotadas para acelerados baseados em grafos de fluxo de dados: **DSA** (*Domain Specific Accelerator*), ou Acelerador de Domínio Específico; **rDSA** (*Reconfigurable Domain Specific Accelerator*), ou Acelerador de Domínio Específico Reconfigurável; **HPCGRA** (*High Performance Coarse Grained Accelerator*).

1.1 Objetivos e contribuições

Uma das lacunas fundamentais para a adoção de FPGAs em sistemas de computação de alto desempenho é a dificuldade de programação e síntese de aceleradores baseados nesta arquitetura. Visando contribuir para uma solução deste problema, esta Tese apresenta três abordagens para síntese de aceleradores de domínio específicos que possam ser empregados em FPGAs, facilitando assim o uso de aceleradores de *hardware* em sistemas heterogêneos CPU-FPGA. A Figura 1.1 apresenta uma visão geral das três abordagens, onde uma aplicação representada pelo Grafo de Fluxo de Dados (GFD) é empregada em aceleradores em FPGA de três formas distintas: como um **DSA** (*Domain Specific Accelerator*), ou Acelerador de Domínio Específico, onde a aplicação é mapeada diretamente em FPGA; um **rDSA** (*Reconfigurable Domain Specific Accelerator*), ou Acelerador de Domínio Específico Reconfigurável, onde uma arquitetura de CGRA é utilizada como uma camada de abstração sobre o FPGA para permitir um rápido mapeamento de aplicações no formato de GFD; e o **HPCGRA** (*High Performance Coarse Grained Architecture*), onde é possível criar arquiteturas de CGRAs personalizadas com blocos de operações mais complexas e interconexões flexíveis para ser utilizado como uma camada de abstração sobre o FPGA para permitir a execução de aplicações em GFD com maior desempenho.

Desta forma, um conjunto de ferramentas e modelos de arquiteturas de aceleradores foram desenvolvidos e publicados em diferentes trabalhos, que embarcam as seguintes contribuições:

- Modelo de desenvolvimento de *hardware* por meio de geradores de código no modelo *Register Transfer Level* (RTL) para execução de algoritmos em aceleradores em FPGAs na nuvem e sistemas heterogêneos CPU-FPGA [14, 13], que incluem:
  - Gerador específico de código parametrizado para o algoritmo *K-means*.
  - Replicação automática dos aceleradores explorando o paralelismo espacial juntamente com mecanismos para distribuir a configuração.
  - Melhor utilização da interface de transferência de dados entre host e o acelerador.
  - Gerador de *hardware* para acelerar ação de busca de estados atratores em Redes Reguladoras de Genes.
  - Avaliação do custo-benefício de aceleradores em diferentes tipos de plataformas CPU-FPGA na nuvem.

- Nova arquitetura de acelerador genérico virtualizado sobre o FPGA criando assim um ambiente CPU-CGRA para a execução de aplicações descritas no formato de grafo de fluxo de dados [99, 16].
  - Mapeamento em tempo de execução de aplicações descritas no formato GFD para CGRA;
  - Modelo de CGRA com rede global ômega capaz de executar múltiplas *threads* (homogêneas ou heterogêneas) no modelo de SMT (*Simultaneous Multithreading*) em um esquema de pipeline temporal.
  - Arcabouço para geração de código RTL para a arquitetura que permite a variação de parâmetros específicos da arquitetura, como quantidade de blocos de processamento e estágios da rede ômega a fim de fornecer modelo de exploração de espaço de projeto para CGRAs com rede global.
  - Mapeador e roteador de algoritmos descritos na forma de grafos de fluxo de dados acíclicos para modelos de CGRAs com rede global ômega.
- Gerador de arquiteturas de aceleradores CGRA para implantação em plataformas heterogêneas CPU-FPGA, com Interconexões Multidimensionais nas quais o projetista pode escolher de maneira arbitrária a topologia da rede de interconexão entre os PEs e a capacidade de processamento e recursos dos PEs do CGRA [98, 27].
  - Modelo de descrição de arquiteturas com interconexões multidimensionais usando notação de objetos JavaScript (JSON).
  - Modelagem de operações por meio de grafos a fluxo de dados acíclicos usando a notação JSON.
  - Arcabouço para geração e implantação de aceleradores com interconexões multidimensionais em sistemas heterogêneos CPU-FPGA.

1.2 Publicações

Esta Tese é resultado de um conjunto de trabalhos científicos publicados, ao longo do Doutorado do autor, nos seguintes eventos e jornais:

- Silva, L.B.; Ferreira, R.D.S.; Canesche, M.; Menezes, M.M.; Vieira, M.D.; Penha, J.C.; Jamieson, P.; Nacif, J.A.M. Ready: A fine-grained multithreading overlay framework for modern cpu-fpga dataflow applications. ACM Transactions on Embedded Computing Systems, v. 18, n. 5s, p. 56:1-56:20, out. 2019. DOI: 10.1145/3358187.

- Silva, L.B.; Canesche M.; Ferreira R.D.S.; Nacif, J.A.M. HPCGRA - An Orthogonal Designed CGRA Generator for High Performance Spatial Accelerators. In: SIMPÓSIO EM SISTEMAS COMPUTACIONAIS DE ALTO DESEMPENHO (SSCAD), 21. , 2020, Online. Anais [...]. Porto Alegre: Sociedade Brasileira de Computação, 2020 . p. 25-36. DOI: https://doi.org/10.5753/wscad.2020.14055.

- Silva, L.B.; Canesche, M.; Penha, J.C.; Carvalho, W.; Comarela, G.; Nacif, J.A.M.; Ferreira, R.D.S. An Open Source Custom K-means Generator for AWS Cloud FPGA Accelerators. In: SIMPÓSIO BRASILEIRO DE ENGENHARIA DE SISTEMAS COMPUTACIONAIS (SBESC), 11. , 2021, Evento Online. Anais [...]. Porto Alegre: Sociedade Brasileira de Computação, 2021 . p. 173-180. ISSN 2237-5430.

- Silva, L.B.; Penha, J.C.; Canesche, M.; Ribeiro, D.V.; Nacif, J.A.M.; Ferreira, R.D.S. An Open-Source Cloud-FPGA Gene Regulatory Accelerator. In: SIMPÓSIO EM SISTEMAS COMPUTACIONAIS DE ALTO DESEMPENHO (SSCAD), 22. , 2021, Belo Horizonte. Anais [...]. Porto Alegre: Sociedade Brasileira de Computação, 2021 . p. 240-251. DOI: https://doi.org/10.5753/wscad.2021.18527.

- Silva, L.B.; Canesche, M.; Penha, J.C.; Campos, J.; Nacif, J.A.M.; Ferreira, R.D.S. Fast flow cloud: A stream dataflow framework for cloud FPGA accelerator overlays at runtime. Concurrency and Computation: Practice and Experience, v. 35, n. 17, e6977, 2023. DOI: 10.1002/cpe.6977.

- Silva, L.B.; Penha, J.C.; Ribeiro, D.V.; Silva, A.; Nacif, J.A.M.; Ferreira, R.D.S. HPyC-FPGA - Integração de Aceleradores em FPGA de Alto Desempenho com Python para Jupyter Notebooks. In: SIMPÓSIO EM SISTEMAS COMPUTACIONAIS DE ALTO DESEMPENHO (SSCAD), 23. , 2022, Florianópolis/SC. Anais [...]. Porto Alegre: Sociedade Brasileira de Computação, 2022 . p. 193-204. DOI: https://doi.org/10.5753/wscad.2022.226383.

- Silva, M.A.; Silva, L.B.; Penha, J.C.; Nacif, J.A.M.; Ferreira, R.D.S. KCGRA – Uma Arquitetura Reconfigurável de Domínio Específico para K-means. In: SIMPÓSIO EM SISTEMAS COMPUTACIONAIS DE ALTO DESEMPENHO (SSCAD), 24. , 2023, Porto Alegre/RS. Anais [...]. Porto Alegre: Sociedade Brasileira de Computação, 2023 . p. 25-36. DOI: https://doi.org/10.5753/wscad.2023.235892.

1.3 Organização do texto

Este texto está organizado em sete capítulos. No Capítulo 1, é dada uma introdução geral motivando o trabalho. O Capítulo 2 apresenta os fundamentos necessários para o entendimento do texto e também os sistemas e arquiteturas utilizados nos trabalhos. No Capítulo 3 são descritos trabalhos relacionados, demonstrando o diferencial em relação às contribuições feitas por esta Tese. O Capítulo 4 descreve a primeira etapa do trabalho com geração de aceleradores específicos, onde são apresentados dois estudos de caso realizados no intuito de entender os desafios do problema de projeto de aceleradores. No Capítulo 5, a partir das lições aprendidas na etapa anterior, é apresentada uma arquitetura reconfigurável para aceleração de algoritmos em FPGAs. O Capítulo 6 avança ainda mais na generalidade da solução, descrevendo uma metodologia para síntese de aceleradores com topologias genéricas (multidimensionais), a qual é sintetizada a partir de uma descrição de alto nível do acelerador. Além disto, o capítulo propõe um arcabouço de *software* para geração e implantação de aceleradores em sistemas heterogêneos CPU-FPGA. Por fim, o Capítulo 7 apresenta as conclusões do trabalho e aponta possíveis trabalhos futuros.

Capítulo 2

Fundamentos

Este capítulo apresenta os fundamentos sobre as arquiteturas e tecnologias utilizadas nesta tese. A Seção 2.1 descreve os FPGAs e as principais plataformas de computação de alto desempenho que utilizam esses dispositivos como aceleradores reprogramáveis. Duas plataformas de FPGA em nuvem são apresentadas: AWS e HARPv2. Ambas as plataformas foram avaliadas e comparadas nesta tese. A Seção 2.2 apresenta as arquiteturas de CGRAs existentes, com ênfase em suas formas de interconexões, demonstrando os principais tipos de redes.

2.1 *Field Programmable Gate Arrays* - FPGAs

O Field-Programmable Gate Array (FPGA), ou Arranjo de Portas Programáveis em Campo, é uma arquitetura reconfigurável que permite a construção de diferentes circuitos em um mesmo *hardware*. Os FPGAs estão em ascensão devido a novas tecnologias produzirem dispositivos com maior poder computacional, embora tenham surgido na década de 1980. Os FPGAs representam a evolução dos antigos dispositivos PAL (*Programmable Logic Array*), ou Arranjo de Lógica Programável e GAL (*Generic Array Logic*), ou Arranjo de Lógica Genérica, dispositivos que se tornaram populares no desenvolvimento de SOCs (*System on Chip*), ou Sistemas em *Chip*, onde são usados como ambiente de testes.

O FPGA é essencialmente um conjunto de unidades lógicas organizadas de forma bidimensional e interligadas por uma rede de interconexão programável. As unidades lógicas de um FPGA podem ser simplificadamente entendidas como tabelas verdade, ou seja, pequenas memórias capazes de armazenar funções lógicas. Essas memórias são chamadas de LUT (*Lookup Table*), ou Tabela de Busca. Os fabricantes de FPGAs adotam tamanhos diferentes de LUTs para equilibrar a flexibilidade e a eficiência dos circuitos. LUTs menores, permitem um mapeamento mais granular e podem ser mais eficientes em termos de área e consumo de energia para determinadas aplicações. Por outro lado, LUTs maiores, podem implementar funções lógicas mais complexas com menor necessidade de interconexões entre as unidades lógicas, o que pode reduzir a latência e melhorar o desempenho geral do circuito. Nos FPGAs modernos, essas

unidades lógicas são chamadas de CLB (*Configurable Logic Block*), ou Blocos Lógicos Configurável. Os CLBs são blocos mais complexos do que as LUTs, permitindo a execução de funções mais elaboradas. As conexões entre os CLBs são formadas por barramentos em linhas e colunas interconectados por comutadores posicionados nos cruzamentos. Os pinos do encapsulamento dos FPGAs são configuráveis como pinos de entrada ou saída para a conexão externa dos circuitos sintetizados.

A Figura 2.1 apresenta a estrutura interna básica de um FPGA genérico, onde os Blocos Lógicos (*Logic Blocks*), os Blocos de Conexão (*Connection Block*(CB)) e os Comutadores (*Switch Box*) podem ser visualizados. Com o avanço tecnológico, os FPGAs modernos são equipados com blocos de maior granularidade. Estruturas como blocos de memória RAM (*Random Acces Memory*), ou Memória de Acesso Aleatório, com capacidades variando entre dois e vinte quilobytes, e palavras de tamanho variando de 16 a 36 *bits*, têm sido incorporadas. Esses blocos de memória são distribuídos entre os CLBs. Além dos blocos de memória RAM, também podem ser incluídos no mesmo encapsulamento blocos de DSP (*Digital Signal Processing*), ou Processamento Digital de Sinais e ALU (*Arithmetic Logic Unit*), ou Unidade Lógico-Aritmética capazes de realizar operações lógicas e aritméticas, como soma, subtração e multiplicação, entre outras. Se forem necessários mais blocos de memória RAM, os próprios CLBs podem ser utilizados como módulos de memória de poucos *bits*. Alguns fabricantes incorporam processadores nos circuitos integrados FPGA. Por exemplo, o Zynq-7000 da AMD Xilinx conta com um processador ARM Cortex A9 dual core integrado [51].

2.1.1 Serviço de FPGA na Nuvem da AWS

Nesta seção, apresenta-se a plataforma de computação heterogênea reconfigurável da AWS. As FPGAs na nuvem democratizam o acesso ao *hardware*, porém uma lacuna considerável permanece na ausência de modelos e exemplos prontamente disponíveis para os desenvolvedores projetarem aceleradores de *hardware*/*software*. Recentemente, grandes empresas como Intel/Altera, Microsoft, Alibaba e Amazon apresentaram plataformas para computação de alto desempenho com FPGAs na nuvem [48, 77, 4, 6]. No entanto, é um desafio desenvolver para um acelerador em FPGA na nuvem. A AWS oferece instâncias com FPGAs por meio do serviço de computação elástica EC2. Este serviço permite alugar sistemas com configurações customizadas, como a quantidade de núcleos de processador, tamanho da memória principal, presença dispositivos aceleradores como GPU e FPGA.

As instâncias da família F1 possuem FPGAs como dispositivos aceleradores. Existem três tipos de instâncias F1: f1.2xlarge, f1.4xlarge e f1.16xlarge, cada uma com configurações diferentes em relação ao número de processadores, memória principal

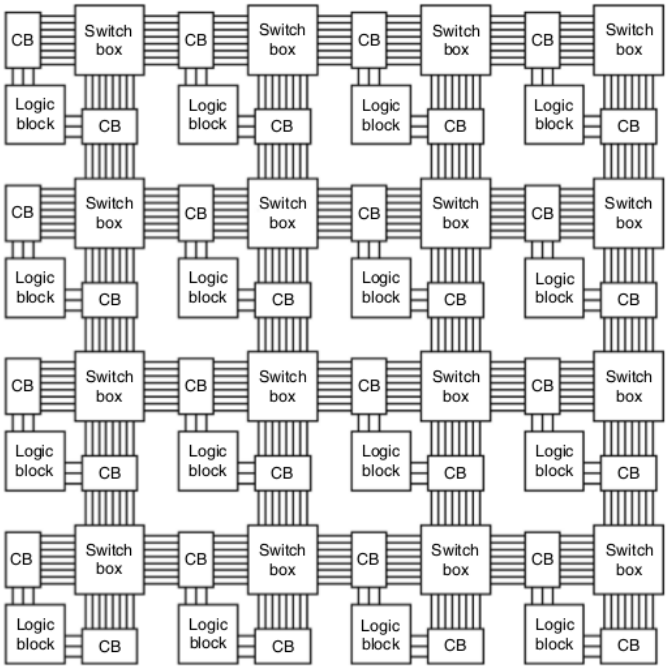


Figura 2.1: Estrutura interna básica dos FPGAs. Fonte: [71].

e dispositivos FPGAs. A Tabela 2.1 mostra os recursos para cada categoria de instância. A instância f1.2xlarge é a mais simples, com apenas um dispositivo FPGA. No entanto, dependendo da aplicação, o uso de vários FPGAs pode ser mais interessante, pois a melhoria de desempenho compensaria o custo adicional. Além disso, é possível utilizar a AWS sob demanda ou reservar uma instância por um, ou três anos. Assim, os valores podem variar ao longo do tempo e conforme a demanda do produto.

Tabela 2.1: Tipos de instâncias AWS EC2 F1

Instância	**vCPU**	**Memória (GB)**	**FPGA Dispositivos**	**Preço/hr($) Sobre demanda**
f1.2xlarge	8	122	1	1,65
f1.4xlarge	16	244	2	3,30
f1.16xlarge	64	976	8	13,20

A Figura 2.3 apresenta um diagrama de blocos da arquitetura AWS EC2 F1 para um dispositivo FPGA. Cada instância possui um processador hospedeiro com 64 GB de memória DDR4, um dispositivo FPGA VU9P UltraScale+ com quatro memórias DDR4 externas de 16 GiB, totalizando 64 GiB. O gargalo do sistema é o PCI Express X16 de 16 Gb/s que conecta a CPU hospedeira ao dispositivo FPGA. Similar aos dispositivos GPU, a memória global do FPGA é desacoplada do espaço de memória da CPU, e todos os dados devem ser explicitamente transferidos de/para o dispositivo hospedeiro/FPGA. A memória externa DDR4 do FPGA possui um alto rendimento

de 64 Gb/s. O dispositivo FPGA Ultrascale+ possui 6.840 blocos DSP. A Figura 2.2 mostra a estrutura interna de um bloco de DSP, com um multiplicador de inteiros de 27x18 *bits*, um pré-somador de 27 *bits* para economia de energia, um acumulador de 48 *bits* que pode ser construído em cascata com acumuladores maiores e uma unidade lógica com as operações bit a bit: AND, OR, NOT, NAND, NOR, XOR e XNOR. Além de várias unidades DSP, o FPGA VU9P possui 2.586.150 células lógicas, 2.364.480 flip-flops, 2.160 blocos de RAM (total de 75,9 Mb) e 960 blocos de Ultra RAM (total de 270 Mb).

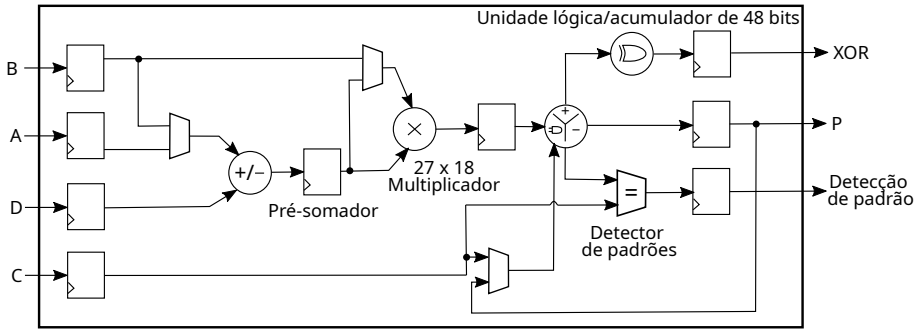


Figura 2.2: Bloco DSP Ultrascale+.

Embora seja um FPGA grande, a interface da AWS usa uma área significativa do dispositivo. O FPGA é dividido em duas regiões: SHELL e lógica personalizada (CL). O SHELL é a parte estática responsável pelo controle de execução e interface de comunicação com o aplicativo do usuário mapeado para a região CL reconfigurável. A plataforma utiliza o protocolo AXI4 para se comunicar com a memória externa, onde é possível instanciar vários controladores. A Figura 2.3 mostra quatro controladores AXI4. A configuração mínima requer pelo menos um AXI4 conforme destacado na Figura 2.3 por uma caixa preta. A largura do barramento é configurável. No entanto, conforme mostrado na Figura 2.3, a implementação física da AWS foi projetada usando quatro barramentos de 512 *bits*. Portanto, o uso de 512 *bits* (ou múltiplos) na lógica personalizada é recomendado para atingir uma taxa de transferência maior.

2.1.2 Plataforma de pesquisa de arquiteturas heterogêneas HARPv2

A Intel apresentou pela primeira vez o projeto HARP em [84], criando um sistema de computação reconfigurável baseado em FPGA fortemente acoplada a uma CPU por uma memória cache coerente de forma transparente. A primeira versão do protótipo possui um processador CPU Xeon e uma FPGA Stratix V com memória compartilhada usando o *Intel QuickPath Interconnect* (QPI). Este protótipo foi inicialmente enviado em 2015 [47] para universidades selecionadas para pesquisa. Em seguida, a Intel ofereceu

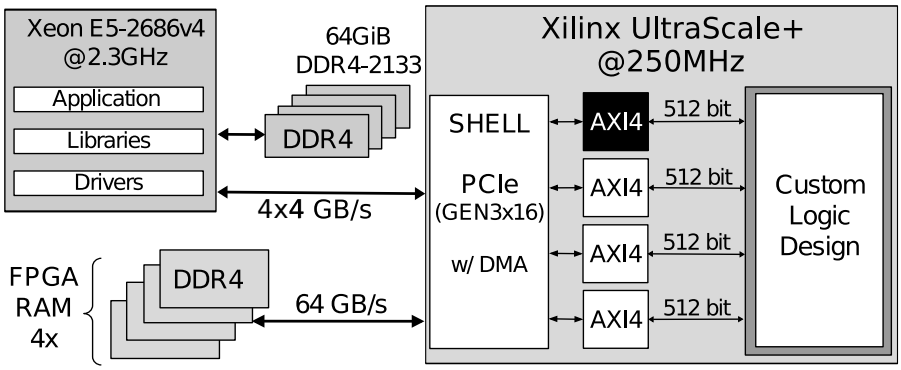


Figura 2.3: Plataforma AWS EC2 F1.

uma segunda versão, o HARPv2, que possui o Xeon E5-2600 v4 e uma FPGA Arria 10. Além disso, o HARPv2 foi instalado em *clusters* em instituições nos EUA e na Europa e está disponível para mais de cem universidades e institutos de pesquisa [47]. Embora a Intel tenha lançado a plataforma há algum tempo, ela ainda é uma plataforma de pesquisa.

A Figura 2.4 mostra o componente principal do HARPv2, que combina um processador Broadwell Xeon de 14 núcleos e um FPGA Arria 10. Além disso, o sistema possui três canais de comunicação entre a CPU e o FPGA: dois PCIe e um QPI. A taxa de transferência máxima entre a CPU e o FPGA é de aproximadamente 25 GiB/s. Como mencionado anteriormente, um protocolo de memória cache coerente implementa um sistema heterogêneo de memória compartilhada CPU/FPGA. A interface de comunicação do FPGA tem largura de 512 *bits*. Em relação às camadas de *software*/*hardware*, a API OPAE gerencia a comunicação dos canais PCIe/QPI para fornecer métodos de envio/recebimento de dados da/para a unidade funcional do acelerador no FPGA.

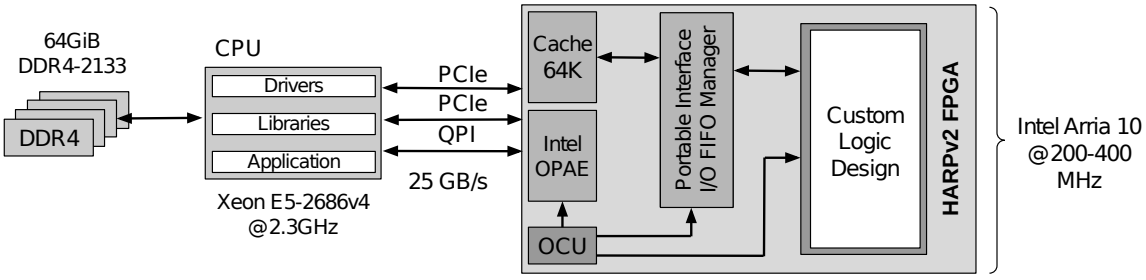


Figura 2.4: Plataforma HARPv2.

2.2 *Coarse Grained Reconfigurable Architecture*-CGRAs

Nesta seção, são apresentadas os CGRAs (*Coarse-Grained Reconfigurable Architectures*), ou Arquiteturas Reconfiguráveis de Grão Grosso, existentes, bem como as formas de interconexões mais comumente utilizadas. Os CGRAs surgiram como uma alternativa aos FPGAs para reduzir o tempo e a complexidade de síntese de algoritmos. Ao contrário dos FPGAs, os quais são arquiteturas reprogramáveis de grão fino no nível de *bits*, os CGRAs são programáveis no nível da palavra para reduzir o esforço de mapeamento. Nos CGRAs, as operações são mapeadas diretamente em unidades funcionais, ao contrário das aplicações para FPGA, que precisam mapear cada operação na forma de equações lógicas binárias, exigindo várias etapas de compilação, como posicionamento e roteamento (P&R). Esse tipo de compilação é comumente conhecida como síntese, e é importante observar que as etapas de posicionamento e roteamento são problemas NP-completos [49].

O P&R em um CGRA difere do FPGA tradicional devido um CGRA possuir unidades funcionais onde as operações de uma aplicação são mapeadas diretamente nessas unidades. Essas unidades funcionais também são conhecidas como Elementos de Processamento (PEs do inglês *Processing Element*). O projeto de um CGRA enfrenta muitos desafios, como estruturar e implementar a topologia de conexão entre os PEs, a interface de comunicação com a memória externa, os bancos de registradores e os recursos de roteamento. Além disso, a falta de disponibilidade e padronização das plataformas de CGRA impedem a ampla adoção pela comunidade de *software*.

Em [114] Wijtvliet *et. al* apresenta as principais arquiteturas de CGRAs do estado da arte das duas décadas passadas. Esse trabalho é um bom ponto de partida para o estudo e compreensão das classificações de arquiteturas reconfiguráveis de grão grosso. Segundo Wijtvliet *et. al* existem muitas definições informais e conflitantes de CGRAs. Isso se dá devido os CGRAs serem um tipo de arquitetura amplamente estudada em ambientes acadêmicos, mas, por outro lado, não possuem muito espaço na indústria. Este fato pode ser explicado devido não haver uma padronização em sua estrutura e forma de programação. Por outro lado, isso torna o campo de pesquisa de novas arquiteturas em aberto, tornando se uma tendência atualmente com a saturação do desempenho de arquiteturas clássicas de processadores.

Para classificar um CGRA, Wijtvliet *et. al* propuseram um modelo baseado em 4 dimensões. A primeira dimensão trata da estrutura da arquitetura que possui as propriedades: tipo de rede de interconexão, granularidade, hierarquia de memória e banco de registradores. A segunda dimensão trata do controle usado durante a execução da aplicação e possui as seguintes propriedades: escalonamento, reconfiguração, roteamento da rede, operadores. A terceira dimensão trata da forma

de integração da arquitetura em um sistema computacional e possui as seguintes propriedades: integração, acoplamento e compartilhamento de recursos. E por fim, a última dimensão é a respeito das ferramentas disponíveis em cada arquitetura, onde, as seguintes propriedades são analisadas: compilador e ferramentas de posicionamento, roteamento e exploração do espaço de projeto.

Existem arquiteturas clássicas, como as descritas em [100, 44, 35], que foram desenvolvidas para serem usadas como coprocessadores em sistemas heterogêneos e servem como base para pesquisas de novas arquiteturas de CGRA. Embora existam poucos CGRAs comerciais disponíveis [60, 37], existem vários CGRAs que foram desenvolvidos para serem virtualizados em FPGAs [55, 18, 65]. O uso de CGRAs como uma camada de abstração sobre os FPGAs tem ganhado a atenção da comunidade acadêmica [101]. Esses CGRAs são conhecidos como Overlays, onde um conjunto de recursos de granulação grossa oculta a complexidade do FPGA, permitindo que um acelerador de *hardware* execute diferentes aplicativos de forma transparente. As arquiteturas de Overlays baseadas em CGRA melhoram o tempo de compilação de aplicações em relação aos FPGAs. Por exemplo, o Overlay proposto em [69] relata uma melhoria de 40% na taxa de transferência em comparação com um projeto mapeado para um FPGA usando a linguagem Vivado HLS.

Um fator crucial para o desempenho de uma arquitetura de CGRA é a quantidade de PEs, pois todos os PEs podem executar operações em paralelo. O número de PEs em uma arquitetura é limitado pela tecnologia de fabricação de circuitos em silício para CGRAs implementados diretamente em chip, ou pela tecnologia da arquitetura de FPGAs para CGRAs sintetizados como uma camada de abstração sobre os FPGAs. Para CGRAs sintetizados em FPGAs, o termo utilizado na literatura é "Overlays CGRAs", nesta tese, o termo CGRA é explicitamente utilizado para arquiteturas virtualizadas sobre o FPGA.

O CGRA simplifica o processo de mapeamento de aplicações para o FPGA. No entanto, implementação de um *hardware* de um CGRAs não é uma tarefa simples, pois o desenvolvedor enfrenta limitações de recursos lógicos do FPGA e de frequência máxima de operação. Devido a essas limitações, vários modelos de CGRAs têm surgido nos últimos anos, cada um utilizando uma estratégia diferente de organização dos elementos da arquitetura para facilitar o mapeamento de algoritmos, reduzir o consumo de recursos e, consequentemente, aumentar o desempenho.

Uma das características arquitetônicas principais de um CGRA é a sua rede de interconexão. Uma grande classe de CGRAs baseia-se em um modelo de malha 2D, como o ADRES [76]. Para esse tipo de CGRA, o P&R é demorado, pois a solução de mapeamento deve encontrar uma solução que minimiza a distância entre os operadores e o roteamento deve equilibrar os caminhos dos dados entre os

operadores da aplicação. A Figura 2.5(a) mostra um grafo de fluxo de dados simples mapeado em uma malha 2D (veja a Figura 2.5(b)). Embora haja apenas uma aresta $G \rightarrow C$ com distância de dois saltos, é necessário balancear todos os caminhos roteados, inserindo registradores nas arestas $F \rightarrow C$ e $B \rightarrow A$ (veja as Figuras 2.5(c) e (d)). Portanto, a interconexão em um CGRA requer recursos de filas programáveis, conhecidas como filas elásticas. Embora a malha 2D seja uma rede de interconexão escalável e de baixo custo, sua baixa conectividade aumenta a complexidade do P&R. Os algoritmos de P&R para malha 2D [49, 76] são demorados para serem integrados em abordagens que exigem a compilação para a arquitetura em tempo de execução.

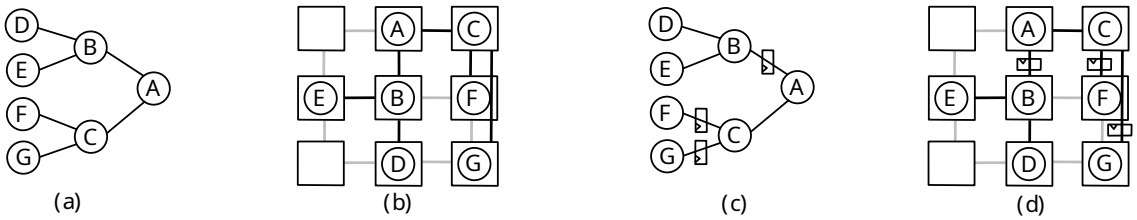


Figura 2.5: Malha CGRA 2D: (a) Grafo defluxo de dados; (b) Posicionamento e roteamento iniciais; (c) Balanceamento adicionando registradores nos caminhos roteados; (d) Roteamento final incluindo registradores.

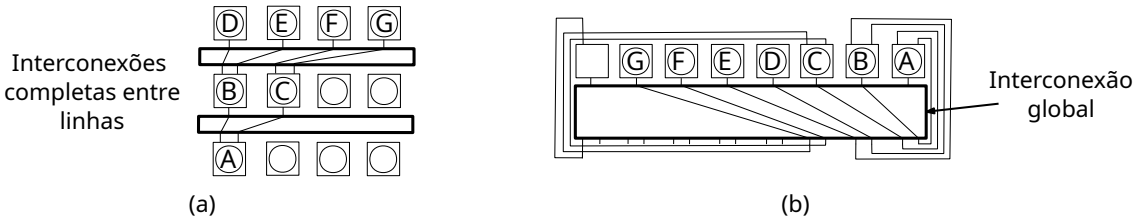


Figura 2.6: Topologias com suporte P&R rápido: (a) Faixa linear (baixa ocupação, conexões completas entre linhas); (b) Interconexão global (baixa frequência de *clock*).

Além dos modelos com malhas 2D, várias topologias de interconexão foram estudadas, como grupos hierárquicos de malhas 2D [104, 95], faixas lineares [87], e redes globais [39]. As faixas lineares reduzem a complexidade do P&R, como mostrado na Figura 2.6(a) para o grafo de fluxo de dados representado na Figura 2.5(a). No entanto, a ocupação da linha é baixa e as redes entre faixas consomem uma quantidade significativa de recursos por exigirem redes de interconexão completas. Uma rede global melhora a ocupação e elimina o desperdício de recursos, como mostra a Figura 2.6(b). Além disso, assumindo um escalonamento válido, o mapeamento P&R é $O(N)$ para uma rede global do tipo *crossbar*. No entanto, seu custo de área é proibitivo para números de entradas e saídas maiores que 32 em FPGAs [39]. Uma rede multiestágio reduz o uso de recursos [39], mas o atraso de

propagação da rede limita a frequência máxima de *clock* do projeto, e a rede deve ter pelo menos $2\log(N) - 1$ estágios para não ser bloqueante ou rearranjável [112].

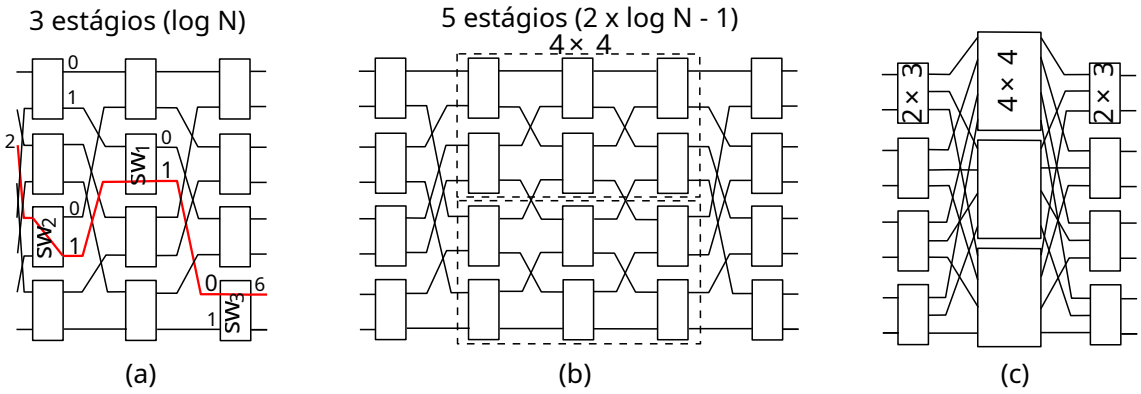


Figura 2.7: Exemplos de multiestágios: (a) $\log N$ estágios bloqueantes; (b) $2\log N - 1$ rearranjáveis; (c) Não bloqueante(Exemplo: Clos).

Uma rede de interconexão multiestágio torna-se bloqueante com base no número de comutadores por estágio e no número de suas portas. A Figura 2.7(a) mostra uma rede Ômega bloqueante com 8 entradas e saídas e comutadores de 2 portas, resultando em $\log_2(n) = \log_2 8 = 3$ estágios. É simples provar que essa rede é bloqueante por existirem $N! = 8! = 40320$ permutações possíveis, mas a rede pode gerar apenas um subconjunto de $2^C = 2^{12} = 4096$ permutações de mapeamento, onde C é o número de comutadores com 2 portas.

A Figura 2.7(b) mostra uma rede Benes rearranjável com $2\log_2(n) - 1 = 2\log_2(8) - 1 = 5$ estágios. Uma rede rearranjável pode realizar permutações arbitrárias se o roteamento for feito em uma ordem específica. No entanto, o aumento do número de comutadores e interconexões aumenta tanto o uso de recursos quanto a latência.

Por fim, a Figura 2.7(c) mostra uma rede Clos não bloqueante, onde o roteamento pode ser feito em qualquer ordem. No entanto, os comutadores em uma rede Clos consomem uma quantidade significativa de recursos. A primeira e a última camada possuem comutadores com 2 portas de entrada e 3 portas de saída, enquanto o estágio intermediário possui comutadores com 4 portas de entrada e saída. Em comparação com a rede Benes, o estágio intermediário da rede Clos possui três comutadores de 4 portas de entrada e saída, enquanto a rede Benes possui apenas dois, como mostrado pela linha tracejada na Figura 2.7(b). Os três estágios intermediários são agrupados para criar os comutadores de 4 portas de entrada e saída. Portanto, há uma compensação entre o uso de recursos e a capacidade de roteamento da rede.

