**WESTERLEY CARVALHO OLIVEIRA**

# DESIGN EXPLORATION OF MACHINE LEARNING DATA-FLOWS ONTO HETEROGENEOUS RECONFIGURABLE HARDWARE

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

Orientador: Ricardo dos Santos Ferreira

**VIÇOSA - MINAS GERAIS**
**2023**

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

T
O48e
2023

Oliveira, Westerley Carvalho, 1991-
Design exploration of machine learning data-flows onto heterogeneous reconfigurable hardware / Westerley Carvalho Oliveira. – Viçosa, MG, 2023.
1 dissertação eletrônica (72 f.): il. (algumas color.).

Texto em inglês.
Orientador: Ricardo dos Santos Ferreira.
Dissertação (mestrado) - Universidade Federal de Viçosa, Departamento de Informática, 2023.
Referências bibliográficas: f. 66-72.
DOI: https://doi.org/10.47328/ufvbbt.2023.570
Modo de acesso: World Wide Web.

1. Arquitetura de computador. 2. Fluxo de dados (Computadores). 3. Aprendizado do computador. I. Ferreira, Ricardo dos Santos, 1969-. II. Universidade Federal de Viçosa. Departamento de Informática. Programa de Pós-Graduação em Ciência da Computação. III. Título.

CDD 22. ed. 004.22

Bibliotecário(a) responsável: Bruna Silva CRB-6/2552

**WESTERLEY CARVALHO OLIVEIRA**

# DESIGN EXPLORATION OF MACHINE LEARNING DATA-FLOWS ONTO HETEROGENEOUS RECONFIGURABLE HARDWARE

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

APROVADA: 21 de junho de 2023 .

Assentimento:

gov.br
Documento assinado digitalmente
WESTERLEY CARVALHO OLIVEIRA
Data: 20/09/2023 10:55:03-0300
Verifique em https://validar.iti.gov.br

Westerley Carvalho Oliveira
Autor

gov.br
Documento assinado digitalmente
RICARDO DOS SANTOS FERREIRA
Data: 20/09/2023 14:37:36-0300
Verifique em https://validar.iti.gov.br

Ricardo dos Santos Ferreira
Orientador

*Dedico este trabalho aos meus pais e amigos.*

# AGRADECIMENTOS

Agradeço aos meus pais Maria Elizabeth e Joelcio por todo o suporte e incentivo que precisei. Aos meus avós Levy, Celita, Lilá e Januário (in memoriam) pelo amor e carinho. Aos meus amigos por sempre estarem ao meu lado.
A todos os professores que muito agregaram aos meus conhecimentos.
À Universidade Federal de Viçosa pelas oportunidades oferecidas.
Ao Professor Ricardo dos Santos Ferreira pela dedicação, paciência apoio e compreensão.
O presente trabalho foi realizado com apoio da Coordenação de Aperfeiçoamento de Pessoal de Nível Superior – Brasil (CAPES) – Código de Financiamento 001.

*Whenever, then, anything in nature seems to us ridiculous, absurd, or evil, it is because we have but a partial knowledge of things, and are in the main ignorant of the order and coherence of nature as a whole - B. Spinoza, Tractatus Politicus.*

# ABSTRACT

OLIVEIRA, Westerley, M.Sc., Universidade Federal de Viçosa, June, 2023. **Design Exploration of Machine Learning Data-Flows Onto Heterogeneous Reconfigurable Hardware**. Advisor: Ricardo dos Santos Ferreira.

This work explores the placement and routing of Machine Learning applications dataflow graphs on different heterogeneous Coarse-Grained Reconfigurable Architectures (CGRA). We analyze three different types of processing element (PE) heterogeneity, the first concerning the interconnection pattern, the second being on the kind of operations a single PE can execute, and the last concerning the PE buffer resources. This analysis aims to propose a fair reduction to the overall cost in comparison to the homogeneous CGRA architecture. We compare our results with the homogeneous case and one of the state-of-the-art tools for placement and routing (P&R). Our algorithm executed, on average, 52% faster than VPR 8.1 (Versatile Place and Route), which is an open-source academic tool designed for the FPGA placement and routing phases, reaching better mapping in 66% of cases and achieving the same results in 26% of cases. Furthermore, a heterogeneous architecture reduces the cost without losing performance in 76% of the cases considering multiplier heterogeneity. We propose a novel heterogeneous buffer architecture that minimizes the buffer resources by 56.3% for K-means dataflow patterns. We also show that a heterogeneous border chess architecture outperforms a homogeneous one. In addition, our mapping reaches optimal instances of single tree dataflows compared to classical Lee/Choi and H-Trees.

**Keywords:** Reconfigurable architecture. CGRAs. Placement. Routing.

# RESUMO

OLIVEIRA, Westerley, M.Sc., Universidade Federal de Viçosa, junho de 2023. **Exploração do Espaço de Projeto de arquiteturas reconfiguráveis heterogêneas para grafos de fluxos de dados de aplicações de aprendizado de máquina**. Orientador: Ricardo dos Santos Ferreira.

Esta dissertação explora o posicionamento e roteamento de grafos de fluxo de dados de aplicações de aprendizado de máquina em variadas arquiteturas reconfiguráveis de grão grosso (CGRA). Foram analisados três tipos de heterogeneidade dos elementos de processamento (PE), o primeiro referente ao padrão de interconexões, o segundo sobre o tipo de operações que um único PE pode executar e o último acerca dos recursos disponíveis ao PE. Essa análise almeja propor uma boa redução no custo total em comparação a um CGRA homogêneo. Os resultados foram comparados com o caso homogêneo e uma das ferramenta do estado da arte para posicionamento e roteamento (P&R). O algoritmo proposto executou, em média, 52% mais rápido do que o VPR 8.1 (Versatile Place and Route), que é uma ferramenta acadêmica de código aberto feita para as fases de posicionamento e roteamento de FPGAs, alcançando um melhor mapeamento em 66% dos casos e conseguindo os mesmos resultados em 26% dos casos. Além disso, uma arquitetura heterogênea diminui reduz o custo sem perda de performance em 76% dos casos considerando a heterogeneidade de multiplicadores. Uma nova arquitetura com heterogeneidade de buffers é proposta e ela reduz a quantidade necessária de recursos de buffers em 56.3% para padrões de fluxo de dados do K-means. Também é mostrada uma arquitetura xadrez-borda heterogênea que supera a arquitetura homogênea. Além disso, o mapeamento proposto atinge configurações ótimas para fluxos de dados de árvore única quando comparado aos clássicos posicionamentos de Lee/Choi e árvores H.

**Palavras-chave:** Arquiteturas Reconfiguráveis. CGRAs. Posicionamento. Roteamento.

# LISTA DE FIGURAS

# LISTA DE TABELAS

# SUMÁRIO

# Capítulo 1

# Introduction

In response to the need for greater energy efficiency, flexibility, and performance in computing, traditional integrated circuit technologies have become inadequate, leading to the power wall problem. Computer architects must therefore optimize designs to improve power and energy efficiency, while also allowing for dynamic system reconfiguration and adaptation. This is especially important for expensive new circuit technologies, which have a short lifecycle and high nonrecurring engineering cost. To balance energy efficiency and flexibility in computing fabrics, computer architects must find ways to meet the demands of our ever-changing world (Podobas et al., 2020). As Dennard's scaling has come to an end and Moore's law appears to be on the brink of obsolescence, computer researchers and practitioners are actively pursuing and exploring alternative forms of computing. These include intrusive technologies such as quantum computing and neuromorphic computing, as well as non-intrusive technologies such as reconfigurable architectures (Podobas et al., 2020; Hennessy and Patterson, 2019).

Field-Programmable Gate Arrays (FPGAs) are digital devices that can be reconfigured and programmed to perform a wide range of computational tasks. Compared to other types of processors, such as Application-Specific Integrated Circuits (ASICs), General-Purpose Processors (GPPs), Graphics Processing Units (GPUs), and Digital Signal Processors (DSPs), FPGAs offer unique advantages and disadvantages that are important to consider when choosing the right processing solution for a particular application.

ASICs are custom-built integrated circuits that are designed to perform specific tasks with high efficiency and performance, but the design process is time-consuming and costly. In contrast, FPGAs can be programmed and reprogrammed to perform different tasks and can be quickly prototyped and deployed. However, FPGAs may have lower performance and higher power consumption compared to ASICs.

GPPs are designed to be general-purpose computing devices and can handle a wide range of tasks. They are easy to program and widely available. However, their performance may not be optimal for certain specialized tasks as well as the energy efficiency. GPUs are designed to accelerate the processing of graphics and other highly

parallelizable tasks. However, they are not always the most efficient choice for irregular dataflow tasks, which require more complex computation patterns that are not easily parallelized. DSPs are specialized processors designed to handle digital signal processing tasks, such as audio and video processing. They offer high performance and low power consumption for these tasks but may not be as versatile as other processors.

The challenge of achieving a balance between flexibility and energy efficiency in computing has become increasingly difficult. ASICs offer energy efficiency however it have low flexibility, while GPPs, GPUs, and DSPs provide high flexibility but they are energy inefficient. FPGAs are reconfigurable systems between ASICs (hardwired) and GPP/GPU/DSP processors (programmable at software level). Although, FPGAs have potential to meet the criteria, they are difficult to program and consume more energy than ASICs. To meet the demand for this balance, a novel architecture or an adaptation to FPGA designs , such as heterogeneous grids for example, are needed. This is illustrated by the increasing use of domain-specific accelerators for machine learning and big data, as they can provide the desired balance. Reconfigurable architectures also allow for a much higher level of parallelism than traditional architectures. By utilizing a reconfigurable system, large calculations or operations can be split up and handled at the same time, dramatically reducing the time it would take to complete the same task on a regular system. This makes reconfigurable architectures ideal for applications that require a large amount of parallelism, such as machine learning or video processing (Liu et al., 2019b; Podobas et al., 2020).

## 1.1 Motivation

The biggest motivation for studying and developing faster and more efficient architectures nowadays is the huge demand that come from Machine Learning and Deep Learning Applications. These applications are used in many different areas including speech recognition, image classification, bioinformatics (Krizhevsky et al., 2012; Jo et al., 2017; Thompson et al., 2021). All of those demanding huge amounts of computations, hence the need for more efficient architectures.

In this scenario, dataflow computing (Ferreira et al., 2005) is a computing paradigm that is based on the idea of processing data as it flows through a computational graph, rather than relying on traditional instruction-based processing. In dataflow computing, operations are performed on data as soon as it becomes available, without the need to fetch and decode instructions. The instructions are embedded and placed in the dataflow graph structure.

By avoiding the need to fetch and decode instructions, dataflow computing can eliminate the overhead associated with these tasks. This can result in a lower energy

consumption compared to traditional instruction-based computing. In addition, dataflow computing is well-suited to the parallel processing requirements of machine learning tasks, as it can efficiently process large amounts of data in parallel by exploiting temporal and spacial parallelism.

Dataflow computing requires a hardware architecture that can efficiently implement the computational graph used to process data (Ferreira et al., 2005). While Field-Programmable Gate Arrays (FPGAs) are a popular choice for implementing dataflow architectures (Ferreira et al., 2011a), they require bit-level configuration, which can make mapping the dataflow graph to physical hardware more complex. In contrast, Coarse-Grained Reconfigurable Arrays (CGRAs) offer a word-level configurable architecture that can reduce the complexity of mapping the dataflow graph to physical hardware. CGRAs can be optimized for specific types of dataflow graphs, making them a good choice for implementing dataflow computing architectures. In addition, CGRAs can reduce the placement and routing complexity that is often a challenge with FPGA implementations.

The papers presented in this dissertation are mainly about the placement of dataflow graphs on different types of CGRA (Coarse-Grained Reconfigurable Architectures). The choice of studying CGRAs come from the fact that they have great flexibility in terms of which types of application can be run on them despite being less efficient for specific tasks than, for example, the systolic array (Liu et al., 2020), which is very efficient, but lacks flexibility.

## 1.2 Problem

The placement and routing of graphs on rectangular grids is an NP-Complete problem that has been studied for a long time and has also been approached in different manners, for example we have greedy algorithms (Ferreira et al., 2005), integer linear programming (Walker and Anderson, 2019), SAT-solvers (Donovick et al., 2019), reinforcement learning (Liu et al., 2018), graph traversal approaches (Ferreira et al., 2013b; Vieira et al., 2021), run-time approaches (Ferreira et al., 2014, 2016), genetic algorithms (Silva et al., 2006) and simulated annealing based algorithms like the one used by VPR (Luu et al., 2011), which happens to be the algorithm to which the results of this work were compared to. The placement is particularly difficult because there are many different constraints that have to be taken into account. The first constraint is the communication constraint, which says that the data-flow graph should be placed in such a way that the communication between the nodes is minimized. The second constraint is the resource constraint, which says that the data-flow graph should be placed in such a way that the resources of the CGRAs are used efficiently. The third constraint is the placement constraint, which says that the data-flow graph

should be placed in such a way that the area of the CGRAs is used efficiently.

In addition to the placement problem on CGRAs, which is already a challenge, there is also the problem of equalizing all the paths that lead a signal to the same point, ensuring that a signal coming from two different paths arrives at the correct time wherever it is needed. This problem is not exclusive to CGRA placement scenarios, as it is also present in other placement scenarios such as Superconductive Single Flux Quantum (SFQ) circuits (Pasandi and Pedram, 2019) and quantum-dot cellular automata (QCA) (Walter et al., 2018; Fontes et al., 2018). In (Pasandi and Pedram, 2019), SFQ circuits are mapping by generating a minimum-area solution that is guaranteed to be fully path balanced. In conventional SFQ circuits, a fully path balanced solution is essential to ensure that the circuit operates correctly. In contrast, in QCA circuits or CGRA dataflow scenarios, it is possible to address the unbalanced paths by reducing the throughput (Torres et al., 2018) or adding extra resources (Nowatzki et al., 2018).

One solution to this problem in CGRA domain is to add First-In-First-Out (FIFO) buffers to the CGRA Processing Element units (PE). These FIFO buffers can help balance the path and ensure that signals arrive at the correct time. However, adding FIFO buffers to all PEs can significantly increase the cost and waste resources when the mapping requires only a few FIFOs.

To address this issue, a possible solution is to distribute these resources in a heterogeneous CGRA instead of a homogeneous one. In a heterogeneous CGRA, resources are distributed based on the requirements of the mapping, which can help save resources and reduce costs. For example, if the mapping requires only a few FIFOs, these resources can be concentrated in a subset of the PEs, while the other PEs can be optimized for other tasks.

By creating a heterogeneous CGRA, designers can optimize the resources to better fit the application requirements and reduce costs. This approach provides a more efficient solution compared to a homogeneous CGRA with idle resources. Therefore, the question of where to distribute these resources in a heterogeneous CGRA becomes crucial for optimizing the performance and cost of the system.

The placement problem in this work was addressed using a simulated annealing algorithm, NeoSA. This algorithm was used to find solutions to different arrangements of CGRA grids with three axis of heterogeneity considered, such as the processing element's interconnectivity, functionality and buffer availability. These types of heterogeneity affect the total cost of the CGRA architecture, thus making heterogeneous and cheaper configurations more desirable.

## 1.3 Hypothesis

The main point of this work is to evaluate the premise that a simulated annealing algorithm is a suitable approach to explore the heterogeneity in CGRA architectures. Additionally, this work also aims to test if it is worthwhile to run multiple instances and how the adjustment of architecture resources affects the quality of the solution.

## 1.4 Goals

Following the testing of the hypothesis previously mentioned, this work aims to explore the following:

- Show that the proposed Simulated Annealing algorithm can generate solutions with good quality and running time compared to those of the state of the art
- Demonstrate that such solutions are optimal up to a certain graph size
- Propose domain specific architectures for machine learning dataflows

## 1.5 Contributions

This dissertation presents various contributions to the community, such as a novel Simulated Annealing algorithm for the placement of graphs that generate quality solutions while being flexible and fast. In addition, a GPU version of the algorithm which can run multiple instances simultaneously is also presented. Another contribution is the novel heterogeneous CGRA architecture Chess, that will be further discussed in later chapters.

Furthermore, it is shown that some graphs are impossible to be placed with a balanced solution, demanding an architecture with buffers that equalize the paths. Optimal solutions for binary trees without losing path balancing are also presented. Lastly, different heterogeneous architectures that are suitable for the placement of machine learning algorithms are discussed.

## 1.6 Structure

The dissertation is organized as follows. First, Chapter 2 will provide a brief background to facilitate the understanding of the next sections. Chapter 3 contains the paper A Design Exploration of Scalable Mesh-based Fully Pipelined Accelerators that

was published in the 2020 International Conference on Field-Programmable Technology (ICFPT) in Maui, USA. Chapter 4 contains the paper Heterogeneous Reconfigurable Architectures for Machine Learning Dataflows that was published to the journal Concurrency and Computation: Practice and Experience Special Edition 2022. This paper was an extension of a previous one entitled Design Exploration of Machine Learning Data-Flows onto Heterogeneous Reconfigurable Hardware that was presented in the XXI Simpósio em Sistemas Computacionais de Alto Desempenho (WSCAD 2020) and won the Best Paper Award in the ocasion. Finally, Chapter 5 contains a conclusion summing up the main results and contributions of this work.

# Capítulo 2

# Background

## 2.1 FPGAs and CGRAs

In recent years, the use of computer architecture has become increasingly important to support advances in technology. While Field Programmable Gate Arrays (FPGAs) and Coarse-Grained Reconfigurable Arrays (CGRAs) are types of hardware that can be used to build reconfigurable accelerator architectures, they have different characteristics and capabilities that makes each of them suitable for distinct applications.

FPGAs are integrated circuits that can be programmed to perform logic operations or to emulate the behavior of a processor. These circuits are composed of reconfigurable logic blocks, which can be programmed to perform various tasks. FPGAs are typically used for tasks that require high speed and complex logic, such as video processing, high performance computer network routers, signal processing, and embedded systems. They are also used to quickly prototype new designs, as their reconfigurability allows for rapid development. They are flexible and can be reprogrammed to perform different tasks, but this flexibility comes at the cost of power consumption and lower performance (clock rates) compared to dedicated hardware.

CGRAs are also reconfigurable circuits, but they are designed to be more simple than FPGAs. CGRAs are composed of coarse-grained logic blocks, which are larger than the logic blocks of FPGAs. This allows them to perform more complex operations per cycle such as a float adder or a multiplication, resulting in higher performance. CGRAs are typically used for tasks that require high performance, such as data processing, scientific computing, and artificial intelligence.

Both FPGAs and CGRAs have their own advantages and disadvantages. FPGAs are more flexible than CGRAs, as they can be programmed to perform various tasks at bit level. They are also typically easier to use and more cost-effective. However, FPGAs require a larger number of logic blocks to perform the same task than the CGRAs, which increases the complexity of the placement and route steps. Although, CGRAs are less flexible than FPGAs, modern FPGAs includes coarse-grained modules as DSP blocks and RAM blocks. Therefore, the last FPGA generations have a heterogeneous architecture that mixes bit level lookup-table units, and word-level blocks

as CGRAs. Both are also more energy efficient and can be used for a wider range of tasks. However, CGRAs are more expensive as they are not commercial products as FPGAs.

