KU LEUVEN

FACULTY OF
ENGINEERING SCIENCE

SNAX-CGRA: System-level Optimization of a CGRA Processor for Efficient AI Acceleration

Claudio Clemente

Thesis submitted for the degree of
Master of Science in
Electrical Engineering, option
Electronics and Chip Design

Supervisor:
Prof. dr. ir. Marian Verhelst

Assessor:
Prof. dr. ir. Matthew B. Blaschko

Assistant-supervisors:
Dr. Guilherme Pereira Paim
Dr. Jun Yin

Academic year 2023 – 2024

Preface

I extend my heartfelt appreciation to all who've been by my side on this extraordinary voyage. Firstly, my gratitude goes to Prof. Marian Verhelst for entrusting me with this thesis. I also extend my thanks to Prof. Matthew B. Blaschko for their evaluation.

A special acknowledgment goes to my daily mentors, Dr. Guilherme Pereira Paim and Dr. Jun Yin. Their unwavering support and guidance were indispensable, enabling me to complete my thesis within a challenging four-month span. Their mentorship not only broadened my knowledge but also refined my skills significantly.

Claudio Clemente

Contents

Abstract

The exponentially growth in the artificial intelligence domain has led to a demand of specific hardware able to deliver edge-cutting performance. Moreover, development of specialized hardware leads to improvements on a wider spectrum than performances alone. In the research field, models can be considered failures or successes based on their ability to effectively integrate theoretical concepts with hardware implementation [19]. This integration is crucial because it determines whether theoretical advancements can be practically applied in real-world systems. When we talk about bonding theory and hardware, we're referring to the intricate process where abstract, often highly complex theoretical models are translated into practical applications that can be executed on physical devices. To successfully bond of theory and hardware we must ensure that the theoretical models are not only mathematically sound and scientifically robust but also feasible for implementation within the constraints of current hardware technologies. It requires a deep understanding of both the theoretical underpinnings and the practical limitations and capabilities of the hardware. This means that theoretical models must be adaptable enough to be transformed into software algorithms that can efficiently run on hardware platforms, whether these are general-purpose processors, specialized chips, or integrated systems. In essence, the success of this bonding process is a testament to the validity and utility of the theoretical models. If researchers can demonstrate that their theoretical work can be realized in physical form, functioning as intended under real-world conditions, it showcases the practical relevance and potential impact of their research. Conversely, if the models fail to be effectively integrated with hardware, it suggests that there may be gaps or flaws in the theoretical framework, or that current technology is not yet capable of supporting such advanced algorithms. Thus, the measure of a research model's success is intrinsically linked to its ability to bridge the gap between theory and practice, making the abstract concrete and the conceptual operational.
Hardware performances and flexibility are key parameters when considering how the AI world works. To achieve optimal performances for a specific task execution, it is important to choose carefully the AI model that performs better for that specific task. If the hardware is more powerful, the AI model can performs better. Moreover, there exist different types of models, categorized by the operation execution method they adopt to run models. There are models designed to execute operations sequentially and model designed to execute operation in parallel, therefore there is hardware specifically designed to support each model category. However, there also exists hardware able to support both of these families, potentially able to execute any type

of AI model. Such kind of hardware is preferable due to its versatility. Hardware designed to provide a flexible yet powerful environment for AI models plays a pivotal role in this field [14]. For instance, consider a model that executes 100 independent operations. If this model is executed on a sequential machine, it requires 100 machine cycles to complete all operations. In contrast, if the model is executed on a system composed of 100 parallel units, it would require only one machine cycle to complete all operations. Therefore, a significant factor in determining the success of a model is its ability to achieve good accuracy within a reasonable amount of time, which is strongly dependent on the adopted hardware.
The importance of aligning theoretical models with appropriate hardware cannot be overstated. Research models are evaluated based on their success in bridging the gap between abstract theory and practical application. This is particularly evident when considering the integration of AI models with advanced hardware platforms [29]. Coarse-Grained Reconfigurable Arrays (CGRAs) have emerged as a promising solution [24], offering the high performance and flexibility needed to execute different kernels on a single piece of hardware [35]. In this context, we propose an innovative CGRA integration into the SNAX system . This system exemplifies how theoretical models can be successfully bonded with hardware to achieve optimal performance. We test and report the performance of SNAX, analyzing it against the overhead incurred, thus demonstrating the practical application of theoretical models in real-world hardware environments. Several different AI-related kernels have been mapped and optimized to fulfil the performance of the SNAX-CGRA towards their theoretical computation power. The design has been implemented in TSMC 16nm technology, with a maximal frequency of 250 MHz. It achieves 2.49 GOPs in accelerating FFT kernels, 4.09x faster than the baseline CGRA solution. The average power efficiency is 330.03 MOPs/mW, 4.91x better than a state-of-the-art (SotA) CGRA system-level solution STRELA.