Capítulo 3

Trabalhos Relacionados

Este capítulo apresenta os trabalhos da literatura relacionados à pesquisa desta tese. Na Seção 3.1, são apresentados os trabalhos relacionados à abordagem inicial de síntese direta de aceleradores em FPGA descrita no Capítulo 4. Na Seção 3.2, são apresentados os trabalhos relacionados ao uso de geradores arquiteturas reconfiguráveis para generalizar a síntese de aceleradores em FPGA, descrita nos Capítulos 5 e 6.

3.1 Abordagens baseadas em síntese direta

3.1.1 Redes Reguladoras de Genes

Os modelos de GRN envolvem tarefas com alto custo computacional, sendo abordados com técnicas de HPC (*High Performance Computing*), ou Computação de Alto Desempenho, que utilizam aceleradores paralelos e técnicas de divisão e conquista. Borelli *et al*. [11] propuseram uma abordagem baseada em GPU para inferir grandes modelos de GRN com até 4096 genes. Essa inferência é realizada particionando o modelo em blocos de genes menores, sendo que o tamanho do bloco influencia no tempo de execução. No Capítulo 4, o acelerador foi desenvolvido para computar os estados atratores de GRN, utilizando um algoritmo específico implementado por meio de máquina de estados em hardware. Portanto, apesar do trabalho de Borelli apresentar uma solução que pode ser executada em múltiplas GPUs, ainda seria necessário aplicar um algoritmo para detectar os estados de estabilização, uma vez que apenas a inferência do modelo foi realizada.

O trabalho apresentado em [75] é o que mais se aproxima da abordagem desenvolvida no Capítulo 4. Os resultados desse trabalho alcançam uma ordem de magnitude de aumento de desempenho sobre um processador com múltiplas *threads*, utilizando uma implementação em FPGA. Nessa implementação, o modelo de GRN é primeiro convertido em Verilog para executar em um FPGA conectado a um processador POWER8 da IBM. A implementação em FPGA proposta é cerca de 10^2 a 10^3 vezes mais rápida do que o pacote Boolnet R [80], calculando em média 2

milhões de atratores por segundo. Ao compararmos os resultados das implementações para o mesmo modelo de GRN, o acelerador proposto no Capítulo 4 é 11 vezes mais rápido.

Trabalhos da literatura utilizam diferentes estratégias para realizar a busca de atratores em GRNs. Por exemplo, o trabalho de Wensheng Guo *et al*. [45] reduziu o problema de busca de atratores a um problema de satisfatibilidade e o paralelizou para execução em ambientes multicores. Hongyang Qu *et al*. [92] propuseram o uso BDD (*Binary Decision Diagram*), ou Diagrama de Decisão Binário, descrevendo dois algoritmos: um para modelos de GRN pequenos, com até 50 genes, e outro para modelos de GRN grandes, com mais de 50 genes. No entanto, essas abordagens foram desenvolvidas para execução em *software*, diferente da implementação desenvolvida no Capítulo 4 que foi desenvolvida em *hardware*.

3.1.2 Aceleradores Algoritmos *K-means*

O algoritmo *K-means* é amplamente utilizado em tarefas de agrupamento sem supervisão, onde cria grupos com base nas distâncias entre os pontos de dados em um espaço multidimensional. No entanto, a complexidade da execução depende do número de grupos e das dimensões dos pontos na base de dados. Os números de grupos (K) e dimensões (N) são meta parâmetros do *K-means* sendo escolhidos pelo usuário. No Capítulo 4, foi proposto um gerador de aceleradores para o algoritmo *K-means* em hardware. O gerador proposto recebe como entrada os meta parâmetros K e N e gera um acelerador específico para agrupar pontos com N dimensões em K grupos. Outros trabalhos da literatura também propuseram geradores de *K-means* para hardware. No entanto, nenhum outro trabalho apresentou um estudo desses aceleradores em plataformas na nuvem, realizando uma comparação de custo de execução entre diferentes tipos de máquinas.

Tang e Khalid [107] apresentam uma abordagem de alto nível para implementar o algoritmo *K-means* em FPGA utilizando a linguagem OpenCL. Nessa abordagem, é utilizada a distância de Manhattan em vez da distância euclidiana, pois a distância de Manhattan não requer operações de multiplicação, o que é mais adequado para o FPGA. A implementação pode realizar tanto a classificação quanto o reposicionamento dos centroides em hardware. Isso difere da nossa abordagem, onde apenas a classificação é realizada em hardware. Portanto, não é possível fazer uma comparação direta com o acelerador proposto no Capítulo 4.

Neshatpour *et al*. apresentam uma estimativa para o tempo de execução do algoritmo *K-means* em FPGA [82], porém, essa estimativa não inclui resultados de tempo real de execução. Os mesmos autores também apresentam outra implementação em uma plataforma CPU/FPGA da AMD Xilinx em [81]. Nesse

trabalho, o desempenho é estimado com base no potencial de desempenho do FPGA e da CPU, considerando a comunicação entre os componentes. Essa estimativa sugere um desempenho potencial 150 vezes maior para o *K-means* em relação à CPU. No entanto, os autores não avaliam o sistema em tempo real de execução.

Amaricai [5] apresentou um gerador parametrizável para FPGA, considerando o número de dimensões e centroides. No entanto, o trabalho não fornece informações sobre o tempo de execução, apenas simulações e resultados estimados. É importante observar que uma descrição simulada nem sempre será executada corretamente em hardware, uma vez que a frequência de *clock* não é considerada em um ambiente simulado. Além disso, os autores podem não ter calculado com precisão os tempos de sincronização, leitura e escrita de dados. O gerador proposto no Capítulo 4 inclui todas as fases de um projeto de hardware para FPGAs.

Li *et al*.[68] propuseram um acelerador FPGA *K-means* baseado no paradigma *Map-Reduce*. Os autores avaliaram o desempenho usando entradas de 2 e 4 dimensões para $k = 4$. Em comparação com os resultados apresentados na Seção 4.2.3, nosso gerador demonstra tempos de execução duas vezes mais rápidos que os resultados de Li *et al*.[68] para $n = 2$ e $n = 4$. Lee *et al*.[63] propuseram uma arquitetura com elementos de processamento para FPGA, sendo pelo menos duas vezes mais lenta que nosso gerador para $k = 3$ e $n = 10$.

O trabalho apresentado em [2] é o mais próximo da nossa abordagem, considerando a plataforma FPGA em nuvem, com resultados de tempo de execução. Em [2], o FPGA é usado para calcular a etapa de classificação, enquanto a CPU calcula o reposicionamento. Porém, os autores avaliam apenas os resultados para $K = 8, N = 2$ e $K = 4, N = 4$, para esses mesmo parâmetros, nosso acelerador é mais de 20 vezes mais rápido.

Penha *et al*.[88] apresentam uma implementação em FPGA na plataforma Intel Harpv2, a mais próxima da nossa abordagem em termos da arquitetura do acelerador *K-means*. A principal diferença é a plataforma Intel/Altera sem memória local e três interconexões PCI-bus entre o FPGA e a CPU[23]. Souza e colaboradores apresentam uma implementação do *K-means* em Intel Harpv2 em [102]. No entanto, o tamanho máximo do conjunto de dados avaliado é de 64k pontos, e o número de centroides (k) varia de 256 a 1024, o que não é realista para a maioria das aplicações *K-means*.

Dias *et al*.[32] propõem uma implementação especializada de FPGA totalmente paralela, visando o FPGA Virtex-6. Os autores apresentam apenas resultados de simulação. Paulino *et al*.[86] apresentam 10 implementações OpenCL FPGA de alto nível para explorar diferentes técnicas de codificação. Ao contrário de trabalhos anteriores, os autores apresentam resultados em tempo de execução e validam o projeto em uma placa PCI-Express Alpha Data com um FPGA Kintex-6. Maciel *et al*. [73] propõem uma arquitetura reconfigurável para *K-means* e *K-modes* baseada em

FPGA. No entanto, os resultados experimentais apresentados avaliam apenas pequenos conjuntos de dados, com no máximo 4096 pontos, devido ao tamanho das memórias internas do FPGA.

3.2 Geradores de Arquiteturas Reconfiguráveis

Nesta seção, são apresentados os trabalhos relacionados à geração de arquiteturas reconfiguráveis baseadas em CGRA. A Tabela 3.1 compara as ferramentas descritas nos Capítulos 5 e 6 com outras ferramentas encontradas na literatura. Foram selecionadas apenas as características arquiteturais dos geradores de cada ferramenta, pois a parte de compilação de aplicações em alto nível não está no escopo desta tese. Deste modo, as seguintes características foram analisadas para cada ferramenta: Interconexão, Roteamento, Filas Elásticas, Operadores Personalizados, Código Intermediário, Exportação para RTL e Implantação.

A coluna "Interconexão"compara se os geradores avaliados permitem ao projetista especificar a forma de conexão entre os elementos de processamento de uma arquitetura. Neste caso, as ferramentas que geram arquiteturas personalizadas conforme a aplicação foram avaliadas como "sim", enquanto os geradores que possuem uma estrutura fixa de conexão dos elementos de processamento foram avaliados como "não".

A coluna "Roteamento"compara se os geradores avaliados permitem ao projetista especificar o roteamento interno de um elemento de processamento. Ou seja, dado que um elemento de processamento possua conexão com outros elementos de processamento, é possível realizar o roteamento entre os elementos vizinhos, sem utilizar a ALU, apenas repassando o dado de um vizinho para outro.

A coluna "Filas elásticas"compara se o projetista pode especificar a adição de filas de balanceamento nas entradas das ALU dos elementos de processamento e qual seus tamanhos máximos. Essas filas são importantes para arquiteturas que possuem rede de interconexão no formato *Mesh*. No entanto, são recursos caros. A possibilidade do projetista escolher o tamanho dessas filas, e em quais portas das ALUs e de quais elementos de processamento utilizar, permite a criação de arquiteturas heterogêneas que consomem menos recursos de *hardware*.

A coluna "Operadores personalizados"compara se as ferramentas permitem ao projetista desenvolver uma nova operação para o conjunto de instruções de uma ALU e incorporar essa operação nas ALUs dos elementos de processamento. Essa operação pode ser simples ou mais complexa, formada por outras operações básicas.

A coluna "Código Intermediário"compara se as ferramentas possuem um código intermediário para configurar as arquiteturas geradas. Isso permite que outras ferramentas de compilação possam compilar aplicações e gerar as configurações para

a execução na arquitetura projetada.

A coluna "Exportação para RTL"compara quais ferramentas geram código em linguagens que podem ser sintetizadas para FPGA ou para processo em silício, ou se apenas utilizam estruturas em alto nível para simulação.

Por fim, a coluna "Implantação"compara quais ferramentas geram todo o arcabouço de *hardware* e *software* para a implantação da solução do acelerador projetado em plataformas que possuem FPGAs.

Tabela 3.1: Características das Ferramentas.

Ferramentas	**Características**						
	Interconexão	**Roteamento**	**Filas elásticas**	**Operadores personalizados**	**Código intermediário**	**Exportação para RTL**	**Implantação**
CGRA-ME	sim	sim	não	não	não	sim	não
OpenCGRA	sim	sim	-	sim	não	sim	não
Pillars	sim	sim	sim	sim	não	não	não
FastCGRA	sim	sim	-	não	não	não	não
THRAM/TRAM	não	não	sim	não	não	sim	não
HETA	não	não	não	não	não	sim	não
READY	não	não	-	não	não	sim	sim
HPCGRA	sim	sim	sim	sim	sim	sim	sim

A ferramenta CGRA-ME [22, 21, 7] usa uma API em C++ ou um formato XML para descrever arquiteturas CGRAs. Para isso, foram criados diversos padrões, criando uma linguagem específica, com várias *TAGs* que permitem descrever detalhadamente o funcionamento de todos os componentes de um CGRA. Com isso, é possível montar diferentes caminhos de dados de PEs, dando mais flexibilidade ao projetista. A ferramenta HPCGRA simplifica a descrição criando componentes com maior granularidade. Onde um caminho de dados de um PE é fixo, assim, o usuário se concentra apenas em descrever a arquitetura do CGRA de maneira global sem se preocupar com detalhes do PE. Essa abordagem, embora menos flexível, foi pensada para que um PE possua sempre o mesmo caminho crítico, favorecendo a síntese da arquitetura para o FPGA.

OpenCGRA [106] é uma ferramenta para modelagem, teste e avaliação de arquiteturas de CGRAs. Todo o processo de desenvolvimento de um CGRA é feito por meio da linguagem Python, onde o projetista consegue programar a descrição de todos os módulos. Todo o processo de construção conta com suítes de testes bem definidos, de forma que a arquitetura final seja totalmente testada. Diferentemente do HPCGRA, que possui o foco na construção de CGRAs para serem implantados como aceleradores em sistemas CPU-FPGA, o OpenCGRA é pensado para processo de silício, permitindo que *scripts* realizem a síntese do CGRA em *chip* e gerem relatórios com resultados de área e frequência máxima de operação.

Pillars [46] é uma ferramenta para geração de CGRAs de forma hierárquica, onde o projetista pode descrever os componentes da arquitetura imperativamente na linguagem Scala [96]. Embora o projetista possa desenvolver todo o caminho de

dados de cada módulo de um CGRA, ele precisa utilizar elementos básicos de hardware como ALU, Unidade de entrada e saída, banco de registradores e outros para compor o circuito. O HPGCRA, por outro lado, possui uma abordagem mais focada na interconexão de cada PE e suas operações. Desse modo, a descrição de uma arquitetura de CGRA no HPGCRA é direta e simples, enquanto na ferramenta Pillars é mais detalhada e complexa.

FastCGRA [118] é uma plataforma para modelagem, avaliação e exploração de CGRAs. Os CGRAs podem ser modelados de forma hierárquica, permitindo a construção de CGRAs com múltiplos núcleos que possuem operações diferentes. A interconexão entre as unidades funcionais e entre os núcleos também é personalizada conforme a descrição criada pelo usuário. FastCGRA propõe algoritmos para mapeamento e roteamento de GFDs para os modelos de CGRAs. Além de realizar o mapeamento e o roteamento, esses algoritmos conseguem otimizar a arquitetura do CGRA para comportar melhor esses GFDs, criando modelos de CGRAs otimizados. A modelagem da arquitetura é feita utilizando um dicionário na linguagem Python e pode ser exportada ou importada de arquivos JSON, ou XML. Os GFDs são descritos por meio de uma API em Python. Os principais diferenciais do FastCGRA são permitir que os usuários criem arquiteturas personalizadas de forma hierárquica e que os algoritmos de mapeamento e roteamento particionem os GFDs, dividindo o problema para obter melhores resultados. Por outro lado, o FastCGRA não permite exportar uma arquitetura para uma linguagem de descrição de hardware para que ela possa ser sintetizada em uma plataforma com FPGA ou para um ASIC. Dessa forma, ao contrário das ferramentas READY e HPCGRA, o FastCGRA funciona apenas como uma forma de avaliação de arquiteturas de CGRAs, sem a possibilidade de usar essa arquitetura como um acelerador de hardware em uma plataforma com FPGA de forma automática.

THRAM [64] é uma ferramenta para modelagem de arquiteturas reconfiguráveis baseadas em CGRAs. Este trabalho é uma extensão da ferramenta TRAM [91], onde os autores adicionaram melhorias na abordagem inicial para permitir arquiteturas de CGRA heterogêneas. THRAM possui uma arquitetura de CGRA *template*, onde, por meio de um arquivo JSON, é possível modificar alguns parâmetros da arquitetura. No entanto, ao contrário do HPCGRA, o modelo de interconexão e roteamento dos CGRAs são fixos, cabendo ao projetista alterar apenas a quantidade de PEs e operações das ALUs. As operações das ALUs são pré-determinadas, ou seja, existe uma biblioteca fixa de operadores que o projetista pode escolher adicionar nas ALUs, enquanto o HPCGRA, além de uma biblioteca de operadores básicos, permite a construção de operadores personalizados.

HETA [28] é uma ferramenta para modelagem e exploração do espaço de projetos de CGRAs heterogêneos. A arquitetura base gerada pela ferramenta é uma CGRA

temporal com uma rede de interconexão no modelo ilha, semelhante ao modelo adotado em arquiteturas de FPGAs. A ferramenta permite a especificação de vários parâmetros da arquitetura; no entanto, ao contrário do HPCGRA, não é possível especificar de maneira genérica a rede de interconexão e as possibilidades de roteamento de cada PE. Embora o HETA seja uma ferramenta bastante completa, o foco das arquiteturas geradas é para implantar diretamente em um *chip* e não como aceleradores em FPGA.

Outros trabalhos adotam uma abordagem diferente, gerando arquiteturas específicas para uma ou mais aplicações, como as ferramentas DSAGEN [113, 70]. Essas ferramentas recebem códigos de kernels na linguagem C, anotados com *pragmas*, e geram uma arquitetura espacial específica. A estrutura resultante pode minimizar o comprometimento entre desempenho, eficiência energética e área. Para isso, o compilador possui um conjunto de otimizações, como desacoplar a memória da computação, para representar o código ideal na forma de um grafo de fluxo de dados. O trabalho gera a arquitetura por meio de pequenos blocos que, juntos, conseguem formar qualquer arquitetura funcional. Esses blocos são PEs, Comutadores, Memórias, FIFOs e controladores, que permitem a construção de um ADG (*Architecture Dataflow Graph*), ou Grafo de Fluxo de Dados da Arquitetura. Para um conjunto de *kernels*, um ADG é otimizado iterativamente, adicionando e removendo componentes aleatoriamente, sendo avaliado em cada iteração usando uma função objetivo. Essa função avalia o desempenho por milímetro quadrado até atingir um valor alvo, ou até que o ADG permaneça estável. Na mesma linha, o trabalho [57] apresenta um gerador de Arranjos Sistólicos para realizar computação vetorial. A ferramenta utiliza um compilador [43] como *frontend* e a infraestrutura Chisel [9] para gerar o código Verilog da arquitetura. O HPCGRA também permite a geração de arquiteturas específicas, como mostrado no estudo de caso da Seção 6.7, além de permitir a geração de arquiteturas genéricas.

Capítulo 4

Projeto de Arquiteturas de Aceleradores de Domínio Específico

Com a adesão de placas de FPGAs em ambientes de alto desempenho, tornam-se necessárias metodologias que visem desenvolver DSA (*Domain Specific Accelerator*), ou Aceleradores de Domínio Específico, que executem tarefas em *hardware* reconfigurável, a fim de alavancar o uso desse novo paradigma de projeto de aceleradores.

Este capítulo apresenta um estudo inicial acerca do problema da síntese de aceleradores em FPGA, e tem como foco em entender os desafios da área, e poder assim propor novas soluções. A Figura 4.1 apresenta uma visão geral das ferramentas desenvolvidas neste capítulo, onde as aplicações são modeladas diretamente para o FPGA. Neste contexto, dois estudos de caso foram desenvolvidos visando demonstrar o uso de FPGAs como aceleradores em sistemas heterogêneos.

O primeiro estudo, descrito na Seção 4.1, apresenta um gerador de aceleradores para realizar a busca de estados de estabilização em redes reguladoras de genes. Este problema foi escolhido como estudo de caso devido à rede reguladora ser modelada por meio de equações lógicas, ideais para modelagem em FPGA, por possuir um paralelismo a nível de *bit*.

O segundo estudo de caso, descrito na Seção 4.2.1, apresenta um gerador de aceleradores para otimizar o algoritmo de agrupamento *K-means*. Este algoritmo foi escolhido como estudo de caso, por permitir uma implementação no formato de grafo de fluxo de dados, que possui um paralelismo espacial e temporal ideal para implementações em *hardware*.

Ambos os estudos de caso são algoritmos sintetizados diretamente em FPGA, ou seja, os parâmetros de entrada dos algoritmos são fixos. Por exemplo, no primeiro estudo de caso, os parâmetros de entrada são as equações do modelo da rede reguladora de genes. Já no segundo estudo de caso, os parâmetros de entrada são as dimensões dos dados de entrada e o número de grupos do *K-means*.

A abordagem de síntese direta adotada nestes dois estudos de caso possui vantagens e desvantagens. Por um lado, essa abordagem resulta em aceleradores

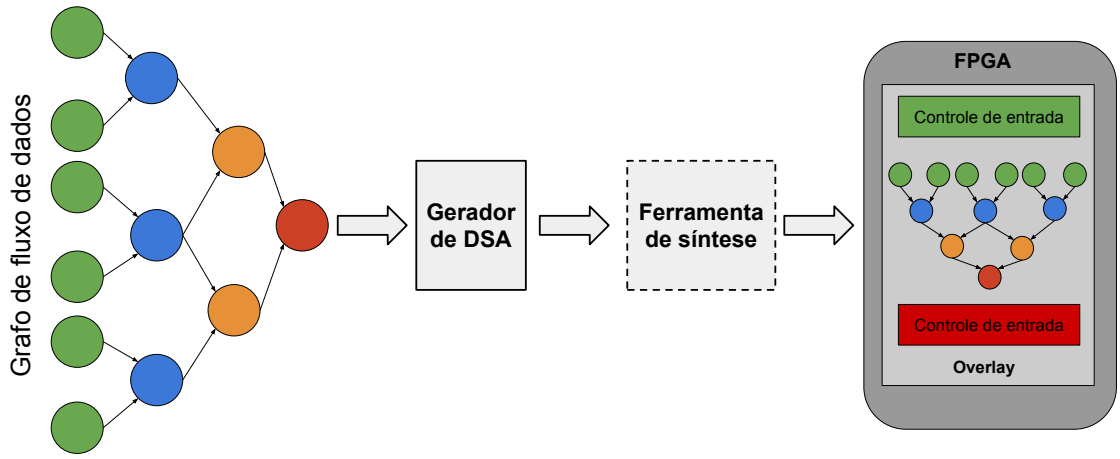


Figura 4.1: Visão Geral dos geradores de aceleradores de domínio específico.

mais eficientes, uma vez que o algoritmo é descrito diretamente em *hardware*, tendendo a possuir melhor desempenho, já que nenhuma camada de abstração é necessária. Isso faz com que funcione como um *baseline* para as propostas que se seguirão. Por outro lado, o projeto do acelerador é complexo, trabalhoso e demorado, uma vez que todo o desenvolvimento foi majoritariamente manual. No entanto, como mostrado nos capítulos seguintes, as lições aprendidas com a estratégia de síntese direta aqui utilizada foram de grande valia no desenvolvimento da metodologia proposta nesta tese.

Este capítulo está organizado da seguinte forma: a Seção 4.1 descreve o projeto do acelerador para redes reguladoras de genes, introduzido pelo autor em [13]. A Seção 4.2.1 apresenta o projeto do acelerador para *K-means*, publicado pelo autor em [14]. Por último, a Seção 4.3 discute os resultados e as lições aprendidas com esses dois estudos de caso, indicando como serão úteis para o desenvolvimento dos próximos capítulos.

4.1 Acelerando buscas de estados de estabilização em Redes Reguladores de Genes

Esta seção apresenta um gerador de aceleradores para realizar a busca de estados de estabilização em redes reguladoras de genes. Este problema foi selecionado por apresentar paralelismo a nível de *bit* ideal para o FPGA devido a sua granularidade fina.

O acelerador proposto resolve um problema específico de determinar os estados de estabilização ou atratores das redes reguladoras. Este problema é NP-difícil como mostrado por Tatsuya Akutsu e colaboradores em [3]. Trabalhos anteriores propõem

estratégias baseadas em BDD [42], SAT usando CPUs [34] e multi-cores [45]. Além disso, também existem abordagens paralelas usando GPUs [78, 11] e aceleradores de *hardware* baseados em FPGA [75, 40]. Redes booleanas podem ser mapeadas eficientemente em *hardware* usando FPGA como já demonstrado em trabalhos anteriores [75, 17]. No entanto, além dos desafios complexos enfrentados pelos programadores que desenvolvem para FPGA, há também um obstáculo adicional para criar uma ferramenta de alto nível que seja acessível para profissionais de outras áreas, como os biólogos que estudam esses modelos.

Embora seja um gerador específico para um nicho de aplicação, sua implementação contribuiu com ideias e conhecimentos que auxiliaram no desenvolvimento da metodologia proposta nesta tese. Um primeiro aspecto decorrente dessa implementação foi a forma de distribuição e identificação de tarefas para elementos de processamento, assim como a coleta dos resultados. Outro aspecto importante foi o comportamento assíncrono das tarefas, onde cada uma tem um tempo de computação diferente, equalizado com o uso de filas para entrega dos resultados na interface com a CPU. Esse paralelismo irregular em tempo também é vantajoso para o FPGA em relação à solução em GPU. Além disso, cada elemento de processamento possui um arcabouço que encapsula o módulo da rede reguladora, específico para cada rede simulada. Os elementos de processamento também foram agrupados para otimizar a comunicação com a interface da CPU. O gerador cria automaticamente todo o *hardware/software* a partir de um conjunto de equações booleanas que descrevem o modelo da rede reguladora de genes em alto nível.

4.1.1 Redes Reguladoras de Genes

Os genes formam sistemas dinâmicos complexos chamados GRNs (*Gene Regulatory Networks*), ou Redes Reguladoras de Genes. Na Biologia, esses modelos evolutivos auxiliam no entendimento da origem e na dinâmica evolutiva de traços fenotípicos [19], onde as interações entre genes são predominantes e críticas durante todo o processo. Em um modelo de GRN, cada vértice possui um estado que pode estar ativado ou desativado. Os estados de todos os vértices formam o estado da rede, que varia conforme a simulação é executada. A GRN pode possuir alguns estados de estabilização, que podem ser um conjunto de estados que formam um ciclo durante a simulação ou apenas um estado fixo, chamados de atratores. Atratores são importantes para o entendimento desses modelos, usados por biólogos para identificar e entender mecanismos causadores de doenças e desenvolver fármacos e terapias [19].

Os modelos de GRNs são abstrações de múltiplas interações entre os genes. É possível representar uma GRN usando um grafo booleano, onde os vértices são os

genes e as arestas são as interações entre os genes. Os modelos GRN são discretos, e evidências experimentais sugerem que a expressão de ativação de um gene é digital e estocástica no nível individual de uma célula, em vez de contínua [19]. Uma operação essencial em modelos GRN é calcular os estados de estabilização ou atratores, tendo uma implicação prática: um tipo de célula pode corresponder a um atrator [45]. No entanto, o número de estados que uma GRN pode possuir é exponencial em função do número de genes.

Uma GRN é modelada por meio de um grafo boleano $G(V, E, F)$, onde V é conjunto de genes $\{g_1, g_2, ..., g_n\}$, E é o conjunto de arestas que representa as interações entre os genes, F é o conjunto de funções de atualização de cada gene. As funções do conjunto F são utilizados para realizar o cálculo booleano para definir o próximo estado de um gene. Durante a simulação de um modelo de GRN é possível realizar o cálculo do próximo estado de forma síncrona, assíncrona e probabilística. A Figura 4.2 apresenta um exemplo de implementação em *hardware* de cada tipo de atualização para um modelo de GRN mostrado na Figura 4.2(a). No modelo de atualização síncrona todos estados são atualizados em simultâneo, usando os estados atuais para calcular o próximo estado como mostrado na Figura 4.2(b). No modelo assíncrono mostrado na Figura 4.2(c) cada gene é atualizado em uma ordem específica e o novo estado de um gene usa o estado mais atual dos demais genes para calcular seu próximo estado. Finalmente, a Figura 4.2(d) ilustra o modo probabilístico onde cada gene tem uma ou mais regras de atualização probabilística. Neste exemplo $v1$ tem 40% de probabilidade de ser atualizado por $v1\&v2$ e 60% de probabilidade por $v1||v2$. Todos os três modelos têm um mapeamento direto para uma implementação de *hardware* em FPGA de forma otimizada, onde granularidade de *bit* é utilizada armazenar cada estado de um gene. Neste estudo de caso delimitamos nosso escopo em desenvolver o acelerador para realizar a atualização dos estados de forma síncrona.

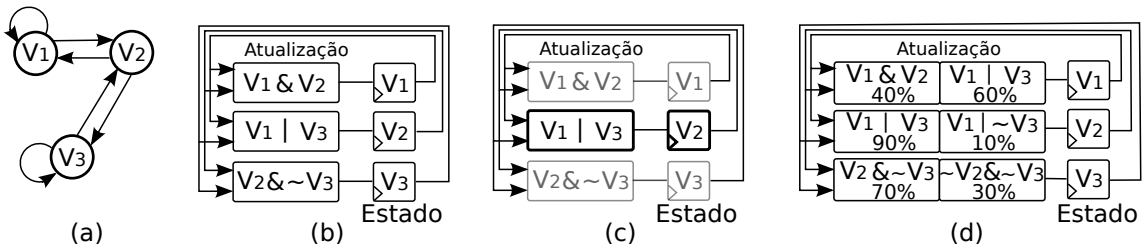


Figura 4.2: Diferentes tipos de atualização de estado em modelos de GRN em *hardware*.

A Figura 4.3(a) apresenta um modelo de GRN simples com apenas dois genes nomeados de G_a e G_b. Os estados dos genes G_a e G_b são atualizados pelas equações 4.1 e 4.2.

$$G_a(t+1) = f_a(t) = G_b(t) \tag{4.1}$$

$$G_b(t+1) = f_b(t) = G_a(t) \& G_b(t) \tag{4.2}$$

Assumindo um estado inicial $G_a = 0$ e $G_b = 0$, como mostrado na Figura 4.3(b). A simulação calcula a evolução dos estados dos genes até que um estado de estabilização ou um ciclo de estados é encontrado. O estado $G_a = 1$ e $G_b = 0$ é um atrator da GRN. Todos os estados possíveis de um modelo devem ser visitados para calcular o diagrama completo dos estados que modelam a dinâmica da GRN. A Figura 4.3(c) mostra o diagrama de caso, para este exemplo, onde existem 4 estados ou $2^{|V|} = 2^2$, onde $|V|$ é a quantidade de genes do modelo de GRN. O exemplo da Figura 4.3 possui um único estado atrator e todos os outros estados convergem para este atrator, isso significa que independente do estado inicial da GRN, após alguns passos de simulação a GRN estabiliza em um atrator e ocorrem mudanças de estados cíclicas neste atrator. Em geral, um modelo de GRN tem pelo menos um atrator com pelo menos um estado. Qualquer mudança nas regras de atualização de um modelo de GRN pode afetar a dinâmica nas trocas de estados durante a simulação. A Figura 4.3(d) mostra um simples exemplo de mudança na função de atualização do gene G_b, onde $G_b(t+1) = G_a(t)$, neste caso, o modelo passa a possuir dois atratores $A1$ e $A2$ e cada atrator possui dois estados.

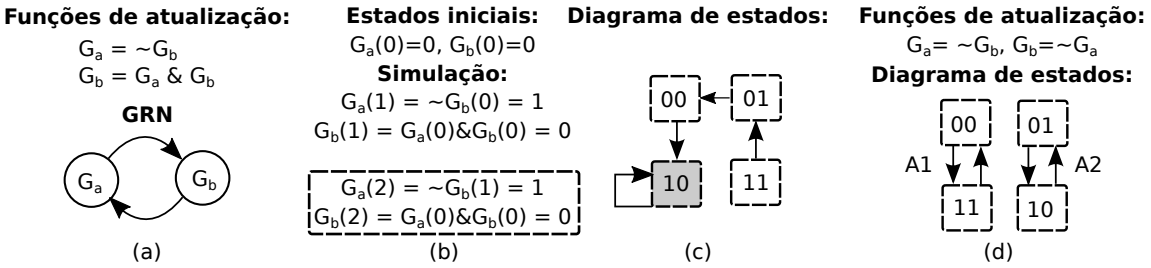


Figura 4.3: (a) Exemplo de um modelo de GRN; (b) Simulação do modelo de GRN; (c) Diagrama de estados do modelo de GRN; (d) Diagrama de estados para a mesma GRN, porém outra função de atualização para o gene G_b.

4.1.2 Gerador e Arquitetura do Acelerador

Esta seção descreve o gerador de código desenvolvido para criar todo arcabouço de *software* e *hardware* necessário para realizar a implantação do acelerador para encontrar estados de estabilização de modelos GRNs. Este arcabouço é gerado na forma de um projeto para a ferramenta Vitis [116], e possui um conjunto de *scripts* para realizar a compilação da parte de *software* e a síntese da parte de *hardware*. O projeto é configurado para gerar um *bitstream* para plataforma de computação

acelerada com FPGA da AWS descrita na Seção 2.1.1 do Capítulo 2.

Para flexibilizar o projeto e dar portabilidade na criação de aceleradores de diferentes modelos de GRN um gerador de código RTL foi desenvolvido. Este gerador é endereçado aos usuários que não possuem conhecimento na área de desenvolvimento de *software* ou possuem muito pouco como, por exemplo, biólogos.

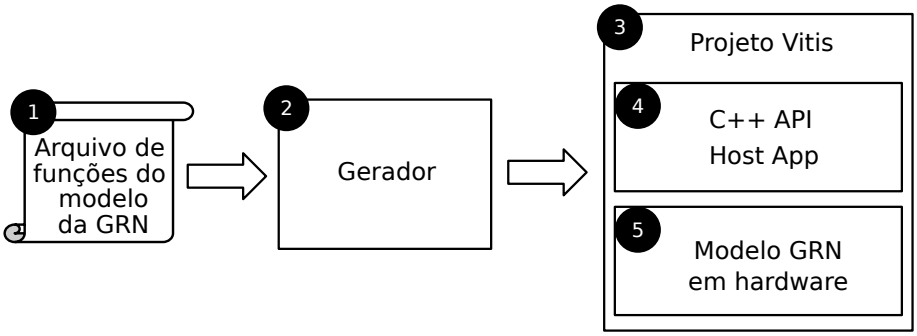


Figura 4.4: Diagrama do arcabouço de *software* do gerador.

A Figura 4.4 apresenta um diagrama do trabalho. No passo, ❶ um arquivo com as funções de atualizações de um modelo de GRN é dado como entrada. Neste arquivo cada linha representa uma função de atualização de um gene. Esta linha deve conter um nome de gene seguido de um sinal de igualdade e depois as operações booleanas envolvendo os demais genes. A Figura 4.5 possui exemplo de GRN com 5 genes, onde as interações entre os genes são realizadas por operações lógicas.

```
1 CtrA = (CtrA or GcrA) and (not CcrM) and (not SciP)
2 GcrA = DnaA and not CtrA
3 SciP = CtrA and not DnaA
4 DnaA = CtrA and CcrM and (not GcrA) and (not DnaA)
5 CcrM = CtrA and (not CcrM) and (not SciP )
```

Figura 4.5: Exemplo de especificação de modelo de GRN.

A partir dessa descrição GRN de alto nível, na etapa ❷ o gerador cria o projeto com os códigos C++ front-end para executar no processador *host* e todo o código RTL para construir o acelerador em FPGA. Foi utilizada a linguagem Python e o pacote Veriloggen [105] para implementar um gerador flexível do RTL. Esta ferramenta permite descrever um código estrutural em Verilog utilizando a facilidade do Python em manipular diversas estruturas de dados. Desse modo é possível criar um acelerador específico para cada modelo de GRN mudando apenas o arquivo com as funções lógicas.

Na etapa ❸, para facilitar na implantação do acelerador em um ambiente com FPGA, o gerador emite como saída um projeto completo para ser utilizado com a ferramenta Vitis da AMD Xilinx. Neste projeto, além dos arquivos de código-fonte

do acelerador e da aplicação que irá executar na CPU, um conjunto de *scripts* para configuração também é gerado. Estes *scripts* conseguem realizar toda a compilação do projeto, incluindo a síntese para o FPGA e o código da CPU nas etapas ❹ e ❺. Desse modo, para utilizar a ferramenta o usuário precisa apenas iniciar uma instância com FPGA na nuvem da AWS e executar os *scripts* para que todo o projeto seja construído.

A Figura 4.6(a) representa a estrutura da arquitetura do acelerador. O código Verilog gerado inclui a interface AWS F1, um esquema de programação de *software/hardware* além do conjunto de elementos de processamento. O módulo acelerador tem três componentes principais: leitor, árbitro e escritor. O gerador suporta vários parâmetros, onde o usuário pode especificar o número de cópias GRN ❼. Para cada cópia, é possível criar uma unidade de Elemento de Processamento (PE) inteiro. O número máximo de cópias depende do tamanho do conjunto de funções de atualização do modelo de GRN, da profundidade dos FIFOs de entrada/saída e das ferramentas da AMD Xilinx que realizam o posicionamento e roteamento no FPGA. As bibliotecas da AWS geram automaticamente as interfaces AXI, impactando diretamente a área ocupada e a vazão. O gerador cria conjuntos de PEs para compartilhar melhor os recursos de memória e superar essas restrições. Cada conjunto e PEs é conectado a um canal de comunicação direto com a memória DDR. Por exemplo, suponha que seja possível implementar 128 cópias do acelerador GRN e o tamanho do conjunto, seja 32 o gerador cria automaticamente, quatro conjuntos, conforme mostrado na Figura 4.6(a), onde cada conjunto possui uma interface AXI e 32 PEs.