FPGAs are often used in applications where the workload is not known in advance, or where the workload may change over time. They are also used to prototype designs before they are implemented in dedicated hardware. CGRAs are typically used in applications where the workload is well-defined and does not change frequently. Therefore, the set of CGRA operators is defined at designer level to match to a specific application domain. What is the best set is an important question, that we address in this dissertation.

CGRAs are designed specifically to accelerate certain types of computations, such as those found in signal processing and machine learning applications. They are typically more power-efficient and faster than FPGAs, but they are less flexible and cannot be easily reprogrammed to perform different domains.

In terms of design and implementation, FPGAs require the use of hardware description languages (HDLs) such as VHDL or Verilog to specify the desired digital logic functions. CGRAs, on the other hand, are typically programmed using higher-level languages such as C or C++.

In recent years, there has been growing interest in using HLS (High level synthesis) to program FPGAs. Here are some advantages and disadvantages of using HLS to program FPGAs, including the long synthesis and placement and routing timing:

- Advantages: (a) Faster development time which can save time in development because programmers can focus on the higher-level abstraction of the problem they are solving, rather than having to worry about low-level hardware details. (b) Portability: allowing programmers to write code that can be easily ported across different FPGA devices, as long as they are compatible with the language.

- Disadvantages: (a) Long synthesis and placement and routing times. (b) HLS can limit the control over hardware details, making it difficult to optimize performance for specific use cases. (c) Higher resource usage: Higher-level languages may require more resources (such as memory and clock cycles) to execute than lower-level languages, which can lead to slower execution and larger hardware requirements.

Overall, FPGAs and CGRAs are both useful tools for accelerating computations in computer architectures, but they have different trade-offs in terms of flexibility, performance, and power consumption. The choice between the two will depend on the specific needs of the application.

## 2.2 FPGA

A field-programmable gate array (FPGA) is an integrated circuit that can be customized for different applications after its fabrication. It consists of millions of logic cells that can be configured to execute a wide array of algorithms. Although the design flow of a traditional FPGA is similar to that of an integrated circuit, it offers many advantages over ICs, including improved performance in some cases. The major advantage of an FPGA over an IC is that it can be reconfigured, which is the equivalent of loading a program into a processor. This reconfiguration process can involve the complete resources of an FPGA or just a portion of them.

An FPGA is composed of Look-up tables (LUTs), Flip-Flops (FFs), wires, Input/Output blocks, and a Configuration Memory. The LUTs perform logical operations, the FFs store the results, wires interconnect the elements, the Input/Output blocks allow for exchanging information with external circuits, and the Configuration Memory configures the operations of the LUTs, the routing network and other functionalities.

The basic architecture of an FPGA is composed of individual cells which are identical circuits constructed from flip-flops and combinational logic using Look-Up Tables (LUTs). This structure is sufficient for implementing various algorithms, although the resulting execution efficiency is limited in terms of maximum frequency.

Nevertheless, the synthesis, placement and routing time is one of the major challenge in FPGA as shown in Figure 2.1, where the design of a convolucional neural network (CNN) in an FPGA-based accelerator requires 14 hours to be generated. The placement and routing are the most timing consuming tasks. The FPGA routing is quite complex due the design size and the bit granularity.

## 2.3 CGRA

A Coarse-grained Reconfigurable Architecture (CGRA) is an alternative to Field Programmable Gate Arrays (FPGA). It has the same logic components, but with a simpler configuration and setup process. Compared to FPGAs, CGRAs can perform more complex operations as they operate at a word level, rather than the single bit level. This allows for more efficient use of resources and faster configuration times.

The primary goal of a CGRA is to reduce the complexity and configuration time required by an FPGA while providing a bit width that is more compatible with the application. CGRAs execute logical operations through Processing Elements (PEs) that perform more complex operations compared to the logical processing units of FPGAs. While FPGAs operate at the single-bit level, CGRAs operate at the word level (Wijtvliet et al., 2016).

A CGRA is mainly composed of the following elements:

- Processing Elements: elements that perform logical and arithmetic operations, they can be either homogeneous or heterogeneous. Homogeneous units perform any operation within a specific instruction set, whereas heterogeneous units perform specific operations, such as arithmetic or load/store functions.
- Interconnection Network: used for interconnecting Processing Units. There are a few different types of interconnection patterns for CGRA's, the most used one being the Mesh pattern, in which each processing element connects to its 4 adjacent neighbours. However there are other types of networks, such as a cluster based architecture, a crossbar network, multistage networks among others. For this dissertation, we'll be focusing on the interconnection patterns stated on figure 2.3).
- Configuration Memory: for configuring the operations performed by PEs and routing through the interconnection network.
- The reconfiguration mode of a CGRA can be either static or dynamic. In static mode, the entire CGRA is configured and all processing elements (PEs) execute the same operation until the next CGRA configuration cycle. In dynamic mode, each PE has its own configuration memory and can switch between configurations in run-time, allowing it to execute multiple operations. In general both configuration modes are generated in compile time. Only a few CGRAs are reconfigurable at run-time.

CGRA architectures differ primarily based on the type of network and processing unit used, with each characteristic impacting implementation costs, mapping algorithms, and processing capacity.The granularity also differentiates CGRAs, referring to the number of bits transferred by the interconnect network and manipulated by the processing units. Lower granularity in a reconfigurable architecture provides more flexibility, while higher granularity reduces complexity and configuration memory requirements.

## 2.4 Placement and Routing on CGRAs

Placement involves deciding where to place the different computation nodes of the application on the PEs of the CGRA. Routing involves determining how to connect these nodes using the interconnect resources of the array, such as wires and switches. Finally, scheduling involves determining when each node should be executed on the hardware resources to minimize latency and maximize throughput.

These three steps can be performed as separate optimization problems, which can simplify the solution process. However, it is important to note that these steps are not entirely independent of each other. For example, if the placement of nodes is not done well, then routing may become difficult or even impossible due to congestion and other constraints. In such cases, it may be necessary to re-do placement in order to enable successful routing.

On the other hand, it is also possible to perform placement, routing, and scheduling as a single optimization problem. This can potentially lead to a more optimal solution, but it also increases the complexity of the problem. Moreover, it may not always be feasible to perform all three steps together due to the high computational cost and resource requirements. In this work, the three steps are performed as separate problem, and our main focus is to find an optimal or near optimal placement to simplify the next steps.

This work focuses mainly in the placement of graphs on CGRAs on a flexible architecture that can be used for diverse applications in exchange for a higher efficiency found in domain-specific architectures. In the following subsections there will be a brief description of how a CGRA works and the different topologies that will be used in this dissertation. In addition we want to find out if the time needed to complete a the process of placement and routing can be decreased. For instance, as one can infer from Figure 2.1, the whole process for a CNN benchmark can take up to 14 hours to be completed (Guo et al., 2023).

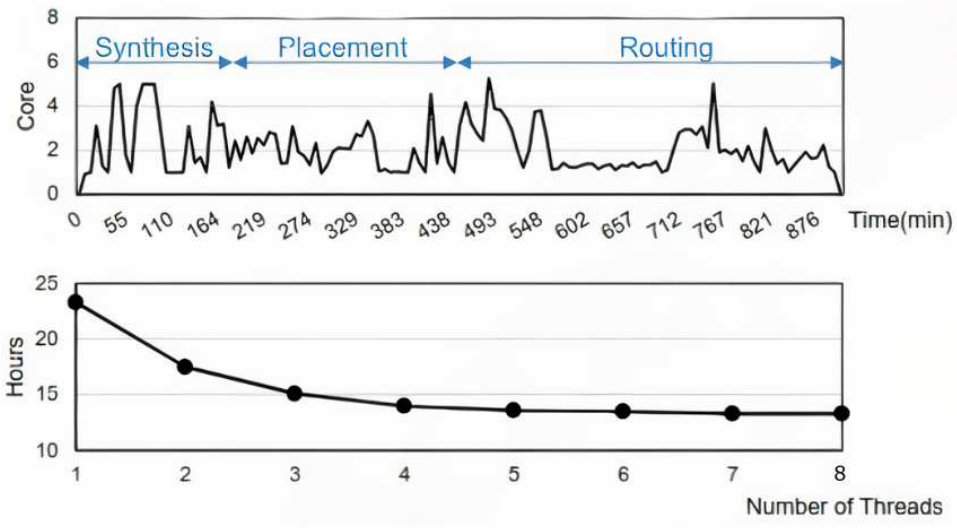



Figura 2.1: The figure above illustrates the active CPU cores during the implementation of a CNN benchmark using Vivado on a 56-core server. The entire implementation process lasts approximately 14 hours, with an average CPU utilization of 2.1 cores. The lower figure presents the runtime as the number of threads is increased. (Adapted from (Guo et al., 2023))

### 2.4.1 Dataflow Graphs and Interconnection Patterns

Dataflow graphs (DFGs) are structures that contain all the information about a particular routine arranged on a set of nodes and a set of edges. A DFG Ferreira et al. (2005) is a graphical representation of a program code where nodes represent instructions and the data flows through the graph. It is a way to visualize and analyze the flow of data within a program. In a DFG, the nodes correspond to individual instructions or operations, while the edges represent the flow of data between these instructions. As mentioned in the previous section, the main goal of this work is finding means to map DFGs from machine learning applications to the CGRA.

Assuming an example of simple sequential graph: Input data → Load Data → Multiply → Add → Store Data → Output. In this example, the input data is first loaded from a memory stream into the CGRA. Then, the loaded data is multiplied by a constant value using a multiply unit. Next, the multiplied data is added to another value using an adder unit. Finally, the result is stored back into memory stream through a store data unit, and the final output is produced. This dataflow graph could be mapped onto a CGRA by assigning the various operations to different processing elements (PEs) on the array. For example, the load data unit could be mapped onto a PE that is designed to read data from a memory stream, while the multiply and add units could be mapped onto separate PEs that specialize in these operations. The store data unit could be mapped onto a PE that is designed to write data back to memory stream.

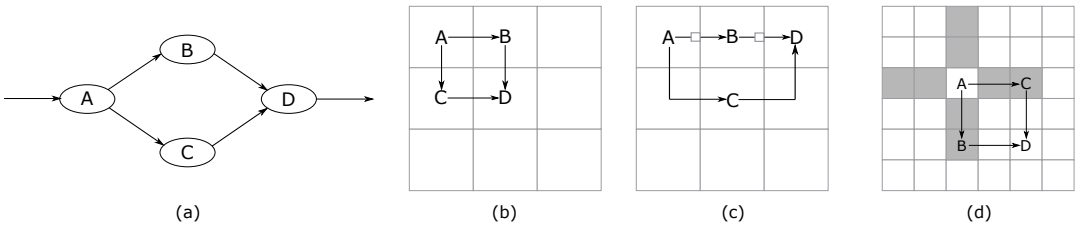



Figura 2.2: (a) A simple dataflow graph; (b) Mapping with the lowest possible wirelength in a mesh grid; (c) Mapping with greater total wirelength and buffers; (d) 1-hop interconnection pattern with an optimal placement.

Figure 2.2(a) shows a very simple dataflow with four nodes (A - D) and the edges that connect them representing the flow of information. In figure 2.2(b) the dataflow is mapped onto a mesh grid in an optimal manner, note that every node in the graph is placed in the nearest cell possible to its neighbouring nodes. This reduces the overall wirelength and by consequence the cost of the architecture. However, it is not always possible to place all nodes in an optimal manner, in those cases longer wirelengths are needed along with buffers that stall the signal, preventing a delay mismatch. A

mapping where this occurs is presented in figure 2.2(c).

The grid interconnection pattern in figure 2.2(b)(c) is called mesh, meaning that every cell only has access to its four immediate adjacent neighbours. Another pattern that is commonly used and provide more flexibility to the placement algorithms is called 1-hop (or meshplus) and is depicted in figure 2.2(d). In this configuration, each cell has access to its four adjacent neighbours and to the ones beyond them as highlighted in grey. Since that is possible, the mapping in figure 2.2(d) also has an optimal cost. Chapters 3 and 4 present two published papers proposed in this work that explore different types of processing element heterogeneity, and will bring more detail into that. One of the contributions of these papers is the Chess interconnection pattern, which is a combination of the already discussed mesh and 1-hop architectures (shown more clearly on figure 2.3). The novel architecture Chess has an average cell degree of 6, with all mesh cells having a degree of 4 and all 1-hop cells having a degree of 8. This means that on average, each Chess cell can reach 6 other cells. Further detail on Chess can be found on the referenced chapters.

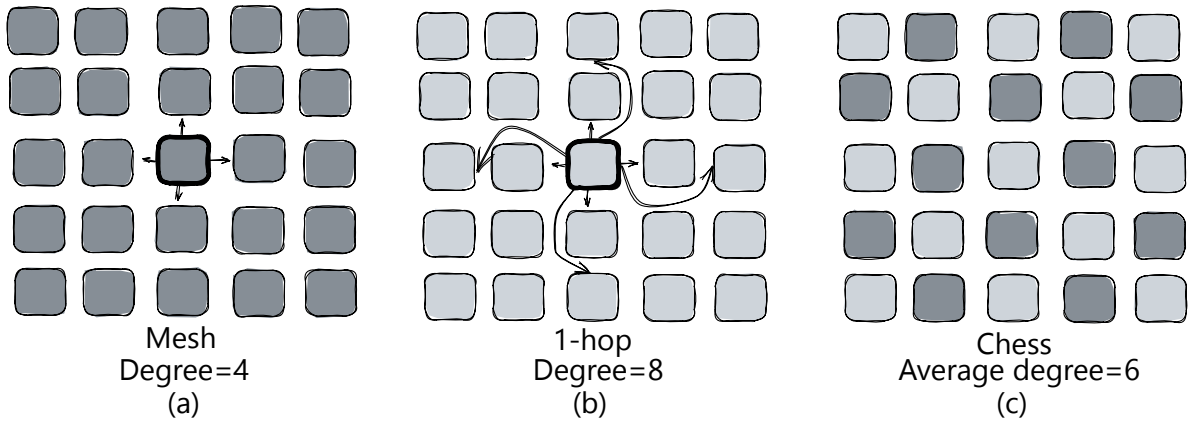



Figura 2.3: (a) Mesh architecture; (b)1-hop architecture; (c) Chess architecture.

## 2.5 Placement and Routing heuristic approaches

In this section we will present different heuristic approaches for the mapping of dataflow graphs on CGRAs.

Integer linear programming (ILP) (Walker and Anderson, 2019) and satisfiability solving (SAT-solvers) (Donovick et al., 2019) are two computational techniques used to find global optimal in various problem domains, including CGRA placement and routing. However, these methods often face scalability challenges when compared to heuristic approaches. This limitation is particularly relevant in the context of coarse-grained reconfigurable architectures, where the maximum graph size is typically cons-

trained to around 30. Consequently, their performance deteriorates as the problem complexity increases, making them impractical for solving problems that surpass a certain size threshold.

Enriching the interconnection structure of CGRA with crossbar (Ferreira et al., 2013a) or multistage networks (Ferreira et al., 2011b) simplifies the placement process by providing global communication capabilities. With these network architectures, each node can directly communicate with any other node in the system, eliminating the need for complex routing algorithms and enabling straightforward placement of components. However, it is important to note that while this solution simplifies placement, it also introduces scalability concerns due to the increasing cost of the network as the system size grows.

In a crossbar network, the cost of the interconnection fabric is proportional to the square of the number of nodes ($O(n^2)$), where $n$ represents the number of components in the system. Therefore, the scalability of crossbar networks is limited due to the quadratic increase in cost, making them less viable for large-scale systems. On the other hand, multistage networks offer a more scalable alternative (Silva et al., 2019). The cost of a multistage network scales with ($O(n \log n)$). While multistage networks alleviate the scalability concerns compared to crossbars, they still introduce a cost increase that limits the design size. A mesh or grid network architecture provides a constant local cost and offers scalability advantages, but it also introduces challenges in terms of placement complexity.

Heuristic approaches provide approximate solutions and scalability without the guarantee of optimality. By employing techniques such as local search based on graph level (Ferreira et al., 2005), genetic algorithms (Silva et al., 2006), graph traversal approaches (Canesche et al., 2020, 2021), or simulated annealing (Luu et al., 2011; Oliveira et al., 2020a), heuristics can efficiently explore large solution spaces and find good-quality solutions within reasonable time frames.

In this work, the placement solution is based on simulated annealing, and we will show that reaches near or optimal placement solutions, manages medium size graphs (up to 200 nodes), and executes in a few seconds.

### 2.5.1 Simulated Annealing

The first proposed approach was the DRESC compiler for CGRA mapping that uses Simulated Annealing (Mei et al., 2003b). This technique is used to explore different placement and routing options to find a valid scheduling of all operations and data dependencies in a CGRA, while minimizing the total routing cost. The algorithm starts by generating an initial schedule that satisfies dependence constraints but with possibly over-subscribed resources. Then, it iteratively reduces resource overuse and

explores different placement and routing options via simulated annealing until a valid placement and routing of all operations and data dependencies are found.

The cost function used during the simulated annealing process is based on the total routing cost, which is the combined resource consumption of all the placed operations and the routed data dependencies. However, this technique has a long convergence time, especially for large dataflow graphs, due to the huge number of possible routes that need to be evaluated (Li et al., 2022).

In section 2.6, more detailing on how a Simulated Annealing algorithm works will be provided along with a deeper understanding of the one used on the experiments done in this work.

### 2.5.2 Modulo Scheduling

The DRESC compiler employs a node-centric modulo scheduling approach where nodes are scheduled and placed first, followed by edge routing. In contrast, edge-centric modulo scheduling (EMS) prioritizes routing efficiency over operation assignment. In node-centric approaches, operations are placed based on heuristic routing costs. The modulo scheduling uses a dynamic CGRA reconfiguration which can switch between configurations in run-time, allowing the processing elements to execute multiple operations. However, when more than one configuration is required, the initialization interval increases and the throughput is reduced. The mapper selects the best candidate for each operation, attempting to route edges from previously-mapped nodes to the candidate. On the other hand, edge-centric approaches prioritize routing efficiency, with the routing function containing the placement decision. When scheduling an operation, the scheduler picks an edge from previously-placed producers or consumers and recursively routes those edges until a suitable slot is found (Li et al., 2022).

While edge-centric approaches can find solutions faster and achieve better quality mapping, they may fall into local minima due to their greedy nature. Prioritizing intelligent routing cost metrics can help address this problem. The quality of mapping with specific priorities depends on efficient heuristics for assigning these priority values to both the operations and the resources (Park et al., 2008).

### 2.5.3 Spatial-Pipeline-Routing

SPR (Spatial-Pipeline-Routing) is a mapping tool (Friedman et al., 2009) designed to efficiently map CGRA applications onto FPGAs. The SPR performs three distinct steps: scheduling, placement, and routing. The scheduling step arranges operations based on data and control dependencies, the placement step assigns operations to

functional units, and the routing step maps data signals between operations using wires and registers.

SPR uses an Iterative Modulo Scheduling (IMS) algorithm (Rau, 1994) for the scheduling step, which is inspired by VLIW (Very Long Instruction Word) architecture. IMS assigns operations to a schedule based on the start time of each instruction, considering resource constraints and dependencies. SPR also uses Simulated Annealing (Kirkpatrick et al., 1983) for placement, with a cooling schedule inspired by VPR (Betz and Rose, 1997), and PathFinder (McMurchie and Ebeling, 1995) and QuickRoute (Li and Ebeling, 2004) for pipelined routing.