List of Figures and Tables

List of Figures

List of Tables

List of Abbreviations and Symbols

Abbreviations

CGRA	Coarse Grain Reconfigurable Array
AI	Artificial intelligence
FPGA	Field-programmable gate array
ASIC	Application-specific integrated circuit
DFG	Data Flow Graph
GPU	Graphics processing unit
CPU	Central processing unit
TPU	Tensor Processing Units
FPU	Floating Point Unit
GeMM	General matrix multiply
PE	Processing element
MAC	Multiply-ACcumulate
CU	Control Unit
ALU	Arithmetic Logic Unit
CSR	Control status registers
I/O	Input output
DPS	Digital Signal Processing
FIFO	First In First Out
TCDM	Tight-Coupled Data Memory
SNN	Spike Neural Network
KNN	K-Nearest Neighbors
CNN	Convolutional Neural Network
DNN	Deep Neural Networks
RISC	Reduced Instruction Set Computing
ReLU	Rectified linear unit
FFT	Fast Fourier Transform
DFT	Discrete Fourier Transform

HPC	High Performance Computing
LLVM	Low Level Virtual Machine
GUI	Graphical User Interface
ISA	Instruction Set Architecture
Kernel II	Initiation Interval
FIR	Finite Input Response
FU	Functional Unit
AGU	Address Generation Unit
LD	Load
Str	Store
DMA	Direct Memory Acess

Chapter 1

Introduction

"*Machine learning researchers mostly ignore hardware despite the role it plays in determining what ideas succeed*"
(The hardware lottery [19], 2020).

1.1 Motivation

The significance of hardware in the realm of artificial intelligence (AI) cannot be overstated. As AI technologies advance rapidly, the demand for sophisticated hardware to support these models intensifies. This trend is particularly evident in the commercial sector, where AI's impact is substantial. For an AI model to be deemed effective, it must deliver cutting-edge performance, competing with existing models that operate on highly specialized hardware, such as Google TPUs or Nvidia GPUs. Consequently, nascent models often face challenges in real-world implementation, as they may lack dedicated hardware unless they are designed with existing hardware compatibility in mind. This necessity to conform to available hardware can constrain the research scope, limiting the exploration of new horizons [19].
Moreover, the dependency on existing hardware shapes the extent to which research can push the boundaries of the unknown. The research field is profoundly influenced by commercial realities, where ideas that meet the stringent requirements of specialized hardware enjoy significant advantages and are more likely to thrive. This dependency underscores why insufficient hardware infrastructure can result in the premature dismissal of even the most sophisticated AI algorithms as failures.
Key parameters that define a successful AI model include a balance of accuracy, execution time, and power consumption, typically requiring a trade-off among these attributes. These factors collectively determine the viability and competitiveness of AI models in both research and commercial applications.

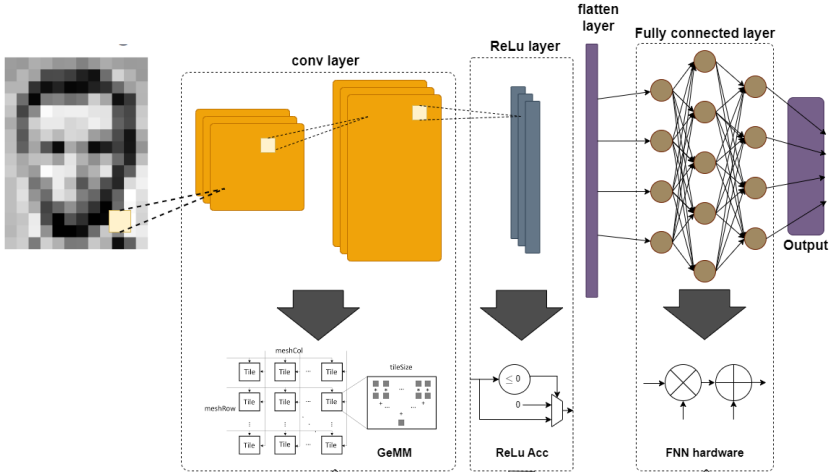


Figure 1.1: Example of layer composition of a neural network (NN) structure. Each layer perform different operations that contribute to generate the Output. In this example a NN that extracts information from an image.