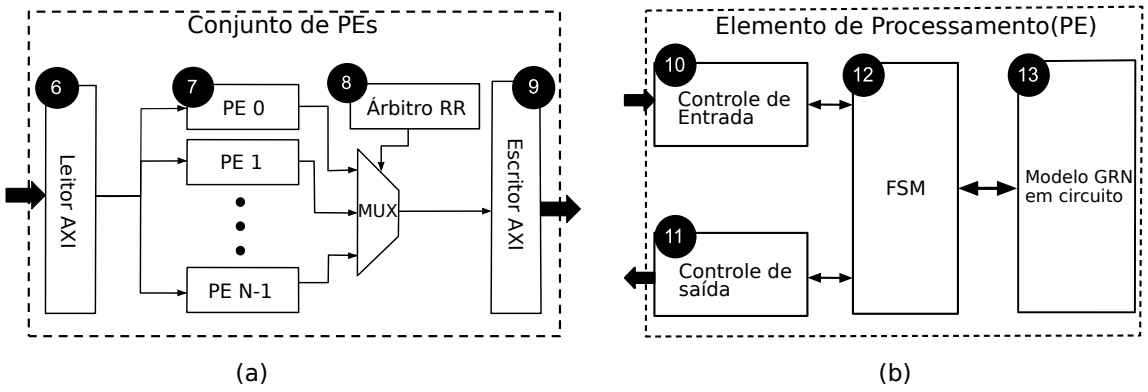


Figura 4.6: Arquitetura do acelerador. (a) Conjunto de PEs que compartilham mesmo canal de comunicação de dados. (b) Arquitetura interna de um elemento de processamento do acelerador.

O conjunto de PEs funciona da seguinte maneira: primeiro, o Leitor AXI ❻ recebe um conjunto de pacotes de configurações. Cada PE é identificado por meio de um ID e possui uma fila de tarefas. Propomos um escalonador distribuído de *software* e *hardware* onde o *host* e o acelerador exploram os trade-offs para equilibrar

melhor a carga de trabalho de computação entre os PEs. O tamanho da configuração é conforme o tamanho do modelo da GRN e pacote de configuração possui três campos: estado inicial, estado final e ID. O controle de entrada de cada PE verifica se o ID da configuração corresponde ao ID do PE e enfileira a configuração na fila de tarefas. Suponha um modelo de GRN com 70 genes. O tamanho do campo de estado inicial e final é um múltiplo de 8 *bits*. Para este exemplo, cada estado requer um pacote de 9 *bytes* (ou 72 *bits*). A interface do leitor é portátil, parametrizada e flexível para suportar outras interfaces com memória externa. Além disso, como já mencionado, o conjunto de PEs maximiza a comunicação e reduz os recursos necessários para implementar a interface AXI. Embora a AWS forneça uma biblioteca genérica para instanciar o maior número possível de interfaces AXI, a instância AWS F1 FPGA tem quatro canais físicos AXI4, onde cada canal tem uma largura de 512 *bits* (64 *bytes*). Desse modo, utilizando um conjunto de PEs para cada canal de entrada e saída, o uso dos canais é otimizado obtendo um melhor consumo de recursos.

Cada conjunto de PEs possui um árbitro ❽ que executa uma política de *round-robin* para enviar os resultados de volta à CPU do *host* por meio do módulo escritor. O número de ciclos para calcular um atrator depende da dinâmica do modelo de GRN e do estado inicial. Portanto, a carga de trabalho do PE é heterogênea. Além disso, um PE pode receber várias tarefas para executar. Por exemplo, o *host* envia o pacote (0,9,3), onde 0 é o primeiro estado, 9 é o último estado e 3 é o ID do PE. Portanto, o PE3 calcula os atratores para os estados 0, 1, 2, . . ., 9 e envia de volta 10 pacotes de saída. Cada pacote de saída tem o ID do estado do atrator, o número de ciclos para atingir esse atrator (transiente) e o tamanho do atrator. Deste modo, o *software host* ajusta dinamicamente a carga de trabalho do PE para um determinado GRN sem criar e instanciar um novo acelerador de *hardware*. Quando o PE termina o cálculo, a máquina de estados para encontrar o atrator insere o resultado na fila de saída, que espera até o próximo intervalo de tempo do escalonador *round-robin* para enviá-lo de volta à CPU. Finalmente, o gravador AXI ❾ manipula o pacote de saída para enviar de volta à CPU *host*.

A Figura 4.6(b) mostra uma visão geral da estrutura interna do PE. Primeiro, o controle de entrada ❿ e o controle de saída ⓫ implementam uma interface de comunicação com o PE assíncrona para mitigar a carga de trabalho desequilibrada. Um PE pode receber mais de um pacote de configuração até que sua fila de entrada esteja cheia. Para cada configuração, o PE calcula o atrator usando o Algoritmo 1 [10] implementado na forma de máquina de estados ⓬. Este algoritmo é ideal para execução em FPGA, uma vez que ele não utiliza memória para encontrar ciclos em diagramas de estados. Em ⓭ um modelo de GRN é implementada em forma circuito, onde a cada ciclo de relógio um próximo estado é calculado conforme as

entradas dos genes. Esse módulo é controlado pela máquina de estados ⓬ onde o cálculo do atrator é realizado.

Algoritmo 1: Algoritmo para busca de atratores em um conjunto de estados[10].

Entrada: E_i = Estado Inicial E_f = Estado Final
Saída: A = Lista de estados atratores T = Lista de transientes L = Lista de comprimento dos atratores

```
início
    para i = E_i; i < E_f; i += 1 faça
        S_0 = E_i;
        S_1 = S_0;
        t = 0;l = 0;
        repita
            PróximoEstado(S_0);
            PróximoEstado(S_1);
            PróximoEstado(S_1);
            t++;
        até S_0 != S_1;
        repita
            ProximoEstado(S_0);
            l++;
        até S_0 != S_1;
        Insere(A,S_0);
        Insere(T,t);
        Insere(L,l);
    fim
fim
```

4.1.3 Biblioteca para utilização do acelerador

Embora a API do OpenCL da AMD Xilinx fornece métodos para realizar a chamada de acelerados genéricos, a lógica de cada acelerador deve ser desenvolvida para fornecer uma interface simples para o usuário final. Para isso, uma biblioteca em C++ foi desenvolvida para fornecer rotinas de chamadas para aceleradores para qualquer modelo de GRN. Desse modo, o usuário precisa apenas informar em um arquivo de entrada o conjunto de estados iniciais e finais e após a execução um arquivo de saída no formato CSV é gerado com todos os dados calculados pelo acelerador. Isso tudo é realizado em poucas linhas de código como mostrado na

Figura 4.7. As linhas de 7 a 10 são apenas a atribuição dos dados para as variáveis usadas como parâmetro na instanciação do objeto da classe GRN na linha 11. Os parâmetros binaryFile e kernel_name são utilizados pela API do OpenCL para configurar o FPGA com o *bitstream* do acelerador. Os parâmetros grn_cfg e grn_outputfile são os arquivos de entrada e saída respectivamente. Na linha 12 é realizada a chamada do acelerador e nas linhas 13 e 14 são escritos os arquivos de saídas com os estados atratores e resultados de desempenho do acelerador.

```
1  #include <host.h>
2  int main(int argc, char *argv[]){
3      if (argc != 5) {
4          std::cout << "Usage: " << argv[0] << " <XCLBIN File> <
               kernel name> <GRN Configuration file> <Output file
               name>" << std::endl;
5          return EXIT_FAILURE;
6      }
7      std::string binaryFile = argv[1];
8      std::string kernel_name = argv[2];
9      std::string grn_cfg = argv[3];
10     std::string outputfile = argv[4];
11     auto grn = Grn(binaryFile, kernel_name, grn_cfg, outputfile);
12     grn.run();
13     grn.saveGrnReport();
14     grn.savePerfReport();
15     return 0;
16 }
```

Figura 4.7: Código C++ de chamada de um acelerador para um modelo de GNR genérica.

4.1.4 Resultados Experimentais

Esta seção apresenta os experimentos realizados para avaliar o acelerador e comparar os resultados com outras plataformas disponíveis na nuvem da AWS. Para isso, três abordagens foram implementadas: (a) uma para CPU sequencial, (b) uma para CPU multi-core utilizando OpenMP e outra (c) GPU utilizando CUDA. O objetivo é comparar o desempenho de arquiteturas de nuvem heterogêneas. Além disso, também foram avaliados o custo para alocar essas arquiteturas nos serviços de nuvem da Amazon [50]. Atualmente há uma grande variedade de dispositivos na AWS, portanto, selecionamos instâncias de baixo custo e alto desempenho para cada plataforma.

A Tabela 4.1 mostra os principais recursos de cada instância da AWS usadas para o experimento. Para FPGA, foi selecionada a instância ***f1.2xlarge***, equipada com 8 vCPUs compostas por processadores Intel Xeon E5-2686 v4 series (Broadwell)

rodando a uma frequência de até 2,2GHz, FPGAs AMD Xilinx VU9P e 122 GB de RAM [50]. Para a plataforma com CPUs multi-core, selecionamos a instância ***c5a.16xlarge***, ideal para algoritmos que requerem processadores de alto desempenho. Essa instância possui 64 vCPUs compostas por processadores AMD EPYC 7002 de segunda geração executados a uma frequência de até 3,3GHz e 128 GB de RAM [50]. Uma versão paralela do Algoritmo 1 foi implementada usando a biblioteca OpenMP para explorar recursos de *multithreading*. Para plataformas com GPU, foram avaliadas três opções da AWS para explorar as compensações custo/desempenho. Como nosso aplicativo não requer recursos específicos, como núcleos tensores nem alta taxa de transferência de memória, é interessante comparar o custo relativo e as compensações de desempenho de uma GPU V100 de 7^a geração, GPU T4 de 8^a geração de baixo consumo e uma GPU K80 de 3^a geração. A instância de menor custo da AWS com GPU é a ***g4dn.xlarge***. Por outro lado, a instância de GPU de maior custo é a ***p3.2xlarge*** com uma GPU V100, 8 vCPUs Xeon E5-2686 v4 de 2,2GHz de frequência e 62 GB de memória RAM.

Tabela 4.1: Propriedades das instâncias da AWS. A coluna preço sobre demanda equivale a uma hora de uso da instância. Os valores foram cotados no ano de 2020.

Nome	Instância	CPU	Memória (GB)	Núcleos	Acelerador	Preço sobre demanda ($)
T4G	t4g.medium	AWS Graviton2	4	2	-	0,0336
T4 GPU	g4dn.xlarge	Xeon Scalable v2	16	4	NVIDIA T4	0,526
K80 GPU	p2.xlarge	Xeon E5-2686 v4	61	4	NVIDIA K80	0,9
FPGA	**f1.2xlarge**	Xeon E5-2686 v4	122	8	**AMD Xilinx VU9P**	**1,65**
C5A	c5a.16xlarge	AMD EPYC 7R32	128	64	-	2,464
V100 GPU	p3.2xlarge	Xeon E5-2686 v4	61	8	NVIDIA V100	3,06

Os experimentos foram realizados utilizando seis modelos de GRNs encontrados na literatura. (a) B_bronch, uma bactéria respiratória e um helminto gastrointestinal (BTC) [109]; (b) CAC é um modelo para o câncer de cólon associado à colite [72]; (c) EGFR é um modelo para o receptor do fator de crescimento epidérmico [94]; (d) CD4 é um modelo para a célula T efetora imune CD4+ [26]; (e) ERB é um modelo para a transdução de sinal do receptor ErbB em células epiteliais mamárias humanas [52]; e (f) SMA é um modelo para a macrofase do sistema imunológico [93]. A Tabela 4.2 mostra as principais características de cada GRN. Para esses GRNs, o tamanho do espaço de estados varia de 10^{15} a 10^{96}, maior que o número de átomos no universo observável.

A Tabela 4.2 classifica o GRN pelo número de genes. O tempo de execução cresce em função do número N de genes. No entanto, também depende da dinâmica da rede controlada pelas funções de atualização nas interações. Primeiro, se o transiente médio e o tamanho do atrator são menores, o acelerador processa cada estado em alguns ciclos. Em segundo lugar, a complexidade das interações e o tamanho de cada

Tabela 4.2: Propriedades dos modelos de GRNs

Nome completo	Nome	Genes	Interações	Operações
B bronchiseptica and T retortaeformis coinfection	BTC	53	160	139
Colitis-Associated Colon cancer	CAC	70	154	118
EGFR & ErbB signaling	EGFR	104	377	370
CD4 T cell signaling	CD4	188	406	301
ErbB (1-4) Receptor Signaling	ERB	247	1,954	2,242
Signaling in Macrophage Activation	SMA	321	553	254

regra de atualização (coluna **Operações**) também têm impacto direto no tempo de execução, bem como na área do FPGA ocupada pelo circuito GRN. Por exemplo, o modelo ERB possui 247 nós, significativamente menor que o modelo SMA, com 321 nós. No entanto, a ERB possui diversas operações cerca de 8,8 vezes maiores que o modelo SMA, resultando em uso maior de recursos do FPGA e maior tempo de execução em CPUs.

Para realizar a análise de desempenho das três plataformas, foram gerados códigos para os seis modelos de GRNs do mundo real, descritos na Tabela 4.2. Para os modelos BTC, CAC, EGFR e CD4, foram explorados 2^{25} estados, enquanto para os modelos ERB e SMA, foram explorados 2^{24} estados. Como os modelos ERB e SMA possuem um maior número de nós e operações, o número de estados explorados foi reduzido pela metade para diminuir o tempo de execução das análises.

Como o número de estados avaliados difere de acordo com cada rede, os resultados foram normalizados para 2^{24}. O tempo de execução foi medido por meio das funções do *chrono::highresolutionclock* inseridas diretamente no código-fonte da chamada do acelerador em C++.

A Figura 6(a) descreve o tempo de execução em milissegundos para os seis aceleradores avaliados na AWS para cada plataforma descrita na tabela 4.1. Classificamos as instâncias da AWS em ordem crescente para o tempo de execução. O acelerador FPGA apresenta um menor tempo de execução para todos os cenários avaliados. A Figura 6(b) mostra o fator de aceleração médio para a abordagem FPGA em comparação com dispositivos multi-core e GPU.

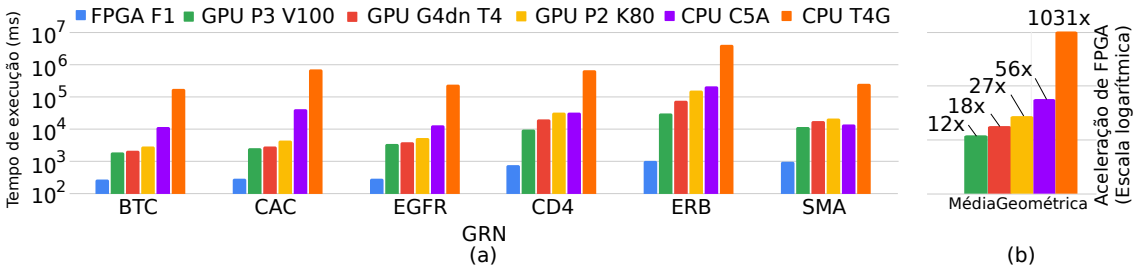


Figura 4.8: (a) Tempo de execução em milissegundos para os GRNs em cada instância da AWS. Menor é melhor; (b) Aceleração média do acelerador em FPGA.

O acelerador FPGA explora o paralelismo espacial em dois níveis. A implementação de *hardware* avalia todas as regras de atualização em paralelo em um único ciclo de *clock*, proporcionando um aumento significativo no paralelismo ao nível de instrução (ILP). Além disso, o gerador cria mais de uma cópia de GRN, aumentando o paralelismo espacial conforme mostrado na Tabela 4.3. Considerando a implementação multi-core, usamos o número máximo de *threads/core*: 2 para as instâncias T4G da AWS e 64 para as instâncias C5A da AWS. Nas placas gráficas, cada *thread* da GPU executa uma computação de um atrator inteira para um determinado estado, explorando o paralelismo SIMT para execução. Embora a frequência operacional da CPU e GPU seja 10x mais rápida que o FPGA, os paralelismos ILP e de tarefas do FPGA superam essa desvantagem para fornecer melhor aceleração. A GPU aproveita o grande número de núcleos para fornecer melhor aceleração do que os dispositivos da CPU. No entanto, cada *thread* da GPU consome inúmeros registradores devido à complexidade das regras de atualização da GRN, reduzindo o número de *threads* de execução nos vários núcleos da GPU. Uma GPU V100 tem 5120 núcleos, sendo cerca de 4,7 vezes mais rápida que uma CPU de 64 núcleos. É importante destacar que é uma comparação simples, pois o custo e o desempenho de um núcleo de CPU não são equivalentes aos núcleos de GPU simples. Em resumo (veja a Figura 4.8(b), o FPGA é 12x, 18x, 27x, 56x e 1031x mais rápido que o GPU V100, GPU T4, GPU K80, CPU de alto desempenho de 64 núcleos, 2-núcleos de CPU de baixo custo, respectivamente.

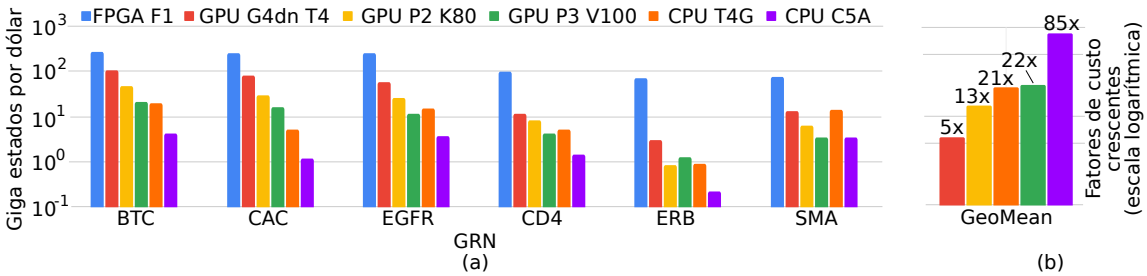


Figura 4.9: Desempenho por dólar: (a) Gigas estados processados por dólar nas seis instâncias avaliadas da AWS. Alto é melhor. (b) Fator de custo crescente, onde menor é melhor.

A computação em nuvem oferece recursos de computação como um serviço, onde vários modelos de preços e tarifas criam novas métricas de custo. Além do tempo de execução, propomos avaliar o desempenho por dólar medindo o número de cálculos do atrator por dólar. A Figura 4.9(a) mostra o número de giga estados processados por dólar para os seis modelos de GRN nas seis instâncias da AWS, onde um número alto significa um melhor desempenho por dólar.

Esses resultados demonstram mais uma vantagem no uso de FPGAs quando comparados a outras abordagens de aceleradores. Embora o custo de um FPGA seja

de US$ 1,65 por hora, o que é 3,1 vezes mais caro que o GPU T4, o FPGA é 18,1 vezes mais rápido que o T4. Em média, o aumento do custo por estado é de aproximadamente 5,7x. Além disso, recursos de GPU são caros como, por exemplo, núcleos tensores da GPU V100, inúteis para computar atratores. Neste caso, mesmo a GPU T4 sendo 1,5x mais lenta que a V100 ela é 6x mais barata. Portanto, é melhor instanciar seis T4 em vez de um V100, que será 4x mais rápido. A Figura 4.9(b) mostra o aumento no fator de custo para calcular o atrator por dólar em relação ao desempenho do FPGA por dólar. Os preços das instâncias da AWS foram relatados na Tabela 4.1. A instância de 64 núcleos tem o pior desempenho por dólar. É 85x pior que o FPGA, enquanto o 2-núcleos é apenas 21x. No entanto, o 2-núcleos é 1031x e 18x mais lento que o FPGA e o 64-core, respectivamente. Portanto, o FPGA alcança o melhor desempenho e o melhor custo por dólar em todos os cenários.

A Tabela 4.3 resume os recursos de FPGA necessários para cada acelerador de *hardware*. Como já mencionado, é possível aumentar o número de cópias do acelerador para redes menores para melhorar o desempenho usando o paralelismo espacial. Por exemplo, os aceleradores BTC, CAC e EGFR podem alocar 128 cópias, organizadas em quatro conjuntos de PEs de 32 cópias cada. Para as redes mais extensas, com mais de 188 nós, as ferramentas de mapeamento e roteamento do FPGA só foram capazes mapear com sucesso 32 cópias. As colunas LUT, REG e LUTasMEM representam o número total de *Lookup tables*, Registradores e Memórias do FPGA para o módulo acelerador. Esses valores não incluem os recursos de interface e as APIs de *hardware* da AWS. A frequência do *clock* foi mantida estável no valor mais alto possível permitido pela plataforma AWS FPGA em 250 MHz representado na última coluna.

Tabela 4.3: Uso de recursos do FPGA

GRN	Cópias	LUT	REG	LUTasMem	Fmax MHz
BTC	128	74,0K(8,3%)	127,2K(6,5%)	18,0K(3,3%)	251
CAC	128	89,2K(9,98%)	153,5K(7,86%)	21,0K(3,83%)	250,38
EGFR	128	110,0K(12,31%)	199,1K(10,20%)	28,2K(5,13%)	251,13
CD4	32	46,4K(4,65%)	82,3K(3,86%)	11,8K(2,05%)	251,13
ERB	32	76,4K(7,68%)	136,3K(6,42%)	19,5K(3,40%)	251,13
SMA	32	69,K(7,01%)	107,5K(5,07%)	15,6K(2,73%)	239,87

Embora a utilização de recursos do FPGA seja relativamente baixa para os circuitos, é importante destacar que os circuitos do controlador de entrada/saída e o SHELL da AWS consomem uma quantidade significativa de recursos do FPGA. Para EGFR, os recursos totais utilizados são 30,6% de LUT, 10,3% de LUTAsRAM e 24,3% de REG. A Figura 5 mostra uma representação gráfica da utilização dos recursos do

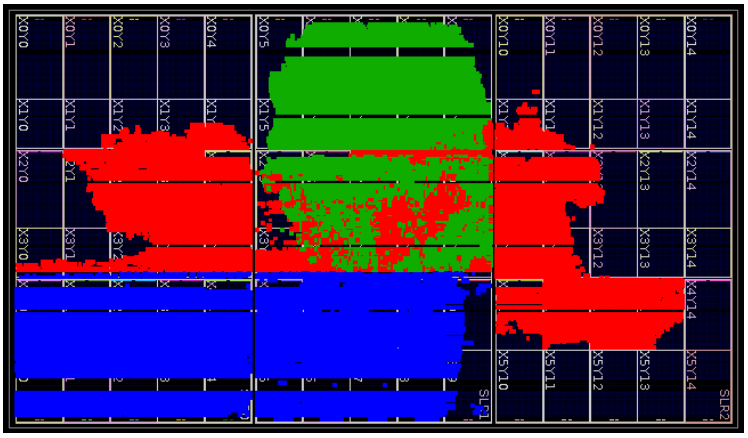


Figura 4.10: Visualização da utilização do acelerador no FPGA para o modelo de GRN EGFR.

FPGA. A área azul representa o AWS SHELL e o acelerador não pode alocar esses quadrantes para colocar a lógica personalizada. A área vermelha são os recursos alocados para entradas/saídas de dados, e apenas a área verde representa o próprio acelerador. O grande desafio em aumentar o número de cópias estava relacionado às limitações internas no FPGA utilizado devido ao número máximo de barramentos adequados para propagação do sinal de *clock*. Apesar das limitações, o acelerador FPGA atinge menor tempo de execução do que execuções *threads* em CPU e GPUs, conforme mostrado nos resultados de análise de desempenho.

4.2 Acelerando a execução do algoritmo *K-means*

K-means é uma técnica de aprendizado de máquina utilizada para extrair conhecimento de uma base dados identificando grupos naturais [41]. Esta seção propõe a implementação de um gerador de aceleradores para otimizar a execução de *K-means* em sistemas heterogêneos com FPGA. O gerador *K-means* tem potencial para paralelização de operações e reutilização de dados, estes dois aspectos podem ser exploradas em dispositivos de FPGA. Este problema foi selecionado como estudo de caso devido o algoritmo *K-means* possuir um padrão regular que trabalha com o pipeline sempre cheio. O *K-means* apresenta uma estrutura de *map-reduce* com um fluxo de dados que permite o acelerador maximizar o uso da interface de transferência de dados entre a CPU e o FPGA. O gerador funciona baseado em dois parâmetros principais são o número de agrupamentos K e o número de dimensões ou atributos N.

4.2.1 *K-means*

K-means é um algoritmo de aprendizado não supervisionado capaz de particionar um conjunto de pontos em grupos K, utilizado por diversas aplicações de mineração de dados [24]. No entanto, o algoritmo avalia cada ponto em cada iteração, e muitas iterações podem ser necessárias até atingir a convergência. Além disso, o processo geral exige um esforço computacional considerável ao lidar com grandes volumes de dados [2].

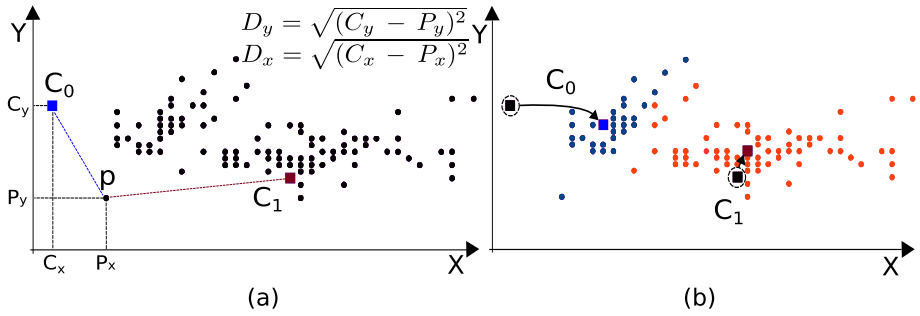


Figura 4.11: Classificação de pontos em um plano 2D.

A Figura 4.11 apresenta um exemplo de classificação de pontos em um plano 2D de dois grupos, ou seja, $K = 2$. Cada grupo tem um centroide C_i, e o algoritmo classifica cada ponto de acordo com o centroide mais próximo. Por exemplo, a Figura 4.11(a) mostra o ponto p, classificado no grupo G_0 uma vez que o centroide mais próximo é C_0. O algoritmo *K-means* executa este passo para todos os pontos da base dados. Depois, o algoritmo atualiza a posição de cada centroide. A nova posição é aquela que minimiza a distância quadrada do centroide do grupo para todos os pontos atribuídos ao grupo. A Figura 4.11(b) apresenta a nova posição dos centroides C_0 e C_1 após a execução de uma iteração. O algoritmo repete esses dois passos (classificação e reposicionamento) até convergir ou atingir o número máximo de iterações.

Neste trabalho foi considerado realizar apenas a etapa de classificação de cada ponto do conjunto de dados de entrada no acelerador de *hardware*, a etapa de reposicionamento do centroide foi realizada a nível de *software* em CPU. Essa estratégia simplifica o *hardware* do acelerador, possibilitando usar paralelismo espacial usando múltiplas cópias do mesmo núcleo de processamento do acelerador.

O algoritmo *K-means* pode ser descrito em termo de operações *map-reduce*. A etapa de classificação pode ser descrita como um mapeamento seguido de duas reduções. A Figura 4.12 mostra o fluxo de dados das operações para N dimensões e K centroides. O mapeamento é responsável pelo cálculo da distância em cada dimensão. A função de distância usada por *K-means* é a distância euclidiana, que, por simplicidade, é comumente substituída por sua versão quadrada (sem efetuar o cálculo da raiz quadrada). Assim, para ponto P e centroide C_i, na dimensão x, é

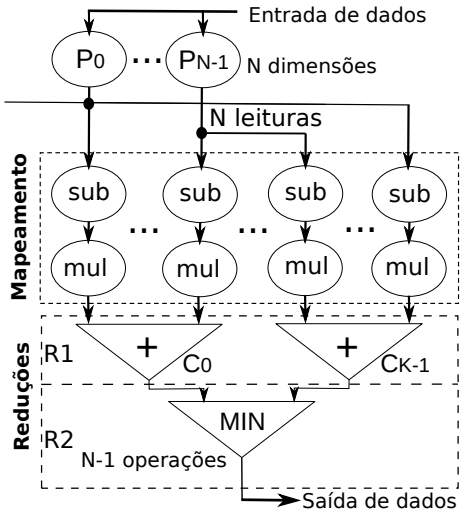


Figura 4.12: Fluxo de dados do algoritmo *K-means*.

necessário calcular $D_{i_x} = (P_x - C_{i_x})^2$. O algoritmo executa esta operação para todos os pontos, dimensões e centroides. Portanto, para cada dado lido da memória são realizadas $2k$ operações (uma subtração e uma multiplicação) como mostrado na Figura 4.12. Esta é a etapa de mapeamento. O segundo passo é uma redução das distâncias de todas as dimensões de um ponto para cada centroide. Para o centroide C_i, a árvore de redução calcula $D_i = D_{i_0} + D_{i_{n-1}}$. O terceiro passo é a redução para encontrar o centroide mais próximo de P, ou seja, o cálculo de $Min(D_0, \ldots, D_{k-1})$. A função Min retorna o índice do centroide mais próximo de P. Este é o passo de classificação *K-means*. Mais especificamente, *K-means* classifica o ponto no grupo $Min(D_0, \ldots, D_{k-1})$, onde a distância ao centroide C_i é dada por $D_i = \sum_{j=0}^{n-1}(P_j - C_{i_j})^2$. Assim requer kn subtrações e multiplicações durante o mapeamento, $k(n-1)$ adições na primeira redução e $k-1$ comparações na segunda redução. Em cada iteração, para m pontos, são necessárias $m[2kn + k(n-1) + k - 1]$ operações. A etapa de classificação é adequada para execução em FPGA, capaz de executar todas as operações em paralelo, dependendo de m, k e n. Outro aspecto relevante para ambas as plataformas é a reutilização de dados, ou seja, o número de operações para cada unidade de dados lida da memória. A reutilização de dados do *K-means* é $\frac{m[2kn+k(n-1)+k-1]}{mn} = 3k - \frac{1}{n}$, onde quanto maior o valor de k a execução é mais favorável para FPGAs. Por outro lado, a largura de banda de transferência de dados de memória limita o desempenho para pequenos valores de k.

A etapa de reposicionamento é responsável por atualizar a posição dos centroides. Para tanto, para cada ponto atribuído ao grupo de centroides C_i, devem ser acumulados os valores de cada dimensão. Além disso, se t pontos são atribuídos ao centroide C_i, sua nova posição para a dimensão j é $C_{i_j} = \frac{\sum_{j=1}^{t} p_j}{t}$. Esta etapa final requer operações $m \cdot n + k \cdot n$. Portanto, a complexidade de classificação é

$O(3k \cdot n \cdot m)$ e a complexidade de reposicionamento do grupo é $O(n \cdot m)$, considerando que, em cenários práticos, o tamanho do conjunto de dados m é muito maior do que k e n, ou seja, $m >> k, n$.

4.2.2 Gerador de aceleradores de *K-means*

Nossa abordagem para geração do acelerador de *K-means* cria automaticamente uma interface apropriada para encapsular o acelerador na lógica personalizada da plataforma com FPGA da AWS. Um módulo de interface baseado em FSM fornece transferências de dados genéricas entre um design de um acelerador e uma interface AXI-4 otimizada para uma largura de banda de 64 *bytes* (512 *bits*). Nosso gerador requer apenas dois parâmetros de entrada: k e n, onde k é o número de centroides e o n é o número de dimensões de dados. Nossa estrutura cria um projeto completo, incluindo *scripts* para construção do *bitstream*.

O gerador é escrito na linguagem Python com o uso da biblioteca Veriloggen [105]. Este modelo de programação é suficientemente abstraído do código Verilog HDL para descrever uma estrutura de *hardware* em Python e pode ser facilmente estendida. Nosso gerador baseado em Veriloggen cria todo o projeto Verilog para ser sintetizado com as ferramentas FPGA. Como já mencionado, a AWS usa FPGAs AMD Xilinx, e todo arcabouço de código foi validado com a ferramenta AMD Xilinx Vivado. Nossa abordagem encapsula toda a complexidade da interface, sincronização, compilação, síntese e comunicação CPU/FPGA. O usuário só precisa criar um projeto e usar um conjunto de *scripts* para compilar e utilizar o acelerador com sua própria base de dados.

Figura 4.13 mostra uma visão de alto nível do acelerador *K-means* em FPGA. Esta aplicação foi selecionada porque é possível mapear diretamente o grafo de fluxo de dados para um acelerador de *hardware*. Portanto, é simples explorar o paralelismo espacial e temporal disponível no grafo de fluxo de dados do *K-means*, que inclui o estágio de mapeamento *sub/mult*, seguido pelas primeiras árvores de redução de soma para calcular a distância do grupo e, finalmente, a última redução para selecionar o grupo de distância mais próximo. O gerador cria todo o código de fluxo de dados para um determinado valor k e n. O código gerado inclui todas as interfaces de entrada e saída, conectando o acelerador FPGA à CPU do host da plataforma. Construímos essas interfaces com base na unidade de E/S de *hardware* proposta em [15] para a plataforma Intel/Altera HARP. Este trabalho mostra como essa interface pode ser adaptada para funcionar plataformas com FPGAs da AMD Xilinx.

Como já mencionado, o código gerado inclui componentes de *hardware* e *software*. O código do *host* é composto por uma biblioteca que encapsula funções do OpenCL da AMD Xilinx. Este código controla as chamadas do acelerador e as transferências

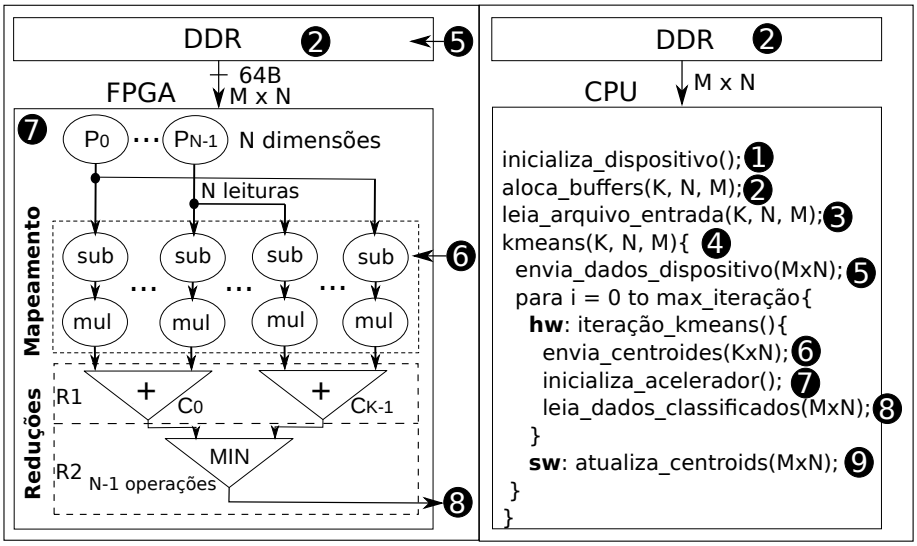


Figura 4.13: Grafo de fluxo de dados do acelerador de *K-means* e Pseudocódigo executado na CPU.

de dados. Figure 4.13 mostra o pseudo-código para as etapas principais de execução do acelerador de *K-means*. Na etapa ❶, o *host* configura o acelerador FPGA enviando o *bitstream*. Esta etapa é executada apenas uma vez e o FPGA executa repetidamente o código do acelerador de *hardware* sem nenhuma reconfiguração para um determinado k e n. Na etapa ❷, a código *host* aloca os *buffers* nas memórias DDR do FPGA. Essas memórias armazenarão os dados durante toda a execução da aplicação. Na etapa ❸, o *host* carrega os dados de um arquivo para os *buffers*. A aplicação será executada em ambos os dispositivos: CPU *host* e FPGA. O passo ❹ é todo o algoritmo *K-means*, incluindo as tarefas de *software* (*host*) e *hardware* (FPGA) e de transferência de dados da CPU para o FPGA. Os passos ❺-❾ na Figura 4.13 detalham as partes do *K-means*. Primeiro, no passo ❺, a CPU envia o conjunto de dados da memória principal para a memória DDR do FPGA, e então a CPU inicia uma nova iteração *K-means*. A cada iteração, primeiro, a CPU envia os centroides para o acelerador em ❻. Na etapa ❼, o acelerador executa de forma bloqueante até classificar todo o conjunto de dados. Na etapa ❽, o FPGA envia os dados classificados para a memória da CPU e, finalmente, na etapa ❾, a CPU atualiza os centroides.