Historically, FPGA mapping tools have used Simulated Annealing for placement and PathFinder for routing. VPR (Betz and Rose, 1997) is a standard mapping tool for FPGA architecture exploration, but it is limited to FPGAs. SPR adopts similar algorithms but extends them to support CGRAs, such as multiplexing of resources across cycles and solving placement and routing issues that arise when using a fixed frequency device. QuickRoute is used to solve the pipelined routing problem.

Overall, SPR is a mature mapping tool (Friedman et al., 2009) that effectively combines VLIW-style scheduling and FPGA placement and routing algorithms for CGRA application mapping. It aims to provide high-quality mappings and support a wide range of architectural features for CGRAs.

## 2.6 Neo Simulated Annealing

while previous work uses simulated annealing in dynamic configurable CGRAs, our approach focus on static CGRAs. A static approach for placement in reconfigurable architectures refers to a method where processing elements (PEs) are allocated to specific components and remain fixed throughout the execution of the system. In this static configuration, the PEs are not shared by multiplexing them in time. Instead, each PE is configured once and can operate independently, potentially working in a pipeline fashion to maximize throughput. Furthermore, static placement eliminates the overhead associated with reconfiguring PEs during runtime. Once the initial configuration is set, the PEs remain in their designated roles, avoiding the need for frequent reconfiguration. This reduces the time and energy required for reprogramming and enhances the overall system efficiency.

In a static placement approach, the parallelism degree of the system is equivalent to the number of operations in the graph, achieving a throughput of $n$ operations per cycle. However, when employing a time-multiplexing approach as previous work (Mei et al., 2003a; Friedman et al., 2009) with a limited number of PEs, denoted as $X$, where $X$ is less than the total number of operations in the graph ($n$), the maximum achievable parallelism degree is limited to $X$. For example, consider a CGRA

with 16 PEs, as proposed by (Mei et al., 2003a). If this CGRA is utilized to execute a graph with 35 operations, it would require at least 3 clock cycles to complete the execution of the graph containing 35 operations.

A simulated annealing algorithm, just as explained in the section before, is a probabilistic technique for approximating the global optimum of a given function. The algorithm works by simulating the physical process of annealing, which is the gradual cooling of a material in order to increase its strength and minimize defects (Bertsimas and Tsitsiklis, 1993). The idea is to start with a random solution and then slowly move towards better solutions by making small changes. The algorithm can be used for both minimization and maximization problems.

There are many different ways to implement a simulated annealing algorithm, but all of them share the same basic structure. The first step is to define a function that needs to be optimized. This function will be used to evaluate the quality of a given solution. The second step is to randomly generate a starting solution. The third step is to define a temperature schedule, which dictates how the temperature of the system should be changed over time. The fourth step is to start the annealing process by making small changes to the current solution and evaluating the new solution using the objective function. If the new solution is better than the current solution, it is accepted and the current solution is updated. If the new solution is worse than the current solution, it is accepted with a probability that is based on the current temperature. The fifth step is to repeat step four until the temperature reaches a minimum value. At this point, the current solution is the best solution that has been found and the algorithm terminates.

There are a few different ways to define the objective function that is used to evaluate solutions. One common approach is to use a function that is specifically designed for the problem at hand. Another approach is to use a general-purpose function such as the sum of squared errors. The choice of objective function will have a significant impact on the performance of the algorithm, so it is important to choose wisely. In this work the objective function was chosen to be the total wirelength cost.

The temperature schedule is another important parameter of the algorithm. There are a few different ways to define the temperature schedule, but the most common approach is to start with a high temperature and gradually decrease it over time. The hope is that the system will be able to explore a large portion of the search space at the beginning when the temperature is high and then slowly converge towards the global optimum as the temperature decreases.

One can make small changes to the current solution in many ways. One common approach is to randomly select a parameter and perturb it by a small amount. Another approach is to select a parameter at random and then move it towards its best known value. The choice of how to make small changes will also have a significant impact

on the performance of the algorithm. For the algorithm used in this thesis, the chosen perturbation was the swap between two nodes in the architecture grid. As stated before, the objective function is the mapping total wirelength, so if after a swap there is a reduction in the wirelength, it means that that configuration is better and therefore it is taken by the algorithm.

Let us consider the placement of the simple graph depicted in figure 2.4(a) onto a mesh architecture. The total wirelength cost of a particular mapping can be calculated by adding up the Manhattan distances between each pair of cells used by the placement of each graph edge minus 1, to consider neighbour cells with zero connection cost. Given that, the total cost of the example placement in figure 2.4(b) is equal to 4 (the number of red arrows for simplicity). When a swap of the contents of cells 0 and 4 happens, node A gets closer to the cells where its neighbours are currently placed, hence reducing the total cost to 2.

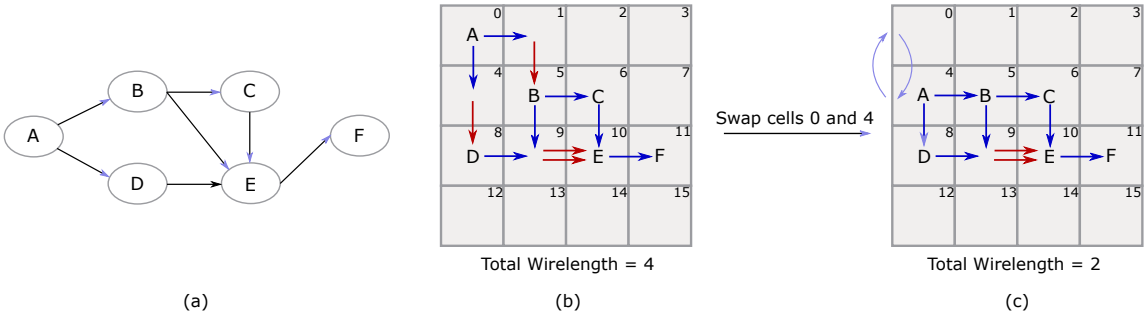



Figura 2.4: (a) A simple dataflow graph; (b) Example mapping with total wirelength 4; (c) Mapping obtained after swapping nodes 0 and 4, with total wirelength 2.

One could think of a greedy algorithm that would sweep the whole grid performing swaps and checking if the cost lowered, choosing the solution with the best possible outcome. However, such an algorithm will most certainly hit a local minimum and not return a satisfactory solution. To contour that, a Simulated Annealing approach, named Neo Simulated Annealind (NeoSA) is used as shown in Algorithm 1.

We start our grid with a random placement of the dataflow graph. Different approaches were tested, like starting the grid with a solution known to have a lower cost, but the algorithm did not seem to do any better in those cases. Then we have to compute the overall wirelength cost of such placement. This is a costly function, since it has to go through all the graph edges, compute the manhattan distances between the cells in which the edge nodes are positioned and add them all up. After that the temperature of the system is set to 100 and then starts the main loop of the algorithm that runs until the temperature breaks the 0.00001 minimum threshold. In each iteration of this loop, the grid is swept two times to choose the two cells that will be

**Algorithm 1** Simulated Annealing Algorithm - NeoSA

```
1:  T ← initial temperature
2:  currentCost ← calculateCost()
3:  while T > 0.00001 do
4:      for n1 ← firstCell to lastCell do
5:          for n2 ← n1.next() to lastCell do
6:              swap(n1, n2)
7:              nextCost ← nextCost − n1.edges() − n2.edges()
8:              annealProb ← exp(−1 × (nextCost − currentCost) / T)
9:              if nextCost ≤ currentCost || random() < annealProb then
10:                 currentCost ← nextCost
11:             else
12:                 undoSwap(n1, n2)
13:     if currentCost = 0 then
14:         break
15:     T ← T × 0.999
```

subjected to the swap. We then removed the costs of the edges connected to the nodes in question, swapped them and then added the costs on their new positions. With this we avoide computing the overall cost in each iteration of the innermost for loop which would greatly increase the running time of the algorithm. In a plain brute force algorithm, if the new cost is lower than the current cost, the algorithm takes it and then goes on. However in this simulated annealing approach the difference is that a worse cost has a chance of being taken based on a annealing probability. If we reach a cost 0 solution, the algorithm stops and have an optimal solution.

# Capítulo 3

# A Design Exploration of Scalable Mesh-based Fully Pipelined Accelerators

## 3.1 Introduction

Domain-specific hardware is a promising architecture choice to achieve power-efficient and high-performance computation. This work focuses on CGRAs as a one part of the heterogeneous computing landscape. In particular, recent work (Canesche et al., 2020) present CGRAs processing spatial dataflows as an approach to exploit parallelism. In this work, we focus on improving the mapping algorithms and performing architecture exploration of the connecting topologies for mesh-based CGRAs.

The contribution of this work is twofold. First, we propose and evaluate a novel Simulated Annealing (SA) mapping algorithm for mesh-based CGRAs, including a new topology called *Chess*. We propose to search for a large SA solution space to find the best possible solution in reasonable amount of time. To explore and exploit CGRA architectures, we have created an experimental framework that takes in a parameterized configuration file and virtually creates, maps a dataflow to, and evaluates a CGRA architecture. We measure the quality of our results based on wire length and required FIFO stages to map a fully-pipelined design correctly. Our second significant contribution is to push the state-of-the-art for CGRAs, which can, currently, find optimal solutions for dataflow graphs up to 20 nodes, to up to 70 nodes. We compare our approach to three others: CGRA-ME (Walker and Anderson, 2019), SAT solvers (Donovick et al., 2019), and VPR (Luu et al., 2011). We can also map graphs with up to 200 nodes by exploring interconnection architectures that provide flexibility to the mapping algorithm.

The remainder of this paper is structured as follows: Section 3.2 provides details on the CGRAs implementation and the respective algorithms that map designs to them. Section 3.3 describes the architecture topologies that we explore for CGRAs, and section 3.4 describes our automated tool for creating an architecture and evaluating those architectures. Section 3.5 describes our mapping algorithm. Section 4.5 shows results for this case study. Finally, section 4.7 concludes the paper.

## 3.2 Background

### 3.2.1 Mesh, Wire Length, and Delay Mismatch

A mesh is a well-known low-cost and scalable interconnection network for CGRAs, where each cell has 4 adjacent local connections (except on the corners and borders). To calculate a cost to map a dataflow to a mesh, assume that local connections have wire cost=0, Figure 3.1(a-c) shows a dataflow design in (a) and various mapping instances in (b), (c), (d), noting that non-local routing wires are colored gray and local wires are bolded. In the (b) instance the longest wire ($lw$) has $cost = 1$, and total wire ($tw$) $cost = 2$. The $tw$ includes both the mapped edges $a \rightarrow c$ and $c \rightarrow d$ (both having $cost = 1$). Figure 3.1(c) depicts an optimal mapping where $tw = 0$ since the mapping only uses local connections to neighbors. Figure 3.1(d) shows another example where the longest local wire $lw = 2$ and total wire $tw = 2$.

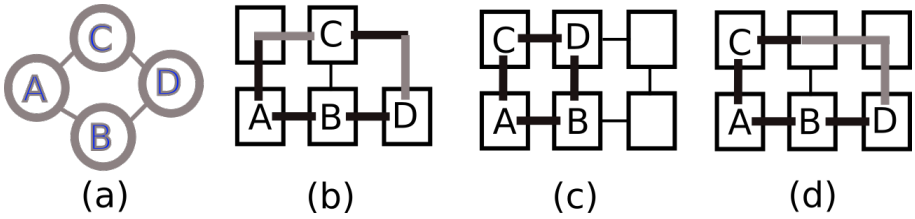



Figura 3.1: (a) Dataflow; (b) lw=1,tw=2; (c) lw=tw=0; (d) lw=2,tw=2.

Most dataflow implementations take advantage of pipeline techniques to improve the maximum clock frequency for an architecture, and in a fully-pipelined dataflow mapping, there can be a delay mismatch between two or more paths if the signals have different arrival times. Figures 3.1(b) and 3.1(d) show two examples of a delay mismatch.

### 3.2.2 Delay FIFO

To matchup pipeline paths, we use delay FIFOs, as described in (Nowatzki et al., 2018), which solves the delay mismatch. A delay FIFO with length $L$ implements a variable size FIFO, where $L$ is the maximum length. We fix the FIFO size at mapping time and implement the delay FIFO at the PE operation inputs. Figure 3.2(a) shows how to solve the mismatch with a FIFO $L = 2$ at node $d$ for the example from Figure 3.1(d). It is possible to reduce the FIFO length to 1 by splitting the FIFO across the path $a \rightarrow b \rightarrow d$, as shown in Figure 3.2(b).

There are graph patterns for mesh-based architectures, where there is no optimal solution, i.e., without FIFOs to perform delay matching. Suppose we compute the

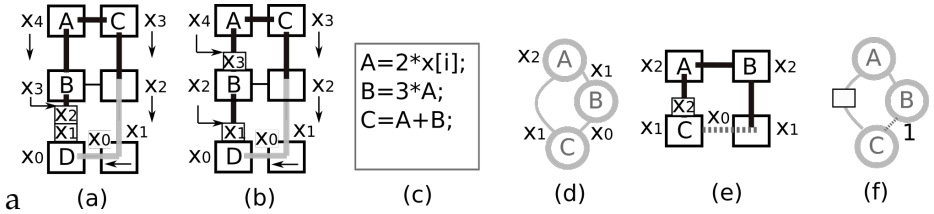



Figura 3.2: (a) FIFO $L = 2$; (b) FIFO $L = 1$; (c) Stream $x_i$ and $x_{i+1}$; (d) Dataflow pattern without optimal mapping in Mesh; (e) Minimal Mesh Mapping with FIFO; (f) Simplified mapped dataflow.

convolution stream $c = 2 * x[i+1] + 6 * x[i]$, where the dataflow is shown in Figure 3.2(c-d). For this example, the minimal mapping requires at least a FIFO size 1, as shown in Figure 3.2(e). Finally, Figure 3.2(f) shows a simplified view of the dataflow. We add the label 1 to the long wire connection (edge $b \rightarrow c$) and use a block to represent a delay FIFO (edge $a \rightarrow c$).

### 3.2.3 Mapping, Routing, Timing, and Metrics

In a stream-based system, the compiler performs Mapping, Routing, and Timing (MRT (Nowatzki et al., 2018)). The mapping or placement phase (M) consists of assigning physical cells to the dataflow PE nodes. The routing phase (R) generates wiring paths to map the dataflow edges. Finally, the timing phase (T) guarantees that there are no delay mismatches. It is possible to perform these three phases separately or simultaneously. Nowatzki *et al.* (Nowatzki et al., 2018) provides an analysis of these trade-offs for very limited graph sizes ($\leq$ 20 nodes). In this work we evaluate dataflows ranging from 20 to 200 nodes, and show that even in the separation of phases approach, we can find optimal and near-optimal solutions with fast execution times in comparison to the results shown in (Nowatzki et al., 2018; Walker and Anderson, 2019; Donovick et al., 2019).

For ASICs and FPGAs, a typical cost function metric is the total wire length as this impacts routing resource costs and hence the area of these chips. For CGRAs, the FIFO size is a more appropriate metric that measures the quality of mappings for a fully pipelined architecture (Nowatzki et al., 2018). The reasons for this are: first, the wire resources are predefined once an architecture is chosen, the routing capacity does not change, being set to bytes or larger (typically 16, 32, or 64-bit transfers of data). Regardless of the routing resources that are used in the CGRA, the final throughput will always be optimal for a valid solution, i.e., generating one result per clock cycle in a fully pipelined architecture. Second, adding FIFOs does not increase the latency because shorter paths are delayed to balance the longest path's overall delay. A routing solution that requires a few long wires will result in a timing solution

that requires small FIFOs or none.

## 3.3 CGRA Interconnection Topologies

As already mentioned, there are simple dataflow graphs for which there is no possible mapping without FIFOs in a mesh topology. This section presents the one-hop, which is a well-know CGRA topology, and a novel topology named Chess. Previous work on CGRA design exploration (Bansal et al., 2004) shows that one-hop mesh has enough flexibility to map dataflows onto CGRAs efficiently. In a one-hop mesh, each cell has 8 adjacent nodes.

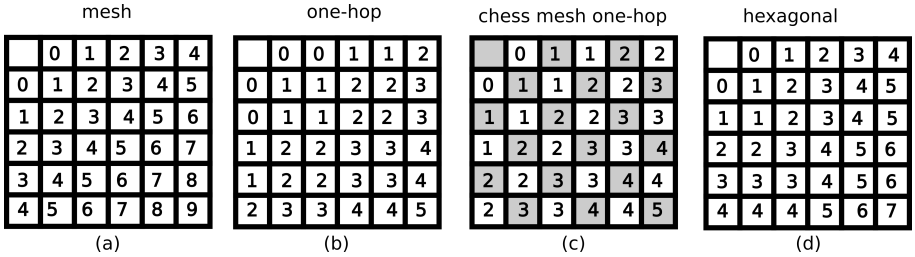



Figura 3.3: All distance from the top-left corner: (a) Mesh; (b) One-hop; (c) Chess; (d) Hexagonal.

One-hop mesh reduces the distances by a factor of 2 in comparison with mesh. Figures 3.3(a) and 3.3(b) show all wire distances ($lw$) from the top-left corner for mesh and one-hop, respectively. However, one-hop has double the number of local interconnections requiring significant hardware resources. Additionally, if we target optimal mapping without FIFOs, the advantage of one-hop is the increased number of adjacent nodes in comparison to mesh. One also might suggest using diagonal connections such as north-east or north-west. However, previous work has already shown that one-hop topologies produce the best results (Bansal et al., 2004).

Our work proposes a novel hybrid topology named Chess (mesh/one-hop). Similar to a chessboard where there are black and white alternating cells, our Chess interconnection pattern has black cells with mesh connections, and white cells with one-hop connections. Although the chess pattern has, on average, 6 local connections per cell, the distance measure is quite similar to one-hop, as shown in Figure 3.3(c). Additionally we show a hexagonal or honeycomb pattern (see Figure 3.3(d)), which also has 6 adjacent cells. Moreover, the chess pattern reduces distance. For instance, the distance from the left top corner to the bottom right corner is 5 and 7 for the Chess and hexagonal, respectively (see Figure 3.3(c-d)). Furthermore, Chess performance is close to one-hop (8 connections) with the cost of a hexagonal (6 connections), as shown in our experiments.

## 3.4 Code Generation for FPGA and ASICs

We also propose a tool to automatically generate the CGRAs, targeting the evaluation of area and power tradeoffs against the two technologies - FPGAs and ASICs. Our tool is a parameterized hardware generator that evaluates the quality/cost of a dataflow mapping as a function of mesh size, interconnection topology, internal PE functionality, delay FIFO size, and the implementation cost of all of these on the target technology. Moreover, our CGRA design includes the configuration memories, and the resulting design is partially reconfigurable at run-time.

Each CGRA consists of a heterogeneous set of PEs and an interconnection network that defines which PEs connect to it. Each PE has a functional unit (FU) and a wrapper interconnection interface. For each PE, the configuration file specifies: 1) word size; 2) delay FIFO length, 3) functional unit operation and opcodes; 4) neighborhood interconnection with other PEs.