These algorithms (models) are composed by many different interconnected parts called *layers*, figure 1.1. Layers are chunks of an algorithm, characterized by different logical functions. Each layer is a sequence of different operations, which can be executed sequentially or in parallel, depending on the nature of the layer itself. To simplify the view, these layers can be seen as a set of different path that the incoming data cross. If each path is not properly supported by a specific system capable of efficiently execute functions, bottlenecks and limitations can be encountered.
One of the key aspects to classify an AI model is the time that it takes to generate the output once the input is given, i.e. how long does the model take to give a result. However the result of the model depends on the execution of every layer, and the computation of a layer depends on the previous one, which is composed by different path. If all the paths do not end at the same time, then the next layer has to wait for its input to arrive to perform part or all operations. This is how bottlenecks are created and that's why it is so important to have structure (so called *accelerators*) to try to remove or reduce these bottlenecks.
History provided some excellent examples of how important it is to have suitable hardware for AI models. In the famous article [23] a model was created to classify cats by using 16,000 CPUs with a 9-layered locally connected sparse autoencoder. One year later, the same autors designed a new specialized hardware,completing the same classification task using only 2 CPUs and 1 GPU [9]. The authors chose to keep the same model and change only the hardware, to highlight the key role that specialized hardware plays in AI applications.
The modern AI accelerators range across different devices such as CPUs, GPUs

- *CPU-based*, hardware dedicated to sequential execution of operations. This is suitable when we have models with small data set [20], like *linear regression* and *K-Nearest Neighbors (KNN)*

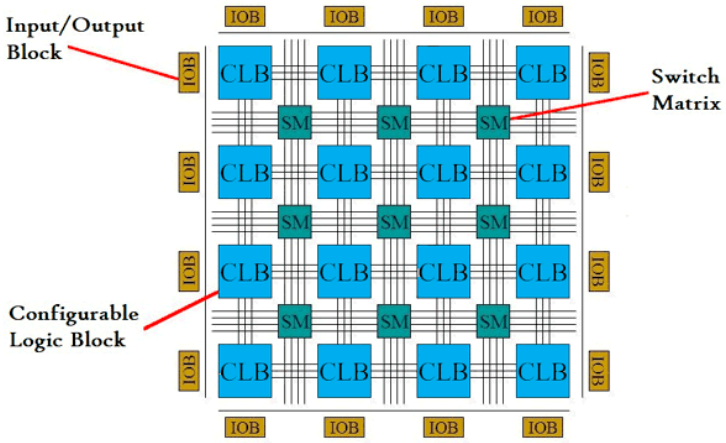


FIGURE 1.2: FPGA architecture. Here it can be seen the area and delay overhead due to the interconnection. The signal as to travel through the interconnection switch that contains much parasitic capacitances due to the transistors that compose the switches.

- *GPU-based*, hardware dedicated to parallel execution of operation, such as image processing.

One popular group of devices that aim to speed up deep learning algorithms are the Deep Learning Processors. This last family relies on huge parallelism of *Multiply-Add-Computations* or in memory computing it exploits different type of techniques to speed up the process like sparsity[7][8] and data locality[7][8] and to save energy it relies on techniques called *quantization* and *pruning* as example works like TPU[21], EIE[18], Eyeriss [8]. These families of specific devices adopt MAC structures to increase performances in deep learning applications. However, the set of instructions that these devices families support is limited to specific operations that are useful only to a restrict portion of models. Models that do not include MAC operations cannot be accelerated by these devices and need some other suitable hardware. Spike neural networks [15] rely on generation of spikes that are evaluated by thresholding or applying activation function on the input. Decision trees is another model that is not based on MAC operation, instead on if-else condition.
Imagine a system designed for high-performance computing tasks, consisting of a TPU-like accelerator, a small CPU for simple arithmetic operations and control, and a generic memory for data fetching. The TPU handles most of the computing intensive operations, while the CPU takes care of basic tasks, ensuring smooth operation. However, a significant challenge arises when we want to introduce another computational kernel into this system.
The existing hardware setup lacks the necessary support to efficiently handle the new kernel, meaning the task would fall to the small CPU. Unfortunately, this CPU is not equipped for such demanding computations and would become a bottleneck, drastically slowing down the entire system. To overcome this issue, we can turn