A plataforma de FPGA utilizada pela AWS recomenda usar uma granularidade para comunicação com a memória de 64 *bytes* ou 512 *bits* [116]. Se um ponto de dados requer menos de 64 *bytes*, a arquitetura explora o paralelismo espacial com replicação, permitindo o processamento de vários pontos em paralelo para maximizar a taxa de transferência de dados. Figure 4.14 mostra a capacidade do gerador de criar múltiplas cópias para explorar o paralelismo espacial manipulando múltiplos pontos de dados. Suponha uma palavra de 16 *bits* e $n = 4$, significando quatro dimensões. Cada ponto requer $4x16$ *bits* ou 8 *bytes*. Portanto, nosso gerador cria oito cópias do acelerador

para computar 8 pontos de dados em paralelo. A Figura 4.14 mostra um exemplo para $n = 2$, onde 16 aceleradores são instanciados. Além disso, para mitigar a latência de comunicação, a interface permite filas e mais de uma instância do acelerador com execução simultânea.

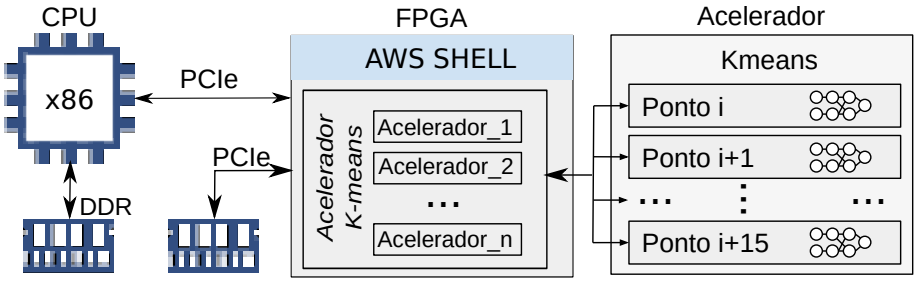


Figura 4.14: Arquitetura do sistema CPU-FPGA com acelerador de *K-means*.

4.2.3 Resultados Experimentais

Para avaliar todo arcabouço de *software* e *hardware* desenvolvido foi gerado um conjunto de aceleradores de *K-means* com diferentes tamanhos de K e N. Para os testes de execução foi utilizada a base de dados *US Census 1990* [31]. Essa base de dados foi usada em vários trabalhos relacionados a aceleradores em FPGA [68, 59, 2, 88]. O tamanho da entrada contém dois milhões de pontos. O desempenho foi avaliado variando os parâmetros n (número de dimensões) e k (número de clusters) de 2 a 32. Os resultados foram comparados com implementações em outras arquiteturas com FPGA Virtex-6 [32], Intel HARPv2 (Arria 10 modelo 10AX115U3F45E2SGE3 FPGA [88]), e placa Alpha Data ADM-PCIE-KU3 com um Kintex-6 XCKU060 FPGA [86]. O tempo de execução dos experimentos foi medido usando as funções da biblioteca *chrono::high_resolution_clock* inseridas diretamente no código da chamada do acelerador. Todos os experimentos foram repetidos 10 vezes e os resultados apresentados são a média das 10 execuções.

A Figura 4.15 mostra o número de operações realizadas pelo acelerador variando o número de dimensões n de 2 a 32, para cada n foram avaliados k de 2 a 16, onde apenas foram considerados potências de 2. O desempenho foi medido em GOPS para a etapa de chamada do acelerador de *hardware*, que executa uma classificação completa sobre todos os pontos de dados m. Os experimentos consideram apenas um módulo DDR. Portanto, o tamanho do conjunto de dados pode ser de até 16 GiB. No entanto, o AWS EC2 F1 possui 4 módulos de memória DDR e, consequentemente, é possível aumentar o número de pontos de dados em até 64 GiB. Por fim, os resultados de execução na AWS foram comparados com o acelerador Intel/Altera HARPv2 [88] considerando apenas a etapa de classificação

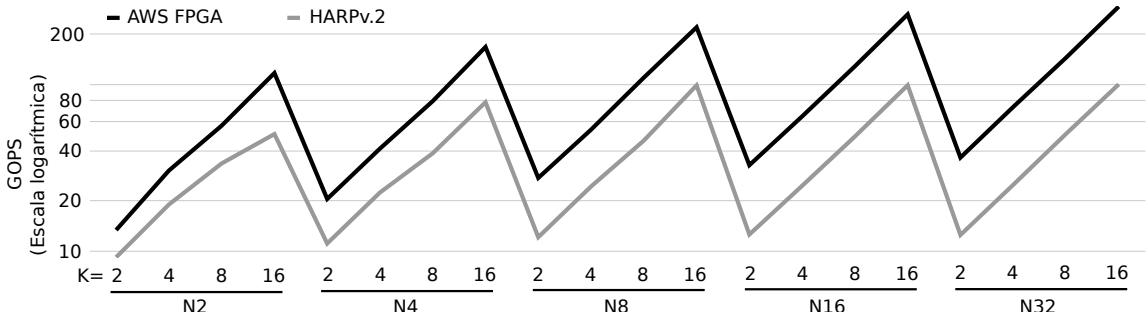


Figura 4.15: Desempenho em GOPS por interação de classificação, incluindo etapas de *hardware* e *software* para o acelerador na plataforma AWS e Intel/Altera HARPv2 [88]

realizada pelo acelerador de *hardware*. Os aceleradores utilizados como comparação possuem o mesmo projeto arquitetônico e diferem apenas na plataforma onde foram executados. No entanto, a plataforma da AWS em nuvem é totalmente diferente, incluindo o dispositivo FPGA e as interfaces de *software* e *hardware* CPU/FPGA. Outro ponto importante é que o FPGA da plataforma HARPv2 não possui memória local. Outro ponto que diverge entre as plataformas comparadas é que a plataforma HARPv2 possui três barramentos PCI para transferir dados da memória principal do sistema de/para o FPGA, onde a largura de banda máxima é de 26 GB/s [88, 23]. O FPGA da plataforma da AWS possui 4 módulos de memórias DDR locais e apenas um barramento PCIe x16 entre o FPGA e a CPU com largura de banda de 16 GB/s.

É possível notar também na Figura 4.15 que o valor de GOPS aumenta em função de k para as mesmas dimensões. Ou seja, quando maior for o valor de k maior será o desempenho da implementação em FPGA. O motivo é a reutilização de $3kdedados - \frac{1}{n}$ operações por byte. O fator limitante mais significativo para plataformas com FPGAs é a leitura de dados.

A Figura 4.15 também mostra que o desempenho do acelerador na AWS é em média $1,98\times$ melhor que o acelerador Intel/Altera HARPv2. O desempenho geral máximo da plataforma HARPv2 é de 58,6 GOPS, e o acelerador de *hardware* na AWS atinge até 294,2 GOPS.

A Figura 4.16 mostra que o tempo de execução de uma iteração não muda para um dado k, pois dependente da quantidade de dados e não de seu reuso (k). O aumento do tempo de execução em função de n é devido à quantidade de dados transferidos. Por exemplo, para 2 milhões de pontos, palavras de 2 *bytes* (ou 16 *bits*) e $n = 2$, serão $2M \cdot 2$ *bytes* $\cdot 2 = 8$MB de dados totais. Para $n = 4$ serão 16 MB de dados totais. Portanto, o tempo de execução cresce linearmente em função de n, ou seja, $n\times$ número de pontos.

A Figura 4.17 apresenta o consumo de recursos do FPGA em função dos valores de k e n. Observa-se que, para um mesmo valor de k, o uso de recursos permanece constante, independentemente da variação de n. Esse comportamento é vantajoso

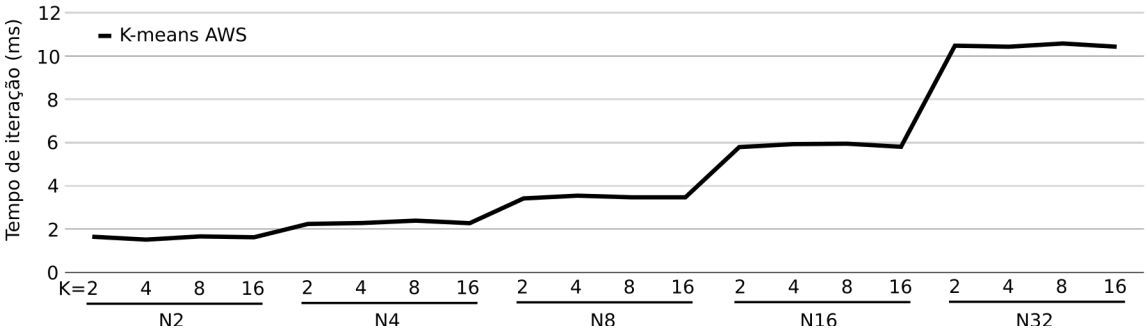


Figura 4.16: Tempo de iteração do acelerador de *k-means* AWS para diferentes valores de k e n.

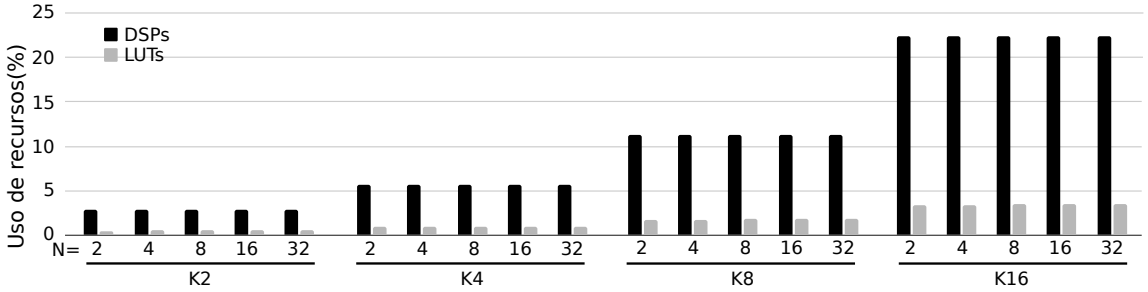


Figura 4.17: Porcentagem de uso de recursos FPGA para conjuntos de aceleradores de *K-means* AWS.

para FPGAs, pois a utilização de recursos influencia diretamente a frequência de *clock* do acelerador. Diferente das Figuras anteriores, fixamos o valor de k e variamos o valor de n dentre de cada faixa no eixo horizontal. A quantidade total de LUTs necessárias são inferiores a 4% do FPGA no pior caso. O número de DSP cresce linearmente com k, e não depende de n. Isso ocorre porque se n for pequeno, replicamos o projeto para processar todos os 64 *bytes* de dados de entrada. Por exemplo, $n = 2$ tem o dobro de cópias em comparação com $n = 4$, portanto $c \times n$ permanece constante, onde c é o número de cópias. No entanto, quando k aumenta, o número de grupos paralelos aumenta e o número de multiplicadores. O valor mais alto de k requer menos de 25% do total de DSPs disponíveis no FPGA AWS EC2 F1. Além disso, o FPGA AWS EC2 F1 tem uma perspectiva favorável, pois o FPGA VU9P da AMD Xilinx tem 6760 DSPs [23] em comparação com os 1518 Arria 10 em HARPv2 [23].

A Tabela 4.4 apresenta uma comparação de desempenho do acelerador na AWS e uma implementação totalmente paralela especializada apresentada em [32] e executada no FPGA Virtex-6. O acelerador proposto neste trabalho calcula mais DPS (*Data Points per Second*), ou Pontos de Dados por Segundos, medidos com base no tempo de execução, incluindo todo o sistema (inicialização, transferência de dados, etc.), enquanto Dias *et al*. [32] estima os resultados com base em dados retirados do

Tabela 4.4: Desempenho em mega pontos de dados por segundo para acelerador SW/HW AWS proposto nesse trabalho e acelerador HW FPGA proposto em [32]

		MDPS (*Mega Data Points per Second*), ou Mega Pontos de Dados por Segundo		
		Proposto neste trabalho		[32]
k	n	Classificação HW	Sw/Hw Total	Hw Total
4	2	1324,06	110,21	88,99
8	2	1205,69	148,71	84,01

relatório de síntese do acelerador.

Tabela 4.5 apresenta uma comparação de desempenho do acelerador proposto com uma implementação OpenCL proposta em [86] na plataforma Alpha Data com uma FPGA Kintex-6 da AMD Xilinx. Nossa implementação atinge um desempenho semelhante ao acelerador $v5$ [86], que utiliza 16 unidades paralelas e otimiza a largura de banda da memória. Por outro lado, o acelerador $v4$ [86] não é otimizado para largura de banda máxima de memória, resultando em degradação do desempenho.

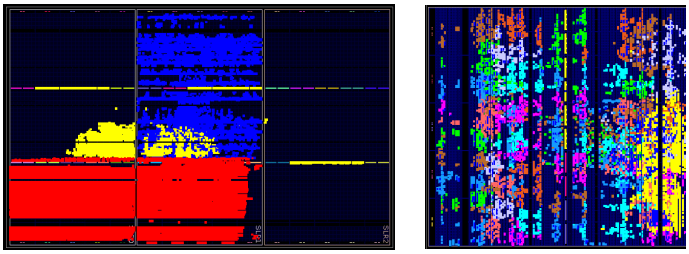
Tabela 4.5: Comparação de desempenho do acelerador Sw/Hw deste trabalho e do acelerador sw/hw baseado em OpenCL proposto em [86].

k	n	Taxa Transf. (MB/s)	Tempo Exec. (ms)	V1 GOPS	Unidade	V5 GOPS	Este Trabalho GOPS
8	8	27,30	38,41	0,33	16	10,91	15,09
16	8	39,00	26,89	0,94	16	31,17	29,98
-	-	-	-	-	-	V4 GOPS	-
8	16	31,50	66,58	0,38	16	2,079	18,83

A Figura 4.18(a) mostra um acelerador para o *K-means* com $k = 16$ e $n = 4$ mapeado no FPGA. O módulo do acelerador é azul; o amarelo mostra nosso módulo de interface de memória DDR/acelerador. Por fim, o vermelho representa o módulo SHELL da AWS. O SHELL é uma região reservada do FPGA e não pode ser alterada pela lógica do usuário. A Figura 4.18(b) ilustra o acelerador $k = 16$ e $n = 4$ com oito cópias. Cada cópia tem uma cor diferente. O desenho se espalha pelas colunas onde estão localizados os blocos de DSPs responsáveis por realizar a multiplicação.

4.3 Discussões

A Seção 4.1 explorou o uso de FPGAs em sistemas heterogêneos para desenvolver um acelerador de *hardware* para redes reguladoras de genes. Este trabalho é pioneiro



(a) (b)

Figura 4.18: Mapeamento do acelerador *K-means* AWS com $k = 16$ e $n = 4$.(a) Área total do FPGA.(b) Área usada apenas pelo acelerador.

na aceleração de GRNs usando Amazon AWS FPGAs. Nosso gerador de acelerador GRN fornece uma interface de fácil utilização para computar atratores sem ser responsável pelo fluxo de projeto de *hardware* e gerenciamento de recursos de FPGA. Além disso, avaliamos o desempenho por dólar, a qual é uma métrica importante em serviços em nuvem. O FPGA oferece alto paralelismo a nível de *bit* na avaliação das equações e escalabilidade para instanciar mais cópias do acelerador, explorando o paralelismo espacial. As filas e o agrupamento dos elementos de processamento otimizaram o desempenho do acelerador com carga de trabalho não uniforme e comunicação centralizada com a CPU. Nosso acelerador é 12x mais rápido que a melhor solução em GPU e atinge um desempenho por dólar 5,7x melhor que a melhor solução de GPU. O código do gerador é aberto e esta disponível em um repositório público[1] e pode ser utilizado também como código base para desenvolver novos aceleradores para instâncias GRN e AWS F1.

A Seção 4.2.1 apresentou um gerador de aceleradores de *K-means* de código aberto para simplificar o projeto de aceleradores de *hardware*/*software* para a plataforma Amazon FPGA. As arquiteturas geradas melhoraram o desempenho especializando-se em um domínio específico e explorando o paralelismo espacial e temporal. O FPGA na nuvem da Amazon democratiza o projeto de aceleradores de *hardware* e *software* especializados. Nosso estudo de caso ilustra como implementar um projeto eficiente usando todas as interfaces de memória. Também mostramos como gerar e integrar um núcleo IP de módulo parametrizado e implantar e executar um acelerador. O código está disponível em um repositório de código aberto[2]. Comparado com o FPGA em nuvem Intel HARPv2 [88], os resultados experimentais apresentaram ganhos de tempo de execução de até $1,98\times$. Nossos aceleradores atingem um desempenho semelhante à exploração de design de alto nível OpenCL apresentada em [86].

Os dois estudos de caso que abordam o uso da síntese direta de aceleradores em

[1] https://github.com/lesc-ufv/grn_hw_accelerator
[2] https://github.com/lesc-ufv/kmeans_aws_f1_hw_generator

FPGAs demostraram um bom desempenho em relação a arquiteturas clássicas, como CPU e GPU, e podemos tirar as seguintes lições destes trabalhos:

- A distribuição de tarefas para um conjunto de PEs por meio de um barramento compartilhado otimiza o uso de recursos e permite o envio de configurações personalizadas para partes específicas do acelerador.
- O uso de filas de entrada e saída simplifica o envio e recebimento de dados para o acelerador.
- O uso de geradores de aceleradores otimiza o tempo de projeto, mesmo para aceleradores específicos, e pode ser usado para generalizar a criação de outras arquiteturas.
- O tempo de síntese elevado torna aceleradores específicos proibitivos para aplicações que precisam atualizar alguns parâmetros, como no *K-means*, em que avaliar uma base de dados com um número diferente de grupos exigiria outra síntese do acelerador.

Capítulo 5

Projeto de Arquitetura de Acelerador Reconfigurável de Domínio Específico

Diferentemente do capítulo anterior, que apresentou geradores de aceleradores para nichos específicos, neste capítulo apresentamos uma ferramenta capaz de gerar um rDSA (*Reconfigurable Domain Specific Accelerator*), ou Acelerador Reconfigurável de Domínio Específico, para a execução de algoritmos no formato de fluxo de dados. A Figura 5.1 apresenta uma visão geral da Ferramenta, chamada *REconfigurable Accelerator DeploY* (READY). READY resultou das lições aprendidas no capítulo anterior, em particular na reconfigurabilidade do acelerador. READY une o modo de execução de aplicações em fluxo de dados com elementos que podem ser reconfigurados em tempo de execução sem a necessidade de realizar uma nova síntese.

READY é um ambiente que suporta o mapeamento de aplicações descritas na forma de GFD em plataformas CPU-FPGA de alto desempenho. A arquitetura proposta é um CGRA reconfigurável em tempo de execução, implementado como uma camada virtual sobre o FPGA. A arquitetura base é composta por um conjunto de elementos de processamento e uma rede de interconexão global, onde qualquer elemento de processamento pode se comunicar diretamente com qualquer outro. Tanto a rede de interconexão como os elementos de processamento são reconfiguráveis em tempo de execução. Além disso, para o CGRA proposto, o desempenho da execução é previsível. Se o grafo de fluxo de dados tem 100 operadores, o CGRA irá executar em pipeline cheio, gerando um resultado por ciclo. Para uma frequência de relógio de 250MHz, o desempenho será de $100 \times 250M = 25$ GOPS.

Um dos principais desafios do *hardware* reconfigurável é o tempo de mapeamento das aplicações. Para os FPGAs, este tempo pode ser da ordem de dias, considerando a complexidade atual dos FPGAs. O CGRA simplifica o processo para o nível de palavras, porém o mapeamento envolve três etapas: *Scheduling*, *Placement* e *Routing* (SPR), o qual é um problema NP-completo [49].

A rede global da arquitetura gerada pelo READY simplifica o mapeamento,

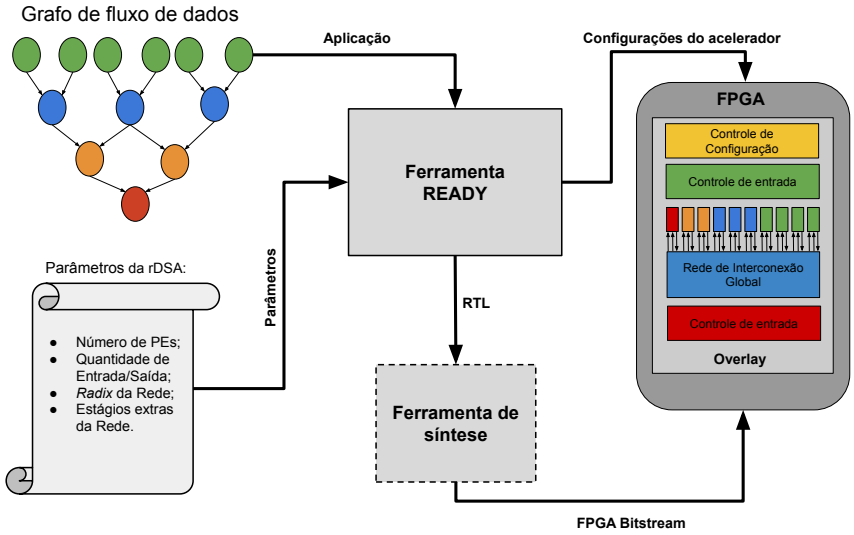


Figura 5.1: Visão geral Ferramenta READY.

porém tem um custo em área e em latência na comunicação. A solução proposta utiliza múltiplas tarefas para ocultar a latência das interconexões. O arcabouço processa descrições de grafos de fluxo de dados usando uma API de alto nível que especifica o paralelismo espacial e temporal explicitamente. A ferramenta é flexível e extensível, e fornece a infraestrutura para explorar diferentes projetos de aceleradores. Para validação, escolhemos um conjunto padrão de operações para os elementos de processamento e usamos uma arquitetura homogênea.

READY foi validado nas plataformas da Intel e da AMD Xilinx, utilizando o *cluster* de pesquisa da Universidade de Paderborn e o serviço de FPGAs na nuvem da AWS, respectivamente. Os resultados experimentais mostram uma melhoria média no tempo de execução de 2x quando comparado a um processador de alto desempenho. Para as aplicações avaliadas, READY também foi 2 ordens de magnitude melhor quando comparado com a abordagem utilizando síntese de alto nível da AMD Xilinx.

5.1 Arquitetura do READY CGRA

Para permitir a exploração de diferentes tamanhos de arquiteturas, foi implementado um gerador parametrizado e modular usando a biblioteca Veriloggen. Esta biblioteca é um arcabouço de ferramentas de processamento de linguagem de *hardware* baseado em Python [105]. Uma arquitetura de CGRA READY é composta por um conjunto de elementos de processamento e por uma rede multiestágio do tipo ômega. A Figura 5.2(a) apresenta um diagrama completo do acelerador. As caixas azuis são dependentes da plataforma e são responsáveis pela comunicação

externa do acelerador, enquanto as caixas brancas são geradas automaticamente de acordo com as configurações desejadas pelo usuário.

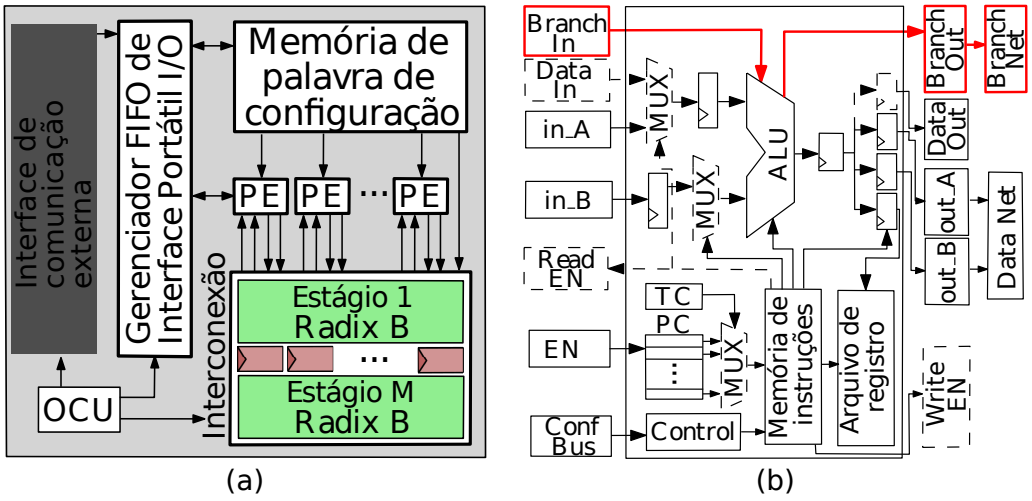


Figura 5.2: (a) Diagrama completo do acelerador: Unidade de controle, Interface de entrada e saída, conjunto de PEs, memória de palavras de configuração e rede de interconexão; (b) Organização interna de um elemento de processamento.

A Figura 5.2(b) mostra a estrutura interna do PE, que possui duas entradas (in_a, in_b) e duas saídas (out_a, out_b). A ALU implementa as mesmas operações presentes na Tabela 5.1. O contador de *threads* (TC) controla o multiplexador de *threads* para selecionar o contador de programa (PC) que endereça a instrução atual a ser executada. A troca de contexto é realizada aumentando o TC a cada ciclo de *clock*, e cada *thread* possui um banco de registradores locais (RF). Os PEs que possuem acesso à memória externa têm uma entrada e uma saída de dados extra, além dos sinais de controle para habilitar a escrita e leitura de dados externos, que são mostrados com as linhas tracejadas na Figura 5.2(b).

O PE possui uma latência de três ciclos de *clock*, que está oculta no sistema geral pela abordagem *multithread*. A entrada $Conf\_bus$ configura a memória de instruções do PE durante a etapa de reconfiguração do CGRA. O TC de cada *thread* possui $log_2(N)$ *bits*, onde N é a quantidade de *threads*. Existem dois sinais para controlar o desvio, representados em vermelho na Figura 5.2(b), que são usados para os operadores condicionais. Esses sinais são conectados em uma rede de interconexão de 1 bit, separada para propagar dinamicamente os sinais de fluxo de controle.

O gerador é capaz de criar redes multiestágio com tamanhos e configurações personalizadas, como o *radix* dos switches internos, o número de estágios extras, o número de entradas e saídas, e a largura em *bits* dos dados da rede. A memória de configuração consiste em um conjunto de memórias otimizadas e distribuídas para cada PE e switches para maximizar a frequência do *clock*, permitindo assim uma comutação de contexto *multithreading* de granulação fina. A unidade de controle,

chamada de *Overlay Control Unit* (OCU), é responsável por controlar as etapas de configuração e execução do CGRA, e por comunicar com o *software* host a finalização do processo.

5.2 Arcabouço de *software* e *hardware* do READY

Nesta seção é apresentada uma visão geral do arcabouço de *software* e *hardware* do READY. O diagrama de blocos mostrado na Figura 5.3 representa o fluxo de execução de um grafo de fluxo de dados em um READY CGRA, com todas as partes do processo.

Na etapa 1 da Figura 5.3 é mostrada a descrição da aplicação em GFD, especificada usando uma API C++. A etapa 2 ilustra o mapeamento do GFD para a arquitetura de destino, criando um objeto CGRA mostrado na etapa 3, que inclui a geração do *bitstream* de configuração do CGRA. A etapa 4 vincula as variáveis do programa às entradas e saídas do GFD. Antes da execução, o *bitstream* do CGRA é enviado para o acelerador no FPGA (etapa 5). Por fim, a etapa 6 é responsável pela execução do acelerador, incluindo as transferências de dados CPU-FPGA sobrepostas. No restante desta Seção, são fornecidos detalhes sobre os principais módulos do READY.

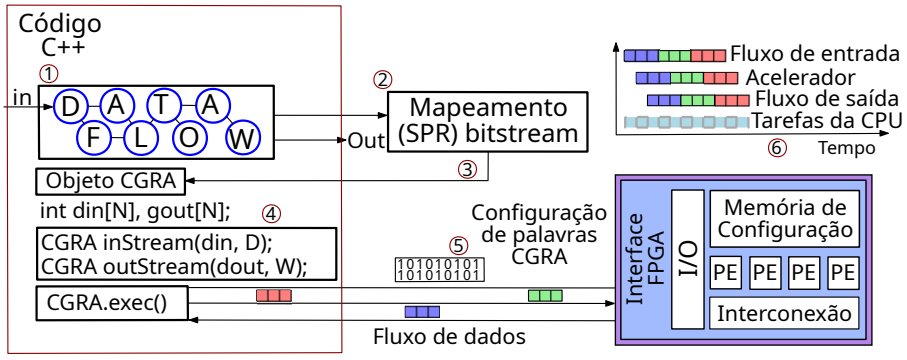


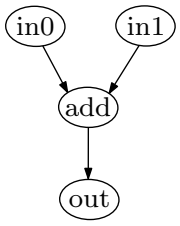
Figura 5.3: Arcabouço READY: (1) Aplicação no formato de grafos de fluxo de dados; (2) Mapeamento; (3) Objeto CGRA; (4) Vinculando os dados locais à aplicação; (5) Tempo de configuração; (6) Execução, incluindo sobreposição de entrada e saída.

5.3 Biblioteca para descrição de Grafo de Fluxo de Dados

Para programar um CGRA na arquitetura READY, foi adotado um modelo de desenvolvimento de algoritmos utilizando grafos de fluxo de dados, que descrevem explicitamente o paralelismo temporal e espacial por meio de uma abordagem

baseada em grafo que tem como lema: "O que você vê é o que você recebe", ou do inglês: *"What You See Is What You Get"* (WYSIWYG). O rótulo 1, na Figura 5.3, descreve esta parte do sistema. A Figura 5.4 mostra um exemplo simples de um projeto de aplicação descrita na forma de um GFD. Essa aplicação realiza a soma de dois fluxos de entrada e possui um fluxo de saída com os resultados das somas.

A Figura 5.4(a) apresenta uma visão gráfica do GFD, onde os nós de entrada de dados (in0 e in1) possuem apenas uma saída, o nó que realiza a operação de soma possui duas entradas e uma saída, e o nó de saída de dados possui apenas uma entrada. A mesma estrutura pode ser descrita utilizando a API C++, como mostrado na Figura 5.4(b), onde primeiramente um objeto do tipo *DataFlow* é instanciado. Esse objeto é responsável por armazenar os objetos operadores do GFD, os quais são os nós. A API possui 19 tipos de operadores, divididos nas categorias: aritméticos, lógicos, condicionais, de controle de fluxo, comparativos, deslocamento e de cópia. A Tabela 5.1 apresenta uma descrição de todos os operadores implementados na biblioteca do READY.



(a)

```
auto df = new DataFlow(0);
auto in0 = new InStream();
auto in1 = new InStream();
auto add = new Add();
auto out = new OutStream();
df → connect(in0, add, add → A());
df → connect(in1, add, add → B());
df → connect(add, out, out → A());
```

(b)

Figura 5.4: Exemplo de aplicação descrita no formato de GFD: (a)Visualização do Grafo; (b) Descrição do GFD em código em C++.

5.4 Mapeamento *multithread* de granularidade fina

Esta seção se concentra em explicar a estratégia utilizada para ocultar a latência da rede global multiestágio, destacando as semelhanças e diferenças entre a abordagem proposta neste trabalho e o *multithreading* refinado de uma arquitetura de GPU convencional.

O *multithreading* refinado foi proposto pela primeira vez para sobrepor a latência de entrada e saída no sistema CDC6600 [110]. Mais tarde, a Denelcor aplicou essa ideia no *Heterogeneous Element Processor* (HEP) [90]. No HEP, cada *thread* possui apenas uma instrução no pipeline do processador, e cada *thread* é independente das demais. O HEP executa uma troca de contexto para outro encadeamento a cada ciclo, de modo que duas instruções de um encadeamento não estejam no pipeline simultaneamente,

Tabela 5.1: Descrição dos operadores implementados na biblioteca do READY.

Operador	Classe	Função	Descrição
Abs()	Aritmética	$\|a\|$	Realiza a conversão do dado de entrada para seu valor absoluto.
Add()		$a + b$	Realiza a soma das entradas.
Sub()		$a - b$	Realiza a subtração das entradas.
Mult()		$a * b$	Realiza a multiplicação das entradas.
And()	Lógicos	$a\&b$	Realiza a operação *and* bit a bit entre as entradas.
Or()		$a\|b$	Realiza a operação *or* bit a bit das entradas.
Not()		$\sim a$	Realiza a operação *not* bit a bit das entradas.
Xor()		$a \hat{} b$	Realiza a operação *xor* bit a bit das entradas.
Beq()	Controle de fluxo	$a == b?1 : 0$	Compara as entradas a e b, caso sejam iguais envia um 1 para saída caso contrário envia 0.
Bne()		$a! = b?1 : 0$	Compara as entradas a e b, caso sejam diferentes envia um 1 para saída caso contrário envia 0.
Slt()		$a<b?1 : 0$	Compara as entradas a e b, caso a seja menor que b envia um 1 para saída caso contrário envia 0.
Sgt()		$a>b?1 : 0$	Compara as entradas a e b, caso a seja maior que b envia um 1 para saída caso contrário envia 0.
Mux()		$a?b : c$	Recebe três entradas, caso a entrada a seja 1 envia a entrada b para saída caso contrário envia a entrada c para saída.
Min()	Comparação	$a<b?a : b$	Compara as entradas a e b, caso a seja menor que b envia a entrada a para saída caso contrário envia a entrada b para saída.
Max()		$a>b?a : b$	Compara as entradas a e b, caso a seja maior que b envia a entrada a para saída caso contrário envia a entrada b para saída.
Shl()	Deslocamento	$a \ll b$	Realiza o deslocamento de bit para esquerda de a, b vezes.
Shr()		$a \gg b$	Realiza o deslocamento de bit para direita de a, b vezes.
PassA()	Cópia	a	Realiza a apenas a cópia dos dados de entrada para a saída.
PassB()		b	Realiza a apenas a cópia dos dados de entrada para a saída.

o que tolera latências de dependência, sobrepondo a latência com trabalho útil de outros encadeamentos.

Recentemente, as arquiteturas de GPU têm adotado uma abordagem semelhante à utilizada nos sistemas HEP e no processador Niagara [61] para lidar com a latência de memória. Essas GPUs modernas executam milhares de *threads* simultaneamente para ocultar as latências de acesso à memória e manter o máximo de unidades funcionais ocupadas. No entanto, essa estratégia demanda uma quantidade significativa de registradores (a GPU NVidia V100, por exemplo, possui 20 MB de espaço de registradores) para armazenar as variáveis locais, o que pode consumir mais de 18% da energia total durante a execução [8].

A arquitetura de CGRA proposta neste capítulo inclui uma rede de interconexão global. Uma abordagem simples seria usar uma *crossbar* global, já que esse tipo de rede, com tamanhos de 128x128 ou 256x256, pode ser projetado em VLSI com um custo de área de silício razoável. No entanto, essa abordagem não é viável em um FPGA, uma vez que os multiplexadores são implementados em FPGAs usando LUTs, que apresentam uma alta latência quando comparados com a abordagem adotada em VLSI [115, 85].

Por outro lado, uma rede multiestágio reduz o custo de área em comparação com uma *crossbar*, mas apresenta a desvantagem de sintetizar a uma frequência de *clock* de 100 MHz ou menos [39]. Para contornar essa limitação, registradores de pipeline foram inseridos entre cada estágio da rede. No entanto, embora isso contribua para

uma melhoria na frequência de *clock*, uma vez que o caminho crítico da rede foi quebrado, a latência de ciclos para um dado passar pela rede aumenta, reduzindo o desempenho da arquitetura na mesma proporção.

Uma abordagem para otimizar o uso dos elementos de processamento e evitar que esses recursos fiquem ociosos por muitos ciclos é utilizar o modelo de *multithreading* de granulação fina. Nesse caso, em uma arquitetura que possua 8 ciclos de latência na conexão entre dois elementos de processamento, é possível executar 8 *threads* de forma simultânea no mesmo pipeline. A cada ciclo, uma *thread* utiliza os recursos de um estágio da rede e dos elementos de processamento.