We evaluate the topologies and delay FIFO impact on FPGA and ASICs to implement different CGRAs. Figure 3.4 shows the area for CGRAs with different configurations implemented as an overlay in FPGA technology. We use Intel's Quartus for the synthesis and target an ARIA10 FPGA.

The area results are then normalized to a baseline as follows. For each grid size, our baseline is the mesh topology without any FIFOs and bypassing routing resources. We considered grid sizes ranging from 81 up to 1,296 PEs.

The Chess and one-hop wires and muxes add an overhead of 1.2 to 1.3× and 1.4 to 1.5×, respectively. Therefore, these topologies that potentially do not need FIFOs can improve the area overhead. Finally, the clock frequency for all designs ranges from 250 MHz to 450 Mhz. All of these designs have a 16-bit word size. Furthermore, we design a 4-bit CGRA with a heterogeneous set of PEs to use all the FPGA DSP units. Half of the PEs have 8 operations that include multipliers, and the other half have no multipliers. The 46×64 CGRA use 40% of the FPGA ALMs with a total of 3,036 PEs, and this CGRA achieves a peak performance of 1.2 Tera 4-bits ops.

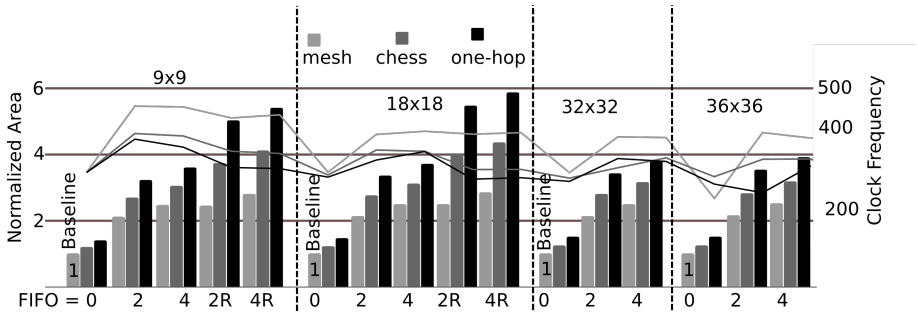



Figura 3.4: Normalized area in number of ALMs for FPGA Overlay design.

We synthesize a set of CGRA designs by using the ASIC flow FreePDK45TM 45nm variant of the FreePDKTM process design kit (FreePDK45, 2019). Figure 3.5 shows the total silicon area for the architectures and the fraction of the area occupied by the registers in black. First, we use the $18 \times 18$ design without routing resources and delay FIFOs as a baseline. The $18 \times 18$ is 3.8× bigger than the 9×9, which shows that the design grows linearly from 81 to 324 PEs. The Chess and one-hop wires and muxes add a small overhead of 1.05× and 1.08×, respectively. Therefore, the topologies without FIFOs improve the mapping without an area overhead. The clock frequency is approximately 1 GHz for these architectures.

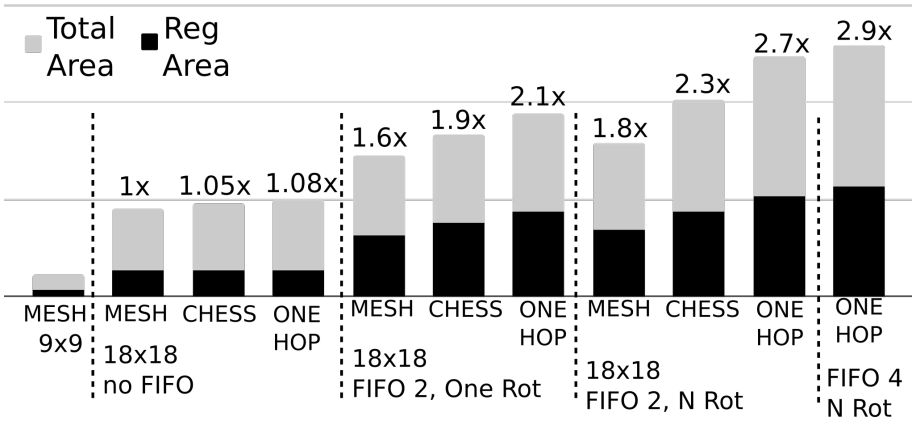



Figura 3.5: Total area and register area for CGRA (ASIC Flow).

## 3.5 Mapping Dataflows with Simulated Annealing

Recent works (Walker and Anderson, 2019; Nowatzki et al., 2018; Liu et al., 2019a) propose integer linear programming (ILP), SAT solvers, and deep reinforcement learning for CGRA mapping. However, even when using state-of-the-art ILP and SAT solvers, the execution time is in the order of minutes. We propose to use a simulated annealing (SA) approach, which is a well-known approach for mapping designs to CGRAs (Mei et al., 2003a) and FPGAs (Luu et al., 2011). Instead of performing random swaps, we propose a heuristic to scan the grid and perform the swaps sequentially.

Our approach performs only the SA mapping with local routing. Next, we execute the detailed routing for the long wires, and finally, we perform the timing step where we consider the FIFO cost function. We also propose to explore the solution space by performing multiple executions of the full SA. Each instance starts from a random permutation of an initial mapping, and the SA minimizes the wire length for each instance. As mentioned in Section 3.2.3, the wire length is just a mapping guideline,

and our goal is to minimize the FIFO size. The mapping does not consider the delay matching requirements, and therefore, the reduced complexity needs less execution time. Furthermore, we implement a GPU version of our SA, where we execute each SA in its thread, differently from previous approaches that parallelize the internals of an SA instance with multiple threads.

## 3.6 Experimental Results

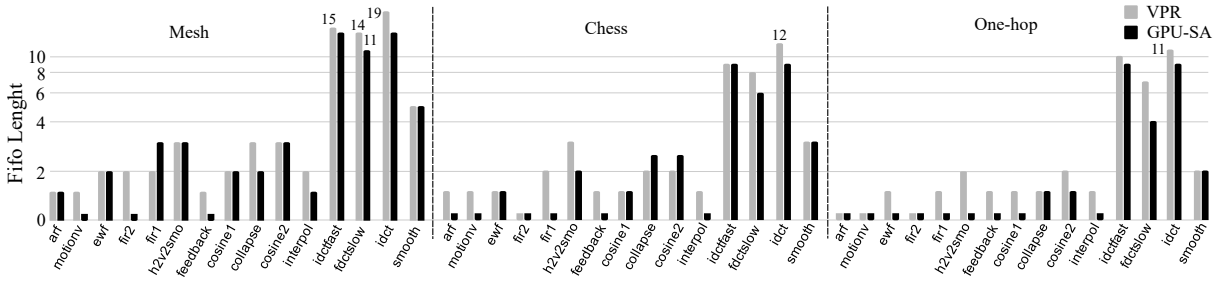



Figura 3.6: FIFO length for UCSB express suite dataflows (University of California, 2020) in three topologies for 1,000 Instances of VPR and our SA approach. Low is better.

We evaluated our mapping technique by using public domain benchmarks extracted from three related works: (a) the CGRA-ME design exploration framework (Walker and Anderson, 2019); (b) a Mediabench set from the University of California, Santa Barbara (University of California, 2020);(c) Stream-Join dataflow control model (Dadu et al., 2019). We include the Stream-Join for irregular flows (Dadu et al., 2019), which handles sparse data with simple dataflow operators and fewer dataflow edges and nodes. We evaluate four dataflow examples from this set (Dadu et al., 2019). In total, we evaluate 37 benchmarks: 14 from CGRA-ME (Walker and Anderson, 2019), 8 stream-joint graphs (Dadu et al., 2019), and 15 from Mediabench (University of California, 2020). In all the experiments we map to a square array of PEs where an $N \times N$ array has an $N = \left\lceil \sqrt{|Nodes|} \right\rceil$.

First, we propose to compare our approach to the ILP-based approach from (Walker and Anderson, 2019) and the SAT-solver approach presented in (Donovick et al., 2019). With a timeout of 10 minutes, the ILP approach (Walker and Anderson, 2019) maps only 22 of the 37 benchmarks with the maximum graph size of 23 nodes, and the SAT approach (Donovick et al., 2019) maps 25 with the maximum graph size of 44. Our approach successfully maps all 37 benchmarks with the maximum graph size of 196, and the average mapping time is a few seconds.

CGRA-ME (of Toronto, 2019) is a framework created for design exploration for CGRA architectures, similar to ours. Although their mapping approaches are generic,

the execution time for the current mapping based on ILP and SA (Walker and Anderson, 2019) are prohibitively long (minutes or hours) for dataflows with more than 20 nodes.

As ILP/SAT solvers do not scale, we propose to compare our SA approach to the VPR tool (Luu et al., 2011). Like our approach, VPR does not perform the timing step for fully pipelined CGRAs. Therefore, we apply our routing and our timing approach to the VPR placement. We execute 1,000 VPR instances with random initial placements to perform a fair comparison. Finally, as the two benchmark sets from (Dadu et al., 2019; Walker and Anderson, 2019) are limited to 25 nodes, we use the UCSB mediabench set (University of California, 2020), in addition to *k-means* benchmarks to compare the SA approaches. This suite is a representative graph selection from *Mediabench*, where the graph size ranges from 10 to 300 nodes.

Figure 3.6 shows the best mapping in 1,000 instances for 15 dataflows in three topologies: mesh, the novel Chess, and one-hop. Our mapping on one-hop reaches the optimal solution for all graph sizes smaller than 66 nodes, while VPR (Luu et al., 2011) reaches the optimal solution for graph sizes smaller than 32 nodes. The average FIFO length for the proposed Chess topology is 2.2 compared to 1.7 obtained by the one-hop and 4.1 for the mesh. Therefore, the Chess reduces the one-hop cost as shown in Section 3.4, and reaches a closer performance to one-hop and 1.9x better than mesh. Between 72 and 102 nodes, there are 3 graphs where one-hop/Chess only require FIFO length of one. For Mediabench (University of California, 2020), in comparison to VPR, our mapping reduces the average FIFO length in 1.22x, 1.42x, and 1.58x for mesh, Chess, and one-hop topologies, respectively.

## 3.7 Conclusion and Future Work

In this work, we performed a design exploration of CGRAs using a SA algorithm and a novel Chess topology. Our results show that our algorithm is flexible and efficient to explore the solution design space for larger dataflows compared to those previously studied. The fully pipelined mapping has the challenge of the path delay mismatch. Therefore, even for small graphs, the problem is hard to solve. We can map dataflows with 20 to 70 nodes in larger CGRAs ($\geq 16$) without delay FIFOs due of the extra connections in the Chess and one-hop topologies. Previous works (Nowatzki et al., 2018) argues that it is hard to map graphs with FIFO of sizes 2, 3 or less and our results show that our SA approach performs it successfully. Finally, our mapping is straightforward to parallelize on a GPU , and it requires a few seconds to perform mappings for 200 nodes.

# Acknowledgment

This study was financed in part by the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior–Brasil (CAPES)– Finance Code 001, FAPEMIG, CNPq, and Intel.

# Capítulo 4

# Heterogeneous Reconfigurable Architectures for Machine Learning Dataflows

## 4.1 Introduction

In the machine learning era, applications for the most diverse purposes are being developed, for example, apps for image classification, speech recognition, and protein fold prediction (Krizhevsky et al., 2012; Jo et al., 2017; Thompson et al., 2021). These applications already require an enormous amount of computations and they are expected to demand more in the next years (Thompson et al., 2021). For example, the expected error for an ImageNet object recognizing deep-learning application should be reduced to 5% by 2025, however, it will demand enormous amounts of computing resources and energy, producing the same amount of carbon dioxide as New York City in one month (Thompson et al., 2021). For instance, the NASNet-A model reduces the error rate of AlexNet in half by using 1,000 times more computing resources (Thompson et al., 2021).

With that in mind, researchers are constantly trying to find alternatives to get around this problem. As stated in the 2018 Turing Lecture (Hennessy and Patterson, 2019), Domain-Specific Architectures (DSA's) are an alternative to solve this problem. We already have architectures such as Field Programmable Gate Arrays (FPGAs) and Coarse Grain Reconfigurable Arrays (CGRAs) that focus on accelerating applications. The current research effort is to enhance the efficiency of these technologies.

Recent work (Fang et al., 2019; Akbari et al., 2019; Liu et al., 2019b; Weng et al., 2019) shows that CGRAs accompanied with spatial dataflows are well suited to exploit parallelism and are tractable for the mapping of dataflows on the architecture. Furthermore, several researchers have shown that this technology is making further progress with efforts in related areas: DSL and compilers, stream input/output (Nowatzki et al., 2017) and architectures (Liu et al., 2019b), bringing up the relevance of research in the area. However, there is still a lack of CGRA tools and a few commercial devices (Liu et al., 2019b).

This paper focuses on the mapping of dataflow graphs (DFG's) on CGRA hetero-

geneous grids. By heterogeneous, we mean that different processing elements (PE) on the CGRA grid might have different properties than another PE on the same row or column. The idea behind these heterogeneous proposed architectures is to reduce the total cost of the architecture compared to the homogeneous one while maintaining the quality of the solutions. We present a simulated annealing-based approach that requires a few seconds to produce a valid solution compared to exact techniques that demand a few minutes (Nowatzki et al., 2018; Walker and Anderson, 2019).

The contributions of this work rely on the evaluation of three different kinds of heterogeneity:

1. **PE Functionality** - The processing element ALU can perform different types of operations. It is possible to reduce the total cost of the architecture by avoiding using the most expensive PEs functionalities.

2. **Buffers** - Another PE resource that has a significant impact on the final cost is the number of buffers that are present on each PE.

3. **Interconnections and Routing Resources** - Depending on how many routing resources and how many direct neighbors each PE has, the total cost can be significantly affected.

Furthermore, as a secondary result, our experiment derives optimal solutions for mapping some tree instances in mesh and mesh-plus architectures. We also demonstrated that it is impossible to map some simple dataflow patterns in mesh topologies without the aid of buffers. This paper is an extension of our conference paper (Oliveira et al., 2020a). In the previous version of this work, our main contribution was to explore homogeneous and heterogeneous CGRA for machine learning dataflows considering only the functionality heterogeneity case. We proposed three heterogeneous architectures that considered multipliers as the critical functionality resource. In this extended version, we have deepened our analysis by considering two other types of heterogeneity, buffers and interconnections. We have used the chess interconnection topology for evaluating interconnection heterogeneity (Oliveira et al., 2020b).

The main research question we aim to answer with this paper is whether a mapping algorithm and novel heterogeneous architectures can implement machine learning dataflows costing less and without losing solution quality. Furthermore, we would like to evaluate the scalability of the mapping for size dataflows between 30 to 200 nodes to overcome prior work limitations (Nowatzki et al., 2018; Walker and Anderson, 2019), where the largest graph size was 38 nodes.

This work is organized as follows. First, Section 4.2 presents dataflow graphs of machine learning applications. Section 4.3 discusses the main CGRA components. Section 4.4 introduces the cost metrics and the mapping approach. Sections 4.5 and 4.6

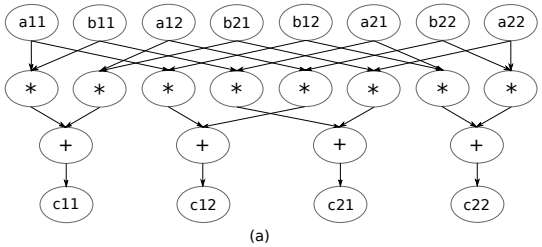
show the experimental results and related work. Finally, Section 4.7 draws the main conclusion and points out future work.

## 4.2 Machine Learning Kernels and Dataflows

This section provides the basic concepts of dataflow computing. First, Section 4.2.1 presents the matrix multiplication dataflow, which is the most common kernel to implement many ML algorithms. Section 4.2.2 shows the convolution, which is also a critical ML kernel. In Section 4.2.3 we briefly describe a $O(N)$ matrix multiplier and the trade-offs between matrix size, performance, and memory bandwidth. Section 4.2.4 draws some insights on embedding tree patterns in mesh architectures, along with a brief discussion of map-reduce patterns. Finally, in Section 4.2.5 we present the K-means study case.

### 4.2.1 Matrix Multiplier Kernel

The graphs we worked with, as mentioned before, come from the most computationally intensive parts of machine learning applications. Each node can be, for example, a multiplier, an adder, a multiplexer, among a few others. Figure 4.1 (a) shows an example of the dataflow graph for a 2x2 matrix multiplication $C = A \times B$. Every clock cycle, two new matrices $A$ and $B$ are set as the dataflow inputs, and a new result $C$ is produced. The latency depends on the multiplier and adder latencies. Figure 4.1 (b) shows a table that summarizes the number of I/Os, multipliers, adders, maximum fan-out for an $n \times n$ matrix, where $n$ ranges from 2 to 8. The number of inputs, outputs, multipliers, adders, and fan-out is $2n^2, n^2, n^3, (n^3 - n^2)$, and $n$, respectively. Therefore, the $4 \times 4$ NVidia tensor core has $64$ multipliers and $48$ adders, plus $16$ adders to perform $D = A \times B + C$.




| | I/Os | mults | adders | fan-out |
|---|---|---|---|---|
| 2x2 | 12 | 8 | 4 | 2 |
| 3x3 | 27 | 27 | 18 | 3 |
| 4x4 | 48 | 64 | 48 | 4 |
| 5x5 | 75 | 125 | 100 | 5 |
| 6x6 | 108 | 216 | 180 | 6 |
| 7x7 | 147 | 343 | 294 | 7 |
| 8x8 | 192 | 512 | 448 | 8 |

(b)

Figura 4.1: (a)Dataflow graph for a full 2x2 matrix multiplication (b)Data summary for matrix multiplications up to 8x8

The dataflow graphs that compose our set of benchmarks were selected among algorithm kernels that are often used in machine learning applications, such as matrix

multiplications and convolutions. Also, we selected binary trees since it is a typical reduction pattern in ML.

### 4.2.2 Convolutions

The convolution operation has been considered one of the essential operations in image processing systems and other applications that use convolutional neural networks (CNN) (Jo et al., 2017; Krizhevsky et al., 2012). However, the number of computations grows very fast according to the size of the convolution kernel. This has made researchers (Wei et al., 2017) strive to reduce the cost of these operations.

The main idea behind the convolution is shown in Figure 4.2(a). On the 3x3 convolution, a 3x3 filter is applied to the elements of the original matrix. The elements are multiplied element-wise and then summed up to return the result element present on the convoluted matrix.

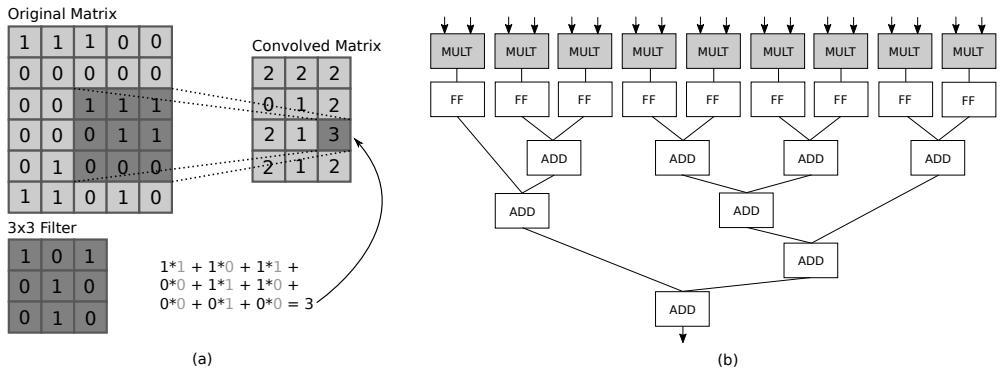



Figura 4.2: (a) Example of a 3x3 convolution (b) Convolution with K=3 dataflow graph

A convolution DFG is a simple reduction tree of multipliers and adders, as shown in Figure 4.2(b) for a $3 \times 3$ filter. The register layer on this example is present to balance the latency of the reduction tree and the multiplier layer, so the whole DFG returns 1 result after 2 clock cycles. Having these in mind, Section 4.2.4 focuses on how to embed trees in two-dimensional architectures. A reduction tree has small data reuse (factor 2) due to many inputs compared to the total number of operations. However, there are opportunities for data reuse as a more significant number of convolution filters over the same input data are presented in most CNN designs.