to two types of programmable devices: *Field Programmable Gate Arrays* (FPGAs) and *Coarse Grained Reconfigurable Arrays* (CGRAs). FPGAs can be configured at compilation time, allowing for flexible hardware design tailored to specific tasks. This adaptability makes them suitable for a wide range of applications, as they can be reprogrammed to optimize performance for different computational kernels. On the other hand, CGRAs offer dynamic reconfiguration capabilities, allowing adjustments at both run time and compilation time. This enables real-time optimization of hardware resources, making CGRAs highly efficient for handling diverse and evolving computational demands. By incorporating FPGAs or CGRAs into our system, we can provide the necessary hardware support for additional kernels, ensuring that the CPU is not overwhelmed and that overall system performance remains high. This approach leverages the strengths of programmable devices to create a flexible and efficient computing environment capable of adapting to various computational challenges.

1.2 Flexibility in Hardware Architectures: From FPGAs to Coarse-Grained Reconfigurable Architectures

In the previous section we stated the problem of not having flexible devices to tackle the introduction of new kernels into already existing hardware, creating bottlenecks and degrading performances. Succesively, to address to this problem, introduced the concept of FPGAs and CGRAs marking their programmable nature, briefly describing them. In this section we will give more insights on the differences between these two architectures, pointing out the problems of the FPGAs and why we are switching to the CGRA structure in AI domains.
At the heart of an FPGA lies an array of Configurable Logic Blocks (CLBs), interconnected by a flexible routing fabric. These CLBs comprise fundamental logic elements such as lookup tables (LUTs), flip-flops, and multiplexers, which can be programmed to implement diverse digital logic functions. This type of architecture provides great flexibility enabling the integration on almost all algorithm, but because of it, to implement even simple operations like *add*, *sub*, *mul* ecc... it needs many blocks and interconnections as well, elements that hide delay, area costs and power consumption, also named *overhead*. While fine-grained hardware flexibility is essential in some contexts, AI kernels necessitate a different kind of adaptability, one rooted in software flexibility. AI models do not need to configure operations at the single-bit level; rather, they operate at a higher, functional level. This means that AI models require a specific set of operations and a distinct configuration for data movement to function effectively. For instance, consider the difference between convolutional neural networks (CNNs) and fully connected networks. Although both types of networks utilize similar operations (such as addition and multiplication), the total number of these operations and the manner in which the computed data is routed differ significantly. This distinction highlights the need for flexible functional logic in AI models. Instead of configuring individual components like adders or

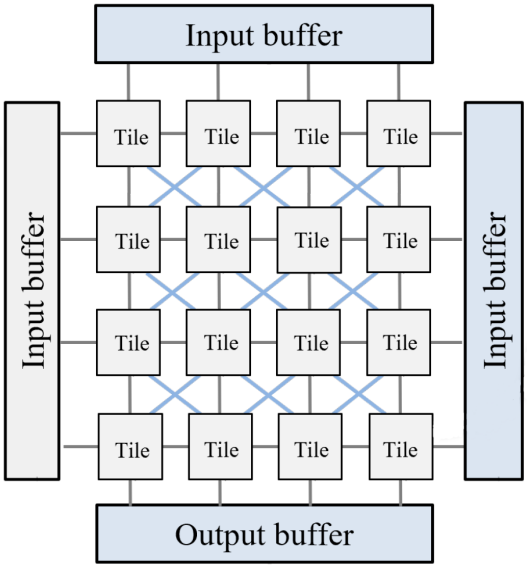


FIGURE 1.3: Generic CGRA architecture. Tiles are connected via a mesh type. Only outermost tiles can fetch data from the memory. Image taken from [31]