Esta abordagem foi implementada da seguinte maneira: durante a execução, um contador de *threads* (TC) controla a alternância de contexto a cada ciclo. A Figura 5.5 mostra uma visão simplificada da nossa abordagem com 3 *threads* representadas pelas cores vermelha, roxa e verde, correspondendo aos grafos das três aplicações: App_1, App_2 e App_3. A cada ciclo de *clock*, uma *thread* é executada nos elementos de processamento, enquanto as outras duas *threads* percorrem a rede de interconexão. Na Figura 5.5(a), App_1 está sendo executado nos PEs. No próximo ciclo, o contador de *threads* (TC) é incrementado, e então App_1 move-se para o primeiro estágio da rede (Figura 5.5(b)), App_2 para o segundo estágio, e App_3 retorna para as unidades do PE. No ciclo seguinte, App_2 move-se para as unidades PE, App_3 para o primeiro estágio, e App_1 move-se para o segundo estágio, conforme mostrado na Figura 5.5(c). No próximo ciclo, o TC retorna à primeira linha da memória de configuração (Figura 5.5(a)) para executar App_1 e repetir o processo.

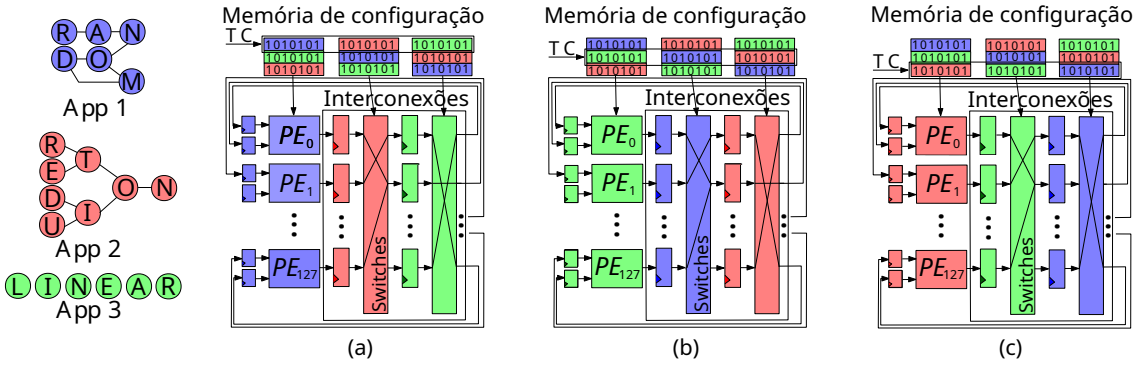


Figura 5.5: Um exemplo de troca de contexto *multithread* com três *threads*: (a) App_1 está alocado para executar no PE, App_2 e App_3 estão alocados para usar os estágios da rede; (b) App_3 está alocado para executar no PE, App_1 e App_2 estão alocados para usar os estágios da rede; (c) App_2 está alocado para executar no PE, App_3 e App_1 estão alocados para usar os estágios da rede.

5.4.1 Comparativo entre modelo de execução READY CGRA e GPU convencional

Um comparativo entre os modelos de execução de *threads* da arquitetura proposta e um modelo de execução de *threads* de GPU convencional é apresentado na Figura 5.6. A Figura 5.6(a) mostra o modelo de execução no formato de fluxo de dados, onde todo o grafo de computação é executado a cada ciclo de *clock* de forma contínua. Nesta Figura, os itens de fluxo são sobrepostos no tempo, e as linhas tracejadas representam intervalos de tempo, enquanto os índices $i + \ldots$ representam os itens de fluxo.

A Figura 5.6(b) apresenta um modelo de processamento de uma GPU convencional, onde o sistema despacha 32 cópias de um nó de fluxo de dados por ciclo para cada *warp*, seguindo um esquema de única instrução múltiplas *threads* (SIMT do inglês *Single Instruction Multiple Threads*). Suponha que o compilador da GPU gere o código mostrado na Figura 5.6(b), e a unidade funcional gaste 10 ciclos para calcular. Suponha que $warp_0$ esteja executando a instrução Z. $Warp_0$ continua a rodar somente se as instruções X e Y já tiverem terminado (mais de 10 ciclos antes). A seguinte instrução R deve esperar pelo menos 9 ciclos, e o escalonamento deve mudar o contexto para outro *warp* que esteja pronto para ser executado.

Neste exemplo, cada *thread* processa apenas um item. Cada *thread* tem um número fixo de registradores pré-alocados durante a vida útil do bloco de *threads* da GPU. Portanto, como a GPU deve acompanhar todos os blocos de *threads* em execução para um determinado núcleo da GPU, ela consome um grande número de registradores.

Diferentemente da solução adotada neste trabalho, o uso de registradores é significativamente reduzido porque o modelo de fluxo de dados usa alguns *threads* para ocultar a latência, aproveitando os registradores da rede multiestágio para armazenar o contexto das aplicações. Outro ponto importante é que é possível utilizar operadores condicionais para resolver caminhos de ramificações divergentes, o que geralmente degrada o desempenho da GPU, uma vez que cada *warp* precisa executar a mesma operação sobre todos os dados ao mesmo tempo.

5.4.2 Desvio condicional de fluxo na arquitetura READY

Os operadores condicionais do READY utilizam um sinal de entrada e saída de desvio de um bit para implementar um mecanismo de fluxo de controle dinâmico. Uma operação de controle (por exemplo, Beq ou Slt) define o sinal de desvio. Utilizamos o termo “desvio” para o controle do fluxo, que é realizado por um multiplexador que recebe o sinal da operação de controle. A Figura 5.7(a) mostra um exemplo simples em que o aplicativo soma 3 a um valor de entrada se o valor da entrada for menor que 2 e subtrai 3 caso contrário.

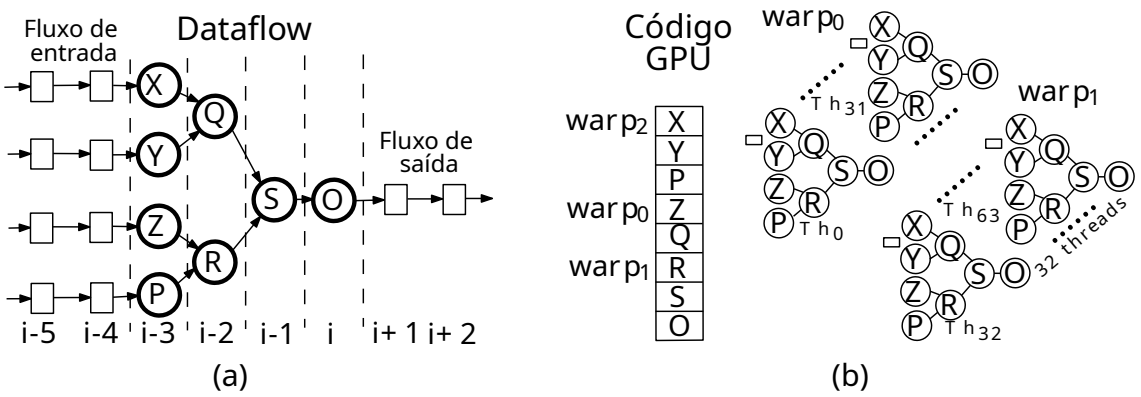


Figura 5.6: (a) Modelo de fluxo de dados baseado em fluxo de dados; (b) Execução em uma GPU convencional

No aplicativo de exemplo mostrado na Figura 5.7, há um fluxo de entradas com os valores 1, 2, 4 e 5. A Figura 5.7(b) mostra a primeira entrada 1 chegando ao segundo estágio do GFD. No próximo ciclo de *clock*, a segunda entrada 2 é encaminhada para o segundo estágio, conforme mostrado na Figura 5.7(c), enquanto o primeiro valor 1 já foi processado pelos nós *Slt*, *Add* e *Sub* e encaminhado para o terceiro estágio na comparação do nó *Mux*. A condição será verdadeira para a primeira entrada, onde o valor 4, oriundo da soma 1 + 3, é passado para a saída. No próximo ciclo, a condição será falsa para o valor de entrada 2. E no próximo ciclo, o *Mux* irá propagar o valor -1, oriundo da subtração de 2 - 3, para a saída B. Finalmente, na Figura 5.7(d), o primeiro dado processado com valor 4 é processado pela saída B.

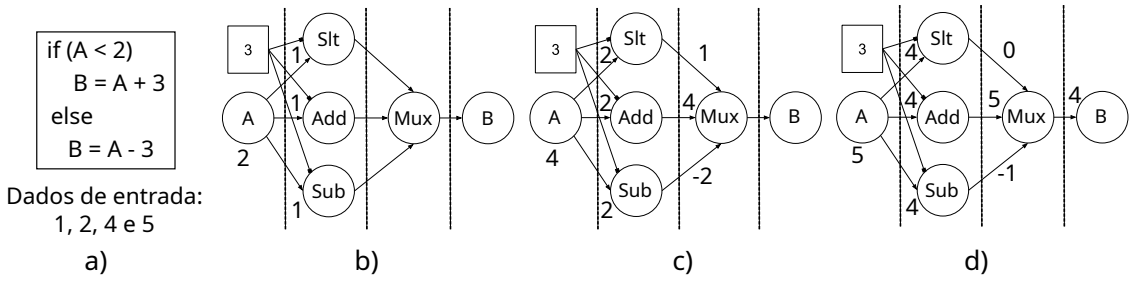


Figura 5.7: Um exemplo simples incluindo operadores condicionais: (a) aplicação de exemplo; (b, c, d) etapas do processamento do fluxo de entrada.

5.5 Biblioteca para chamada do acelerador CGRA

Muitas arquiteturas de CGRA não suportam trocas rápidas de contexto em tempo de execução, pois dependem de reconfiguração total ou parcial [66]. O READY, por sua vez, inclui a reconfiguração como parte do fluxo de execução de uma aplicação. Assim, instantes antes de iniciar a execução, as unidades são configuradas. Para

acelerar uma aplicação no READY, foi implementada uma biblioteca de *software* e *hardware* para as plataformas Intel HARPv2 e Alveo AMD Xilinx, detalhadas nas Seções 2.1.2 e 2.1.1 do Capítulo 2.

A biblioteca possui um conjunto de ferramentas para encapsular a complexidade do projeto. A Figura 5.8 ilustra um exemplo de código para chamada de um CGRA. O usuário declara e inicializa as variáveis de *host* correspondentes, que são *in*0, *in*1 e *out*0. Posteriormente, o CGRA carrega a aplicação no formato de um GFD mapeado, associando as variáveis do *host* com entradas e saídas. Finalmente, o usuário inicializa a execução do acelerador usando um dos dois métodos: o *syncExecute* e o *asyncExecute*. Semelhante a linguagem CUDA de programação das GPUs da NVidia, o aplicativo pode ser executado de forma síncrona, bloqueando o código do *host* até que a tarefa de aceleração termine (*syncExecute*), ou pode ser executado de forma assíncrona onde as tarefas do *host* se sobrepõem à execução do kernel do acelerador (*asyncExecute*).

```
1  ...
2  Cgra cgra;
3  short a[1024], b[1024], c[1024];
4  int len = 1024 * sizeof (short);
5  for (short i = 0; i < 1024; ++i) {
6    a[i] = 0; b[i] = 0; c[i] = 0;
7  }
8  cgra.loadCgraProgram(df->mapped);
9  cgra.setCgraProgramInputStreamByID(df->id, df->in0.id, a, len);
10 cgra.setCgraProgramInputStreamByID(df->id, df->in1.id, b, len);
11 cgra.setCgraProgramOutputStreamByID(df->id, df->out0.id, c, len);
12 cgra.syncExecute(0);
13 for (short i = 0; i < 1024; ++i) {
14  printf("%d", c[i]);
15 }
16 ...
```

Figura 5.8: Código C++: host In/Out Stream, objeto CGRA e método de chamada do acelerador.

Como já mencionado, o rótulo 6 na Figura 5.3 mostra uma linha do tempo de um rastreamento de execução, incluindo o carregamento das palavras de configuração do CGRA, a transferência de dados e a execução do acelerador e do *host*. O sistema inicia enviando as palavras de configuração através do canal de dados para configurar o acelerador de sobreposição. A execução começa assim que o primeiro dado chega, sobrepondo as etapas de transferência e de execução. O código do *host* pode ser executado simultaneamente no modo assíncrono. Durante a execução, caso as entradas de dados parem, toda a execução do CGRA é bloqueada até que os dados voltem a ser entregues para o acelerador.

5.6 Resultados experimentais

Nesta seção, apresentamos os resultados experimentais realizados para validar o arcabouço de *software* e *hardware* do READY nas plataformas HARPv2 e AWS F1. Esta seção está dividida da seguinte forma: na Seção 5.6.1, apresentamos os recursos FPGA usados pelo CGRA gerado para o caso de estudo. Na Seção 5.6.2, realizamos uma comparação do tempo de mapeamento do READY com outros trabalhos da literatura [56]. Na Seção 5.6.4, fazemos uma comparação do desempenho do READY em GOPS com um CGRA da literatura [65]. Na Seção 5.6.5, avaliamos o tempo de transferência de memória na AWS F1 e HARPv2 em função do tamanho dos dados usados pelas aplicações. Na Seção 5.6.6, comparamos o desempenho do acelerador READY com processadores Intel Xeon executando com um e oito *threads*. Finalmente, na Seção 5.7, realizamos um estudo para mostrar que ainda é possível melhorar o desempenho do CGRA aumentando o número de operações realizadas por PE.

5.6.1 Configuração para os experimentos

A plataforma da Intel HARPv2 possui um FPGA Arria 10 e um processador Xeon E5-2680 com 24 núcleos de 2,4GHz, além de 128 GiB de memória RAM. A plataforma AWS F1, por sua vez, conta com um FPGA AMD Xilinx UltraScale+ VU9P, um processador Xeon CPU E5-2686 com 8 núcleos a 2,30GHz e também 128 GiB de memória RAM. Mais detalhes sobre essas duas plataformas podem ser encontrados no Capítulo 2.

Uma arquitetura READY CGRA foi gerada contendo 128 PEs e uma rede ômega com *radix* 8 e um estágio extra. A palavra de processamento da arquitetura foi estipulada em valores inteiros com sinal de 16 *bits*. A Tabela 5.2 resume os recursos de FPGA usados para mapear o READY CGRA. Ele consome 33,2% e 16% do total de recursos de LUT para FPGAs HARPv2 e AWS F1, respectivamente. O CGRA requer 26,5% e 10,7% dos recursos de registradores para HARPv2 e AWS F1, respectivamente. Em relação aos recursos de DSP, o CGRA consome 8,4% e 1,8% para HARPv2 e AWS F1, respectivamente.

O FPGA da plataforma da AWS F1 tem aproximadamente o dobro de LUT e registradores em comparação com o FPGA do HARPv2 e 4× mais recursos de DSPs. O *clock* final é de 200 MHz e 250 MHz para HARPv2 e AWS F1, respectivamente. A configuração de tempo de execução rápido vem ao custo de área adicional para fornecer programabilidade mostrada nas duas últimas colunas na Tabela 5.2. Ele consome cerca de 30% e 40% da área total do acelerador no AWS FPGA e HARPv2, respectivamente. O custo indireto é devido à rede de interconexão totalmente

programável da arquitetura.

Tabela 5.2: Comparação de uso de recursos READY CGRA entre as plataformas Intel HARP v2 e AMD Xilinx AWS F1.

Plataforma	Área total do CGRA			*clock*	FMax	Sobrecarga de programabilidade	
	LUTs	Registradores	DSPs	alvo (MHz)	(MHz)	LUTs	Registradores
HARPv2 recursos	141.174	226.777	128	200	243,7	52.437	79.680
Porcentagem	33%	26,5%	8,4%			12,2%	9,3%
AWS F1 recursos	141.669	207.742	128	250	250,6	54.119	66.005
Porcentagem	16%	10,7%	1,8%			4,6%	2,8%

5.6.2 Avaliação do tempo de mapeamento e roteamento

O READY CGRA é sintetizado uma vez para o FPGA e pode executar um conjunto de aplicações descritas na forma de grafo de fluxo de dados. Nesta seção, é feita uma comparação do tempo de posicionamento e roteamento do READY CGRA com o CGRA proposto em [56] e com a aplicação mapeada diretamente no FPGA utilizando as ferramentas de síntese da AMD Xilinx. A Tabela 5.3 mostra o tempo de execução de posicionamento e roteamento para as ferramentas AMD Xilinx FPGA e CGRA [56] para mapear os *benchmarks* no formato de grafo de fluxo de dados. O CGRA proposto em [56] atinge uma velocidade três ordens de magnitude maior em comparação com as ferramentas AMD Xilinx. READY é ainda melhor para reduzir o tempo de execução do posicionamento e roteamento do CGRA [56] por um fator adicional de três ordens de magnitude.

A Tabela 5.3 mostra os tempos de execução nas colunas 2^o, 3^o, e 4^o. As duas últimas colunas mostram a aceleração em relação às ferramentas da AMD Xilinx.

Tabela 5.3: Tempo de execução do mapeamento e roteamento para as seguintes abordagens: ferramenta da AMD Xilinx FPGA, CGRA [56] e READY.

Benchmark	Ferramenta AMD Xilinx	CGRA [56]	READY	Aceleração do tempo de mapeamento CGRA [56]	READY
Chebyshev	240s	200ms	174,4μs	$1,2 \times 10^3$	$1,4 \times 10^6$
sgfilter	396s	200ms	162,4μs	$1,4 \times 10^3$	$2,4 \times 10^6$
mibench	245s	270ms	163,4μs	$0,9 \times 10^3$	$1,5 \times 10^6$
qspline	242s	170ms	162,1μs	$1,4 \times 10^3$	$1,5 \times 10^6$
Média	274,8s	223,3ms	165,6μs	$1,2 \times 10^3$	$1,7 \times 10^6$

5.6.3 Reconfiguração do CGRA READY

A Tabela 5.4 compara o fluxo de síntese tradicional de FPGA utilizando OpenCL e Maxeler [103], a reconfiguração parcial do FPGA apresentada em [62], ao CGRA proposto em [56] e ao READY CGRA. As abordagens com CGRA e reconfiguração

parcial requerem apenas uma síntese da arquitetura e depois as aplicações são mapeadas diretamente para essa arquitetura. Ao mesmo tempo, o fluxo de FPGA tradicional deve repetir a síntese para cada nova aplicação, conforme mostrado na linha *Compilação de módulo*, o que leva de minutos a horas para ser realizada. A reconfiguração parcial em FPGA (ou PR FPGA) também requer minutos para compilar uma única aplicação e inseri-la na região do PR no FPGA. Os CGRAs reduzem o tempo de compilação para a escala de milissegundos/microssegundos pelo mapeamento rápido.

Além disso, as abordagens baseadas em CGRAs geram um *bitstream* para o CGRA em vez de *bitstream* para o FPGA. Em relação ao tempo de reconfiguração do módulo, o Maxeler FPGA PR substitui qualquer módulo em milissegundos, pois o *bitstream* do módulo FPGA já está sintetizado. Ao mesmo tempo, os CGRAs requerem apenas microssegundos para realizar a reconfiguração do módulo. No entanto, o CGRA proposto em [56] não está disponível em plataformas com FPGA de alto desempenho como as Alveo da AMD Xilinx e Harpv2 da Intel.

Finalmente, o READY é o único que possui um sistema de reconfiguração parcial no nível CGRA em alguns nanossegundos. O READY permite reconfigurar uma única conexão ou elemento de processamento sem a necessidade de reconfigurar toda a arquitetura. Essa abordagem diminui o tamanho do *bitstream* do CGRA e acelera o processo de reconfiguração para a execução de uma nova aplicação.

Tabela 5.4: Tempo de design, compilação e reconfiguração em quatro abordagens: OpenCL/Maxeler fluxo tradicional para FPGA [103], Maxeler Reconfiguração Parcial [62], CGRA DSP [56] e READY.

Time	Maxeler OpenCL	Maxeler Reconfiguração Parcial (PR)	CGRA DSP	READY
Sínteses	horas	horas (1x)	horas (1x)	horas (1x)
Compilação de módulo	horas	34 min	223 ms	182 μs
Reconfiguração de módulo	-	5 ms	42 μs	36 μs
Em tempo de execução	-	sim	sim	sim
CGRA Configuração Parcial	-	-	-	32 ns
Implementação em FPGAs de alto desempenho	sim	sim	no	sim

5.6.4 Avaliação de desempenho

A avaliação de desempenho do READY foi realizada por meio de comparação com outras abordagens baseadas em CGRA sintetizados em FPGA. O CGRA apresentado em [56] relata apenas os resultados da simulação. Portanto, os trabalhos [65, 67], extensões do trabalho apresentado em[56], foram selecionados para realizar uma comparação entre implementações em FPGA.

Li et al.[65] propõem um sistema de um acelerador com um CGRA com uma matriz linear de unidades funcionais, denominado aqui de **TM CGRA**, e Li et al.[67]

estendem seu trabalho para integrar CGRA em um FPGA usando a interface PCIe, denominado aqui de **TM-PCI CGRA**. A Figura 5.9 mostra o desempenho em GOPS para comparar os **TM CGRA** e **TM-PCI CGRA** com o **READY** implementado na plataforma HARPv2 e na nuvem da AWS, usando quatro benchmarks propostos por Li et al. [65]. A abordagem **TM CGRA** usa um FPGA Zynq XC7Z020 a 233 MHz. A abordagem **TM-PCI CGRA** usa um núcleo Xeon E5-1650 e uma placa AMD Xilinx VC707 conectada usando um slot PCIe Gen2, com frequência de *clock* do FPGA de 250 MHz. O **READY** da AWS F1 atinge melhor desempenho em 250 MHz, conforme mostrado no gráfico da Figura 5.9, usando mapeamento espacial em vez de uma abordagem de multiplexação de tempo.

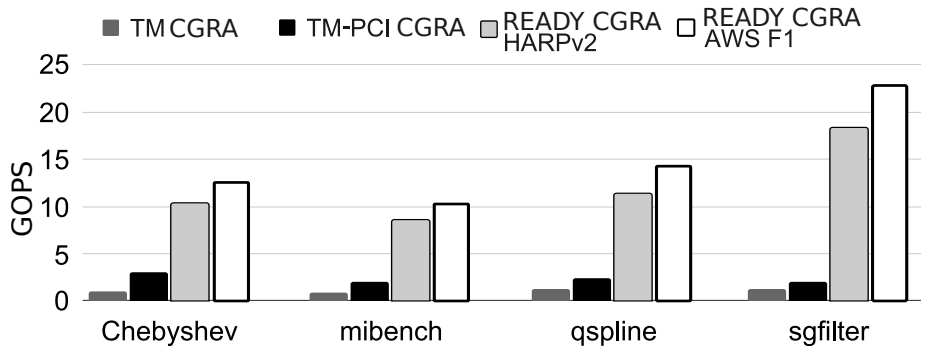


Figura 5.9: Comparação de desempenho em GOPS: TM CGRA [65], TM-PCI CGRA [67] e READY CGRA em HARPv2 e AWS F1. Maior é melhor.

5.6.5 Desempenho limitado pela Transferência de Memória na AWS e HARPv2

Como já mencionado, AWS F1 e HARPv2 possuem abordagens diferentes para conectar o FPGA e o processador host. Existe uma degradação do desempenho do READY em GOPS devido à largura de banda limitada das plataformas. Desse modo, foram avaliadas quatro aplicações: FIR, K-means, Sobel e Mul-Add, onde a transferência de dados entre o processador host e o acelerador influencia diretamente no desempenho.

A aplicação FIR é um Filtro de Impulso Finito com 63 *taps* e 124 operações, e K-means é o algoritmo de agrupamento não supervisionado com 8 grupo (K) e 4 dimensões(N). Sobel é o filtro de imagem de detecção de borda com uma máscara 3×3. Por fim, criamos um GFD de teste Mul-Add para avaliar o desempenho máximo usando todos os CGRA PEs e In/Outs do READY. O HARPv2 funciona melhor se o tamanho dos dados for menor que 64 MiB, graças à conexão QPI fortemente acoplada no processador e o uso de memória compartilhada. No entanto, se o tamanho dos dados for de pelo menos 128 MiB, o AWS F1 terá um desempenho melhor, pois a

frequência do *clock* é de 250 MHz em comparação com 200 MHz do HARPv2, como mostrado na Figura 5.10.

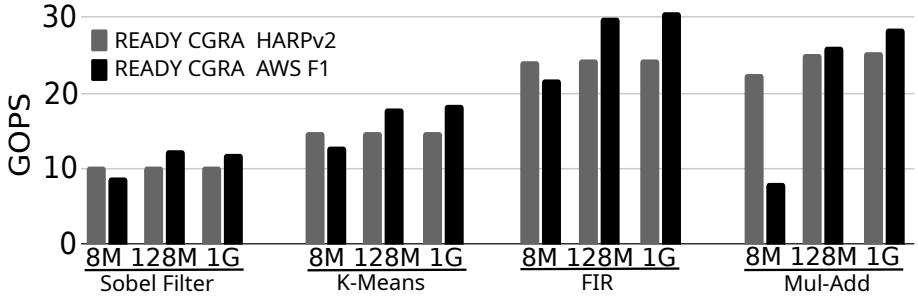


Figura 5.10: Comparação de desempenho em GOPS entre READY CGRA nas plataformas HARPv2 e AWS F1 em função do tamanho dos dados. Maior é melhor.

5.6.6 Desempenho de FPGA/CPU

Também foi avaliado o desempenho do READY em comparação com o processador Xeon de alto desempenho. A Figura 5.11 mostra o tempo de execução em milissegundos para processar 1 GiB de dados para os quatro aplicativos: FIR, K-means, Sobel e Mul-Add. O READY CGRA AWS F1 é, em média, 9,4 vezes mais rápido que o processador host com um único *thread*. Em comparação com a configuração do Xeon com oito *threads*, o READY CGRA F1 é 2,29 vezes mais rápido. Foram utilizadas oito *threads* do Xeon para deixar a comparação justa, uma vez que o CGRA READY usado para esse caso de estudo executa oito *threads* de forma temporal para ocultar a latência da rede, como mostrado na Seção 5.1.

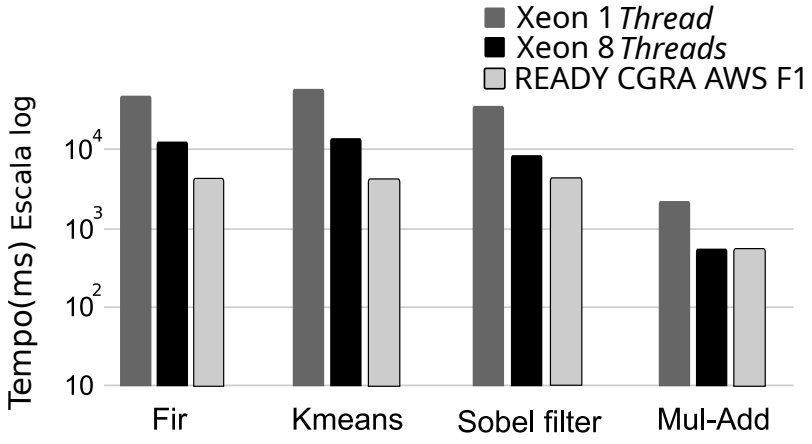


Figura 5.11: Tempo de execução em milissegundos para Xeon com um e oito *Threads* e READY CGRA na plataforma AWS F1 com 8 *threads*. Menor é melhor.

A Tabela 5.5 apresenta os valores de tempo de execução para a Figura 5.11. Os benchmarks FIR, K-means e Sobel têm quase o mesmo tempo de execução. A razão é

simples: todos os fluxos de dados são executados em um pipeline para produzir um resultado por ciclo de *clock*, e o fluxo de dados de entrada tem o mesmo tamanho de elementos de 1 GiB. As pequenas diferenças são devidas ao tempo de configuração, transferência de dados e profundidade do pipeline.

Tabela 5.5: Tempo de execução em segundos para processar 1GiB de dados para o READY CGRA na AWS F1 em comparação com CPU Xeon. Menor é melhor

		Tempo de execução(segundos)			Aceleração	
Benchmarks	Operações (%uso)	Xeon 1 *Thread*	Xeon 8 *Threads*	READY AWS F1	Xeon 8 *Threads*	READY AWS F1
Fir	124(97%)	49,41	12,83	4,31	3,80	11,50
K-means	75(58%)	59,18	13,99	4,32	4,60	13,70
Sobel filter	52(40%)	35,85	8,57	4,30	4,20	8,30
Mul-Add	128(100%)	2,22	0,54	0,55	4,10	4,00
Média		36,66	8,98	3,37	4,10	9,40

O FIR executa 124 operações a 250 MHz, resultando em um desempenho teórico previsível de $124 \times 250M = 31$ GOPS. O desempenho medido, mostrado na Tabela 5.5 para um fluxo de dados de 1 GiB, é de $\frac{124 \times 1G}{4.31} = 30.8$ GOPS, que está próximo do valor teórico previsto de 31 GOPS. Já o K-means tem uma baixa ocupação com 75 operações de 128 PEs (58% das unidades do CGRA), resultando em um desempenho teórico de $75 \times 250M = 18,7$ GOPS e desempenho medido de $\frac{75 \times 1G}{4,32} = 18,6$ GOPS. O K-means atinge a melhor aceleração, sendo $13,7\times$ mais rápido em comparação com o Xeon de oito *threads*. O código K-means possui comandos condicionais para selecionar o grupos, o que gera comandos de ramificação e diminui o desempenho do processador.

O código sintético Mul-Add possui oito fluxos de dados, reduzindo o tempo de execução do CGRA por um fator de 8. Apesar de ser um código simples, ele é vetorizado pelo compilador e melhora significativamente o desempenho do processador Xeon. Apesar disso, o READY atinge o mesmo desempenho que um processador Xeon de oito *thread*. Este benchmark foi proposto para validar e testar o CGRA usando um caso de execução que inclui todos os elementos de processamento e conexões de entrada e saída.

5.7 Aumentando o desempenho do CGRA

O READY CGRA realiza uma única operação por PE. Uma estratégia para melhorar o desempenho do CGRA é aumentar a granularidade do PE. Por exemplo, o CGRA proposto em [56] dobra o desempenho usando duas unidades de DSP por PE.

Para testar essa abordagem, um estudo de caso foi proposto, onde o PE do READY CGRA foi estendido para realizar até duas operações, conforme mostrado na Figura 5.12. Para testar essa abordagem, uma aplicação de teste foi implementada

para calcular a seguinte operação: $\prod_{i=1}^{16}(a * a + 3)_i$, onde a é o fluxo de dados de entrada. Com essa abordagem, o CGRA atinge um desempenho de 56,3 GOPS para um fluxo de dados de 1GiB. Essa implementação foi validada usando a plataforma AWS F1 com um CGRA executando a 250 MHz. Em comparação com a mesma operação executada na CPU, o CGRA foi $13,9\times$ e $2,5\times$ mais rápido para uma e oito *threads*, respectivamente.

Figura 5.12: (a) CGRA com 128 PEs com uma operação; (b) 128 PEs com duas unidades funcionais.

5.8 HLS em comparação a READY

Os dois principais fabricantes de FPGA, AMD Xilinx e Intel Altera, fornecem estruturas de síntese de alto nível para programadores de *software* C/C++. Essas ferramentas traduzem códigos C/C++ anotados com diretivas em códigos em linguagem de *hardware* como o Verilog, que são posteriormente sintetizados para o FPGA.

Nesta seção, foi avaliado o AMD Xilinx HLS [116], onde os resultados foram comparados com o READY em termos de tempo de execução. O AMD Xilinx HLS pode traduzir um código C/C++ em *hardware* sem nenhuma anotação. Porém, quando nenhuma anotação é realizada no código C++, a ferramenta não é capaz de usar bem os recursos da arquitetura do FPGA, como, por exemplo, o paralelismo espacial. A ferramenta AMD Xilinx HLS fornece um conjunto de anotações definidas como *pragmas* na linguagem C++ para otimizar o código de *hardware* gerado [117]. As mais comumente utilizadas são: *UNROLL*, *PIPELINE* e *DATAFLOW*.

A Tabela 5.6 apresenta o tempo de compilação, simulação e síntese de quatro benchmarks introduzidos na Tabela 5.5 usando a ferramenta AMD Xilinx HLS. A coluna **Síntese** mostra o tempo de síntese do código necessário para executar a aplicação no FPGA. Mesmo para uma aplicação simples, o tempo de compilação demanda mais de uma hora. A coluna **CPU Sim. Comp.** é o tempo gasto para criar um código para simular a aplicação em uma arquitetura x86, que é uma simulação de *software*. A coluna **HW Sim. Comp.** apresenta o tempo para gerar o código RTL para fornecer uma simulação de *hardware* com precisão de ciclo, incluindo todos os

circuitos necessários para emular a execução do FPGA. Neste caso, é observado que o tempo aumenta para alguns minutos.

Finalmente, as colunas **CPU Sim.** e **HW Sim.** relatam o tempo de simulação ao nível de *software* e *hardware*, respectivamente. A simulação de *hardware* é demorada; portanto, foram avaliados apenas um pequeno fluxo de dados de entrada com 512 dados.

Tabela 5.6: Resultados experimental do tempo com Vitis AMD Xilinx High-Level Synthesis para quatro benchmarks. O tempo de simulação foi medido para um fluxo de dados 512.

Benchmark	Tempo de Síntese	CPU Sim. Comp.	HW Sim. Comp.	CPU Sim.	HW Sim.
Fir	2h 27m	42s	7m 9s	0.12s	71s
K-means	2h 13m	41s	6m 44s	0.1s	88s
Sobel filter	2h 9m	42s	7m 38s	3.6s	42s
Mul-Add	1h 40m	41s	6m 40s	2.9s	38s

Vitis permite códigos em C++ puro e códigos com anotações usando diretivas de compilação. O código *C++ Puro* é quando o programador não procede com nenhuma alteração no código-fonte C++. Depois que o Vitis compila esse código, a ferramenta fornece comentários ao programador gerando tabelas de relatórios de *hardware*. No entanto, isto exige conhecimento de *hardware* para entender esses relatórios, como os conceitos de latência, pipeline e ciclos de *clock*. Além disso, o Vitis fornece formas de onda de sinais, que novamente exigem experiência de *hardware* para interpretar.

O programador pode usar algumas diretivas de alto nível (pragmas) para otimizar os números do relatório (latência, ciclos de *clock*). Alguns exemplos de pragmas são: *INTERFACE*, *ARRAY_PARTITION*, *UNROLL*, *PIPELINE* e *DATAFLOW*. O pragma *INTERFACE* define como o código C++ transfere código para o FPGA. Cada *pragma* tem alguns parâmetros. Por exemplo, o modo de operação de *INTERFACE* pode ser definido como *s_axilite*, que implementa todas as portas de comunicação através da interface AXI4-Lite. O pragma *ARRAY_PARTITION* otimiza o acesso aos vetores com a possibilidade de separá-los em blocos de memória ou mesmo em registradores individuais. O pragma *UNROLL* aplica otimizações de *loop*. O pragma *PIPELINE* sobrepõe as execuções de iteração do *loop*. Finalmente, o pragma *DATAFLOW* cria pipelines no nível da tarefa para aumentar o rendimento do acelerador em *hardware* [116].

A Figura 5.13 apresenta o tempo total de execução do Vitis HLS em C++ Puro e READY para os quatro kernels avaliados em instâncias FPGA AWS F1. Medimos o tempo de execução instrumentando o código C++ com funções da biblioteca padrão *std::chrono*. Os fluxos de dados READY executados no CGRA são, em média, duas ordens de magnitude mais rápidos que os kernels Vitis HLS C++ Puro. A arquitetura

CGRA otimiza a leitura/escrita de dados em rajadas de 512 *bits* da RAM do FPGA, enquanto o HLS utiliza apenas a largura de *bits* especificada na aplicação em C++, ou seja, 16 *bits*. Portanto, para gerar um acelerador eficiente, os programadores também devem considerar os recursos da interface da placa. Além disso, a interface de comunicação CGRA utiliza quatro bancos de memória e o HLS utiliza apenas um. O programador pode otimizar esses códigos usando pragmas e tipos de dados da API AMD Xilinx HLS. Estes ajustes de código estão fora do escopo deste trabalho.

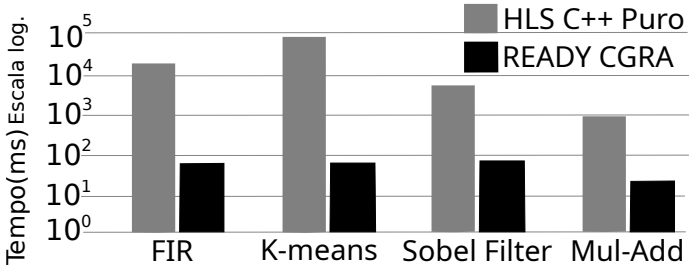


Figura 5.13: Tempo de execução em milissegundos para HLS C++ Puro e READY CGRA na AWS F1 para fluxos de dados de 8 milhões. Menor é melhor.