### 4.2.3 Linear Matrix Multiplier

The dataflow of Figure 4.1 (a) requires $O(n^3)$ multiplier/adder units and $O(n^2)$ I/O units to perform one $n \times n$ matrix multiplication per cycle. Figure 4.3 (a) exemplifies a different approach for a 3x3 systolic matrix multiplication with $O(n^2)$ multiplier/adder units and $O(n)$ I/O units, which requires $O(n)$ cycles to compute one matrix

multiplication. This approach reduces the I/O requirements. The nodes $mul_0, mul_3$, and $mul_6$ on the leftmost column receive the elements of each row of matrix A, which will be multiplied by matrix B, stored in each $mul$ node. After that, the result is added to the data that comes from the upper node and sent to the next one. The bottom nodes will at each clock cycle release an element of the result matrix C. An FPGA implementation for this design was presented in (Wei et al., 2017).

Figure 4.3 (b) shows the corresponding dataflow graph for the O(N) matrix multiplier. The multipliers on the left are colored in dark grey and represent the input nodes, while the adders of the bottom line are the output nodes. Our approach is more flexible, where both matrix multiplier designs (see Figure 4.1 and Figure 4.3) could be implemented. Finally, matrix multiply operations consume around 70% of the total runtime in ML implementations (Qin et al., 2020).

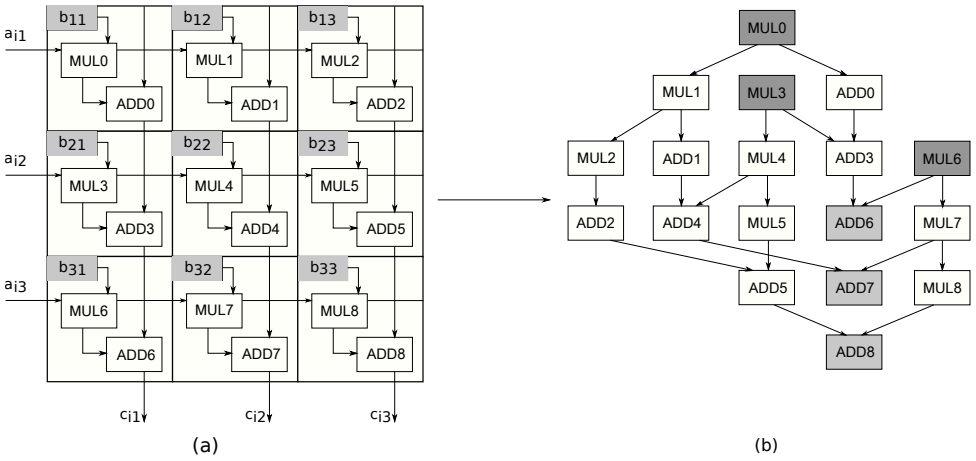



Figura 4.3: (a) $O(N)$ linear array for matrix multiplication (b) dataflow graph representation;

### 4.2.4 Trees and Map-Reduce Patterns

Dataflow graphs are not random graphs, and tree patterns are frequently found in parallel algorithms to implement the reduced steps of map-reduce algorithms. Therefore, an important issue is how to map a tree in fully pipelined mesh architectures efficiently. In Very Large Scale Integration (VLSI) circuit research, a well-known approach to balance delay times is the H-tree, which is a self-similar fractal (Browning, 1980), as shown in Figure 4.4(a). Although it has a symmetric layout, the occupancy rate is low, with many bypassing cells. For a tree with 16 leaves and 31 nodes in total, the total area is 49 cells, and there are 6 long wires ($cost = 1$ due to a bypassing cell) shown in bold in Figure 4.4(a). As the long wires or bypassing cells are equally distributed, and the tree is fully balanced, there is no delay mismatch, and therefore,

there is no need to include buffers to equalize the pipeline path lengths. Considering mesh interconnection, a single 15 and 31 node H-tree are not optimal since they require routing resources and mapped area is not minimal.

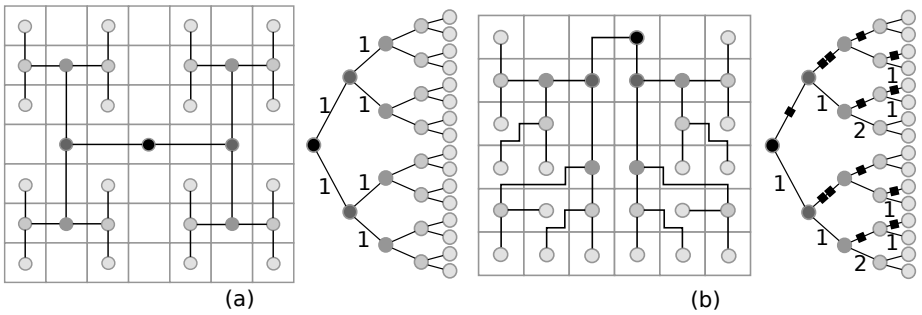



Figura 4.4: (a) 7x7 mesh and the H-tree configuration; (b) Lee/-Choi tree in a 6x6 mesh.

tableArea, occupancy, delay FIFO, and latency for H-Tree and Lee/Choi.

| | H-tree | | | |
|---|---|---|---|---|
| Size | Area | Occ | FIFO | Lat |
| 15 | 21 | 71% | - | 4 |
| 31 | 49 | 63% | - | 6 |
| 63 | 105 | 60% | - | 11 |
| 127 | 225 | 56% | - | 14 |
| | Lee/Choi | | | |
| 15 | 16 | 94% | 1 | 6 |
| 31 | 36 | 86% | 2 | 8 |
| 63 | 81 | 78% | 1 | 9 |
| 127 | 144 | 88% | 2 | 12 |

Lee and Choi (Lee and Choi, 1996) propose an algorithm to reduce the total area, as shown in Figure 4.4(b), where the same tree requires only 36 nodes in a $6 \times 6$ mesh. Nevertheless, the Lee/Choi tree has different path lengths from the leaves to the root, and it requires nine long routing wires, where two of them have $cost = 2$. When there is a long wire in one pipeline path that converges to one node, we should insert a buffer in the other path to equalize the delay. The black rectangular boxes at the rightmost tree in Figure 4.4(b) represent these delay buffers. The worst case is a buffer of size $2$, at the third level. A single 31 node Lee/Choi tree has the minimum area. However, it is not optimal since it requires routing and buffer resources.

Table 4.2.4 presents a mapping comparison of H-tree, and Lee/Choi approaches for full binary trees. Since it has a low occupancy rate, the H-Tree pattern needs a more extensive area to accommodate the structure. However, it doesn't require any buffers. On the other hand, the Lee/Choi approach reduces the area in cells once its occupancy rate is much higher but requires an architecture with variable length buffers to implement the pipeline correctly. In addition, such an architecture demands more area for memory and control. For instance, a FIFO buffer with $length = 2$ increases the area by 9%, using a 55nm technology library, as shown in (Nowatzki et al., 2018). Therefore, the area increase of 36% for a full 31-tree in the H-tree layout is mitigated to 23% compared to Lee/Choi trees due to the FIFO overhead area cost.

A map-reduce algorithm dataflow graph comprises two main phases: mapping and reduction. The reduction phase is, in most cases, a tree. Therefore, to compose our set of benchmarks, we chose binary trees followed by a tail of nodes after each leaf.

### 4.2.5 K-means

*K-means* is an unsupervised learning algorithm capable of partitioning a set of points in $k$ groups (or clusters). This algorithm is commonly used for clustering data points together in data-mining applications and data-analysis techniques (Choi and So, 2014; Abdelrahman, 2016; He et al., 2020). K-means is an example of a map-reduce algorithm, in which the classification phase performs one mapping followed by two reductions, as shown in Figure 4.5.

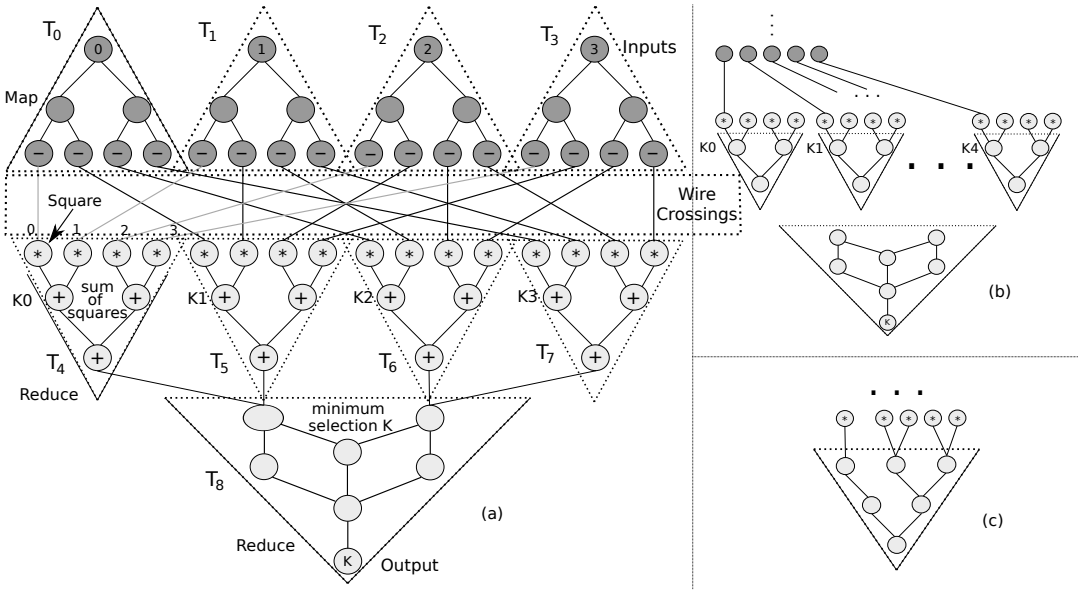



Figura 4.5: (a) K-means dataflow: One map followed by two reduces; (b) $k = 5, n = 4$; (c) $k = 4, n = 5$.

The mapping is responsible for the distance computation in each dimension for the current stream input $X$. For ease of explanation, assume a $K-means$ example $n = 4$ dimensions and $k = 4$ clusters, as shown in Figure 4.5. First, the input values should be distributed to all centroid distance units. The second step is a reduction to sum the distances ($K_i$) of all dimensions of a point $X$ to each cluster centroid $C_i$. In this step, $k$ is the number of clusters ($k = 4$ for our example) and $1 \leq i \leq k$. The third step is the reduction to find the closest centroid to $X$, i.e., the computation of $Min(K_1, \ldots, K_k)$. Hence, the algorithm performs $kn$ subtractions and multiplications during the mapping to compute the distances. The first reduction requires $k(n-1)$ additions to add the distances, and the second reduction requires $k-1$ comparisons to find the closest centroid to $X$. Each iteration computes $m[2kn + k(n-1) + k - 1]$ operations for $m$ points, and the arithmetic intensity is $\frac{m[2kn+k(n-1)+k-1]}{mn} = 3k - \frac{1}{n}$. Therefore, for larger values of $k$, the arithmetic intensity reaches a level that makes the problem suitable for our hardware implementation.

Embedding a *K-means* dataflow into a mesh is challenging. First, we should connect each input stream $x_i$ in Figure 4.5(a), to all $k$ map-reduce subgraphs. For instance, $x_0$ is close to the subgraph of $k_0$ in Figure 4.5(a). However, it should be close to all $k_j$ subgraphs. Furthermore, we should place $x_1, x_2$, and $x_3$ to guarantee that they are simultaneously close to all $k_j$ subgraphs. For this example $k = 4, n = 4$, there are 4 expanding trees ($T_0, \ldots, T_3$) followed by a full cross-edge pattern to connect to 4 reduction trees ($T_4, \ldots, T_7$), which should converge to a final reduction tree $T_8$. The expanding trees spread the input data, then each reduction tree ($T_4, \ldots, T_7$) perform the sum-of-squares for each centroid, i.e., the sum of squared Euclidean distances. And, finally, the tree $T_8$ selects the closest centroid with the minimum distance. Moreover, if we add a new cluster $k_4$, as shown in Figure 4.5(b), each input should be close to five subgraphs. Also, if we insert a new input dimension $x_4$, as shown in Figure 4.5(c), the first reduction tree is no longer a full binary tree, and we must insert registers to provide a correct balance. Finally, the second reduction is not a tree since there are reconvergent paths. Therefore, *K-means* is an interesting case study where there is: (a) map-reduce patterns; (b) different tree patterns (full and not full); (c) multiple fan-out inputs connected to disjoint subgraphs; (d) reduction with reconvergent paths.

## 4.3 Main CGRA Components

The CGRA is composed of an array of interconnected processing elements (PE's). Unlike FPGAs, which work at a bit level, CGRAs work at a coarse grain, meaning they work at the word level. This fact brings benefits such as reduced mapping for more complex architecture, but on the other hand, very few commercial architectures and design toolchains are availableLiu et al. (2019b). This Section presents three key points to exploit: operations, interconnections, and buffers.

### 4.3.1 Processing Elements

We can categorize CGRA architectures into homogeneous and heterogeneous. In this case, homogeneous means that every PE present on the grid is identical to one another. Thus, each of those PEs can perform the same kind of operations regardless of the grid. In contrast, a heterogeneous configuration presents two or more different types of PE's that can also perform various types of operations. It looks pretty straightforward that having a homogeneous configuration is the best way to go because it is less restricted, allowing better placements to be performed. However, this increases the entire architecture cost since every PE should be able to perform all operations. In this work, we propose a design exploration of heterogeneous CGRAs.

In our scenarios, the operation that causes a single PE to be more expensive is the multiplication (Luo and Lee, 2000). Therefore we divided our PE's into 2 groups, which can perform multiplication and those that can not.

In Figure 4.6(a), all the PE's are identical and can perform multiplications. These PE's are represented by grey-scale squares. Figure 4.6(b), however, shows multiplier PE's disposed as in a checkerboard, where nodes immediately adjacent to a special PE do not have a multiplier. We also have a column-wise configuration presented in Figure 4.6(c) and the configuration with multipliers on the borders of the grid Figure 4.6(d). At this point, it is worth noting that the IO nodes of the dataflow graph must be placed on the grid's borders for all of these cases.

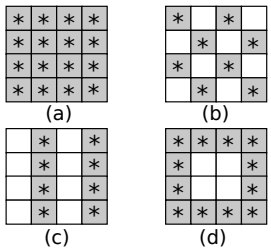



Figura 4.6: (a) Homogeneous grid; (b) checkerboard; (c) columns; (d) borders.

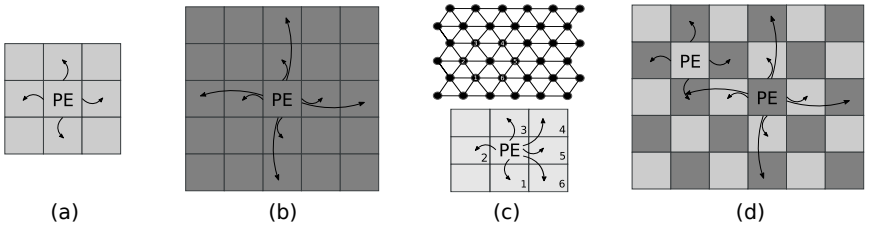



Figura 4.7: Homogeneous Interconnections: (a) Mesh; (b) Mesh-plus; (c) Hexagonal; (d) Heterogeneous Chess interconnection.

### 4.3.2 Interconnections

In addition to the multiplier heterogeneous setups proposed, we also perform a design exploration of three different interconnection architectures. Mesh, the simplest one, in which each PE can communicate with its 4 immediate neighbors without extra wires. Mesh-plus can jump one cell in each direction, in addition to the four connections present in mesh. Each PE can access 6 other PEs around it on the hexagonal configuration, resembling a honeycomb pattern. Figure 4.7 illustrates these architectures.

We also utilize a heterogeneous topology named Chess (Oliveira et al., 2020b), which consists of a grid with both mesh and mesh-plus cells arranged in a chessboard configuration. That means that throughout the grid, the cells are alternating between mesh and mesh-plus, as shown in Figure 4.7(d). The chess topology performs almost as well as the mesh-plus topology (Oliveira et al., 2020b). It has a very similar performance while maintaining a lower overall cost with an average of 6 instead of 8 local connections. Moreover, the chess topology covers a more extensive grid area in fewer steps than mesh and hexagonal, with performance comparable to mesh-plus.

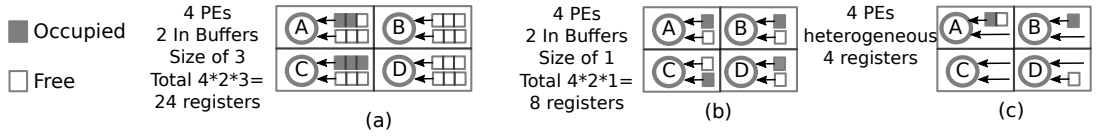



Figura 4.8: (a) Homogeneous Buffer size 3 PE; (b) Homogeneous Buffer size 1 PE; (c) Heterogeneous buffer size architecture.

### 4.3.3 Delay Mismatch Buffer

Buffers solve the delay mismatch problem by balancing the pipeline paths, and they are the most expensive resources in fully-pipelined CGRAs (Nowatzki et al., 2018). Furthermore, the buffer sizes are defined at design time considering the worst scenario. However, the occupation rates could be low. For instance, Figure 4.8(a) depicts an example of a mapped homogeneous architecture with 4 processing elements (PE). Each PE has two input buffers with the size of 3 registers. Therefore, the architecture requires $4 \times 2 \times 3 = 24$ registers, where 4 is the number of $PE$ with $2$ input buffers. Nevertheless, for this example, at run-time only 5 of 24 registers are used (grey boxes). Figure 4.8(b) shows another architecture where each PE has two input buffer size of one register, which reduces the overall cost by a factor of 3. In spite of that, it could be insufficient to map larger dataflows. We propose to evaluate heterogeneous buffer size architectures as shown in Figure 4.8(c), where the $PE_C$ has no buffers, $PE_B$ and $PE_D$ has only one input buffer of size 1, and $PE_A$ has one input buffer of size 2. It reduces the overall cost, and the mapping is still feasible. The challenge is to determine where the buffers should be placed at design time.