multipliers at a granular level, the focus is on how these components are utilized within the entire model's architecture and data flow.
CGRAs (figure 1.3) embody this principle of software flexibility. CGRAs allow for the reconfiguration of functional units (such as switching an adder for a multiplier) and data flow paths at a higher level of abstraction. This reconfiguration applies to the entire architecture, enabling different models and functions to be implemented efficiently. By adjusting the functional logic and data pathways, CGRAs can adapt to various AI workloads, providing the necessary flexibility without the need for low-level hardware adjustments. They does not rely on CLB structures but instead adopt specialized architectures to perform operations in such a way that overhead is reduced. These architectures are organized into blocks called *tiles*. They are blocks that can be configured to select which *functional units* . The tiles are interconnected in such a way only neighbouring tiles can communicate with each other (called *king mesh*). With such architectures and specialized functional units CGRAs can achieve better performances than FPGAs.
This flexibility ensures an environment capable of supporting any algorithm that can be translated into a Data Flow Graph (DFG), whether AI-related or not. Detailed explanations of this capability will be provided later.
In conclusion, similar to FPGAs, this system retains reconfigurability even after tape-out, thus preserving the freedom to select and execute various kernels, while still delivering high performances by adopting specialized Processing Elements (PEs). Within the AI domain, this concept extends to the capacity to run multiple AI models and, critically, models that have yet to be developed.

1.3 State-of-the-art

This section provides a comprehensive review of existing research pertaining to the adopted CGRAs and their integration into computing systems. By examining various architectural proposals and their implications, we aim to contextualize our research within the current landscape of CGRA-based computing, focusing on off loading operations related to memory access and integrating the architecture into complex systems.

1.3.1 Offloading Kernels into CGRAs: Proposals and Innovations

As explained in section 1.1 using CPU for non hardware supported kernels might induce huge bottlenecks, slowing down the whole process. Introducing a generic purpose processor to sustain new-bork kernels and allow to off load computing intensive algorithms is fundamental for the overall system performances.
First, we examine proposals focused on enhancing the offloading of kernels into CGRAs. In Softbrain [26] streaming engines with regular access patterns are proposed to simplify this process, this paper shows how CGRAs can achieve similar performances to domain specific accelerator for DNNs, at the expenses of area and power. The dMT-CGRA [39] utilizes multithreading techniques to parallelize tasks on the accelerator allowing to avoid redundant stores by keeping data locally, targeting high-performance computing (HPC) and comparing their approach with GPUs. Similarly, REVEL [41] suggests using CGRAs as vector lanes within a Vector Processing Unit (VPU), combining systolic computation with dataflow. This approach incorporates streaming engines for data transfer from main memory and includes scratchpad memories within each lane, aiming to compete with Out-of-Order (OOO) CPUs at high frequencies. Next, the most similar to this work together with CDA [42], DSAGEN [40] introduces a hardware-software codesign framework to generate an architecture tailored to specific applications for offloading, using streaming engines for memory communication. Meanwhile, CDA [42] focuses on unrolling data flow graph (DFG) subgraphs to boost performance, targeting HPC and benchmarking against other high-performance CGRAs and GPUs. This work is a combination of the work of DSAGEN[40] and CDA[42], creating a complex system controlled by a RISC-V[1], offloading memory managements to ad-hoc modules and unrolling DFG nodes to achieve better computational performances.
These works gave important information on how CGRAs can compete with ASICs. Softbrain gives good performances, however cannot support irregular loops, which plays a big role in AI algorithms, to fill this requirement [25] was used instead. Furthermore Softbrain offloaded memory accesses to an external module that led to significant improvements. However these approach still face challenges as their module can only allow regular access to the memory, while irregular one is fundamental for DFG processing, as later will be explained. dMT-CGRA starts to use multithreading techniques avoiding redundant stores and keeping the values inside the CGRA fabric, this technique is also used in our case [31] by moving the value in interest across tiles.

In REVEL the authors used scratchpads for each lane, which result in a big overhead, we avoided this since we are now working with a Tight Coupled Data Memory (TCDM). Lastly, OpenCGRA [31] was reviewed, which is a combination of the previous mentioned work, enable both static dataflow and time multiplexed, enabling data re-use across threads and instead of multiple scratchpads for each lane, it uses a single external scratchpad, having much potential for HPC. However despite having combined multiple good aspect of different work, OpenCGRA did not explored the offload principles that Softbrain and DSAGEN used by using streamers to access memory and the whole control logic is implemented inside the CGRA which consists in increasing the kernel II.