5.9 Discussões

Diferentemente dos trabalhos apresentados no Capítulo 4 que são aceleradores de aplicações específicas, este Capítulo se concentrou e desenvolver uma arquitetura de uma ferramenta para síntese de aceleradores genéricos chamada READY. READY é um arcabouço de *software* e *hardware* para geração e implantação de CGRAs como acelerador em plataformas heterogêneas CPU-FPGA. READY compila descrições de grafo de fluxo de dados de alto nível e gera automaticamente a configuração para um CGRA em tempo polinomial $O(n^2 \log n)$, onde n é o número de arestas do grafo de fluxo de dados. Este processo é três ordens de magnitude mais rápido do que trabalhos anteriores. READY também gera código HDL que pode ser mapeado e executado automaticamente em duas plataformas CPU-FPGA. Foram apresentados resultados de desempenho de um caso de estudo de uma arquitetura de CGRA com 128 unidades de processamento e 8 *threads*. Em geral, os resultados apresentados pelos trabalhos anteriores são baseados em simulações e não em um sistema real. Os resultados deste trabalho mostram ser possível mapear e executar aplicações de grafo de fluxo de dados eficientemente, e o desempenho do sistema resulta em uma aceleração de até 2x em comparação com um sistema moderno de processador *multithreading*. READY é extensível e flexível, permitindo uma exploração de diferentes arquiteturas. Além disso, READY é de código aberto e disponível para a

comunidade de pesquisa acadêmica, fornecendo uma base para mais pesquisas sobre aceleradores para sistemas reconfiguráveis CPU-FPGA.

Embora o READY CGRA tenha apresentado bons resultados de desempenho e tempo de compilação de aplicações em GFDs existe espaço para exploração de novas abordagens. Dessa forma, as seguintes lições foram retiradas deste capítulo:

- O uso de uma arquitetura de CGRA com rede global otimiza o tempo de síntese de aplicações em relação aos FPGAs.
- O uso de CGRA como uma camada virtual sobre o FPGA apresenta maior uso de recursos em relação a aplicações que usam abordagem de síntese direta.
- Especializar os PE para realizar operações mais complexas, aumentam o desempenho.
- O uso de uma rede global inviabiliza a escalabilidade do projeto, não permitindo um aumento na quantidade de PEs para obter mais desempenho.
- É importante avaliar diferentes formas de interconexão entre os PEs de um CGRA para aumentar a escalabilidade em relação à quantidade de PEs. Nesse sentido, uma ferramenta que auxilie os projetistas de *hardware* nessa tarefa é de grande valia.

Capítulo 6

Geração de Arquiteturas de Aceleradores Reconfiguráveis de Domínio Específicos

Este capítulo apresenta uma metodologia para a síntese de instruções e interconexões genéricas em aceleradores para sistemas CPU-FPGA. Como demonstrado no Capítulo 4, o uso de aceleradores para aplicações específicas apresenta desempenho superior em comparação com as mesmas aplicações executadas apenas na CPU. No entanto, como mencionado em capítulos anteriores, projetar aceleradores específicos impõe uma série de desafios, como o tempo de síntese das ferramentas de FPGA e o acoplamento do acelerador. Embora a abordagem adotada no Capítulo 5 apresente uma metodologia para superar esses desafios, ainda é necessário explorar novos aceleradores reconfiguráveis para problemas específicos. Isso implica em projetar instruções e interconexões personalizadas para cada novo acelerador, para minimizar o uso de recursos e otimizar a comunicação da arquitetura, permitindo escalar o tamanho dos aceleradores para obter melhores desempenhos.

O Capítulo 5 apresenta uma plataforma para desenvolvimento chamada READY, onde é possível reconfigurar a arquitetura para a execução de diferentes aplicações. READY é um arcabouço de *software* e *hardware* que possui um gerador que permite a geração de CGRAs variando apenas a composição dos elementos de processamento e os parâmetros internos da rede global de interconexão entre os PEs. Na Seção 5.7 do Capítulo 5, é demonstrado que é possível aumentar o desempenho da arquitetura especializando o conjunto de instruções, porém READY não permite esse tipo de especificação. Outra limitação do READY é o uso de uma rede global fixa (i.e. interconexão) que limita o aumento do número de PEs do CGRA, e, portanto, o desempenho da arquitetura resultante. Outras formas de interconexão entre PEs poderiam ser exploradas de modo a minimizar a comunicação entre os PEs para o dado problema específico. Além disso, o READY usa uma rede multiestágio que tem o número de conexões definido por uma potência de 2. O READY foi avaliado com

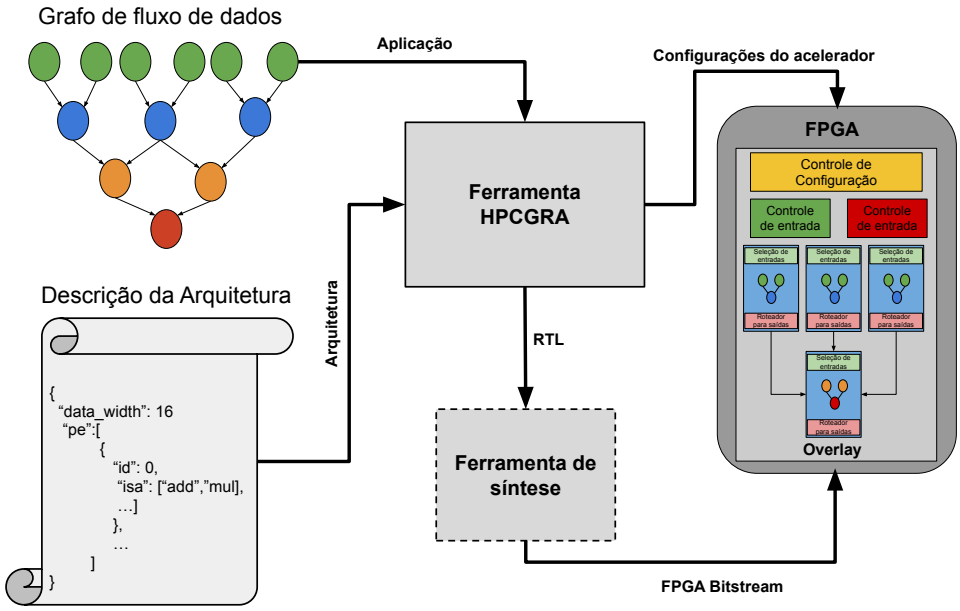


Figura 6.1: Visão Geral da Ferramenta HPCGRA.

256 conexões, se aumentarmos a arquitetura, o salto é para uma arquitetura de 512 conexões.

Este capítulo apresenta uma ferramenta chamada HPCGRA (*High Performance Coarse Grained Accelerator*), ou Acelerador Reconfigurável de Alto Desempenho, que permite superar, por meio de uma descrição em alto nível, ambas as limitações descritas anteriormente, ou seja: (a) a especificação de novas instruções; e (b) a exploração de interconexões genéricas entre os PEs. A Figura 6.1 apresenta uma visão geral da ferramenta HPCGRA. Dada uma descrição da arquitetura e uma aplicação no formato de grafo de fluxo de dados como entrada, a ferramenta é capaz de gerar o RTL para o acelerador correspondente, além da configuração necessária para a execução da aplicação no acelerador. Dessa forma, é possível desenvolver aceleradores reconfiguráveis que executam aplicações no formato de fluxo de dados com instruções e interconexões personalizadas, permitindo a aceleração de algoritmos em aceleradores reconfiguráveis especializados.

6.1 Arquitetura básica do acelerador

A arquitetura básica de um acelerador gerado pelo HPCGRA é composta por módulos de entrada e saída de dados, controle de configuração e execução e um conjunto de elementos de processamento. Todos os PEs podem ter uma ou mais entradas e saídas conectadas às filas externas de dados. Cada PE possui uma ALU (*Arithmetic Logic Unit*), ou Unidade Lógica e Aritmética, um roteador interno e uma unidade de

configuração. Essa estrutura básica é o núcleo de qualquer acelerador gerado pelo HPCGRA. No entanto, existem inúmeras formas de organizar os PEs. Cada PE pode implementar várias operações, então cabe ao projetista construir o acelerador que melhor se adapte ao conjunto de aplicações que se deseja acelerar.

O gerador do HPCGRA permite ao usuário descrever ortogonalmente todos os eixos apresentando na Figura 6.2(a) de uma arquitetura de forma independente. Para isso, uma entidade PE é modular, e o gerador adiciona automaticamente os recursos necessários para roteamento, configuração e computação. Como mostrado na Figura 6.2a) a ALU ① é o núcleo da arquitetura, onde são realizadas as computações. A partir da ALU, outros componentes são adicionados, conforme a descrição da arquitetura. Os componentes ② da fila elástica são conectados às portas de entrada da ALU, essas filas são usadas para adicionar atrasos reconfiguráveis nos caminho de dados de um PE. A dimensão de roteamento ③ é responsável por rotear as entradas do PE e a saída da ALU para as saídas do PE. A próxima dimensão é a interconexão ④, onde cada PE é conectado com outros PEs vizinhos ou com módulos de saída e entrada externas. E, finalmente, a dimensão ⑤ de configuração está vinculada a todos os componentes. Neste caso, cada componente é configurado independentemente, sendo possível reconfigurar apenas a operação que a ALU irá realizar sem mudar a configuração dos demais componentes.

Para permitir a configuração de cada componente individualmente, foi desenvolvido um mecanismo de reconfiguração parcial para o acelerador, permitindo uma reconfiguração eficiente e rápida. Esse mecanismo gera automaticamente um barramento de configuração que passa por todos os PEs da arquitetura em forma de pipeline. O barramento utiliza registradores para cada PE ao longo da rota, encurtando o caminho crítico. A Figura 6.2b) mostra o mecanismo implementado em uma arquitetura de exemplo no formato 2D. A largura do barramento de configuração pode ser personalizada, ou seja, o usuário escolhe se as palavras de configuração serão enviadas inteiras para os PEs ou de forma serial, permitindo trocar uso de recursos por tempo de configuração. Cada configuração é formada por três campos principais: um campo com o identificador único de cada PE, um campo para identificar qual módulo interno do PE a configuração é destinada e um campo com o dado da configuração. A largura de *bits* de todos esses campos forma a largura de *bits* da palavra de configuração completa. É importante observar que, devido à palavra de configuração que será enviada para todos os PEs, a maior palavra de configuração entre todos os PEs será utilizada. Isso ocorre porque cada PE pode possuir uma largura de palavra de configuração diferente, dependendo das funcionalidades implementadas em cada PE, que dependem do modelo projetado pelo usuário.

A estratégia de projeto do gerador torna a ferramenta muito versátil, uma vez que

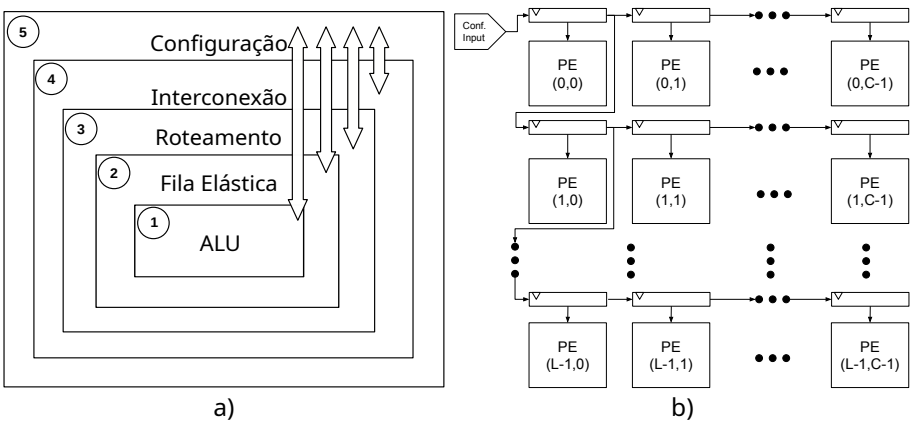


Figura 6.2: Configuração do acelerador: a) Abordagem ortogonal. b) Modelo de reconfiguração.

é possível construir aceleradores genéricos no formato tradicional de um CGRA ou até mesmo um acelerador específico no formato de uma aplicação em fluxo de dados. Isso pode ser feito de muitas maneiras, como, por exemplo, embarcar uma operação personalizada na ALU que realiza toda a computação de uma aplicação em um único PE, ou gerar o acelerador onde cada PE corresponde a um nó de computação do fluxo de dados da aplicação. Também é possível misturar essas duas abordagens, criando operações que são subconjuntos dos nós de um fluxo de dado de uma aplicação para os PEs e realizando a interligação entre os PEs.

A execução de uma aplicação em um acelerador segue sempre o mesmo processo. Primeiro, o acelerador recebe informações globais sobre quais filas de entrada e saída de dados serão utilizadas. Em seguida, o acelerador recebe todas as configurações dos PEs que serão utilizados na execução. Após essa etapa de configuração, as filas de entrada de dados são liberadas para enviar os dados para os PEs que possuem entradas e receber dados dos PEs que possuem saídas para serem escritos nas filas de saídas.

Assim, todo o controle de execução não depende do arranjo de PEs. Como os PEs são configurados de forma estática no início da execução, cada PE apenas processa os dados que chegam em suas entradas quando esses dados possuem um bit de validação configurado em um. Após o processamento, ele entrega em suas saídas os dados processados com um bit de validação configurado em um para que o próximo PE possa processar e assim por diante. Esses dados válidos são propagados por todos os PEs do acelerador conforme as configurações, até que cheguem a uma saída e sejam escritos nas saídas externas do acelerador.

6.2 Ferramenta HPCGRA

HPCGRA é uma ferramenta que permite a descrição detalhada de um acelerador por meio de uma descrição de alto nível, seguindo a notação de objetos da linguagem JavaScript (JSON)[58]. A partir dessa descrição, a ferramenta gera automaticamente todo o código RTL de um acelerador na linguagem Verilog. HPCGRA é um projeto de código aberto baseado na biblioteca Veriloggen[105], escrita em Python, que fornece recursos de alto nível para produzir código Verilog.

Especificamos um conjunto de elementos que podem ser utilizados na descrição do acelerador para realizar a transformação do código JSON em um modelo de *hardware*. É importante destacar que o formato proposto é ortogonal, o que significa que é possível definir as principais características da arquitetura seguindo os eixos apresentados na Figura 6.2a).

O formato desenvolvido permite a descrição individual de cada elemento de processamento. Essa abordagem foi pensada para projetar arquiteturas heterogêneas, onde cada PE pode possuir características únicas. Essa flexibilidade simplifica a elaboração de diferentes formatos de arquitetura de acelerador, seguindo o modelo básico apresentado na Seção 6.1. É possível definir o conjunto de operações para cada PE e também especificar seus vizinhos. A Seção 6.3 detalha cada campo do formato da descrição, desenvolvido de forma genérica e ortogonal.

A ferramenta possui um conjunto de arquiteturas predefinidas que podem ser geradas com uma única linha de comando, sem a necessidade de um arquivo JSON de entrada. Os usuários só precisam definir o número de PEs e qual o tipo da rede de interconexão, isso permite gerar CGRAs de arquiteturas conhecidas da literatura. A Tabela 6.1 mostra os principais modelos de interconexão disponíveis na ferramenta. Essa facilidade é útil para a prototipagem rápida de projetos homogêneos, nos quais todos os PEs possuem o mesmo conjunto de instruções.

Tabela 6.1: Padrões de interconexão para o gerador HPCGRA de alto nível.

Modelo	**Descrição**
Mesh	Cada PE se comunica com seus vizinhos adjacentes.
One Hop	Cada PE se comunica com seus vizinhos adjacentes e com os vizinhos adjacentes de seus vizinhos.
Chess	Igual ao *Mesh* para PEs ímpares e igual ao *One Hop* para PEs pares.
Diagonal	O mesmo que o *Mesh* com a adição dos vizinhos das diagonais.
Hexagonal	Cada PE possui 6 vizinhos, formando um padrão semelhante a uma colmeia.
Torus	Igual ao *Mesh*, mas os PE das bordas são vizinhos dos PEs das bordas opostas.

6.3 Especificação de formato

Nesta seção, apresentamos o formato proposto para descrever um acelerador, seguindo a notação de um arquivo no formato JSON. Temos cinco objetos principais definidos como chave e valor:

1. O campo **"data_width"** é um inteiro que define a largura, em *bits*, dos dados de processamento da arquitetura. Este valor é constante para uma arquitetura e não pode ser alterado após a geração do código HDL.

2. O campo **"conf_bus_width"** é um inteiro, em potência de dois, que define a largura, em *bits*, do barramento de configuração. Este valor pode variar de 1 bit até o máximo suportado, influenciando diretamente no tempo de reconfiguração da arquitetura, já que os dados de configuração serão serializados.

3. O campo **"axi_bus_data_width"** é um inteiro, em potência de dois, que define a largura, em *bits*, do barramento de comunicação com a memória externa. Para atribuir um valor a este campo, é necessário conhecer a plataforma alvo para a qual o acelerador será implantado.

4. O campo **"operations"** é um arranjo de objetos que definem operações personalizadas desenvolvidas pelo usuário para especializar uma arquitetura. Este campo é detalhado na Seção 6.4.

5. O campo **"pe"** é um arranjo de objetos que descreve cada elemento de processamento individualmente.

Um objeto **"pe"** contém sete atributos: *"id"*, *"num_istream"*, *"num_ostream"*, *"neighbors"*, *"routes"*, *"elastic_queue"* e *"isa"*. O atributo *"id"* é um identificador único de cada PE, sendo um valor inteiro que vai de zero até a quantidade de PEs menos um. Os atributos *"num_istream"* e *"num_ostream"* representam a quantidade de entradas e saídas externas que o PE possui, respectivamente. O atributo *"neighbors"* é uma lista de IDs de outros PEs que são vizinhos do PE atual, permitindo a geração de qualquer arquitetura de rede de interconexão entre os PEs. O atributo *"routes"* determina o roteamento interno de cada PE. Esse recurso é importante em arquiteturas de CGRAs 2D, mas é um recurso caro. Para facilitar a exploração de possíveis tipos de roteamento interno, foi desenvolvido um mecanismo para criar roteamentos personalizados. Neste atributo, o usuário pode inserir quantos roteamentos possíveis o PE pode realizar.

A Figura 6.3 demonstra dois exemplos de roteamento gerados. Na Figura 6.3(a), o valor passado para o atributo *"routes"* é igual a 1, denominado neste trabalho de *"one_routing"*. Dessa forma, o PE pode realizar o roteamento de apenas um dado,

que pode ser de um vizinho para outro, de uma entrada para um vizinho (caso o PE possua entrada externa) e de uma saída da ALU para um vizinho. Na Figura 6.3(b), é um exemplo onde o valor passado para o atributo *"routes"* é igual a N, onde N é o número de entradas provenientes de outros PEs vizinhos ou entrada externa. Neste caso, é possível realizar qualquer roteamento interno do PE para os PEs vizinhos, denominado neste trabalho de *"full_routing"*. É possível também passar o valor 0 para o atributo *"routes"*. Neste caso, o PE não realizará nenhum roteamento, repassando a saída da ALU para todas as saídas do PE, denominado neste trabalho de *"no_routing"*. Qualquer valor diferente de zero, um e N, implicará em um mecanismo que permitirá apenas o número passado de entradas a serem roteadas. Ou seja, caso o valor seja 3 e o PE seja vizinho de 6 PEs, é possível rotear um subconjunto de 3 vizinhos paralelamente.

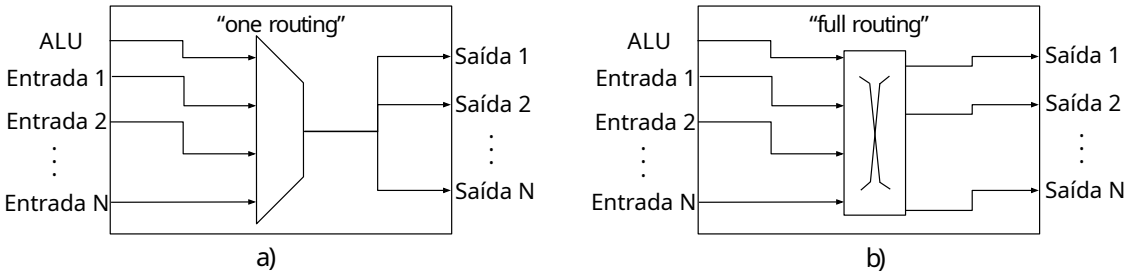


Figura 6.3: Mecanismo de roteamento PE. a) *"one_routing"*, b) *"full_routing"*.

Um problema comum em CGRAs com duas dimensões, é evitar atrasos incompatíveis após as etapas de posicionamento e roteamento (P&R) de uma aplicação [83]. Esse problema ocorre quando dois ou mais caminhos possuem tempos de chegada diferentes ao atingir um determinado PE, devido às decisões de rotas tomadas pelo P&R. Esse problema pode ser corrigido adicionando filas elásticas nas entradas do PE [83]. Essas filas são um recurso programável onde é possível definir seu tamanho em tempo de configuração. Assim, é possível criar atrasos em alguns caminhos de dados e resolver as incompatibilidades de chegada do dado a um PE geradas durante as etapas P&R. O atributo *"elastic_queue"* é um vetor de inteiros, onde cada índice do vetor é referente a cada entrada da ALU do PE. Em outras palavras, caso a ALU possua operações com três entradas, um vetor com três inteiros pode ser passado, com valores diferentes de tamanho de fila elástica para cada entrada.

O campo *"isa"* corresponde ao conjunto de instruções do PE. Um PE executa cálculos com dados provenientes dos PEs vizinhos, dados de entrada externos e com registradores internos que armazenam constantes. Na versão atual da ferramenta, foi adicionado um conjunto de instruções com as principais operações básicas. Estas operações são especificadas usando uma lista no campo *"isa"*, facilitando a escolha

das operações que cada PE pode realizar. As principais operações disponíveis são aritméticas e lógicas, rotuladas como *"add"*, *"sub"*, *"mul"*, *"and"*, *"or"*, *"not"*, *"madd"*, *"macc"*, *"addadd"*, *"subsub"*, *"addsub"*, e *"mux"*. Destaca-se a operação de multiplicação e acumulação (*macc*), comum em muitas aplicações como multiplicações de matrizes.

Para exemplificar o formato de entrada utilizado para gerar uma arquitetura de CGRA, a Figura 6.4 apresenta um exemplo de arquitetura para um acelerador CGRA 2D com 4 PEs e uma rede de interconexão tipo *Mesh*. Os PEs na primeira coluna são PEs de entrada, enquanto os PEs na última coluna são PEs de saída.

```
{
"data_width": 16,
"conf_bus_width": 8,
"axi_bus_data_width": 512,
"pe": [
    {
     "id": 0, "num_istream":1, "num_ostream":0,
     "routes":0, "elastic_queue":[0,0],
     "neighbors":[ 1, 2], "isa":[ "sub","add"]
    },
    {
     "id": 1, "num_istream":0, "num_ostream":1,
     "routes":0, "elastic_queue":[0,0],
     "neighbors":[ 1, 2], "isa":[ "or","and"]
    },
    {
     "id": 2, "num_istream":1, "num_ostream":0,
     "routes":0, "elastic_queue":[0,0],
     "neighbors":[ 1, 2], "isa":[ "or","and"]
    },
    {
     "id": 3, "num_istream":0, "num_ostream":1,
     "routes":0, "elastic_queue":[0,0],
     "neighbors":[ 1, 2], "isa":[ "mux","sub"]
    }
  ]
}
```

Figura 6.4: Exemplo de uma arquitetura de CGRA *Mesh* 2x2 descrita em JSON.

6.4 Formato para geração de operações personalizadas

Nesta seção, apresentamos o formato criado para a definição de operações personalizadas que podem ser adicionadas ao conjunto de instruções de um PE,

especializando assim uma arquitetura de acelerador. Para garantir compatibilidade com o formato utilizado na descrição do acelerador, adotamos o uso de objetos JSON.

Toda operação personalizada deve seguir o modelo de execução de um fluxo de dados, o que significa que um novo operador deve ter entradas processadas por nós de operações intermediárias antes de serem enviadas para algum nó de saída. Cada operação deve ter pelo menos uma entrada e uma saída, sendo que os nós intermediários podem ser operações básicas ou outras operações personalizadas. É importante destacar que um operador personalizado com mais de uma entrada e saída exigirá que um PE possa receber essas entradas de dados de fontes externas ou de outros PEs vizinhos e enviar os dados processados para outros PEs vizinhos ou para saídas externas. Portanto, um operador com N entradas e M saídas precisa garantir que o número de entradas externas mais o número de vizinhos seja maior ou igual a N, e que o número de saídas externas mais o número de PEs vizinhos seja maior ou igual a M.

Toda operação possui uma latência expressa em ciclos de *clock*. As operações básicas apresentam uma latência de um ciclo, enquanto as operações personalizadas podem exigir diferentes quantidades de ciclos. A definição da latência de uma operação personalizada é responsabilidade do projetista, que pode adicionar registradores ao operador, aumentando sua latência. Para garantir a sincronização no fluxo de dados da arquitetura, a ferramenta balanceia todas as operações, ajustando suas latências para que correspondam à da operação com maior latência.

Um objeto JSON que descreve um operador é composto por quatro atributos: *"opcode"*, *"inputs"*, *"outputs"* e *"dataflow"*. O atributo *"opcode"* é uma *string* correspondente ao nome único do operador. O campo *"inputs"* é um vetor de *strings* com nomes únicos para cada entrada. O campo *"outputs"* é um vetor de *strings* que contém rótulos de nós do operador, especificando o nome da saída de cada nó usando um ponto. Por exemplo, se um operador tem um nó interno com o rótulo *"add1"* e a saída do operador deve ser desse nó, o campo *"outputs"* deve conter a *string "add1.out0"*, onde "out0" é o nome da saída do nó *"add1"*. É importante notar que um nó interno de um operador também é considerado um operador. Para os operadores padrão da biblioteca do HPCGRA, as saídas são sempre nomeadas com "out" seguido de um inteiro a partir de 0. Por fim, o atributo *"dataflow"* é um vetor de objetos do tipo *"node"*. Um objeto *"node"* representa um nó no grafo de fluxo de dados do operador e possui quatro campos: *"type"*, *"label"*, *"inputs"* e *"outputs"*. O campo *"type"* é uma *string* com o nome de um operador, que pode ser um operador personalizado ou um operador básico da biblioteca padrão. O campo *"label"* é um rótulo em *string* para o nó, funcionando como um nome de objeto no escopo do fluxo de dados do operador. O campo *"inputs"* é um vetor de strings onde cada

posição corresponde a uma entrada do nó. Cada entrada é o nome de outro nó, seguido pela saída deste nó que fornecerá o fluxo de dados para o nó em questão no escopo do grafo de fluxo de dados. Se a entrada do nó vier de uma das entradas, então o nome dado à entrada do operador deve ser passado neste vetor. Por fim, o campo *"outputs"* é um vetor de strings com os nomes das saídas do nó, padronizadas com a *string* "out" seguida do número correspondente à saída, por exemplo, *"out0"* é a primeira saída do nó.

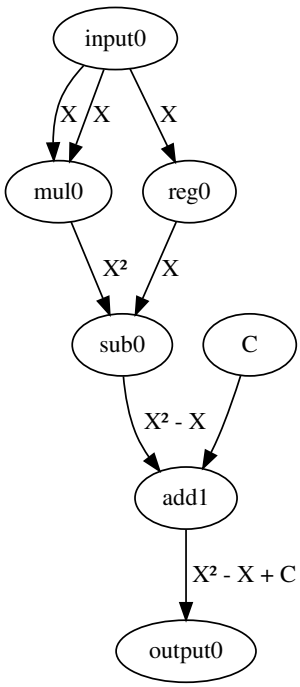
6.4.1 Exemplo de um operador personalizado

A fim de ilustrar melhor como é descrito um operador personalizado nesta subseção, é apresentado um exemplo de operador que realiza o cálculo de um polinômio de segundo grau com a seguinte fórmula:

$$X^2 - X + C \tag{6.1}$$

Neste caso, o polinômio possui uma entrada X e uma constante C, que precisa ser inicializada com algum valor constante. Este polinômio requer a realização de uma multiplicação, uma adição e uma subtração para cada valor de entrada. Essas operações podem ser visualizadas no formato de fluxo de dados na Figura 6.5a) e o código correspondente no formato JSON na Figura 6.5b).

Na Figura 6.5a), o primeiro nó do grafo de fluxo de dados é uma entrada que representa o valor de X da equação. Esse nó é representado na descrição do operador no campo *"inputs"* na Figura 6.5b). Este campo é um arranjo que contém todas as entradas do operador. Neste operador que possui apenas uma entrada ela foi nomeada como "in0". Uma vez que a entrada é feita o próximo nó representa a operação X^2. Para isso é utilizada uma multiplicação onde o valor de X é passado para as duas entradas do operador. Esta operação é descrita no objeto nó do campo *"dataflow"* da Figura 6.5b), que possui o rótulo *"mul0"*. Para balancear o grafo de fluxo de dados da operação um nó registrador é adicionado para a entrada *"in0"* no nó *"reg0"* que é descrito no objeto de rótulo *"reg0"* na descrição. Com os valores de X^2 e X representados nas arestas do grafo da Figura 6.5a), é realizada a operação de subtração desses dois valores de acordo com Equação 6.1. Essa operação é descrita pelo objeto de rótulo *"sub0"* na Figura 6.5b). Em seguida é realizada a adição da saída do nó *"sub0"* com o valor constante *"C"*. Este valor é configurado em tempo de execução, para isso um nó do tipo *"const"* com rótulo *"c0"* é descrito no formato da Figura 6.5b). Por fim para a saída com resultado final da equação a saída do nó de rótulo *"add1"* é mapeada para a saída do operador, isto é descrito no campo *"outputs"* do formato da Figura 6.5b), onde a saída *"out0"* do nó *"add1"* é passado no campo *"outputs"*.




```
{
 "opcode": "poly2",
 "inputs": ["in0"],"outputs":["add1.out0"],
 "dataflow": [
   {
     "type": "mul","label":"mul0",
     "inputs":["in0","in0"],"outputs":["out0"]
   },
   {
     "type":"reg","label":"reg0",
    "inputs":["in0"],"outputs":["out0"]
   },
   {
     "type":"sub","label":"sub0",
     "inputs":["mul0.out0","reg0.out0"],"outputs":["out0"]
   },
   {
     "type":"const", "label":"c0",
    "inputs":[0],"outputs":["out0"]
   },
   {
     "type":"add","label":"add1",
     "inputs":["sub0.out0","c0.out0"],"outputs":["out0"]
   }
  ]
}
```

b)

Figura 6.5: Exemplo de um operador personalizado. a) Grafo de fluxo de dados da equação 6.1. b) Descrição do operador no formato JSON proposto.

6.5 Formato de configuração para uma arquitetura genérica

Nesta seção, é apresentado um código de montagem para configurar qualquer arquitetura de um acelerador gerado pelo HPCGRA. O código de montagem simplifica a configuração para um acelerador em alto nível. Essa linguagem de montagem é usada como um formato intermediário, permitindo que aplicações escritas em diferentes linguagens sejam compiladas para este formato, tornando esta ferramenta mais versátil. Os campos das instruções da linguagem são dependentes da definição da arquitetura, ou seja, a linguagem é única para cada modelo de arquitetura.

A linguagem possui três tipos de instruções: uma instrução relativa à operação que um PE executa na ALU, uma instrução relativa ao roteamento interno de um PE e uma instrução para configurar dados de propriedades de execução de um acelerador. Cada instrução possui um formato específico, como mostrado na Figura 6.6. O primeiro formato é usado para configurar qual operação um PE realiza e depende do conjunto de operações do PE em que ela é utilizada. Nos exemplos das linhas 2, 3 e 4 da Figura 6.6, as operações *add*, *sub* e *mul* serão configuradas nos PEs com identificadores iguais a 0, 3 e 5, respectivamente. Cada exemplo possui argumentos distintos: na linha 2,

os argumentos são da entrada externa 0 com a fila elástica de tamanho 1 e os dados provenientes do PE com identificador 1; na linha 3, os argumentos são de dados provenientes dos PEs com identificadores 2 e 3; e na linha 4, o segundo argumento é uma constante literal de valor igual a 10.

O segundo formato, mostrado na linha 6 da Figura 6.6, é para a instrução de roteamento. Ela possui três argumentos: o primeiro é o identificador do PE no qual ela será atribuída, o segundo é a origem do dado que será roteado e o terceiro é o destino para o qual o dado será enviado. A origem pode ser de uma saída da ALU do PE (como no exemplo da linha 7 da Figura 6.6), de uma entrada externa (como no exemplo da linha 8 da Figura 6.6) ou de um identificador de um PE vizinho (como no exemplo da linha 9 da Figura 6.6). O campo destino pode ser um identificador de um PE vizinho (exemplos das linhas 7 e 8 da Figura 6.6) ou para uma saída externa (como no exemplo da linha 9 da Figura 6.6).

O terceiro formato, mostrado na linha 11 da Figura 6.6, é para a instrução de configuração de propriedade. Esta instrução é usada principalmente para configurar constantes em operadores personalizados durante a etapa de configuração do acelerador. Ela possui três argumentos: o primeiro é o identificador do PE no qual essa configuração será atribuída, o segundo é a propriedade e o terceiro é o valor. No exemplo da linha 12 da Figura 6.6, a instrução é usada para configurar o valor 2024 na constante de nome "c0"para a operação personalizada poly2, criada como exemplo na Seção 6.4.

```
1  1° Formato : <operação> $<ID do PE> [ #<tamanho da fila> ] $<1° argumento> ...
2      Exemplos:   add $0 #1 $istream[0] $1
3                  sub $3 $2 $5
4                  mul $5 $7 10
5
6  2° Formato :  route $<ID do PE> $<origem> $<destino>
7      Exemplos: route $0 $alu[0] $3
8                route $0 $istream[0] $4
9                route $0  $1 $ostream[0]
10
11 3° Formato : set $<ID do PE> <propriedade> <valor>
12     Exemplo : set $10 poly2.c0 2024
```

Figura 6.6: Formato das instruções de configuração de um acelerador com exemplos de uso.

Para ilustrar como é configurada uma aplicação, a Figura 6.7 apresenta um exemplo que calcula um polinômio de terceiro grau $X^3 + X^2 - X + C$. Na Figura 6.7(a), temos o grafo de fluxo de dados do polinômio. Na Figura 6.7(b), o código na linguagem de configuração. Na Figura 6.7(c), uma arquitetura de CGRA com 4 PEs configurados para executar a aplicação. Todos os PEs do CGRA podem realizar 3

operações: "add", "mul"e "poly2". A instrução "poly2"é uma operação personalizada, conforme exemplo na Seção 6.4. O uso da operação "poly2"permite que apenas 4 PEs realizem o cálculo de um polinômio de grau 3. Observa-se que o grafo da Figura 6.7(a) possui 5 nós de operações, mas, como a arquitetura possui a operação "poly2"embarcada, precisamos apenas calcular o valor de X^3 e somar o resultado com o resultado da operação "poly2".

Para isso, no PE0, é inserido o valor de X, multiplicado por ele mesmo na ALU, ao mesmo tempo, em que é roteado para os PEs 1 e 3, conforme as instruções mostradas nas linhas 1 e 3 do código da Figura 6.7(b), e destacado nas linhas azuis da Figura 6.7(c). No PE0, também é roteado o valor calculado de X^2 da ALU para o PE2, conforme a linha amarela da Figura 6.7(c) e descrito na linha 2 do código da Figura 6.7(b). No PE1, é realizada a operação personalizada "poly2"com o valor de X proveniente do PE0, descrito na linha 4 do código da Figura 6.7(b) e roteado o resultado para o PE3, destacado na cor vermelha na Figura 6.7(c). No PE2, é calculado X^3 multiplicando o X^2 calculado no PE0 com o valor de X proveniente do PE3, descrito na linha 6 do código da Figura 6.7(b). Para garantir que os dados sejam calculados corretamente, é necessário um balanceamento no caminho do X^2 que chega antes do valor de X no PE2. Para isso, foi configurada uma fila elástica na entrada da ALU com tamanho 1 na porta de entrada do valor de X^2. Na linha 7 do código da Figura 6.7(b), a saída da ALU do PE2 é roteada para o PE3, destacado na cor verde na Figura 6.7(c). Por fim, no PE3, é realizado o cálculo final somando o valor de X^3 com o valor de $X^2 - X + C$ calculado no PE1 e roteando o resultado da ALU para a saída externa do CGRA, descrito nas linhas 8 e 9 do código da Figura 6.7(b) e destacado na cor roxa na Figura 6.7(c). Na linha 10 do código da Figura 6.7(b), é configurado o valor 2024 para a constante C do polinômio.