## 4.4 Mapping Algorithm and Metrics

This work brings forward a design exploration of CGRA architectures. As already mentioned, we propose a simulated annealing (SA) algorithm to map dataflows on the homogeneous and heterogeneous architectures. We divided the mapping problem into two phases, as described in the following. First, to find a suitable placement solution for each proposed dataflow graph, the SA approach aims to minimize the wire length used, similar to previous approaches (Mei et al., 2003a). This was done by checking, at each iteration of the algorithm, if the cost of swapping two nodes would increase or decrease the total length cost of the solution, always keeping track of the positions of the graph's multipliers and IOs.

After that, we proceed to the routing phase, which aims to validate the solution by identifying a viable routing that demands the least amount of delay buffers per PE since that is crucial to the total cost of the architecture, even more than the total wirelength (Nowatzki et al., 2018). For most dataflow edges, the routing is straightforward

as the nodes are placed in adjacent cells. Therefore, only a tiny subset of edges requires a routing strategy to implement the maze routing algorithm.

### 4.4.1 Cost Metrics and Optimal Mapping

The considered cost metric for the mapping algorithm is the total wirelength, despite it not being the most impactful factor on the overall cost of the architecture. If we are able to find a perfect solution with zero wirelength, it is guaranteed that the architecture will not need any buffers to implement it. So when we guide our algorithm to minimize the wirelength, we are also indirectly trying to minimize the needed number of buffers per processing element in the final architecture, while maintaining a simpler implementation. Furthermore, the wirelength is a local metric, that can be easy evaluated, while buffer allocation is a global metric that should consider all graph paths.

### 4.4.2 Non-optimal Pattern in Mesh Interconnections

It is straightforward to prove that it is not possible to map the dataflow pattern shown in Figure 4.9(a) optimally on a simple mesh CGRA. This is important because it shows that a mesh architecture can not implement some types of graphs, even very small ones, without the use of buffers. Therefore, heuristic mapping should be developed by keeping in mind non-optimal graph pattern. We demonstrated this by using a visual proof. Figure 4.9(b) shows the grid after the placement of node A on a random cell and the set of cells that can hold its neighbors without increasing the mapping's overall cost, namely $neighA = A_N, A_E, A_S, A_W$. We then proceed to the placement of node B, shown in one of these positions. Figure 4.9(c) shows the placement of B in position $A_E$, making $neighA = A_N, A_S, A_W$ and $neighB = B_N, B_E, A_S$. We now still have to place node C on the grid. However, there is no intersection between neighA and neighB, meaning $neighA \cap neighB = \emptyset$. The same would happen for whichever position we might have chosen for B because of the symmetry of the problem.

Note that it is possible to place the graph using PEs with size 1 buffers, however, the goal is to find a placement that doesn't need any buffers since they increase the overall cost. We could also perform the placement using an architecture with higher connectivity, such as chess or mesh-plus (Figure 4.9(d)), but once again they are more expensive.

### 4.4.3 NeoSA Algorithm

Figure 4.9(e) shows how NeoSA works. It is a simulated annealing based algorithm that sweeps the grid twice to select 2 cells, and try to perform a swap. If the cost is

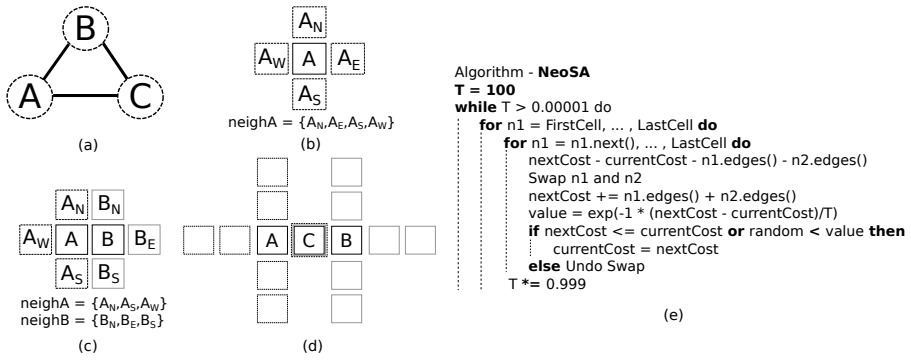



Figura 4.9: (a) Dataflow graph; (b) Visual proof for mesh; (c) Visual proof for distance 1; (d) Optimal mesh-plus solution; (e) NeoSA Pseudocode

lower after the swap, the algorithm continues. If the cost is worse, the swap can either be undone or kept according to an annealing probability. The actual code used for the experiments can be found at this GitHub page [1].

## 4.5 Experiments and Results

The experiments proposed in this work aimed to compare the performance of our algorithm (when executed with a homogeneous grid) to the state-of-the-art tool Versatile Place&Routing (VPR) 8.1Murray et al. (2020). After that, the second comparison was between both our homogeneous and heterogeneous approaches. It is important to highlight again that we worked with three different types of heterogeneity: the PE interconnection heterogeneity by considering the Chess architecture. The multiplier resource heterogeneity means that PEs may or may not have the capacity to perform multiplications since this is an expensive operation. And lastly, for the K-means benchmarks, we proposed an architecture that is heterogeneous regarding the presence or absence of buffers on the PEs.

Section 4.5.1 presents the evaluated dataflow benchmarks. In Section 4.5.2, we demonstrated that our SA approach can find optimal mappings for trees and map-reduce trees, considering homogeneous architectures. Section 4.5.3 evaluates the execution time of our SA approach in comparison the VPR tool (Murray et al., 2020). Section 4.5.4 brings the results for all three types of heterogeneity. In Section 4.5.4 we included the Chess interconnection topology in comparison to the other homogeneous ones. Section 4.5.4 brings an analysis between the results of the homogeneous case and the results of all three proposed multiplier heterogeneous configurations. Section 4.5.4 brings the same analysis but strictly for the K-means benchmark set. And lastly, Section 4.5.4 discusses the heterogeneity in terms of buffer resources.

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

### 4.5.1 Benchmarks

Table 4.1 summarizes the benchmarks we used in our experiments. A graph for a binary tree benchmark is denoted, for example, tree_n_X_t_X, where the number following n indicates the number of nodes on each connected component. The number following t is the number of connected components. So, following this notation, tree_n_31_t_3 is a graph with 3 binary trees of 31 nodes each. Benchmarks 1 through 11 represent these trees, followed by systolic matrix multiplications and a classic matrix multiplication like the one shown in Figure 4.1. In addition, we have convolutions ranging from $2 \times 2$ matrices to $5 \times 5$ matrices. The next subset of our benchmarks is the Map-reduce graphs, which are very similar to the trees, differing by a longer tail of nodes following the tree leaves, as shown in Section 4.2.4. And lastly, we have K-means dataflows. The first five have K=4, and N ranging from 4 to 8, and the remaining four have fixed N=4, and K ranging from 5 to 8, where $K$ is the number of clusters and $N$ is the number of data dimensions.

Tabela 4.1: List of benchmarks.

| ID | Benchmarks | Nodes | Mults | ID | Benchmarks | Nodes | Mults | ID | Benchmarks | Nodes | Mults |
|---|---|---|---|---|---|---|---|---|---|---|---|
| **1** | tree_n_15_t_1 | 15 | 8 | **14** | matmul44 | 32 | 16 | **27** | tree_n_15_t_3_r_3 | 117 | 24 |
| **2** | tree_n_31_t_1 | 31 | 16 | **15** | matmul55 | 50 | 25 | **28** | tree_n_15_t_4_r_3 | 156 | 32 |
| **3** | tree_n_63_t_1 | 63 | 32 | **16** | matmul66 | 72 | 36 | **29** | K4N4 | 63 | 16 |
| **4** | tree_n_127_t_1 | 127 | 64 | **17** | matmul22normal | 24 | 8 | **30** | K4N5 | 86 | 21 |
| **5** | tree_n_15_t_2 | 30 | 16 | **18** | convolution22 | 11 | 4 | **31** | K4N6 | 97 | 27 |
| **6** | tree_n_31_t_2 | 62 | 32 | **19** | convolution33 | 26 | 9 | **32** | K4N7 | 112 | 30 |
| **7** | tree_n_63_t_2 | 126 | 64 | **20** | convolution44 | 47 | 16 | **33** | K4N8 | 123 | 35 |
| **8** | tree_n_15_t_3 | 45 | 24 | **21** | convolution55 | 74 | 25 | **34** | K5N4 | 82 | 21 |
| **9** | tree_n_31_t_3 | 93 | 48 | **22** | tree_n_15_t_1_r_3 | 39 | 8 | **35** | K6N4 | 95 | 25 |
| **10** | tree_n_15_t_4 | 60 | 32 | **23** | tree_n_31_t_1_r_3 | 79 | 16 | **36** | K7N4 | 111 | 30 |
| **11** | tree_n_31_t_4 | 124 | 64 | **24** | tree_n_63_t_1_r_3 | 159 | 32 | **37** | K8N4 | 123 | 34 |
| **12** | matmul22 | 8 | 4 | **25** | tree_n_15_t_2_r_3 | 78 | 16 | | | | |
| **13** | matmul33 | 18 | 9 | **26** | tree_n_31_t_2_r_3 | 158 | 32 | | | | |

For all the benchmarks, the placement algorithm aimed to minimize the square area used. For instance, considering a homogeneous grid and a graph with 18 nodes, the smaller grid is a $5 \times 5$ architecture. The minimal size will be fixed by the minimum number of I/Os, multipliers, and general-purpose PEs for the heterogeneous ones.

Take for instance the heterogeneous borders model in Figure 4.6(d). If the side of the grid has size N, one could place at most 4*(N-1) multipliers on its borders. Thus, if a graph happened to have more multipliers than slots available, we had to nudge the grid's dimensions so it could fit. This happened, for example, on the bigger tree graphs where there are many leaves as inputs.

### 4.5.2 Trees and Map-Reduce

An interesting result of our mapping approach is the pattern that the graph tree_n_15_t_4 took after the placement on the homogeneous grid. The trees followed the H-tree pattern from the root and then took a form almost like a puzzle piece. These "puzzle pieces"then arranged themselves to fit tightly into the grid, using it with an occupation of 93.75% on an $8 \times 8$ grid against the occupation of 71.43% on a $7 \times 12$ grid for the H-tree configuration. Therefore, we reduced the area in 31.25% in comparison to the H-tree. For the same instance, the Lee/Choi (Lee and Choi, 1996) structure requires 12.5% more area with buffer size 2. Our approach does not require buffers in this case. An illustration of this solution is shown in Figura 4.10.

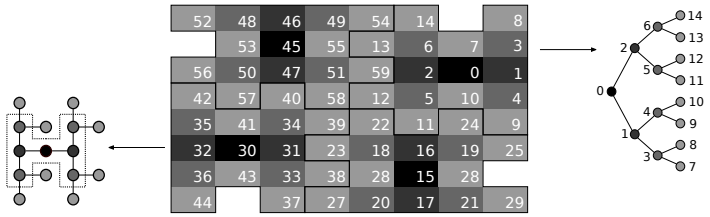



Figura 4.10: Puzzle-piece pattern for balanced trees on a homogeneous grid

Figure 4.11 (a) and (b) shows that there is an optimal mapping for the dataflows of binary trees of size 31 and 63 mapped on a mesh-plus interconnection topology where the 63-tree has a 98.4% area occupation rate. The root is in the darkest shade in both cases, and the nodes are filled from darker to lighter from the root to the leaves. We labeled the cell numbers following a breadth-first traversal (by level). Therefore, our mapping improves H-tree and Lee/Choi approaches by reaching optimal solutions for single 15-node tree in mesh interconnection, and for a single 31 and 63-node tree in mesh-plus interconnection with minimum area and neither routing nor buffer resources are required.

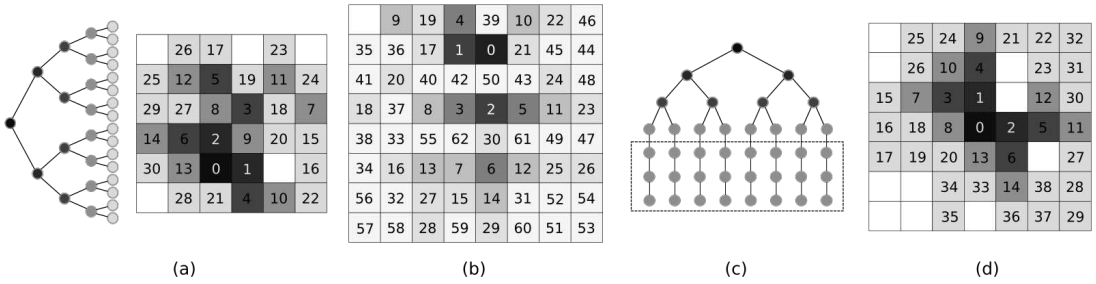



Figura 4.11: Optimal full trees in mesh-plus interconnection: (a) 31 nodes; (b) 63 nodes; (c) Map-reduce Tree; (d) Map-reduce mapped Tree.

Finally, Figure 4.11(c) shows a map-reduce example of the 15-node binary tree (reduction phase) and a map phase of 3 operations per leaf, alongside its perfect

solution on a mesh-plus grid, in Figure 4.11(d).

### 4.5.3 Mapping Execution Time

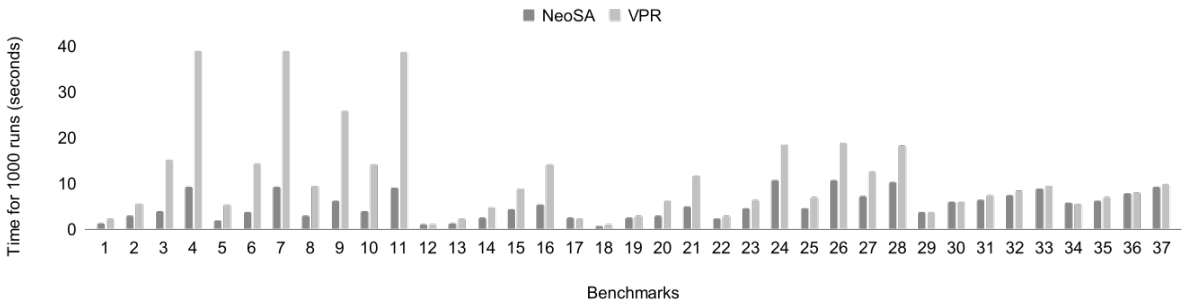



Figura 4.12: Mapping execution time comparison between VPR and our Simulated Annealing mapping (NeoSA).

To perform the placement, we used a simulated annealing-based algorithm that aimed to reduce the total wire length of the solution. To do so, the grid was scanned sequentially twice, a first time to select the reference node and a second time to select the destination node. Then, the cost of performing a swap between these two nodes is calculated. If the swap diminishes the overall wire length, it is executed. On the other hand, the swap increases the total length. It is only executed with an annealing probability that depends on the temperature. We started our algorithm with a temperature of 100 and decremented it to 99.9% of its value after each annealing step.

It is important to state that the total wire length cost is only a guideline to the placement. We used the number of buffers needed in each PE as the final metric since it is more appropriate in terms of total cost (Nowatzki et al., 2018). That happens because a homogeneous architecture in which all PE's have buffer resources will have many of those buffers inactive. So, to mitigate this waste of resources, the chosen solution will have the lowest amount of buffers per PE.

As a reference to evaluate our simulated annealing implementation, we will compare to the VPR tool (Murray et al., 2020) the state-of-the-art on SA for FPGA. Our target is a CGRA, and although VPR was developed for FPGA, it still out-performs recent CGRA mapping as presented in the CGRA-ME design exploration tool (Chin et al., 2018). VPR was executed 1000 times for each graph, each time using a different random seed. Other parameters were all set to the tool's standard. Among the 1000 results, the one with the lowest buffer size cost was chosen.

Figure 4.12 shows a comparison between the execution time (1000 instances) for our algorithm (named NeoSA for short) and the one performed on VPR.

It is notable that for all benchmarks, our algorithm outperforms VPR. Especially for the larger graphs, such as tree_n_31_t_4, VPR ends up being very slow, while

we can perform the placement four times faster. All the experiments were executed using the 16 cores, through openMP, of an Intel(R) Xeon(R) CPU E5-2630v3 2.4GHZ compiled with GCC 5.4.0 and the -O3 option. NeoSA is on average 52.8% faster than VPR considering the 37 evaluated dataflows. The NeoSA average execution time to map 1000 instances is of 5.3 seconds.

### 4.5.4 Heterogeneity

#### Interconnection

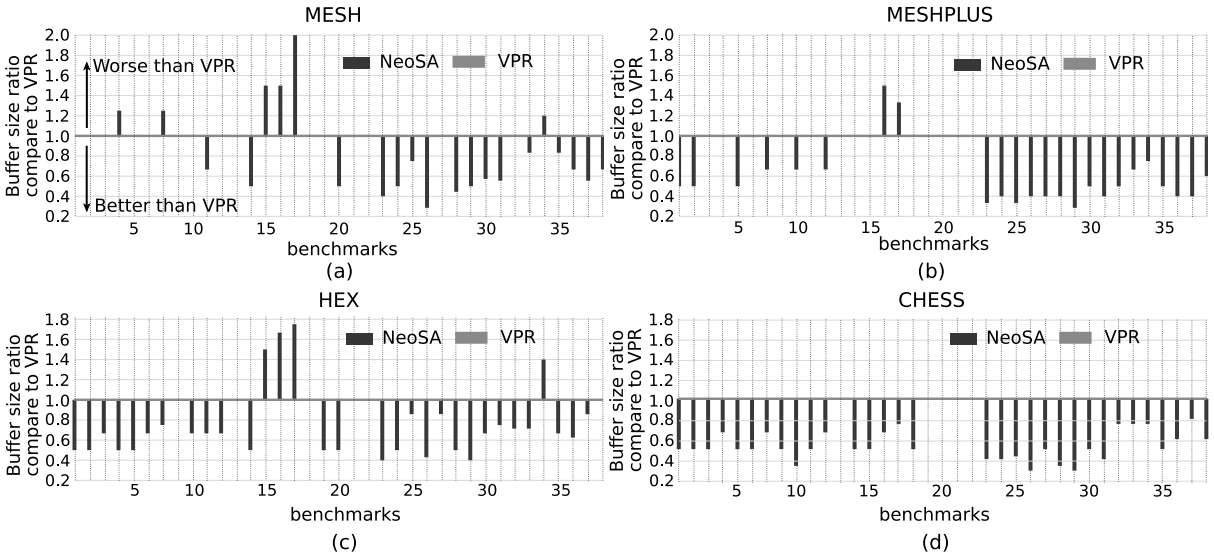



Figura 4.13: Comparison between our homogeneous Simulated Annealing (NeoSA) approach and VPR8 (Murray et al., 2020) considering all four interconnection topologies: (a) mesh, (b) mesh-plus, (c) hexagonal, (d) chess.