1.3.2 Integration of CGRAs into Embedded Systems: Design Strategies and Energy Efficiency

The integration of CGRAs into embedded System on Chips (SoCs) has also been explored in several state-of-the-art works. These studies are summarized here and will be used for comparison with our system in result section. IPA [11] features an ultra-low-power CGRA integrated with an OpenRISC processor, utilizing clock-gating mechanisms to enhance energy efficiency. UE-CGRA [36] employs an RV32IM RISC-V processor to control the CGRA accelerator, implementing sophisticated Very Large-Scale Integration (VLSI) mechanisms to adjust the clock frequency of the CGRA processing elements (PEs) for control-driven applications, thereby reducing energy consumption. RipTide [16] presents an ultra-low-power CGRA integrated with a RISC-V core that includes RV32EMC extensions. Lastly, [37] propose a streaming elastic CGRA microarchitecture that can efficiently compute data- and control-driven applications. It focuses on managing processing capabilities over time and space, aiming to improve energy efficiency and performance compared to existing CGRA architectures. The goal is to integrate this CGRA into a system-on-chip (SoC) to serve as a general-purpose accelerator for offloading computing-intensive tasks from the processor, enhancing overall system performance and energy efficiency. By reviewing these works, we aim to provide a comprehensive background and context for the design choices made in our proposed CGRA system, ensuring that our approach is well-informed and builds upon established research in the field.
As the aforementioned works, we will implement a RISC-V based core called Snitch [43] to control the CGRA, clock-gating is a fascinating technique that can deliver good energy efficiency and ideally implement it. A 4x4 structure like STRELA and IPA will be adopted since it can deal with most of the implemented kernels having a kernel II of 1 without increasing too much the overhead. However instead of having a unidirectional flow of data like STRELA (from top to bottom), in the SNAX-CGRA implementation the loads are happening on the top and left side and stores from the bottom and right side, increasing the maximum bandwidth and processing capabilities.

1.3.3 Conclusion

After reviewing these works and the considerations done in the previous subsections, we decided to adopt and explore the base architecture of [31], exploiting its capabilities. In section 3 we will remove the scratchpad inserting the CGRA in a more complex system, the SNAX, using an external Tight Coupled Data Memory (TCDM). In addition we will off-load some operation from the CGRA following Softbrain and DSAGEN by using modules, in this case Address Generation Units (AGUs) instead of stream engines, to interface with the memory, enabling irregular access to the memory.

1.4 Goal of the Thesis

The primary objective of this research is to integrate a CGRA within a complex computing system, specifically the SNAX framework. The SNAX system is based on a RISC-V architecture, utilizing the Snitch core as the central processing unit (CPU) that acts as the controller for the CGRA accelerator. Additionally, the system incorporates a flexible memory architecture with individual, independently accessible memory banks, creating an optimal environment for executing artificial intelligence (AI) kernels. The following sections outline the main contributions of this work

1.4.1 Integration of the CGRA within the Complex System

This research details the methodologies and challenges associated with integrating the CGRA into the existing SNAX system. It addresses both hardware and software aspects to ensure seamless communication and coordination between the CGRA and the Snitch core. The integration process involves designing and implementing interface protocols that allow the CGRA to interact efficiently with the CPU and other system components. This includes the development of control signals, data paths, and synchronization mechanisms to ensure coherent operation across the system.

1.4.2 Creation of Custom Control and Status Registers (CSRs) for CGRA Management

The study introduces and documents the development of specialized Control and Status Registers (CSRs) tailored to manage and monitor the CGRA operations. These CSRs provide the necessary control mechanisms to facilitate efficient utilization of the CGRA within the system. Custom CSRs are designed to enable fine-grained control over CGRA functionalities, including configuration, status monitoring, and execution control. The design process includes specifying the register map, defining the control signals, and ensuring that the CSRs integrate seamlessly with the RISC-V instruction set architecture.

1.4.3 Optimization of Data Exchange between Memory and CGRA

This research focuses on optimizing the data transfer processes between the flexible memory system and the CGRA to enhance overall performance. Techniques to reduce latency and increase data throughput are explored and implemented. The optimization strategies include designing efficient memory access patterns, leveraging parallelism, and implementing caching mechanisms to minimize data transfer delays. These optimizations are critical for maximizing the performance of AI kernels, which are often data-intensive.