6.6 Avaliação

Esta seção apresenta as avaliações realizadas com a ferramenta HPCGRA. Para avaliar os resultados de síntese, foram geradas arquiteturas de CGRAs 2D com números de PEs em linha e colunas iguais a 9x9, 18x18 e 36x36. Para cada tamanho, foram considerados quatro tipos de interconexões entre os PEs: o modelo Mesh, One-hop, Diagonal e Hexagonal. Todas as arquiteturas foram sintetizadas utilizando a infraestrutura da Intel/Altera, com o FPGA Arria 10AX115U3F45E2SGE3 como plataforma alvo. Este FPGA possui 427.200 blocos lógicos (ALMs) e 1.518 blocos de processamento de sinal digital (DSPs). O *clock* alvo para todas as sínteses foi definido como 400MHz, a qual é a frequência máxima suportada pelo FPGA Arria 10.

Para validar a escalabilidade do gerador, foi sintetizado uma arquitetura de CGRA com um arranjo de 46x66 PEs e com palavras de processamento de 4 *bits*.

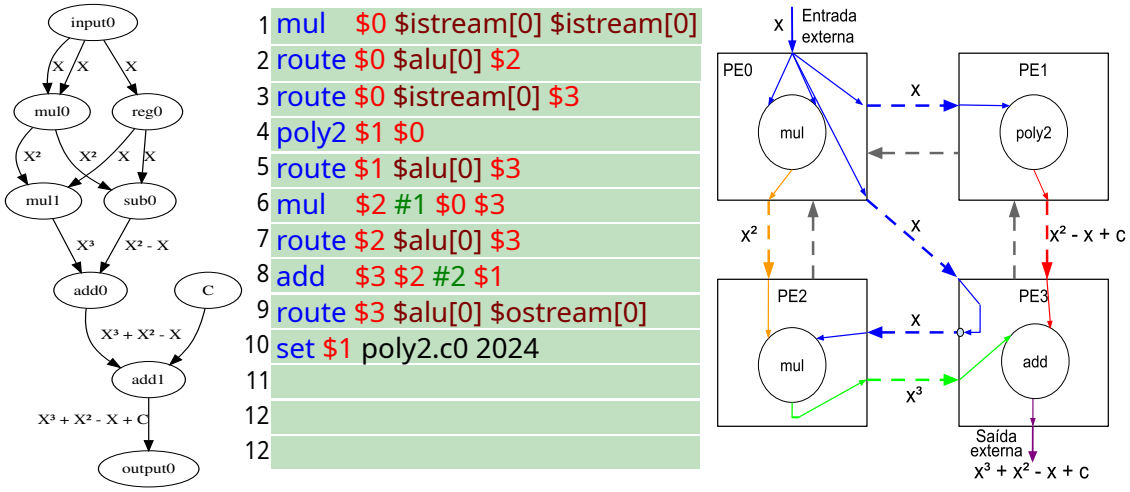


Figura 6.7: Exemplo de código de configuração de uma aplicação para calcular polinômio de terceiro grau. a) Grafo de fluxo de dados da aplicação. b) Código de configuração. c) Arquitetura de um CGRA com 4 PEs.

Como o FPGA Arria 10 possui apenas 1518 DSPs, apenas metade dos PEs realiza multiplicação. O CGRA utilizou 39% dos ALMs e 100% dos DSPs, alcançando uma frequência máxima de 403 MHz. O desempenho teórico desta arquitetura pode ser calculado da seguinte forma: cada PE realiza uma operação a cada ciclo de *clock*, então o número de PEs multiplicado pelo tempo de ciclo em segundos é o número de operações por segundo que a arquitetura pode realizar. Desta forma, o desempenho teórico desta arquitetura é $46 \times 66 \times 400 = 1214400$ MOPS ou 1,2 TOPS, mostrando que o código gerado é escalável.

A Tabela 6.2 apresenta uma comparação entre três tamanhos de CGRAs gerados, cada um com três tipos diferentes de roteamento interno. É importante destacar que as colunas ADRES e HyCube representam CGRAs da literatura, e os dados foram coletados a partir do trabalho apresentado em [108]. A arquitetura ADRES é compatível com o nosso modelo de roteamento *"one_routing"*, como mostrado na Figura 6.3(a). Neste trabalho, essa abordagem utiliza 32% menos ALMs por PE em comparação com o melhor resultado apresentado no trabalho anterior [108]. O modelo HyCube é compatível com a implementação *"full_routing"* mostrada na Figura 6.3(b), onde a abordagem deste trabalho é aproximadamente 50% mais eficiente em recursos do que em [108]. O gerador proposto neste trabalho é escalável e permite que os desenvolvedores aumentem o número de elementos, mantendo a mesma média de uso de recursos por PE, conforme mostrado na Tabela 6.2.

Também foi realizada uma exploração de projetos com várias configurações, variando os padrões de interconexão, recursos de roteamento e recursos de filas elásticas. A Figura 6.8 mostra o uso dos recursos das arquiteturas *Mesh*. As Figuras

Tabela 6.2: Uso de recursos para CGRAs de 16 *bits*. ADRES e HyCube usando FPGA STRATIX 10 [108] e o trabalho proposto usando FPGA ARRIA 10. NR denota "no_routing", OR denota "one_routing"e "FR"denota "full_routing".

	ADRES	HyCube	Our NR	Our OR	Our FR	Our NR	Our OR	Our FR	Our NR	Our OR	Our FR
Tam. Array	4x4	4x4	9x9	9x9	9x9	18x18	18x18	18x18	36x36	36x36	36x36
ALM	5051	5664	6400	11954	13827	26801	49589	58111	108709	202643	237248
DSP	32	32	81	81	81	324	324	324	1296	1296	1296
Média (ALM/PE)	216	330	79	147	170	82	153	179	83	156	183

6.8(a) e 6.8(b) correspondem aos modelos *"no_routing"* e *"full_routing"*, respectivamente. Cada CGRA possui tamanhos de filas elásticas nas entradas da ALU de 0, 2 e 4. Os PEs com filas elásticas maiores que zero possuem uma latência em ciclos de *clock* maior em seu caminho de dados, diminuindo o caminho crítico e melhorando o desempenho da arquitetura, porém o uso de recursos de *hardware* é maior. Observa-se no gráfico da Figura 6.8(b) que a frequência aumenta para CGRAs que possuem filas elásticas de tamanhos 2 e 4, mas diminui para aquelas com tamanho 0.

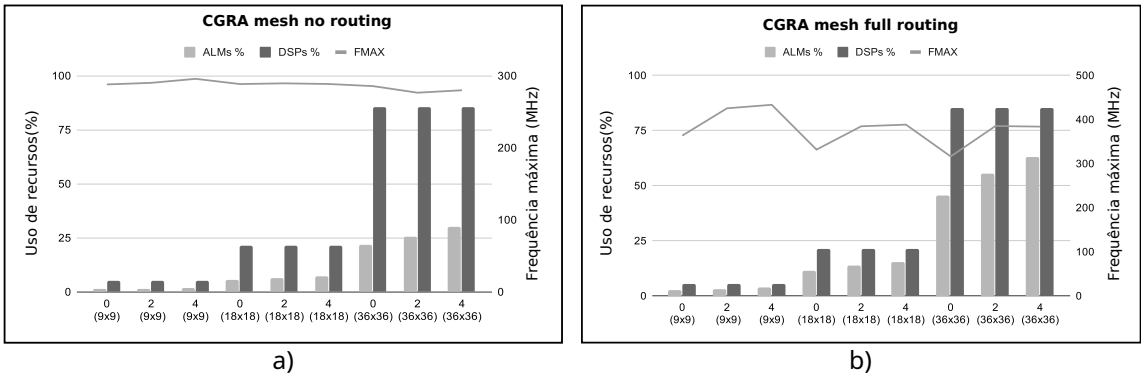
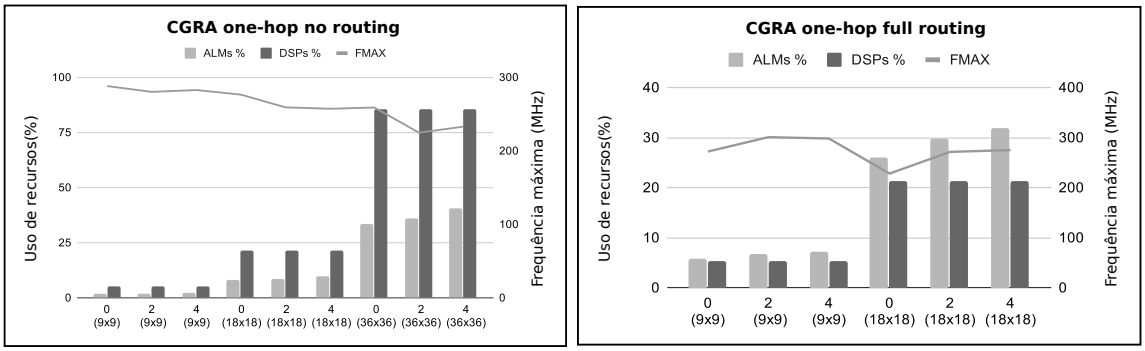


Figura 6.8: Gráfico de uso de recursos e uma frequência máxima. Os gráficos a e b referem-se ao modelo de interconexão *Mesh*; a) sem roteamento interno no PE; b) com todas as possibilidades de roteamento.

A Figura 6.9 ilustra o uso dos recursos da arquitetura *One-hop*. As Figuras 6.9(a) e 6.9(b) correspondem aos modelos *"no_routing"* e *"full_routing"*, respectivamente. Cada CGRA possui tamanhos de filas elásticas nas entradas da ALU de 0, 2 e 4. Os PEs com filas elásticas maiores que zero apresentam um aumento na frequência de *clock*, seguindo o mesmo padrão encontrado na síntese das arquiteturas *Mesh*. O modelo *One-hop* se comunica com mais dois vizinhos em relação ao modelo *Mesh*, aumentando o uso de recursos de *hardware* por PE. Devido a esse fato, a ferramenta de síntese não conseguiu sintetizar CGRAs com roteamento total para o tamanho 36x36.

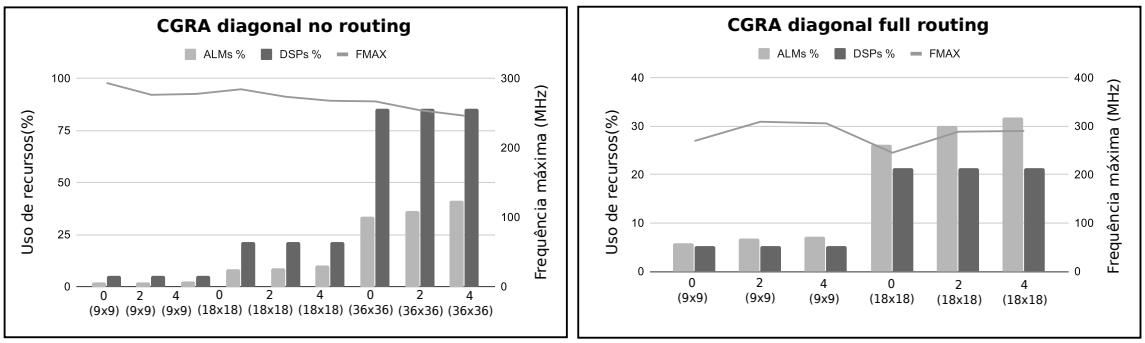
As Figuras 6.10(a) e 6.10(b) ilustram o uso de recursos para o modelo *Diagonal*. Neste modelo, cada PE tem oito vizinhos. Embora o modelo *Diagonal* tenha mais



a) b)

Figura 6.9: Gráfico de uso de recursos e frequência máxima. Os gráficos a e b referem-se ao modelo de interconexão *One-hop*; a) sem roteamento interno no PE; b) com todas as possibilidades de roteamento.

capacidade de roteamento do que o modelo *Mesh*, ele também consome mais recursos. Na versão com roteamento completo *"full_routing"*, apenas os tamanhos 9x9 e 18x18 foram sintetizados pela ferramenta de síntese. Já a versão sem roteamento *"no_routing"* apresentou uma redução na frequência conforme o tamanho da arquitetura aumentou.



a) b)

Figura 6.10: Gráfico de utilização de recursos e frequência máxima para o modelo de interconexão CGRA Diagonal; a) sem roteamento interno no PE; b) com todas as possibilidades de roteamento.

As Figuras 6.11(a) e 6.11(b) apresentam os resultados para a arquitetura *Hexagonal*. Nesta arquitetura, cada PE possui seis vizinhos, semelhante ao modelo *One-Hop*. No entanto, o padrão de conexão entre vizinhos é semelhante ao de uma colmeia. Os CGRAs com este padrão de conexão têm frequência semelhante ao modelo *One-Hop*, mas com menor variação ao aumentar o número de PEs e a quantidade de latência das filas elásticas para a versão sem roteamento. Para a versão com roteamento total, a ferramenta sintetizou CGRAs com tamanho 36x36, porém houve uma queda na frequência de *clock*.

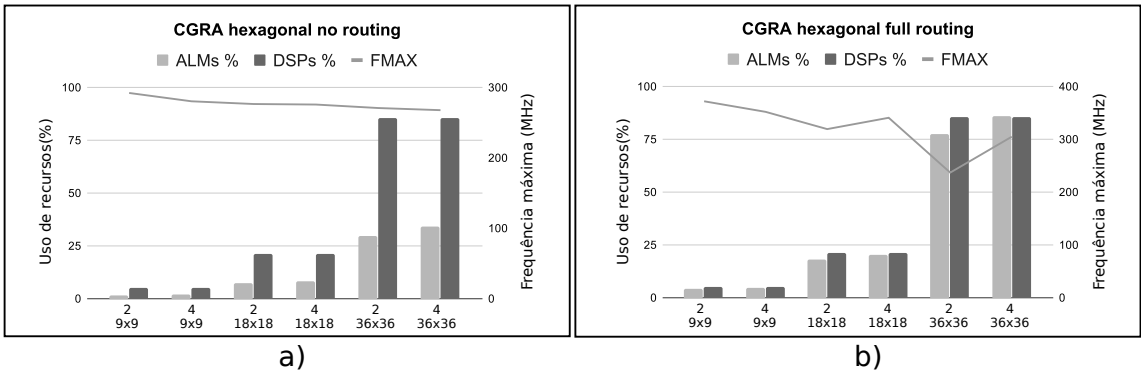


Figura 6.11: Gráfico de utilização de recursos e frequência máxima para o modelo de interconexão CGRA *Hexagonal*; a) sem roteamento interno no PE; b) com todas as possibilidades de roteamento.

6.7 Estudo de caso de uma arquitetura de CGRA especializada em *K-means*

Este estudo de caso visa utilizar a metodologia proposta neste capítulo para desenvolver um acelerador especializado para executar o algoritmo *K-means*. A arquitetura é um CGRA com uma rede de interconexão personalizada, permitindo a conexão entre blocos de operações personalizadas que aceleram o *K-means* com diferentes números de grupos (K) e dimensões (N). A arquitetura foi denominada KCGRA (*K-means Coarse Grained Reconfigurable Architecture*), ou Arquitetura de *K-means* Reconfigurável de Grão Grosso. Diferente da abordagem de síntese direta apresentada no Capítulo 4, este acelerador não requer uma nova síntese para a execução do *K-means* com novos valores de N e K, sendo necessária apenas a reconfiguração da arquitetura em tempo de execução.

A arquitetura do KCGRA consiste em um conjunto de PEs, cada um com dois operadores que encapsulam uma série de operações básicas. O primeiro operador, chamado "Kmeans2xN", realiza a classificação em relação a dois centroides para um ponto de entrada P com até N atributos e dois grupos $K = 2$. O segundo operador, denominado "Filtro", realiza a junção/redução de dois operadores do "Kmeans2xN". Por exemplo, podemos construir uma árvore com três operadores "Filtro"para agrupar quatro operadores "Kmeans2xN"e executar um *K-means* com oito grupos ($K = 8$) e N atributos.

A Figura 6.12(a) apresenta o diagrama geral da arquitetura do KCGRA, onde os operadores em vermelho são do tipo "Kmeans2xN"e os operadores em azul são do tipo "Filtro". O acelerador pode ter até N entradas $I_0, I_1, \ldots, I_{N-1}$, que representam os atributos do ponto a ser classificado. Cada operador "Kmeans2xN"com dois centroides seleciona o centroide mais próximo. Nos próximos níveis, teremos uma

árvore de redução com operadores do tipo "Filtro"que farão a seleção do centroide final mais próximo.

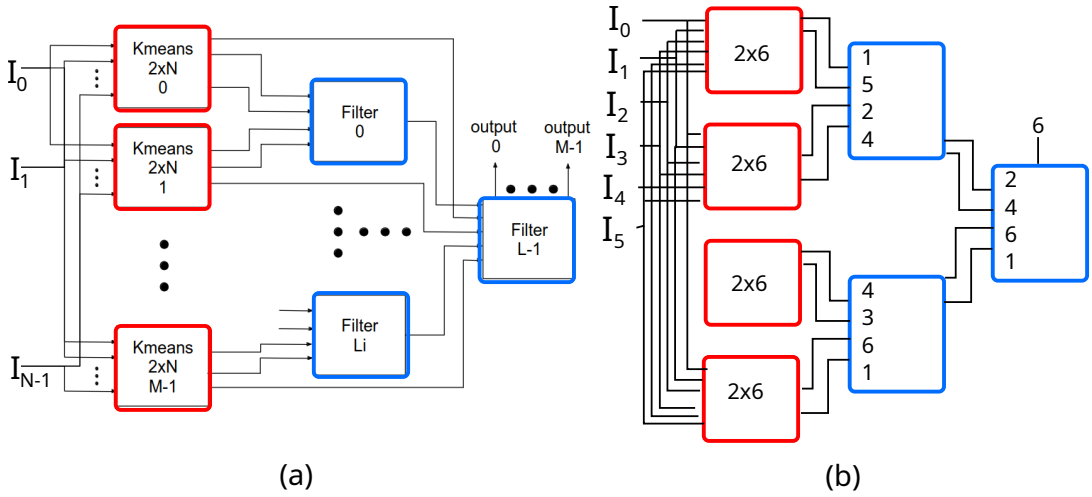


Figura 6.12: (a) Arquitetura genérica para um KCGRA com N atributos; (b) Exemplo de KCGRA para no máximo 6 atributos $N = 6$ e 8 grupos.

A Figura 6.12(b) apresenta um exemplo com 8 grupos ($K = 8$) e 6 atributos ($N = 6$). Cada operador "Kmeans2x6"tem 2 saídas que informam qual é o grupo mais próximo e qual é o valor da distância. Suponha que os centroides estejam distribuídos de cima para baixo na Figura 6.12(b), com o primeiro operador contendo C_0 e C_1, o segundo com C_2 e C_3, o terceiro com C_4 e C_5, e o último com C_6 e C_7. Neste exemplo, os centroides C_1, C_2, C_4 e C_6 foram selecionados com as distâncias 5, 4, 3 e 1, respectivamente. O primeiro filtro escolhe entre C_1 e C_2; como C_2 está mais próximo, com a distância 4, é propagado para o próximo filtro. O mesmo ocorre com C_6, que tem distância 1. Como C_6 está mais próximo que C_4, a saída tem o valor 6 como resposta.

6.7.1 Operadores *K-means*

A Figura 6.13(a) mostra um elemento de processamento com um operador "K-means2xN"com $N = 32$. O elemento de processamento pode ter até 32 entradas externas para receber os atributos. Internamente, o operador tem dois centroides, cada um com N dimensões, e um identificador ID para cada centroide. Desse modo, o operador irá calcular todas as distâncias, somá-las e retornar qual dos dois centroides é o mais próximo. O operador pode ser ajustado para um valor N entre 1 e 32 para o número de atributos. Além disso, pode ser configurado para calcular 1 ou 2 centroides. A saída do operador informa a menor distância e o ID do centroide mais próximo. Esses dados serão repassados para os operadores de "Filtro"para finalizar a classificação.

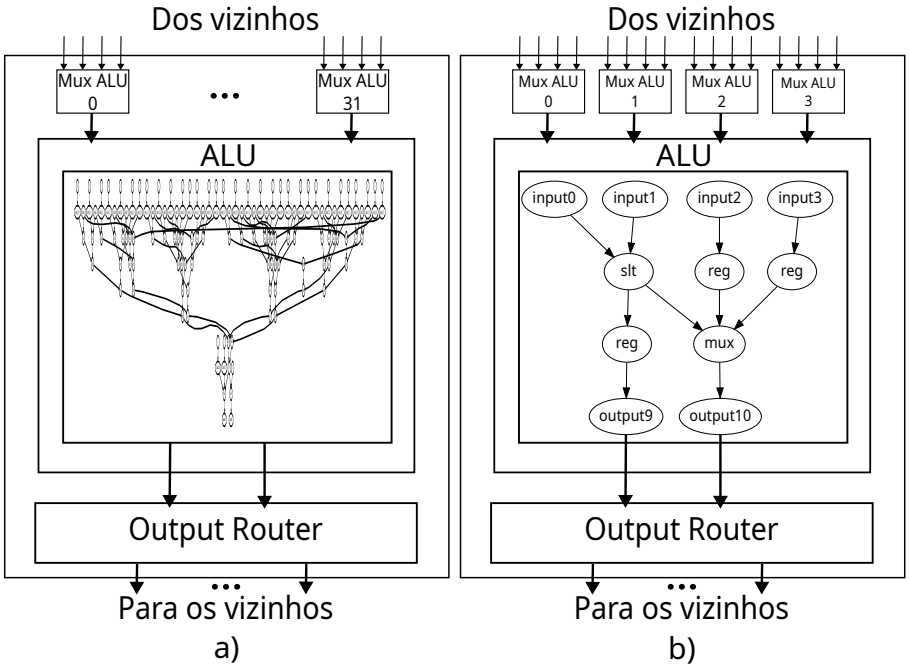


Figura 6.13: Arquitetura dos elementos de processamento do KCGRA.

A Figura 6.13(b) mostra o operador "Filtro"com mais detalhes. Ele recebe dados de 2 operadores "Kmeans2xN". Esses dados são comparados e o operador repassa o índice da menor distância e o ID do centroide mais próximo. Os operadores "Filtro"podem ser configurados para formar uma árvore de redução, entregando no final a classificação do ponto de entrada.

6.7.2 Modo de Reconfiguração do KCGRA

O KCGRA oferece modos de configuração para se adaptar a diferentes valores de K e N, gerando replicações do acelerador que podem classificar mais de um ponto por vez. Por exemplo, é possível ter dois operadores "Kmeans2x4"que processam dois pontos em paralelo, recebendo 8 entradas (4 atributos de cada ponto) e gerando duas saídas com a classificação dos dois pontos.

A Figura 6.14 ilustra dois exemplos de configuração das operações do *K-means*. Para simplificar, consideramos um operador "Kmeans2x4"com $K = 2$ e $N = 4$. Para implementar um *K-means* com $K = 2$ e $N = 2$, podemos subutilizar um operador "Kmeans2x4", configurando $N = 2$. A Figura 6.14(a) mostra que é possível configurar o operador "Kmeans2x4"para receber dois pontos com $N = 2$, processar apenas um deles e calcular o centroide mais próximo, destacado em cinza-claro. Os dois pontos estão conectados a um segundo operador "Kmeans2x4", em cinza-escuro, para selecionar o centroide do segundo ponto. Assim, é possível calcular um *K-means* com $K = 2$ e $N = 2$ processando dois pontos por vez. A

Figura 6.14(b) mostra o grafo de fluxo de dados das operações para cada ponto. Já a Figura 6.14(c) mostra a reconfiguração da mesma arquitetura para calcular um *K-means* com $K = 4$ e $N = 2$. Nesse caso, os operadores são subutilizados, pois recebem apenas dois atributos, e cada operador calcula dois centroides que são enviados a um operador "Filtro". Por fim, a Figura 6.14(d) mostra o grafo de fluxo com as operações para cada ponto no KCGRA.

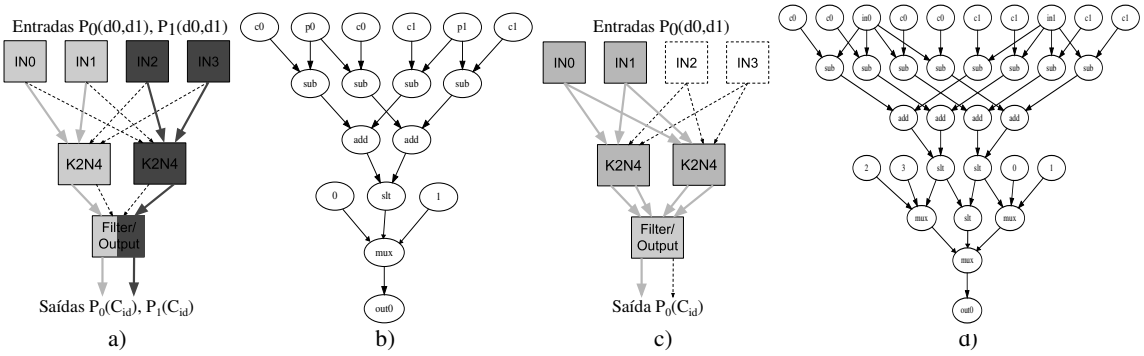


Figura 6.14: KCGRA com 4 entradas e operadores $K = 2$ e $N = 4$: (a) duas cópias em paralelo para K=2 e N=2; (b) Grafo de fluxo de dados com $K = 2$ e $N = 4$; (c) Configuração para $K = 4$ e $N = 2$. (d) Grafo de fluxo de dados para $K = 4$ e $N = 2$

A Figura 6.15 apresenta a configuração do KCGRA para os dois exemplos da Figura 6.14, utilizando o formato apresentado na Seção 6.5. A Figura 6.15(a) mostra em duas colunas a configuração para dois *K-means* com $K = 2$ e $N = 2$, demonstrado na Figura 6.14(a-b), enquanto as duas colunas de código da Figura 6.15(b) correspondem à configuração do *K-means* com $K = 4$ e $N = 2$, demonstrado na Figura 6.14(c-d).

```
00 pass $0 $istream[0]
01 pass $1 $istream[0]
02 route $0 $alu[0] $4
03 route $1 $alu[0] $4
04 Kmeans2x4 $4 $0 $1 0 0
05 route $4 $alu[1] $6
06 pass $6 $4
07 route $6 alu[0] $10
08 route $10 $6 $ostream[0]
```

```
09 pass $2 $istream[0]
10 pass $3 $istream[0]
11 route $2 $alu[0] $5
12 route $3 $alu[0] $5
13 Kmeans2x4 $5 $2 $3 0 0
14 route $5 $alu[1] $8
15 pass $8 $5
16 route $8 alu[0] $10
17 route $10 $8 $ostream[1]
```

a)

```
00 pass $0 $istream[0]
01 pass $1 $istream[0]
02 route $0 $alu[0] $4
03 route $1 $alu[0] $4
04 route $0 $alu[0] $5
05 route $1 $alu[0] $5
06 Kmeans2x4 $4 $0 $1 0 0
07 Kmeans2x4 $5 $0 $1 0 0
08 route $4 $alu[0] $6
09 route $4 $alu[1] $7
10 route $5 $alu[0] $8
```

```
11 route $5 $alu[1] $9
12 pass $6 $4
13 pass $7 $4
14 pass $8 $5
15 pass $9 $5
16 route $6 alu[0] $10
17 route $7 alu[0] $10
18 route $8 alu[0] $10
19 route $9 alu[0] $10
20 Kmeans_filter $10 $6 $8 $7 $9
21 route $10 $alu[1] $ostream[0]
```

b)

Figura 6.15: Formato de configuração: (a) Configuração com duas cópias do *K-means* $K = 2$ e $N = 2$(Figura 6.14(a-b)); (b) Configuração para Figura 6.14(c-d) com $K = 4$ e $N = 2$.

Seguindo a mesma arquitetura do exemplo da Figura 6.14, temos 4 instruções: $pass, route, Kmeans2x4$, e $Kmeans\_filter$. Na Figura 6.15(a), existem quatro instruções *pass* nas linhas 0, 1, 9 e 10, que receberão as 4 entradas (duas para cada ponto) e as atribuirão aos elementos de processamento (PE) 0, 1, 2 e 3, respectivamente. Em

seguida, quatro instruções *route* são usadas para conectar duas entradas em cada um dos dois operadores $Kmeans2x4$ identificados pelos IDs 4 e 5, respectivamente. Na linha 4, o novo operador $Kmeans2x4$ é configurado internamente para processar os dados das entradas, onde os PE_0 e PE_1 foram conectados às entradas 0 e 1 da unidade de cálculo do operador $Kmeans2x4$. O mesmo acontece na linha 13 para os PEs de entrada 2 e 3. Finalmente, as linhas 5-8 e 14-17 fazem a ligação da saída 1 da ALU (que contém o ID do centroide) para repassar os centroides dos dois pontos para as saídas externas do KCGRA.

A Figura 6.15(b) mostra o segundo exemplo, no qual temos apenas um ponto de entrada com 2 atributos e calculamos 4 centroides. Neste exemplo, temos apenas dois atributos de entrada e 4 centroides. Observa-se que apenas as linhas 0 e 1 possuem instruções para leitura na memória. Nas linhas 6 e 7, temos dois operadores $Kmeans2x4$ que receberão os dados, cada um verificando 2 centroides. Em seguida, os resultados de 2 pares de valores, a distância e o ID do centroide de menor distância, totalizando 4 valores, são repassados para o operador $Kmeans\_filter$ da linha 20. Por isso, temos 4 instruções de repasse e roteamento nas linhas 12 a 19. Por fim, na linha 21, o grupo classificado é enviado para a memória externa.

6.7.3 Resultados

Para validar o acelerador de domínio específico KCGRA, foi desenvolvida uma arquitetura com 16 operadores $Kmeans2x32$ e 15 operadores $Kmeans\_filtro$. Para a execução e a medida de tempo, o KCGRA foi sintetizado e testado em uma plataforma com FPGA Alveo U55C conectada a um nó computacional de alto desempenho. Para a comparação de desempenho na mesma plataforma, três aceleradores estáticos foram gerados com a ferramenta apresentada no Capítulo 4 e sintetizados para a mesma plataforma para a execução do *K-means* com $N = 8$ dimensões e $K = 4$, 8 e 16 grupos. O principal objetivo dos experimentos é verificar se o desempenho do KCGRA é equivalente ao desempenho das abordagens de síntese direta.

Todos os experimentos foram realizados em um nó computacional com Linux Ubuntu 16.04, equipado com dois processadores Intel(R) Xeon(R) Silver 4210R, totalizando 20 núcleos, 14 MB de cache L3 e 2,4 GHz de *clock*, juntamente com um FPGA AMD Xilinx Alveo U55C, que possui 1.304K de LUTs, 2.607K registradores e 16 GB de memória HBM (*High Bandwidth Memory*), ou Memória de Alta Largura de Banda. Os experimentos foram executados com 2 milhões de pontos de entrada.

Todas as medidas de tempo foram realizadas considerando a execução completa da classificação dos 2 milhões de pontos, incluindo os tempos de transferência de

entrada e saída de dados. A Tabela 6.3 apresenta uma comparação dos tempos de execução de uma iteração da fase de classificação do algoritmo *K-means* entre duas abordagens: a síntese direta denominada RTL e o KCGRA. Como o *K-means* é limitado pela taxa de transferência de memória para a leitura dos pontos, podemos observar que o tempo de execução não aumenta para diferentes valores de K, pois o valor de N é o mesmo, ou seja, o número de bytes transferidos durante a execução é o mesmo.

Tabela 6.3: Tempo de execução para classificação de 2 milhões de pontos para acelerador com abordagem síntese direta do Capitulo 4(RTL) e KCGRA.

Acelerador	K	N	CPU→FPGA	FPGA→CPU	Tempo Kernel	Acel. Kernel	Tempo Total	Acel. Total
RTL	4	8	4,04	0,39	2,93	1	7,36	1
	8	8	4,29	0,33	2,94	1	7,56	1
	16	8	4,33	0,34	2,92	1	7,59	1
KCGRA	4	8	3,78	0,89	2,59	1,13	7,26	1,01
	8	8	3,61	0,76	2,67	1,10	7,04	1,07
	16	8	3,92	0,82	4,82	0,60	9,56	0,79

O KCGRA com operadores $Kmeans2x32$ pode ser configurado para vários valores de K e N. Os testes da Tabela 6.3 foram realizados com três configurações de K com 4, 8 e 16 centroides para um valor fixo de $N = 8$. Portanto, o KCGRA teve uma perda de desempenho na execução com 16x8, devido a uma subutilização na transferência de dados, uma vez que o KCGRA foi previamente projetado para o valor de $N = 32$, e apenas 16 entradas são utilizadas devido ser possivel executar apenas duas cópias do Kmeans com $K = 16$. Para as configurações 4x8 e 8x8, o KCGRA apresentou o mesmo desempenho, umas vêz que foi possivel utilizar todas as entradas executando 4 cópias em paralelo.

A Figura 6.16 apresenta dois gráficos do ganho de desempenho do KCGRA em relação à abordagem RTL, considerando agora o tempo de execução do kernel apenas, sem considerar a transferência de dados. A Figura 6.16(a) mostra a comparação em GOPS. É possível observar que o KCGRA só fica abaixo da abordagem RTL para o Kmeans16x8, devido à subutilização da arquitetura. Esta subutilização ocorre, pois o operador tem 32 entradas ($N = 32$) e foi configurado para $N = 8$ com $K = 16$, gerando uma subutilização das entradas.

Outro ponto importante é o uso de recursos no FPGA. A Tabela 6.4 apresenta os resultados do uso de recursos dos aceleradores RTL e KCGRA. Os aceleradores foram sintetizados para a mesma plataforma, permitindo uma comparação direta dos valores de LUTs (Unidades lógicas), REGs (Registradores), LUTasMEM (Unidades Lógicas Configuradas como Memórias), BRAM (Blocos Distribuídos de Memória) e DSP (Unidade de Cálculo de multiplicação no Nível de Palavra).

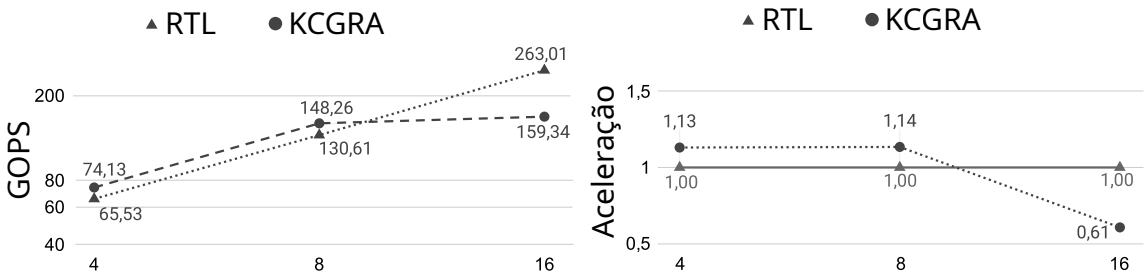


Figura 6.16: Comparativo de desempenho em GOPS para a etapa de classificação das versões de síntese direta do Capítulo 4 e do KCGRA em uma Alveo U55C, para diferentes valores K e N.

Como já mencionado, o acelerador RTL possui uma implementação com os parâmetros do *K-means* constantes no número de dimensões e grupos, diferente do KCGRA, que possui como diferencial a reconfigurabilidade em tempo de execução sem a necessidade de realizar uma nova síntese. Assim, ao gerar um acelerador com valores menores de K e N, será alocada uma quantidade personalizada de recursos.

Como esperado, o KCGRA apresenta um maior uso de recursos de LUTs, LUTasMEM e REGs, uma vez que são alocados recursos para sua configuração máxima ao se criar 16 unidades com operador $Kmeans2x32$ e 15 unidades com operador $Kmeans\_filtro$. O KCGRA pode implementar um *K-means* com 32 centroides de 32 atributos. Essa configuração foi executada para 2M pontos com um desempenho de 668 GOPS.

Tabela 6.4: Comparação utilização de recursos para aceleradores com valores de K e N fixos e KCGRA com K até 32 e N até 32 para o FPGA Alveo U55C.