The main goal of this work was to perform a design exploration of heterogeneous CGRAs to mitigate the total number of buffer resources. Considering the minimal grid size, Figure 4.13(a-c) shows the ratio between the number of buffers needed for our algorithm and VPR 8.0, considering the three previously commented homogeneous interconnections: mesh, mesh-plus, and hexagonal. We considered our baseline to be the VPR 8.0 results, and it is represented in the chart as the black line on the horizontal axis. Our NeoSA outperforms VPR when the ratio is less than 1. Therefore, the bars that are below the black line in Figure 4.13 indicate that NeoSA outperforms VPR. For instance, NeoSA is better than VPR in 16, worst in 6 out of 37 cases, and it reaches the same results in 13 out of 37 cases for the Mesh interconnection. Moreover, for example, Benchmark 13 has a ratio of 0.5 for the Mesh topology as shown in Figure 4.13(a), which means that our algorithm found a solution that needs half the number of buffers needed by the VPR solution. Note that for most of the cases, our algorithm performed as good or better than VPR. We also evaluated the heterogeneous

Chess interconnection pattern. As already mentioned, Chess is a hybrid architecture created by alternating mesh and mesh-plus interconnection nodes, similar to a chessboard. Figure 4.13(d) shows the buffer ratios for Chess, where NeoSA outperforms VPR in 32 out of 37 cases. With the convolution dataflows (benchmarks 18-21) and the matrix multiplication $2 \times 2$, NeoSA performs as well as VPR. Therefore, NeoSA is well adapted to the heterogeneous chess interconnection pattern.

Tabela 4.2: Count of benchmarks for which NeoSA beat the VPR solution for each interconnection pattern.

| | Win Count | | | Average Buffer Size | |
|---|---|---|---|---|---|
| Pattern | NeoSA wins | VPR wins | Ties | NeoSA | VPR |
| Mesh | 16 | 6 | 15 | 2.76 | 3.73 |
| Mesh-plus | 22 | 2 | 13 | 0.92 | 2.16 |
| Hexagonal | 27 | 4 | 6 | 2.32 | 3.46 |
| Chess | 32 | 0 | 5 | 0.97 | 2.49 |
| Total | 97 | 12 | 39 | 1.74 | 2.95 |

In Table 4.2, we present a summary of how many times our NeoSA algorithm found a solution better than VPR, alongside the net count of those results. Notice that the cases where NeoSA and VPR found a solution with equal cost are not considered in this count. For instance, considering the mesh-plus interconnection, our approach is better in 6 out of 21 graphs, VPR is better in 2 cases, and we reach the same buffer size in 13 graphs. Table 4.2 also shows the average buffer size required. The heterogeneous Chess interconnection pattern has the same cost as Hexagonal, in which each cell has 6 local neighbors. However, Chess outperforms Hexagonal by reducing the buffer size by 58% on average. Although mesh-plus chess cells have some one-hop wires in comparison to hexagonal, it could be efficiently implement without degrading the clock cycle as shown in recent CGRA design (Gobieski et al., 2021). Furthermore, Chess reaches a result close to Mesh-plus, which has 33% more wires than Chess interconnection. Finally, these experiments show that NeoSA is competitive to VPR (Murray et al., 2020) for graph size up to 200 nodes, and it is also an open-source tool.

**Processing Elements/Multiplier Heterogeneity**

Figure 4.14 evaluates the buffer size needed on the processing element heterogeneous architectures for the four interconnection patterns. The baseline is the homogeneous solution, and Figure 4.14 depicts the ratio to this baseline for which we compared the three multiplier heterogeneous architectures introduced in Section 4.3.1: checkerBoard, cols, borders. The heterogeneous architectures performs equally or better than homogeneous in 58%, 63%, and 78% of the evaluated cases for cols, checkerboard, and

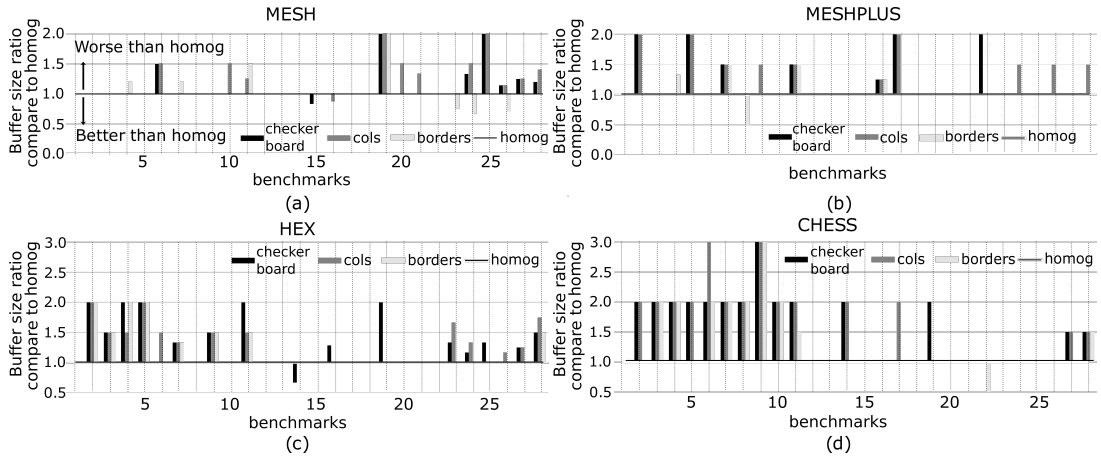



Figura 4.14: Ratio between the best result of each heterogeneous architecture and the homogeneous baseline

borders, respectively. Note that as in Figure 4.13, the vertical axis represents the ratio of the results compared to a baseline, which in this case is the homogeneous case. For ease of explanation, consider the Borders result for benchmark 13 in Figure 4.14(a) and notice that it is equal to 1, meaning that for that benchmark, our algorithm found a Borders solution that is as good as a homogeneous one. With that in mind, one can infer that there is a heterogeneous setup that performs as well as the homogeneous for most of the benchmarks. In some cases, again, it is harder to outperform the homogeneous case cost on the larger graphs, but we still got results that are reasonably close to it.

Therefore in Figure 4.14, when the bar is above the baseline means that this heterogeneous architecture performs worse than the homogeneous one, which is an expected result. Nevertheless, in most cases, the heterogeneous architecture reaches the same results as the homogeneous one, and no bar line is depicted. The pattern that yields the worse results is the one in which the multipliers are placed in columns through the grid, named *cols*, as we can see in in Figure 4.14. For instance, considering the mesh interconnection, *cols* architecture increases the buffer size in 11 out of 28 cases. However, it reaches the same results in 16 out of 28 cases. The *borders* architecture reaches the best results. It is worse than the homogeneous in 25 out of 112 cases, considering all evaluated interconnections. Moreover, the *borders* architecture is better than a homogeneous architecture in 5 cases. It is important to highlight the fact that the chess interconnection pattern is studied considering two axes of heterogeneity (interconnections and multipliers). In this case, to consider Multiplier Heterogeneity we first run a homogeneous chess instance and use it as a baseline to compare the heterogeneous cases. Therefore, in this case, we did not compare chess and hexagonal/mesh/mesh-plus. We compared chess heterogeneous to the chess baseline homogeneous. We could conclude that chess is the most sensible out of the

four interconnection patterns considering tree dataflows.

Tabela 4.3: Count of benchmarks for which NeoSA beat the VPR solution for each interconnection pattern.

| Pattern | CheckerBoard loses | Cols loses | Borders loses |
|---|---|---|---|
| Mesh | 7/28 | 11/28 | 4/28 |
| Mesh-plus | 7/28 | 10/28 | 4/28 |
| Hexagonal | 14/28 | 13/28 | 8/28 |
| Chess | 14/28 | 14/28 | 9/28 |
| Total | 42/112 | 47/112 | 25/112 |

A summary with the number of times that each heterogeneous configuration wasn't able to perform as good as the homogeneous is presented in Table 4.3, where 7/28 for the mesh interconnection and the heterogeneous architecture checkerboard means that the homogeneous one is only better than heterogeneous in 7 out of the 28 graphs.

The last line in Figure 4.14(d) summarizes these totals for each heterogeneous architecture. The border-based and the checkerboard are more suitable for our benchmark set. This is an interesting result since most FPGAs use column patterns as a heterogeneous design to place the memory and DSPs.

**Multiplier Heterogeneity in K-means Dataflows**

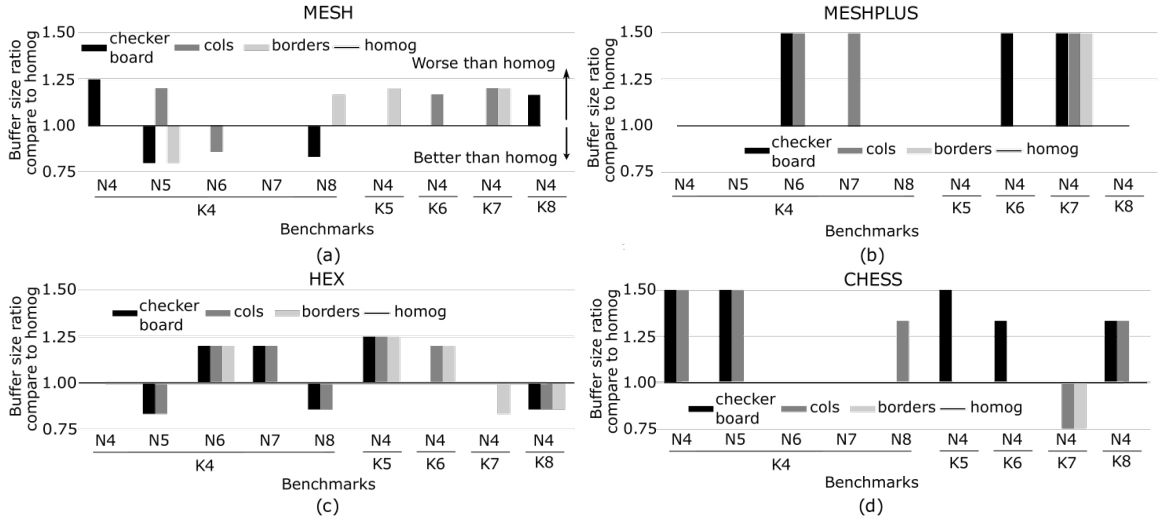



Figura 4.15: Ratio between the best result of each heterogeneous architecture and the homogeneous baseline for the K-means benchmarks

We separated the results for the K-means benchmarks from the others because they configure an interesting study case as mentioned before. They are composed of reduction trees, full of reconvergences, and are a real challenge for placement and routing algorithms. We could observe for the 108 evaluated cases shown in Figure 4.15, the

heterogeneous architecture is worse only in 34 cases, and it outperforms the homogeneous one in 14 cases. We can highlight the *borders* chess architecture, which reaches the same performance as the homogeneous one, and it even better in one case. Therefore, our NeoSA approach targeting more complex dataflow patterns performs well in a border chess CGRA.

**Buffer Heterogeneity**

The first experiment we did for K-means was using a homogeneous grid to check the placement's behavior, the quality of the solutions that could be achieved, and establish a starting point for further analysis. Table 4.4 shows the number of PEs that will have active buffers on average for the K-means benchmark set. The grid size represents the total number of PEs on the grid, and line *Active Buffer PEs* represents how many of those PEs will have active buffers. For instance, on average, for the K4N5 benchmark, 15 out of its 100 PEs will require their buffers to be used. Starting with these results, we then observed that the ratio of PEs that required the presence of buffers in their functional unit inputs was below 20% as inferred from Table 4.4. This encouraged searching for an architecture that could accommodate the placement without demanding all PEs to have buffers. This approach is different from what we did in the previous sections. We studied the grid's heterogeneity in terms of the presence or absence of multipliers in each processing element and interconnection degree.

Tabela 4.4: Number of active buffers for the K-means benchmarks.

| | $K = 4$ | | | | | $N = 4$ | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | N4 | N5 | N6 | N7 | N8 | K5 | K6 | K7 | K8 | Aver |
| Grid Size | 64 | 100 | 100 | 121 | 144 | 100 | 100 | 121 | 144 | 11 |
| Active Buffer PEs | 12 | 15 | 19 | 22 | 25 | 16 | 20 | 24 | 28 | 2( |
| Unused Buffer PEs | 52 | 85 | 81 | 99 | 119 | 84 | 80 | 97 | 116 | 9( |
| Percentage of Unused Buffers | 81% | 85% | 81% | 81% | 82% | 84 % | 80% | 80% | 80% | 81 |

From this point, the goal was to find a buffer size heterogeneous architecture pattern that could keep the quality of the solutions while lowering the overall cost of the structure by saving some buffer resources, which are very expensive (Nowatzki et al., 2018).

To search for such an architecture, we proposed a quantitative approach to define at design time where these buffers should be allocated. First, we analyzed our K-means dataflow set to find where the buffers are required and their size. It is important to highlight two points. First, the buffers are elastic, meaning a size 3 buffer has three registers, and it could implement a queue of size 3,2,1 or no queue whatsoever. Second, for the previous architecture, the buffer size is defined by the worst case in a

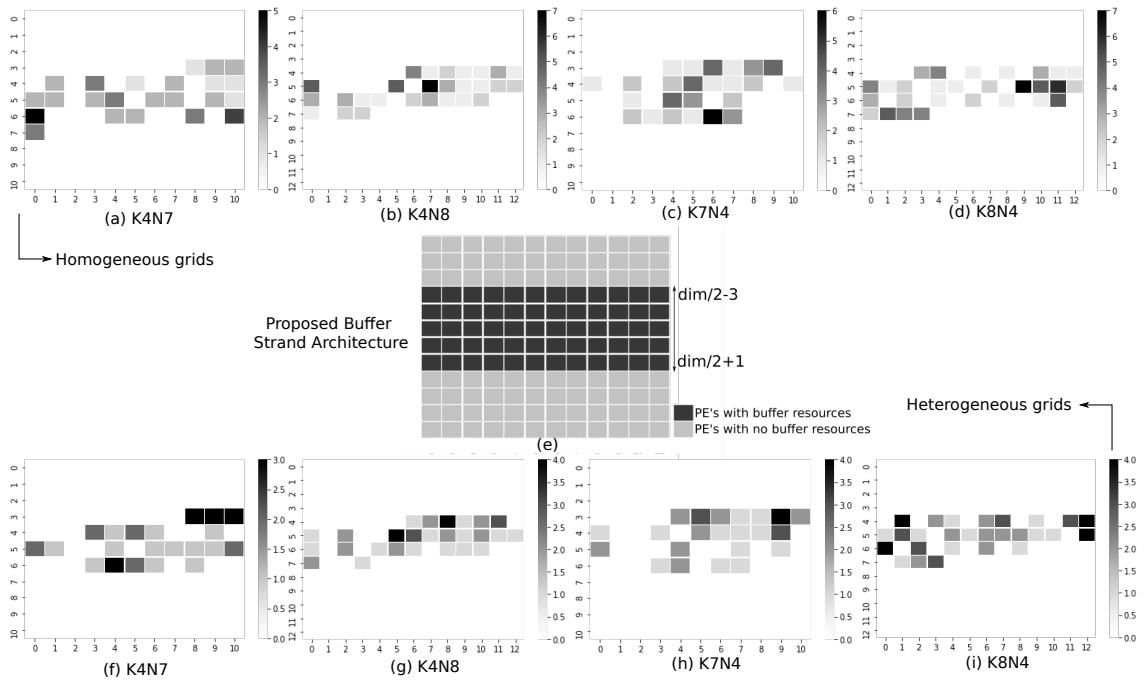



Figura 4.16: Heatmap of the active PEs buffer size: (a-d) Mapped results for homogeneous buffer architectures; (e) Novel proposed architecture; (f-i) Mapped results for the novel heterogeneous architecture.

specific PE, and it is replicated to all PEs. Therefore, at the register level, most buffers are totally unused or partially unused.

To find a good pattern, we first analyzed the results of mappings considering the homogeneous grid. For that, we plotted heatmaps of the grids coloring with darker colors the PEs that require longer buffers (more used registers) for that particular solution. Some of these heatmaps are shown in Figure 4.16(a-d).

We were able to identify that the active PEs tended to lay on a strand approximately in the middle of the grid. Thus, we proposed an architecture, shown in Figure 4.16(e), that consists of a strand of 5 rows in width that goes throughout the grid. We were able to find solutions for all the K-means instances we tested using this architecture. This novel architecture reduces the overall buffer requirements by 56.3% on average. And, these experiments demonstrate that the NeoSA algorithm is able to find valid mapping solutions in heterogeneous buffer architectures. Future work could empirically test whether architectures with different buffer sizes and patterns will map machine learning dataflows.

## 4.6 Related Work

Most CGRA mapping approaches focus on a processing element (PE) architecture shared by multiple instructions in a time-multiplexed fashion (Mei et al., 2003a; Park

et al., 2008; Hamzeh et al., 2012) for homogeneous architectures. In general, the architecture size is small, where the most usual size is a $4 \times 4$ grid. Therefore, the maximum number of parallel operations is $4x4 = 16$. Furthermore, the PE should have the capability to decode the instructions from a local instruction memory, which consumes area and power. On the other hand, fully pipelined architectures are configured once before the execution, and the PE will be dedicated to evaluating a single operation, saving area, and power. Moreover, the architecture sizes are more significant than 16 PEs, and the grid size bounds the maximum number of parallel operations. Therefore, this work focus on fully pipelined architectures (FPAs). Furthermore, our goal is to reduce the resources and energy costs without degrading the mapping and the CGRA performance. One way is to use heterogeneous CGRAs (Chin et al., 2018) or approximate computing CGRAs (Akbari et al., 2019). In the last two decades, several CGRA architectures have been proposed, they have not been detailed here, for more information refer to (Wijtvliet et al., 2016; Liu et al., 2019b).

The CGRA mapping is a NP-complete (Hamzeh et al., 2012), and considering FPAs, the difficulty to find a solution depends on the buffer size as shown in (Nowatzki et al., 2018). Therefore, with large buffer sizes (greater than 5) (Nowatzki et al., 2018), it becomes easier to find a valid mapping solution. However it increases significantly the CGRA area. For instance, a buffer size 7 increases the total silicon area of a CGRA in more than 25% (Nowatzki et al., 2018). A recent work (Nowatzki et al., 2018) shows that integer linear programming (ILP) (Chin et al., 2018; Walker and Anderson, 2019) and satisfiability modulo theory (SMT) (Donovick et al., 2019) approaches are costly in terms of computation time, which could take several minutes for small size graph (less than 20 nodes). These approaches solve the exact problem. Heuristics could be used. Even by using a hybrid approach (ILP plus heuristics) (Nowatzki et al., 2017, 2018; Weng et al., 2019, 2020) to solve the mapping in a reasonable time (a few seconds) for homogeneous FPAs, the FPA should have large buffers (more than 7 slots). Our approach shows that it is possible to save area by providing a design exploration of heterogeneous architecture and small buffers sizes (2 or 3).

A design exploration of homogeneous and heterogeneous architectures by using the CGRA-ME framework was presented in (Chin et al., 2018). However, the main drawback is the maximum dataflow graph size (less than 16 nodes), and the mapping execution time (minutes). Even by improving the two mapping strategies: the integer linerar programming (ILP) and the simulated annealing (SA) in a recent work (Walker and Anderson, 2019), the maximum graph size was 38 nodes. Furthermore, the CGRA-ME SA implementation is inefficient in terms of execution time (minutes to hours). VPR (Murray et al., 2020) is state-of-the-art for simulated annealing for FPGA. Prior work for CGRA mapping (Huang et al., 2013; Vasilyev et al., 2016) use VPR on homogeneous architectures. Our approach is also based on SA, and our

results are competitive to VPR in terms of quality and execution time. Finally, recent work (Zhang et al., 2021; Gobieski et al., 2021) propose two CGRAs. One work (Gobieski et al., 2021) focuses on ultra low-power and energy efficient CGRA generator, and evaluates a 6x6 CGRA. The second work (Zhang et al., 2021) focuses on compiler tools and uses graph partition to reduce the mapping complexity, and then, apply ILP approaches (Nowatzki et al., 2018) in subgraphs with less than 30 nodes.