1.4.4 Optimizations for Implemented AI Kernels (FFT, Convolution, ReLU)

The thesis investigates specific optimizations for key AI kernels, including Fast Fourier Transform (FFT), Convolution, and Rectified Linear Unit (ReLU) operations. These optimizations are aimed at improving execution efficiency on the CGRA. Each kernel is analyzed to identify computational bottlenecks and opportunities for parallel execution. Optimizations such as loop unrolling, data prefetching, and efficient utilization of CGRA resources are applied to enhance performance. Detailed performance evaluations are conducted to quantify the improvements achieved.

Together with the implementation, further exploration was conducted for each section, seeking multiple solutions to the aforementioned challenges. This comprehensive approach ensures that the proposed system is robust, efficient, and adaptable to various AI workloads.

1.5 Thesis outline

- **Chapter 1: Introduction**
  Chapter 1 presents a comprehensive overview of the motivations and goals driving the interest in CGRA architecture. It discusses the broader context and significance of the research, outlining the specific objectives and expected contributions of this thesis. This chapter sets the stage by explaining why CGRA is a focal point for advanced computational research, particularly in the domain of AI kernel execution.

- **Chapter 2: Literature Review**
  Chapter 2 delves into the research landscape relevant to this work. It covers a range of topics including the various tools and development environments utilized in CGRA research. The chapter also reviews different CGRA architectures, highlighting their key features and capabilities. Additionally, it explores the frameworks employed, with a particular focus on the SNAX system, providing a detailed understanding of the foundational technologies that underpin this research.

- **Chapter 3: System Design and Contributions**
  Chapter 3 outlines the primary contributions of this thesis. It begins with a detailed account of the engineering efforts behind openCGRA, describing the process of integrating this architecture into the SNAX system. The chapter then explains the translation of configuration scripts from Python to C, and how these configurations are utilized to automate the CGRA setup within the SNAX environment. Furthermore, it discusses the development and implementation of CSRs, including their design rationale and functional role. This chapter also includes the adopted techniques to adapt the C kernel to a CGRA friendly format and later explain the optimizations to increase the kernels performances. Lastly, the section covers the creation and use of testbenches to validate the systems performance and functionality.

- **Chapter 4: Conclusion and Future Work**
  Chapter 4 provides a thorough analysis of the thesis conclusions. It interprets the results obtained from the research, offering insights into the effectiveness and efficiency of the CGRA integration. The chapter also reflects on the overall impact of the contributions made. It concludes with a discussion on potential future work and possibilities, suggesting avenues for further exploration and improvement in CGRA and related technologies. This includes proposed enhancements and additional research questions.

Chapter 2

Background

Chapter 2 delves into the research landscape relevant to this work, providing a comprehensive overview of the foundational concepts and technologies that underpin this thesis. It begins by introducing the concept of CGRA architecture, explaining the motivations behind choosing CGRA for this research. The chapter highlights the flexibility, efficiency, and performance benefits of CGRA in executing AI kernels, emphasizing its ability to be reconfigured at the functional block level for optimized performance across various computational tasks. Following this conceptual introduction, the chapter discusses the structural components and operational principles of CGRA, detailing its internal architecture and highlighting key features such as parallel processing abilities and scalability. The advantages of CGRA over traditional fixed-function hardware are underscored, supported by diagrams and figures that illustrate the architecture and functionality.
The discussion then transitions to the various tools and development environments utilized in CGRA research, reviewing the software and hardware tools used for designing, simulating, and implementing CGRA architectures. The importance of these development environments in facilitating the creation and optimization of CGRA systems is highlighted.
OpenCGRA framework is introduced with an explanation of its operation and the types of files it generates, emphasizing its role in streamlining the design and configuration process of CGRA architectures. The practical applications and benefits of using OpenCGRA in this research context are thoroughly discussed. The chapter also provides an in-depth look at the SNAX system, integral components of this research. The SNAX system is introduced with a detailed explanation of its architecture and the rationale behind its selection for this study. This section elucidates the synergy between the CGRA, and the SNAX system, offering insights into how these components create an efficient and powerful computing environment. To conclude, this chapter introduce the basics for understanding the *Fast Fourier Transform* algorithm, starting from the sampling criteria and how the *Discrete Fourier Transform* works. The basic hardware component of the FFT is here introduced.