Acelerador	K	N	LUT	LUTasMEM	REG	BRAM	DSP	Reconfiguração
RTL	4	8	9.350	306	11381	7	384	02h 44m 25s
	8	8	17.632	318	21.768	7	768	02h 56m 28s
	16	8	34.361	402	40.041	7	1.536	03h 23m 53s
KCGRA	2..32	2..32	153.494	6298	158497	0	0	2,34ms

Enquanto as implementações HLS e RTL usam módulos BRAM e DSP para o cálculo das distâncias, a versão do KCGRA não utiliza DSPs. O cálculo da distância no operador $Kmeans2x32$, por escolha de projeto, utiliza a distância de "Manhattan", que foi sintetizada usando LUTs, uma escolha automática da ferramenta de síntese.

Por fim, a coluna **reconfiguração** mostra o tempo para gerar a configuração para programação do FPGA. Enquanto as versões estáticas precisam recompilar todo o projeto, o que pode demorar horas, o KCGRA, no pior cenário, para configurar todas as unidades, requer 2,34 ms. O tempo do KCGRA depende apenas do tamanho da configuração, e estimamos o pior caso. Ou seja, o KCGRA ocupa um espaço compatível com suas funcionalidades, gera um desempenho competitivo com a versão estática e pode ser configurado para outros valores de K e N em apenas 2,34

ms, ou seja, seis ordens de grandeza mais rápido. Além disso, o KCGRA comprova que a metodologia apresentada neste capítulo pode ser aplicada e estendida para o desenvolvimento de aceleradores de domínio específicos com bons resultados de desempenho.

6.8 Discussões

Neste capítulo, foi apresentada uma metodologia para o desenvolvimento de aceleradores com topologias genéricas. Para isso, foi proposta a ferramenta HPCGRA, que gera código Verilog de arquiteturas de aceleradores a partir de uma descrição em alto nível utilizando a notação de objetos em JavaScritp (JSON). Esta ferramenta é de código aberto e está disponível no GitHub[1]. O código Verilog gerado pela ferramenta é genérico e sintetizável para diferentes plataformas de FPGAs, como Intel/Altera ou AMD Xilinx. Os resultados da síntese mostram que a frequência das arquiteturas é estável, mesmo com o aumento do tamanho das mesmas. O gerador é escalável; mesmo aumentando o número de PEs, a relação ALM/PE permanece estável. Foi possível sintetizar arquiteturas com 3036 PEs, com desempenho teórico de 1,2 TOPS. A ferramenta também pode exportar um projeto para ser sintetizado em placas Alveo da AMD Xilinx, criando todos os *scripts* necessários para a construção do *bitstream* final.

Com a metodologia desenvolvida para a geração de aceleradores com topologias genéricas, a ferramenta HPCGRA consegue gerar arquiteturas de CGRAs especializadas para a execução de aplicações específicas. Essa metodologia foi adotada no estudo de caso da Seção 6.7, onde foi proposta a arquitetura chamada KCGRA. Essa arquitetura é formada por dois operadores personalizados que foram desenvolvidos utilizando o formato JSON para descrever um grafo de fluxo de dados. Esses operadores encapsularam partes da aplicação do algoritmo *K-means* em elementos de processamento interligados por uma rede personalizada, permitindo a execução da etapa de classificação do *K-means* com diferentes tamanhos de grupos (K) e dimensões (N), apenas reconfigurando o acelerador em tempo de execução. Um KCGRA com 16 operadores, cada um capaz de lidar com até 32 atributos, foi sintetizado e executado em uma placa Alveo U55C, sendo comparado com aceleradores baseados em síntese direta, onde o KCGRA demonstrou possuir desempenho equivalente ou superior em tempo de execução.

Este capítulo é uma das contribuições centrais desta Tese. Ele propõe o HPCGRA, uma ferramenta que permite a geração de aceleradores genéricos baseados CGRAs. HPCGRA equaciona a síntese de instruções e interconexões genéricas, dois dos

[1]https://github.com/lesc-ufv/hpcgra

principais problemas de pesquisa levantados nos estudos de caso do Capítulo 3.

Capítulo 7

Conclusão e Trabalhos Futuros

A pergunta fundamental desta tese foi entender se existe uma metodologia genérica para a síntese de aceleradores em FPGAs. A resposta a essa pergunta foi desenvolvida em etapas ao longo dos capítulos desta tese.

Inicialmente, para compreender o problema de pesquisa, foram realizados projetos de síntese direta de aceleradores para FPGA, conforme apresentado no Capítulo 4. Na Seção 4.1, foi proposto um gerador de aceleradores para encontrar estados de estabilização em redes reguladoras de genes, modeladas por equações booleanas, o que é ideal para dispositivos FPGA. Os resultados mostraram que a implementação em FPGA pode ser até 12 vezes mais rápida que a implementação em GPU de alto desempenho. Também foi demonstrado que, em um contexto de computação em nuvem, o uso do FPGA para esse problema é 5,7 vezes melhor que o uso de GPUs, quando analisado o número de estados cobertos de um modelo de GRN por dólar. Na Seção 4.2.1, foi apresentado um gerador de aceleradores para o algoritmo *K-means* para FPGA. Foi mostrado que o uso de FPGA como acelerador para este problema consegue otimizar o tempo de execução, aproveitando o paralelismo intrínseco do FPGA. Esses estudos destacaram que, embora os aceleradores específicos em FPGA apresentem bons resultados de desempenho, os tempos gastos durante o processo de síntese tornam essa abordagem insatisfatória para cenários onde os parâmetros de entrada precisam ser alterados para realizar alguma análise. Assim, no Capítulo 5, foi introduzido o uso de uma camada de abstração sobre o FPGA, visando otimizar o tempo de síntese de aceleradores.

Uma vez compreendido em profundidade o problema de pesquisa no Capítulo 4, uma nova etapa do trabalho se iniciou, focada em pesquisar uma plataforma que generalizasse o projeto de aceleradores em FPGAs. No Capítulo 5, é apresentado um arcabouço de *software* e *hardware* para a geração e implantação de um acelerador genérico para FPGA denominado READY. A arquitetura resultante do READY é um CGRA com rede de interconexão global chamada READY CGRA. Neste modelo, o tempo de posicionamento e roteamento de aplicações para esta arquitetura é realizado em tempo polinomial. Esse diferencial no tempo de P&R permite que uma aplicação seja compilada e executada no acelerador em tempo de execução. Os

resultados de desempenho do READY CGRA superam o tempo de execução de uma CPU de alto desempenho. Porém, este modelo de organização da arquitetura, onde todos os PEs são conectados por meio de uma rede global, possui uma baixa escalabilidade. Foi possível realizar a síntese das arquiteturas READY CGRA com até 128 PEs, limitando o desempenho do acelerador. Uma forma de contornar essa dificuldade foi adicionar mais operações em um mesmo PE, aproveitando melhor os recursos de roteamento da rede global. Um experimento mostrou que ao adicionar mais operações por PE, a síntese da arquitetura não foi afetada, e o desempenho foi o dobro no caso em que cada PE conseguia executar duas operações. Esse resultado mostrou que uma possível melhoria da arquitetura seria a criação de PEs com uma granularidade maior, onde uma pequena parte do grafo de fluxo de dados pudesse ser executada de uma vez no mesmo PE. Esta abordagem foi então incorporada, sendo tratada no Capítulo 6. A ferramenta resultante desta extensão chamada HPCGRA permite a criação de operadores personalizados para os aceleradores, além de permitir o uso de outras topologias de interconexão entre os PEs da arquitetura, além da rede global originalmente proposta no READY CGRA.

No Capítulo 6, é apresentada uma metodologia para a síntese de aceleradores genéricos. Esta metodologia é guiada pela ferramenta HPCGRA, um gerador para arquiteturas reconfiguráveis em plataformas de alto desempenho que reúne todas as contribuições dos Capítulos 4 e 5. Os resultados mostraram ser possível sintetizar arquiteturas com grandes quantidades de PEs sem comprometer a frequência de operação do sistema, tornando-o escalável. O modelo de descrição de um acelerador em alto nível permite ao projetista criar diferentes formas de interconexões, além de possibilitar o desenvolvimento de novos operadores que podem ser incorporados diretamente no caminho de dados dos PEs. Tudo isso é feito com uma interface de configuração padronizada em alto nível que permite a configuração de cada parte da arquitetura parcialmente. Para demonstrar o poder de personalização do gerador, foi proposta uma arquitetura de CGRA especializada em acelerar o algoritmo *K-means* como estudo de caso, permitindo a execução do *K-means* com diferentes parâmetros de entrada sem a necessidade de uma nova síntese.

Os resultados desta tese apontam para algumas direções para trabalhos futuros, visando ampliar a adoção do uso de FPGA como aceleradores em sistemas computacionais de alto desempenho. Durante a pesquisa, duas direções se destacaram, relacionadas à: (1) integração ao fluxo de projeto; e (2) acesso aos dados pelo CGRA.

A primeira direção é motivada pelas barreiras existentes para a integração de aceleradores ao *toolchain* de *software*. Portanto, é fundamental realizar mais pesquisas em novas ferramentas que integrem aplicações em *software* com aceleradores em *hardware* de maneira transparente.

A segunda direção futura está relacionada ao desempenho do acesso aos dados pelo CGRA. Como apresentado na Seção 6.1, a arquitetura base de um acelerador gerado pela ferramenta HPCGRA recebe os dados por meio de filas no formato de *streams*, realiza o processamento em um modelo de fluxo de dados e envia a saída para filas de saída também no formato de *streams*. No entanto, a interface com a memória externa só consegue ler os dados incrementando o ponteiro no padrão [i+1]. Mahdi propôs em [1] uma unidade capaz de gerar *streams* de dados para CGRAs, chamada DSU (*Data Stream Unit*), ou Unidade de *Stream* de Dados. Em um trabalho futuro, esta unidade poderia ser incorporada à arquitetura base gerada pelo HPCGRA, permitindo um acesso mais flexível à memória externa por meio de outros padrões de acesso.

Referências Bibliográficas

[1] Mahdi Abbaszadeh. *Efficient data streaming for a tightly-coupled CGRA*. PhD thesis, University of Toronto (Canada), 2023.

[2] Tarek S Abdelrahman. Accelerating K-Means Clustering on a Tightly-Coupled Processor-FPGA Heterogeneous System. In *IEEE International Conference ASAP*, 2016.

[3] Tatsuya Akutsu, Satoru Kuhara, Osamu Maruyama, and Satoru Miyano. A system for identifying genetic networks from gene expression patterns produced by gene disruptions and overexpressions. *Genome Informatics*, 9:151–160, 1998.

[4] Alibaba Cloud. What are fpga-accelerated instances? https://www.alibabacloud.com/help/doc-detail/163932.htm, 2024. Acessado: 02/04/2024.

[5] Alexandru Amaricai. Design Trade-offs in Configurable FPGA Architectures for K-Means Clustering. *Studies in Informatics and Control*, 26(1):43–48, 2017.

[6] Amazon. Elastic Compute Cloud - Amazon EC2 - AWS. http://aws.amazon.com/ec2/, 2024. Acessado: 02/04/2024.

[7] Jason Anderson, Rami Beidas, Vimal Chacko, Hsuan Hsiao, Xiaoyi Ling, Omar Ragheb, Xinyuan Wang, and Tianyi Yu. Cgra-me: An open-source framework for cgra architecture and cad research : (invited paper). In *2021 IEEE 32nd International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, pages 156–162, 2021.

[8] Hodjat Asghari Esfeden and et al. Corf: Coalescing operand register file for gpus. In *Proceedings of International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 701–714. ACM, 2019.

[9] Jonathan Bachrach, Huy Vo, Brian Richards, Yunsup Lee, Andrew Waterman, Rimas Avižienis, John Wawrzynek, and Krste Asanović. Chisel: constructing hardware in a scala embedded language. In *DAC Design Automation Conference 2012*, pages 1212–1221. IEEE, 2012.

[10] Amartya Bhattacharjya and Shoudan Liang. Median attractor and transients in random boolean nets. *Physica D: Nonlinear Phenomena*, 95(1):29–34, 1996.

[11] Fabrizio F Borelli and *et al*. Gene regulatory networks inference using a multi-gpu exhaustive search algorithm. *BMC bioinformatics*, 14(18):1–12, 2013.

[12] Andrew Boutros, Eriko Nurvitadhi, Rui Ma, Sergey Gribok, Zhipeng Zhao, James C Hoe, Vaughn Betz, and Martin Langhammer. Beyond peak performance: Comparing the real performance of ai-optimized fpgas and gpus. In *2020 International Conference on Field-Programmable Technology (ICFPT)*, pages 10–19. IEEE, 2020.

[13] Lucas Bragança, Jeronimo Costa Penha, Michael Canesche, Dener Ribeiro, José Augusto M Nacif, and Ricardo Ferreira. An open-source cloud-fpga gene regulatory accelerator. In *Symposium on High Performance Computing Systems (WSCAD)*, 2021.

[14] Lucas Braganca and *et al*. An open source custom k-means generator for aws cloud fpga accelerators. In *Brazilian Symp on Computing Systems Engineering (SBESC)*, 2021.

[15] Lucas Bragança, Fredy Alves, Jeronimo Costa Penha, Gabriel Coimbra, Ricardo Ferreira, and José A. M. Nacif. Simplifying HW/SW Integration to Deploy Multiple Accelerators for CPU-FPGA Heterogeneous Platforms. In *IEEE SAMOS*, 2018.

[16] Lucas Bragança, Michael Canesche, Jeronimo Penha, Josué Campos, José Augusto M. Nacif, and Ricardo S. Ferreira. Fast flow cloud: A stream dataflow framework for cloud fpga accelerator overlays at runtime. *Concurrency and Computation: Practice and Experience*, 35(17):e6977, 2023.

[17] Lucas Bragança and *et al*. Exploring the dynamics of large-scale gene regulatory networks using hardware acceleration on a heterogeneous cpu-fpga platform. In *IEEE International Conference on ReConFigurable Computing and FPGAs (ReConFig)*, 2017.

[18] Davor Capalija and Tarek S Abdelrahman. A high-performance overlay architecture for pipelined execution of data flow graphs. In *IEEE International Conference on Field programmable Logic and Applications*, 2013.

[19] Alvaro Chaos and *et al*. From genes to flower patterns and evolution: dynamic models of gene regulatory networks. *Journal of Plant Growth Regulation*, 25(4):278–289, 2006.

[20] Yuze Chi, Weikang Qiao, Atefeh Sohrabizadeh, Jie Wang, and Jason Cong. Democratizing domain-specific computing. *Commun. ACM*, 66(1):74–85, dec 2022.

[21] S Alexander Chin, Kuang Ping Niu, Matthew Walker, Shizhang Yin, Alexander Mertens, Jongeun Lee, and Jason H Anderson. Architecture exploration of standard-cell and fpga-overlay cgras using the open-source cgra-me framework. In *Int Symposium on Physical Design*, 2018.

[22] S Alexander Chin, Noriaki Sakamoto, Allan Rui, Jim Zhao, Jin Hee Kim, Yuko Hara-Azumi, and Jason Anderson. Cgra-me: A unified framework for cgra modelling and exploration. In *Int Conf on Application-specific Systems, Architectures and Processors (ASAP)*, 2017.

[23] Young-Kyu Choi, Jason Cong, Zhenman Fang, Yuchen Hao, Glenn Reinman, and Peng Wei. In-depth analysis on microarchitectures of modern heterogeneous cpu-fpga platforms. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, 12(1):1–20, 2019.

[24] Yuk-Ming Choi and Hayden Kwok-Hay So. Map-Reduce Processing of K-Means Algorithm with FPGA-Accelerated Computer Cluster. In *IEEE International Conference ASAP*, 2014.

[25] Jason Cong, Zhenman Fang, Michael Lo, Hanrui Wang, Jingxian Xu, and Shaochong Zhang. Understanding performance differences of fpgas and gpus. In *2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 93–96, 2018.

[26] B Conroy and *et al*. Design, assessment, and in vivo evaluation of a computational model illustrating the role of cav1 in cd4+ t-lymphocytes. *Frontiers in immunology*, 5, 2014.

[27] Matheus da Silva Alves, Lucas Bragança Silva, Jerônimo Penha, Ricardo Ferreira, and José Augusto M Nacif. Kcgra–uma arquitetura reconfigurável de domínio específico para k-means. In *Anais do XXIV Simpósio em Sistemas Computacionais de Alto Desempenho*, pages 25–36. SBC, 2023.

[28] Yuan Dai, Jingyuan Li, Qilong Zhu, Yunhui Qiu, Yihan Hu, Wenbo Yin, and Lingli Wang. Heta: A heterogeneous temporal cgra modeling and design space exploration via bayesian optimization. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 2023.

[29] Jeff Dean, David Patterson, and Cliff Young. A new golden age in computer architecture: Empowering the machine-learning revolution. *IEEE Micro*, 38(2):21–29, 2018.

[30] R.H. Dennard, F.H. Gaensslen, Hwa-Nien Yu, V.L. Rideout, E. Bassous, and A.R. LeBlanc. Design of ion-implanted mosfet's with very small physical dimensions. *IEEE Journal of Solid-State Circuits*, 9(5):256–268, 1974.

[31] Dua Dheeru and Efi Karra Taniskidou. Machine learning repository. http://archive.ics.uci.edu/ml, 2017. Acessado em: 02/04/2024.

[32] Leonardo A Dias, João C Ferreira, and Marcelo AC Fernandes. Parallel implementation of k-means algorithm on fpga. *IEEE Access*, 8:41071–41084, 2020.

[33] Keith Diefendorff. Power4 focuses on memory bandwidth. *Microprocessor Report*, 13(13):1–8, 1999.

[34] Elena Dubrova and Maxim Teslenko. A sat-based algorithm for finding attractors in synchronous boolean networks. *IEEE Trans. on Comp. Biology and Bioinformatics*, 2011.

[35] Carl Ebeling, Darren C Cronquist, and Paul Franklin. Rapid—reconfigurable pipelined datapath. In *Field-Programmable Logic Smart Applications, New Paradigms and Compilers: 6th International Workshop on Field-Programmable Logic and Applications, FPL'96 Darmstadt, Germany, September 23–25, 1996 Proceedings 6*, pages 126–135. Springer, 1996.

[36] Lieven Eeckhout. Is moore's law slowing down? what's next? *IEEE Micro*, 37(04):4–5, 2017.

[37] Renesas Eletronics. Stp engine (ip core), 2024. Acessado: 02/04/2024.

[38] Hadi Esmaeilzadeh, Emily Blem, Renee St. Amant, Karthikeyan Sankaralingam, and Doug Burger. Dark silicon and the end of multicore scaling. In *Proceedings of the 38th annual international symposium on Computer architecture*, pages 365–376, 2011.

[39] Ricardo Ferreira and et al. An fpga-based heterogeneous coarse-grained dynamically reconfigurable architecture. In *Proceedings of the 14th international conference on Compilers, architectures and synthesis for embedded systems*, pages 195–204. ACM, 2011.

[40] Ricardo Ferreira and Julio Vendramini. Fpga-accelerated attractor computation of scale free gene regulatory networks. In *Field Programmable Logic and Applications (FPL)*, 2010.

[41] Edward W Forgy. Cluster Analysis of Multivariate Data: Efficiency Versus Interpretability of Classifications. *Biometrics*, 21:768–769, 1965.

[42] Abhishek Garg and *et al*. An efficient method for dynamic analysis of gene regulatory networks and in silico gene perturbation experiments. In *Int Conf on Research in Computational Molecular Biology*, 2007.

[43] Hasan Genc. A dsl for systolic arrays. https://github.com/hngenc/systolic-array, 2020. Acessado em: 11/08/2020.

[44] Seth Copen Goldstein, Herman Schmit, Matthew Moe, Mihai Budiu, Srihari Cadambi, R Reed Taylor, and Ronald Laufer. Piperench: a co/processor for streaming multimedia acceleration. *ACM SIGARCH Computer Architecture News*, 27(2):28–39, 1999.

[45] Wensheng Guo, Guowu Yang, Wei Wu, and Mingyu Sun. A parallel attractor finding algorithm based on boolean satisfiability for genetic regulatory networks. *PloS one*, 9(4), 2014.

[46] Yijiang Guo and Guojie Luo. Pillars: An integrated cgra design framework. In *Third Workshop on Open-Source EDA Technology (WOSET)*, 2020.

[47] PK Gupta. Accelerating datacenter workloads. In *26th International Conference on Field Programmable Logic and Applications (FPL)*, 2016.

[48] Prabhat K Gupta. Xeon+ fpga platform for the data center. In *Fourth Workshop on the Intersections of Computer Architecture and Reconfigurable Logic*, volume 119, page 2, 2015.

[49] Mahdi Hamzeh, Aviral Shrivastava, and Sarma Vrudhula. Regimap: register-aware application mapping on coarse-grained reconfigurable architectures (cgras). In *ACM Proceedings of the 50th Annual Design Automation Conference*, 2013.

[50] Sadegh Hashemipour and Maaruf Ali. Amazon web services (aws)–an overview of the on-demand cloud computing platform. In *Int Conf for Emerging Technologies in Computing*. Springer, 2020.

[51] Scott Hauck and Andre DeHon. *Reconfigurable computing: the theory and practice of FPGA-based computation*, volume 1. Morgan Kaufmann, 2010.

[52] Tomáš Helikar, Naomi Kochi, Bryan Kowal, Manjari Dimri, Srikumar M Raja, Vimla Band, Hamid Band, and Jim A Rogers. A comprehensive, multi-scale dynamical model of erbb receptor signal transduction in human mammary epithelial cells. *PLoS One*, 8(4):e61757, 2013.

[53] R. A. Iannucci. Toward a dataflow/von neumann hybrid architecture. In *Proceedings of the 15th Annual International Symposium on Computer Architecture*, ISCA '88, page 131–140, Washington, DC, USA, 1988. IEEE Computer Society Press.

[54] Intel. Intel high level synthesis compiler. https://www.intel.com.br/content/www/br/pt/software/programmable/quartus-prime/hls-compiler.html, 2024. Acessado: 02/04/2024.

[55] Abhishek Kumar Jain, Xiangwei Li, Pranjul Singhai, Douglas L Maskell, and Suhaib 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)*, pages 1–8. IEEE, 2016.

[56] Abhishek Kumar Jain, Douglas L Maskell, and Suhaib A Fahmy. Resource-aware just-in-time opencl compiler for coarse-grained fpga overlays. *arXiv preprint arXiv:1705.02730*, 2017.

[57] Liancheng Jia, Liqiang Lu, Xuechao Wei, and Yun Liang. Generating systolic array accelerators with reusable blocks. *IEEE Micro*, 40(4):85–92, 2020.

[58] JSON. Introducing json. https://www.json.org/json-en.html, 2020. Acessado em: 25/07/2020.

[59] Roman Kaplan, Leonid Yavits, and Ran Ginosar. Prins: Processing-in-Storage Acceleration of Machine Learning. *IEEE Transactions on Nanotechnology*, 2018.

[60] Changmoo Kim, Mookyoung Chung, Yeongon Cho, Mario Konijnenburg, Soojung Ryu, and Jeongwook Kim. Ulp-srp: Ultra low-power samsung reconfigurable processor for biomedical applications. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, 7(3):1–15, 2014.

[61] Poonacha Kongetira, Kathirgamar Aingaran, and Kunle Olukotun. Niagara: A 32-way multithreaded sparc processor. *IEEE micro*, 25(2):21–29, 2005.

[62] Charalampos Kritikakis and Dirk Koch. Enabling dynamic system integration on maxeler hls platforms. *Journal of Signal Processing Systems*, 92(9):887–905, 2020.

[63] Dajung Lee, Alric Althoff, Dustin Richmond, and Ryan Kastner. A Streaming Clustering Approach Using a Heterogeneous System for Big Data Analysis. In *IEEE/ACM ICCAD*, 2017.

[64] Jingyuan Li, Yunhui Qiu, Guowei Zhu, Qilong Zhu, Wenbo Yin, and Lingli Wang. Thram: A template-based heterogeneous cgra modeling framework supporting fast dse. In *2023 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 1–5. IEEE, 2023.

[65] Xiangwei Li, Abhishek Kumar Jain, Douglas L Maskell, and Suhaib A Fahmy. A time-multiplexed fpga overlay with linear interconnect. In *IEEE Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1075–1080, 2018.

[66] Xiangwei Li, Cheng Fei Phung, and Douglas L Maskell. Fpga overlays: hardware-based computing for the masses. In *Proceedings of the Eighth International Conference on Advances in Computing, Electronics and Electrical Technology-CEET*, pages 25–31, 2018.

[67] Xiangwei Li, Kizheppatt Vipin, Douglas L Maskell, Suhaib A Fahmy, and Abhishek Kumar Jain. High throughput accelerator interface framework for a linear time-multiplexed fpga overlay. In *2020 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 1–5. IEEE, 2020.

[68] Zhehao Li, Jifang Jin, and Lingli Wang. High-Performance K-Means Implementation Based on a Simplified Map-Reduce Architecture. *arXiv preprint arXiv:1610.05601*, 2016.

[69] 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)*, pages 56–63, Dec 2015.

[70] Sihao Liu, Jian Weng, Dylan Kupsh, Atefeh Sohrabizadeh, Zhengrong Wang, Licheng Guo, Jiuyang Liu, Maxim Zhulin, Rishabh Mani, Lucheng Zhang, et al. Overgen: Improving fpga usability through domain-specific overlay generation. In *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 35–56. IEEE, 2022.

[71] Vinícius Duarte Lopes. *Uma heurística polinomial para escalonamento de loops em arquiteturas reconfiguráveis de grão grosso*. PhD thesis, Universidade Federal de Viçosa, Av. Peter Henry Rolfs, s/n - Campus Universitário, Viçosa - MG, 36570-900, 7 2013.

[72] Junyan Lu and *et al*. Network modelling reveals the mechanism underlying colitis-associated colon cancer and identifies novel combinatorial anti-cancer targets. *Scientific reports*, 5(1):1–15, 2015.

[73] Lucas Andrade Maciel, Matheus Alcântara Souza, and Henrique Cota de Freitas. Reconfigurable fpga-based k-means/k-modes architecture for network intrusion detection. *IEEE Transactions on Circuits and Systems II: Express Briefs*, 67(8):1459–1463, 2019.

[74] Jonathan Mak and Alan Mycroft. Limits of parallelism using dynamic dependency graphs. In *Proceedings of the Seventh International Workshop on Dynamic Analysis*, WODA '09, page 42–48, New York, NY, USA, 2009. Association for Computing Machinery.

[75] Matteo Manica, Raphael Polig, Mitra Purandare, Roland Mathis, Christoph Hagleitner, and Maria Rodriguez Martinez. Fpga accelerated analysis of boolean gene regulatory networks. *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, 17(6):2141–2147, 2020.

[76] Bingfeng Mei, Serge Vernalde, Diederik Verkest, Hugo De Man, and Rudy Lauwereins. Adres: An architecture with tightly coupled vliw processor and coarse-grained reconfigurable matrix. In *International Conference on Field Programmable Logic and Applications*, pages 61–70. Springer, 2003.

[77] Microsoft. Project catapult. https://www.microsoft.com/en-us/research/project/project-catapult/, 2021. Acessado: 02/04/2024.

[78] Andrzej Mizera, Jun Pang, and Qixia Yuan. Gpu-accelerated steady-state computation of large probabilistic boolean networks. *Formal Aspects of Computing*, 31(1):27–46, 2019.

[79] Duncan JM Moss, Srivatsan Krishnan, Eriko Nurvitadhi, Piotr Ratuszniak, Chris Johnson, Jaewoong Sim, Asit Mishra, Debbie Marr, Suchit Subhaschandra, and Philip HW 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*, pages 107–116, 2018.

[80] Christoph Müssel, Martin Hopfensitz, and Hans A Kestler. Boolnet—an r package for generation, reconstruction and analysis of boolean networks. *Bioinformatics*, 26(10), 2010.

[81] Katayoun Neshatpour et al. Energy-Efficient Acceleration of Big Data Analytics Applications Using FPGA. In *IEEE International Conference on Big Data*, 2015.

[82] Katayoun Neshatpour et al. Big Biomedical Image Processing Hardware Acceleration: A Case Study for K-Means and Image Filtering. In *Int. Symposium on Circuits and Systems*, 2016.

[83] Tony Nowatzki, Newsha Ardalani, Karthikeyan Sankaralingam, and Jian Weng. Hybrid optimization/heuristic instruction scheduling for programmable accelerator codesign. In *Proceedings of the 27th International Conference on Parallel Architectures and Compilation Techniques*, pages 1–15, 2018.

[84] Neal Oliver, Rahul R Sharma, Stephen Chang, Bhushan Chitlur, Elkin Garcia, Joseph Grecco, Aaron Grier, Nelson Ijih, Yaping Liu, Pratik Marolia, et al. A reconfigurable computing system based on a cache-coherent fabric. In *ReConFig*, pages 80–85, 2011.

[85] Giorgos Passas, Manolis Katevenis, and Dionisios Pnevmatikatos. Crossbar nocs are scalable beyond 100 nodes. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 31(4):573–585, 2012.

[86] Nuno Paulino, João Canas Ferreira, and João MP Cardoso. Optimizing opencl code for performance on fpga: k-means case study with integer data sets. *IEEE Access*, 8:152286–152304, 2020.

[87] Nuno MC Paulino, João Canas Ferreira, and João MP Cardoso. Dynamic partial reconfiguration of customized single-row accelerators. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 27(1):116–125, 2018.

[88] Jeronimo Costa Penha, Lucas Bragança, Kristtopher Coelho, Michael Canesche, Jansen Silva, Giovanni Comarela, José Augusto M Nacif, and Ricardo Ferreira. A gpu/fpga-based k-means clustering using a parameterized code generator. In *2018 Symposium on High Performance Computing Systems (WSCAD)*, pages 61–69. IEEE, 2018.

[89] Tekla S. Perry. David patterson says it's time for new computer architectures and software languages. https://spectrum.ieee.org/david-patterson-says\-its-time-for-new-computer-architectures-and-software-languages, 2021. Acessado: 17/11/2021.

[90] A Pipelined. Shared resource mimd computer by b. In *Smith et al. and published in the Proceedings of the 1978 International Conference on Parallel Processing*, 1986.

[91] Yunhui Qiu, Yuhang Cao, Yuan Dai, Wenbo Yin, and Lingli Wang. Tram: An open-source template-based reconfigurable architecture modeling framework. In *2022 32nd International Conference on Field-Programmable Logic and Applications (FPL)*, pages 61–69. IEEE, 2022.

[92] Hongyang Qu, Qixia Yuan, Jun Pang, and Andrzej Mizera. Improving bdd-based attractor detection for synchronous boolean networks. In *Proceedings of the 7th Asia-Pacific Symposium on Internetware*, Internetware '15, page 212–220, New York, NY, USA, 2015. Association for Computing Machinery.

[93] Sobia Raza and *et al*. A logic-based diagram of signalling pathways central to macrophage activation. *BMC systems biology*, 2(1):36, 2008.

[94] Regina Samaga and *et al*. The logic of egfr/erbb signaling: theoretical properties and analysis of high-throughput data. *PLoS computational biology*, 5(8):e1000438, 2009.

[95] Karthikeyan Sankaralingam and et al. Exploiting ilp, tlp, and dlp with the polymorphous trips architecture. In *30th Annual International Symposium on Computer Architecture, 2003. Proceedings.*, pages 422–433. IEEE, 2003.

[96] Scala. A programming language that scales with you: from small scripts to large multiplatform applications. https://docs.scala-lang.org/pt-br/tour/tour-of-scala.html, 2024. Acessado em: 17/03/2024.

[97] Muhammad Shafique and Siddharth Garg. Computing in the dark silicon era: Current trends and research challenges. *IEEE Design & Test*, 34(2):8–23, 2017.

[98] Lucas Silva, Michael Canesche, Ricardo Ferreira, and José Augusto Nacif. Hpcgra-an orthogonal designed cgra generator for high performance spatial accelerators. In *Anais do XXI Simpósio em Sistemas Computacionais de Alto Desempenho*, pages 25–36. SBC, 2020.

[99] Lucas Bragança Da Silva, Ricardo Ferreira, Michael Canesche, Marcelo M Menezes, Maria D Vieira, Jeronimo Penha, Peter Jamieson, and José Augusto M Nacif. Ready: A fine-grained multithreading overlay framework for modern cpu-fpga dataflow applications. *ACM Transactions on Embedded Computing Systems (TECS)*, 18(5s):1–20, 2019.

[100] Hartej Singh, Ming-Hau Lee, Guangming Lu, Fadi J Kurdahi, Nader Bagherzadeh, and Eliseu M Chaves Filho. Morphosys: an integrated reconfigurable system for data-parallel and computation-intensive applications. *IEEE transactions on computers*, 49(5):465–481, 2000.

[101] Hayden Kwok-Hay So and John Wawrzynek. Olaf'17: Third international workshop on overlay architectures for fpgas. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '17, page 1, New York, NY, USA, 2017. Association for Computing Machinery.

[102] Matheus A Souza, Lucas A Maciel, Pedro Henrique Penna, and Henrique C Freitas. Energy efficient parallel k-means clustering for an intel® hybrid multi-chip package. In *2018 30th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, pages 372–379. IEEE, 2018.

[103] Ivan Stanojević, Mladen Kovačević, and Vojin Šenk. Application of maxeler dataflow supercomputing to spherical code design. In *Exploring the DataFlow Supercomputing Paradigm*, pages 133–168. Springer, 2019.

[104] Steven Swanson, Ken Michelson, Andrew Schwerin, and Mark Oskin. Wavescalar. In *Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture*, page 291. IEEE Computer Society, 2003.

[105] Shinya Takamaeda-Yamazaki. Pyverilog: A python-based hardware design processing toolkit for verilog hdl. In *International Symposium on Applied Reconfigurable Computing*, pages 451–460. Springer, 2015.

[106] Cheng Tan, Chenhao Xie, Ang Li, Kevin J Barker, and Antonino Tumeo. Opencgra: An open-source unified framework for modeling, testing, and evaluating cgras. In *2020 IEEE 38th International Conference on Computer Design (ICCD)*, pages 381–388. IEEE, 2020.

[107] Qing Y Tang and Mohammed AS Khalid. Acceleration of k-means algorithm using altera sdk for opencl. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, 10(1):1–19, 2016.

[108] Ian Taras and Jason H Anderson. Impact of fpga architecture on area and performance of cgra overlays. In *2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 87–95. IEEE, 2019.

[109] Juilee Thakar, Ashutosh K Pathak, Lisa Murphy, and Isabella M Cattadori. Network model of immune responses reveals key effectors to single and co-infection dynamics by a respiratory bacterium and a gastrointestinal helminth. *PLoS computational biology*, 8(1), 2012.

[110] James E Thornton. Parallel operation in the control data 6600. In *Proceedings of the October 27-29, 1964, fall joint computer conference, part II: very high speed computer systems*, pages 33–40. ACM, 1964.

[111] Top500. Top500 list. https://www.top500.org/lists/top500/2023/11/, 2023. Acessado: 24/03/2024.

[112] Abraham Waksman. A permutation network. *Journal of the ACM (JACM)*, 15(1):159–163, 1968.

[113] Jian Weng, Sihao Liu, Vidushi Dadu, Zhengrong Wang, Preyas Shah, and Tony Nowatzki. Dsagen: Synthesizing programmable spatial accelerators. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pages 268–281. IEEE, 2020.

[114] Mark Wijtvliet, Luc Waeijen, and Henk Corporaal. Coarse grained reconfigurable architectures in the past 25 years: Overview and classification. In *2016 International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation (SAMOS)*, pages 235–244. IEEE, 2016.

[115] Ting Wu, Chi-Ying Tsui, and Mounir Hamdi. A 2 gb/s 256* 256 cmos crossbar switch fabric core design using pipelined mux. In *2002 IEEE International Symposium on Circuits and Systems. Proceedings (Cat. No. 02CH37353)*, volume 2, 2002.

[116] Xilinx. Vitis unified software platform documentation. https://docs.amd.com/v/u/2020.1-English/ug1393-vitis-application-acceleration, 2024. Acessado: 02/04/2024.

[117] AMD Xilinx. Amd xilinx vitis hls. https://www.xilinx.com/products/design-tools/vitis/vitis-hls.html, 2024. Acessado: 03/02/2024.

[118] Su Zheng, Kaisen Zhang, Yaoguang Tian, Wenbo Yin, Lingli Wang, and Xuegong Zhou. Fastcgra: A modeling, evaluation, and exploration platform for large-scale coarse-grained reconfigurable arrays. In *2021 International Conference on Field-Programmable Technology (ICFPT)*, pages 1–10, 2021.