## 4.7 Conclusion

We studied three types of heterogeneity in this work. Firstly we demonstrated that our algorithm is well adapted to find solutions considering three homogeneous architectures and the heterogeneous Chess interconnection topology. Next, the multiplier heterogeneous setups results showed us that it is possible to find good enough placements for the graphs tested in this work that reduce the number of multiplier resources on the CGRA grid, therefore reducing its overall cost. Lastly, we proposed an architecture that does not require buffers in its entirety aimed at the K-means benchmark set. This architecture was able to map solutions for all of the dataflows in the set, reducing the overall cost considerably compared to the homogeneous case. A few other experiments that could be done and explored in future work consist of fixing some of the multipliers and I/Os in a good position, found on a pre-processing stage. This could be done on graphs with a regular and repetitive structure, such as the systolic array. This approach could guide the simulated annealing to a better or even an optimal solution. Future work could also include dataflow merging, which identifies graph similarities to generate a dedicated and straightforward reconfigurable architecture by minimizing the routing resources. We also plan to compare SA-based placement to genetic (Silva et al., 2006) and divide-and-conquer search approaches.

## Acknowledgments

This work was carried out with the support of the Coordernação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) - Financing Code 001. Financial support from FAPEMIG, CNPq, and UFV.

# Capítulo 5

# Conclusion

In conclusion, this master's thesis has contributed to the field of reconfigurable architectures, specifically Coarse-Grained Reconfigurable Architectures (CGRAs), by introducing a flexible mapping approach for dataflow graphs. Furthermore, an extensive analysis has been conducted to investigate the potential benefits of using CGRAs for Machine Learning applications.

The proposed mapping approach is based on simulated annealing, which offers a powerful optimization technique for effectively mapping dataflow graphs onto CGRAs. This approach ensures efficient utilization of resources while considering performance constraints. Additionally, a novel heterogeneous buffer architecture has been introduced, which effectively reduces costs without compromising overall performance for machine learning dataflow patterns. This buffer architecture optimally handles the varying dataflow characteristics of machine learning algorithms, further enhancing the efficiency of the system.

Moreover, the thesis introduces a novel interconnection topology known as *Chess* between mesh and one-hop topologies, which provides an alternative and efficient method for connecting components in a reconfigurable architecture in comparison to hexagonal topology which has the same node degree. The Chess topology offers advantages such as reduced wire length and improved overall system performance. Additionally, a parameterized hardware generator has been proposed, enabling rapid evaluation of different architectures by considering wire length and buffer costs.

Our research provides a framework for exploring mesh architectures and more complex interconnection topologies to efficiently map dataflow graphs of up to seventy operators without FIFOs. We demonstrated that the heterogeneous border chess architecture is more efficient than a homogeneous one, and our mapping algorithm achieved better results than VPR 8.1 (Luu et al., 2011). Additionally, our algorithm reached optimal results for single tree dataflows compared to classical Lee/Choi and H-Trees. Altogether, our findings demonstrate the potential of leveraging heterogeneous architectures for the efficient design of Machine Learning applications.

Future research concerning the mapping of dataflow graphs onto CGRA architectures could focus on improving the efficiency of the mapping algorithm. This could

include exploring new search strategies and optimization techniques that better exploit the potential of the CGRA's architecture and reduce the number of cycles needed to map a graph onto the CGRA. Additionally, research could be conducted to investigate how to better utilize the CGRA's reconfigurability in order to reduce the number of operations needed to complete a task. A further research direction could be to explore how to better integrate compilers, schedulers, and other application development tools with the CGRA's hardware (Silva et al., 2020) to simplify the process of designing and implementing dataflow graphs on the CGRA.

# Referências Bibliográficas

Abdelrahman, T. S. (2016). Accelerating K-Means Clustering on a Tightly-Coupled Processor-FPGA Heterogeneous System. In *IEEE International Conference ASAP*.

Akbari, O., Kamal, M., Afzali-Kusha, A., Pedram, M., and Shafique, M. (2019). X-cgra: An energy-efficient approximate coarse-grained reconfigurable architecture. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*.

Bansal, N., Gupta, S., Dutt, N., Nicolau, A., and Gupta, R. (2004). Network topology exploration of mesh-based coarse-grain reconfigurable architectures. In *DATE*, volume 1. IEEE.

Bertsimas, D. and Tsitsiklis, J. (1993). Simulated annealing. *Statistical science*, 8(1):10–15.

Betz, V. and Rose, J. (1997). Vpr: A new packing, placement and routing tool for fpga research. In *International Workshop on Field Programmable Logic and Applications*, pages 213–222. Springer.

Browning, S. A. (1980). The tree machine: A highly concurrent computing environment.

Canesche, M., Carvalho, W., Reis, L., Oliveira, M., Magalhaes, S., Jamieson, P., Nacif, J. M., and Ferreira, R. (2021). You only traverse twice: A yott placement, routing, and timing approach for cgras. *ACM Transactions on Embedded Computing Systems (TECS)*, 20(5s):1–25.

Canesche, M., Menezes, M., Carvalho, W., Torres, F., Jamieson, P., Nacif, J. A., and Ferreira, R. (2020). Traversal: A fast and adaptive graph-based placement and routing for cgras. *IEEE Transactions on CAD of Integrated Circuits and Systems*.

Chin, S. A., Niu, K. P., Walker, M., Yin, S., Mertens, A., Lee, J., and Anderson, J. H. (2018). Architecture exploration of standard-cell and fpga-overlay cgras using the open-source cgra-me framework. In *International Symposium on Physical Design*.

Choi, Y.-M. and So, H. K.-H. (2014). Map-Reduce Processing of K-Means Algorithm with FPGA-Accelerated Computer Cluster. In *IEEE International Conference ASAP*.

Dadu, V., Weng, J., Liu, S., and Nowatzki, T. (2019). Towards general purpose acceleration by exploiting common data-dependence forms. In *International Symposium on Microarchitecture*.

Donovick, C., Mann, M., Barrett, C., and Hanrahan, P. (2019). Agile smt-based mapping for cgras with restricted routing networks. In *Int. Conf. on ReConFigurable Computing and FPGAs (ReConFig)*. IEEE.

Fang, Z., Javadi, F., Cong, J., and Reinman, G. (2019). Understanding performance gains of accelerator-rich architectures. In *2019 IEEE 30th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, volume 2160, pages 239–246. IEEE.

Ferreira, R., Cardoso, J. M., Toledo, A., and Neto, H. C. (2005). Data-driven regular reconfigurable arrays: design space exploration and mapping. In *Embedded Computer Systems: Architectures, Modeling, and Simulation: 5th International Workshop, SAMOS 2005, Samos, Greece, July 18-20, 2005. Proceedings 5*, pages 41–50. Springer.

Ferreira, R., Denver, W., Pereira, M., Quadros, J., Carro, L., and Wong, S. (2014). A run-time modulo scheduling by using a binary translation mechanism. In *2014 International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS XIV)*, pages 75–82. IEEE.

Ferreira, R., Denver, W., Pereira, M., Wong, S., Lisba, C. A., and Carro, L. (2016). A dynamic modulo scheduling with binary translation: Loop optimization with software compatibility. *Journal of Signal Processing Systems*, 85:45–66.

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

Ferreira, R., Rocha, L., Santos, A., Nacif, J., Wong, S., and Carro, L. (2013b). A run-time graph-based polynomial placement and routing algorithm for virtual fpgas. In *IEEE International Conference on Field programmable Logic and Applications (FPL)*.

Ferreira, R., Vendramini, J. G., Mucida, L., Pereira, M. M., and Carro, L. (2011a). 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.

Ferreira, R. S., Cardoso, J. M., Damiany, A., Vendramini, J., and Teixeira, T. (2011b). Fast placement and routing by extending coarse-grained reconfigurable arrays with omega networks. *Journal of Systems Architecture*, 57(8):761–777.

Fontes, G., Silva, P. A. R., Nacif, J. A. M., Neto, O. P. V., and Ferreira, R. (2018). Placement and routing by overlapping and merging qca gates. In *2018 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 1–5. IEEE.

FreePDK45 (2019). Freepdk45tm 45nm process design kit. `https://www.eda.ncsu.edu/wiki/FreePDK45:Contents`.

Friedman, S., Carroll, A., Van Essen, B., Ylvisaker, B., Ebeling, C., and Hauck, S. (2009). Spr: an architecture-adaptive cgra mapping tool. In *Proceedings of the ACM/SIGDA international symposium on Field programmable gate arrays*, pages 191–200. ACM.

Gobieski, G., Mai, K., Lucia, B., and Beckmann, N. (2021). Snafu: An ultra-low-power, energy-minimal cgra-generation framework and architecture. In *International Symposium on Computer Architecture (ISCA)*.

Guo, L., Maidee, P., Zhou, Y., Lavin, C., Hung, E., Li, W., Lau, J., Qiao, W., Chi, Y., Song, L., et al. (2023). Rapidstream 2.0: Automated parallel implementation of latency insensitive fpga designs through partial reconfiguration. *ACM Transactions on Reconfigurable Technology and Systems*.

Hamzeh, M., Shrivastava, A., and Vrudhula, S. (2012). Epimap: Using epimorphism to map applications on cgras. In *Proceedings of the 49th Annual Design Automation Conference*, pages 1284–1291.

He, Z., Wang, Z., and Alonso, G. (2020). Bis-km: Enabling any-precision k-means on fpgas. In *The 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 233–243.

Hennessy, J. L. and Patterson, D. A. (2019). A new golden age for computer architecture. *Communications of the ACM*, 62(2):48–60.

Huang, Y., Ienne, P., Temam, O., Chen, Y., and Wu, C. (2013). Elastic cgras. In *Proceedings of the ACM/SIGDA international symposium on Field programmable gate arrays*, pages 171–180. ACM.

Jo, J., Cha, S., Rho, D., and Park, I.-C. (2017). Dsip: A scalable inference accelerator for convolutional neural networks. *IEEE Journal of Solid-State Circuits*, 53(2):605–618.

Kirkpatrick, S., Gelatt Jr, C. D., and Vecchi, M. P. (1983). Optimization by simulated annealing. *science*, 220(4598):671–680.

Krizhevsky, A., Sutskever, I., and Hinton, G. E. (2012). Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*.

Lee, S.-K. and Choi, H.-A. (1996). Embedding of complete binary trees into meshes with row-column routing. *IEEE Trans on Parallel and Distributed Systems*, 7(5).

Li, S. and Ebeling, C. (2004). Quickroute: a fast routing algorithm for pipelined architectures. In *Proceedings. 2004 IEEE International Conference on Field-Programmable Technology (IEEE Cat. No. 04EX921)*, pages 73–80. IEEE.

Li, Z., Wijerathne, D., and Mitra, T. (2022). Coarse grained reconfigurable array cgra. *Book Chapter in Springer Handbook of Computer Architecture*.

Liu, D., Yin, S., Luo, G., Shang, J., Liu, L., Wei, S., Feng, Y., and Zhou, S. (2018). Data-flow graph mapping optimization for cgra with deep reinforcement learning. *IEEE Trans on Computer-Aided Design of Integrated Circuits and Systems*, 38(12).

Liu, D., Yin, S., Luo, G., Shang, J., Liu, L., Wei, S., Feng, Y., and Zhou, S. (2019a). Data-flow graph mapping optimization for cgra with deep reinforcement learning. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*.

Liu, L., Zhu, J., Li, Z., Lu, Y., Deng, Y., Han, J., Yin, S., and Wei, S. (2019b). A survey of coarse-grained reconfigurable architecture and design: Taxonomy, challenges, and applications. *ACM Computing Surveys (CSUR)*, 52(6):1–39.

Liu, Z.-G., Whatmough, P. N., and Mattina, M. (2020). Systolic tensor array: An efficient structured-sparse gemm accelerator for mobile cnn inference. *IEEE Computer Architecture Letters*, 19(1):34–37.

Luo, Z. and Lee, R. B. (2000). Cost-effective multiplication with enhanced adders for multimedia applications. In *Int Symp on Circuits and Systems (ISCAS)*. IEEE.

Luu, J., Kuon, I., Jamieson, P., Campbell, T., Ye, A., Fang, W. M., Kent, K., and Rose, J. (2011). Vpr 5.0: Fpga cad and architecture exploration tools with single-driver routing, heterogeneity and process scaling. *ACM TRETS*, 4(4):32.

McMurchie, L. and Ebeling, C. (1995). Pathfinder: A negotiation-based performance-driven router for fpgas. In *Proceedings of the 1995 ACM third international symposium on Field-programmable gate arrays*, pages 111–117.

Mei, B., Vernalde, S., Verkest, D., De Man, H., and Lauwereins, R. (2003a). Adres: An architecture with tightly coupled vliw processor and coarse-grained reconfigurable matrix. In *International Conference on Field Programmable Logic and Applications*.

Mei, B., Vernalde, S., Verkest, D., De Man, H., and Lauwereins, R. (2003b). Exploiting loop-level parallelism on coarse-grained reconfigurable architectures using modulo scheduling. *IEE Proceedings-Computers and Digital Techniques*, 150(5):255–261.

Murray, K. E., Petelin, O., Zhong, S., Wang, J. M., ElDafrawy, M., Legault, J.-P., Sha, E., Graham, A. G., Wu, J., Walker, M. J. P., Zeng, H., Patros, P., Luu, J., Kent, K. B., and Betz, V. (2020). Vtr 8: High performance cad and customizable fpga architecture modelling. *ACM Trans. Reconfigurable Technol. Syst.*

Nowatzki, T., Ardalani, N., Sankaralingam, K., and Weng, J. (2018). Hybrid optimization/heuristic instruction scheduling for programmable accelerator codesign. In *Int Conf on Parallel Architectures and Compilation Techniques (PACT)*.

Nowatzki, T., Gangadhar, V., Ardalani, N., and Sankaralingam, K. (2017). Stream-dataflow acceleration. In *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*, pages 416–429. IEEE.

of Toronto, U. (2019). Cgra - modelling and exploration. `http://cgra-me.ece.utoronto.ca/`.

Oliveira, W., Canesche, M., Reis, L., Nacif, J., and Ferreira, R. (2020a). Design exploration of machine learning data-flows onto heterogeneous reconfigurable hardware. In *Simp. em Sistemas Computacionais de Alto Desempenho, WSCAD*.

Oliveira, W., Canesche, M., Reis, L., Torres, F., Silva, L., Jamieson, P., Nacif, J., and Ferreira, R. (2020b). A design exploration of scalable mesh-based fully pipelined accelerators. In *2020 International Conference on Field-Programmable Technology (ICFPT)*, pages 233–236. IEEE.

Park, H., Fan, K., Mahlke, S. A., Oh, T., Kim, H., and Kim, H.-s. (2008). Edge-centric modulo scheduling for coarse-grained reconfigurable architectures. In *Int Conf on Parallel architectures and compilation techniques (PACT)*.

Pasandi, G. and Pedram, M. (2019). A dynamic programming-based, path balancing technology mapping algorithm targeting area minimization. In *2019 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 1–8. IEEE.

Podobas, A., Sano, K., and Matsuoka, S. (2020). A survey on coarse-grained reconfigurable architectures from a performance perspective. *IEEE Access*, 8:146719–146743.

Qin, E., Samajdar, A., Kwon, H., Nadella, V., Srinivasan, S., Das, D., Kaul, B., and Krishna, T. (2020). Sigma: A sparse and irregular gemm accelerator with flexible interconnects for dnn training. In *Int Symp on High Performance Computer Architecture (HPCA)*.

Rau, B. R. (1994). Iterative modulo scheduling: An algorithm for software pipelining loops. In *Proceedings of the 27th annual international symposium on Microarchitecture*, pages 63–74.

Silva, L., Canesche, M., Ferreira, R., and Nacif, J. A. (2020). 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.

Silva, L. B. D., Ferreira, R., Canesche, M., Menezes, M. M., Vieira, M. D., Penha, J., Jamieson, P., and Nacif, J. A. M. (2019). Ready: A fine-grained multithreading overlay framework for modern cpu-fpga dataflow applications. *ACM Transactions on Embedded Computing Systems (TECS)*, 18(5s):1–20.

Silva, M., Ferreira, R., Garcia, A., and Cardoso, J. (2006). Mesh mapping exploration for coarse-grained reconfigurable array architectures. In *Int Conf on Reconfigurable Computing and FPGA's (ReConFig)*.

Thompson, N. C., Greenewald, K., Lee, K., and Manso, G. F. (2021). Deep learning's diminishing returns: The cost of improvement is becoming unsustainable. *IEEE Spectrum*, 58(10):50–55.

Torres, F. S., Silva, P. A., Fontes, G., Nacif, J. A., Ferreira, R. S., Neto, O. P. V., Chaves, J., and Drechsler, R. (2018). Exploration of the synchronization constraint in quantum-dot cellular automata. In *2018 21st Euromicro Conference on Digital System Design (DSD)*, pages 642–648. IEEE.

University of California, S. B. (2020). Express benchmarks. `https://web.ece.ucsb.edu/EXPRESS/benchmark/`.

Vasilyev, A., Bhagdikar, N., Pedram, A., Kvatinsky, S., and Horowitz, M. (2016). Evaluating programmable architectures for imaging and vision applications. In *International Symposium on Microarchitecture (MICRO)*.

Vieira, M., Canesche, M., Bragança, L., Campos, J., Silva, M., Ferreira, R., and Nacif, J. A. (2021). Reshape: A run-time dataflow hardware-based mapping for cgra overlays. In *2021 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 1–5. IEEE.

Walker, M. J. and Anderson, J. H. (2019). Generic connectivity-based cgra mapping via integer linear programming. In *Symposium on Field-Programmable Custom Computing Machines (FCCM)*.

Walter, M., Wille, R., Große, D., Torres, F. S., and Drechsler, R. (2018). An exact method for design exploration of quantum-dot cellular automata. In *2018 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 503–508. IEEE.

Wei, X., Yu, C. H., Zhang, P., Chen, Y., Wang, Y., Hu, H., Liang, Y., and Cong, J. (2017). Automated systolic array architecture synthesis for high throughput cnn inference on fpgas. In *Design Automation Conference (DAC)*.

Weng, J., Liu, S., Dadu, V., and Nowatzki, T. (2019). Daegen: A modular compiler for exploring decoupled spatial accelerators. *IEEE Computer Architecture Letters*.

Weng, J., Liu, S., Dadu, V., Wang, Z., Shah, P., and Nowatzki, T. (2020). Dsagen: Synthesizing programmable spatial accelerators. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pages 268–281. IEEE.

Wijtvliet, M., Waeijen, L., and Corporaal, H. (2016). Coarse grained reconfigurable architectures in the past 25 years: Overview and classification. In *Int Conf on Embedded Computer Systems: Architectures, Modeling and Simulation (SAMOS)*.

Zhang, Y., Zhang, N., Zhao, T., Vilim, M., Shahbaz, M., and Olukotun, K. (2021). Sara: Scaling a reconfigurable dataflow accelerator. In *International Symposium on Computer Architecture (ISCA)*.