2.1 CGRA

In the domain of AI model creation, various system designs have emerged, with CGRA standing out as a fundamental component (Charitopoulos et al., 2021 [6]). These CGRA designs offer a template-based approach, where adjustable factors partially determine the reconfigurable architecture's structure. This enables the creation of flexible and adaptive AI models, particularly in terms of kernels. In recent years, there has been a notable surge in the popularity of CGRA algorithms, largely attributed to the growth of *High-Performance Computing* (HPC). HPC involves the utilization of supercomputers and parallel processing techniques to solve complex computational problems, aggregating computing power to achieve significantly higher performance than conventional desktop computers or workstations [27]. By leveraging parallelism, tasks are distributed across multiple processors or machines, effectively reducing computation time. This aligns closely with the nature of CGRA architectures, as their ability to distribute workload across multiple processing elements makes them well-suited for such tasks.
There exists a strong correlation between HPC and *Data Flow Graph* (DFG) processing, where parallel operations are employed to accelerate hardware performance, akin to the concept of multiple parallel nodes in CGRA architectures. DFG processing can be utilized within HPC systems to manage and schedule parallel tasks, serving as a methodology to achieve high-performance computing. Given that CGRA functions as a DFG processor, it emerges as a compelling candidate for HPC applications. Moreover, the increasing demand for large-scale models in AI development necessitates enhanced performance and power efficiency in hardware (Cao et al., 2023)[5]. CGRA architectures have demonstrated excellence in meeting these requirements.
Additionally, CGRA, also referred to as DFG processors, play a crucial role due to their ability to execute any composition of operations represented by a DFG. A DFG is a graphical depiction of data dependencies among operations, where *nodes* represent computations and *directed edges* signify data dependencies between these operations. The organization of the graph into levels reflects the logical execution of nodes, with horizontally aligned nodes considered to be on the same level. Nodes on the same level are executed concurrently, while those on different levels are executed sequentially at different time instants.

Understanding CGRA as DFG processors holds significance because they possess the capability to execute any composition of operations represented by a DFG.

With more focus we want to execute operations that are **often executed** and **repeated** and within complex algorithms, most of the time, these set of operations can be individuated as [4] and [3] do for complex-valued least-mean-square, and they take the name of *kernels*. AI models algorithm are composed by many kernels. If we take a classic *Deep Neural Network* (DNN) it is made by multiple fully connected layer, each one of them with a activation function at the output, this function could vary like a *ReLu* (if the output is less then 0 then the function gives 0 otherwise the output is let through) or some other complex function. Taking as example a

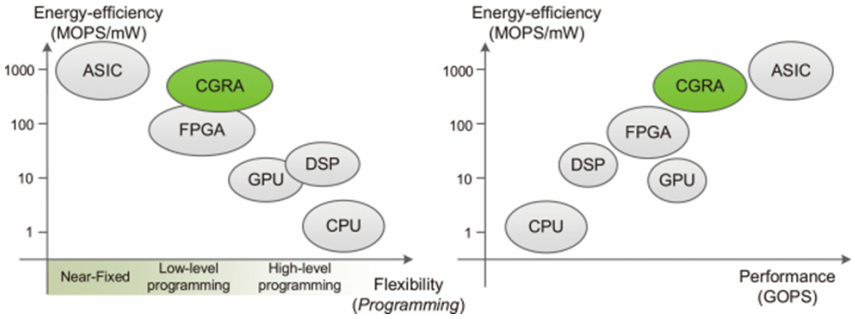


FIGURE 2.1: CGRA in flexibility/performance spectrum

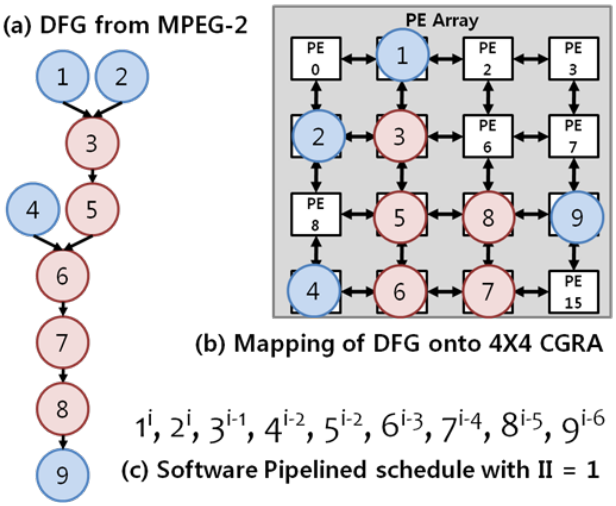
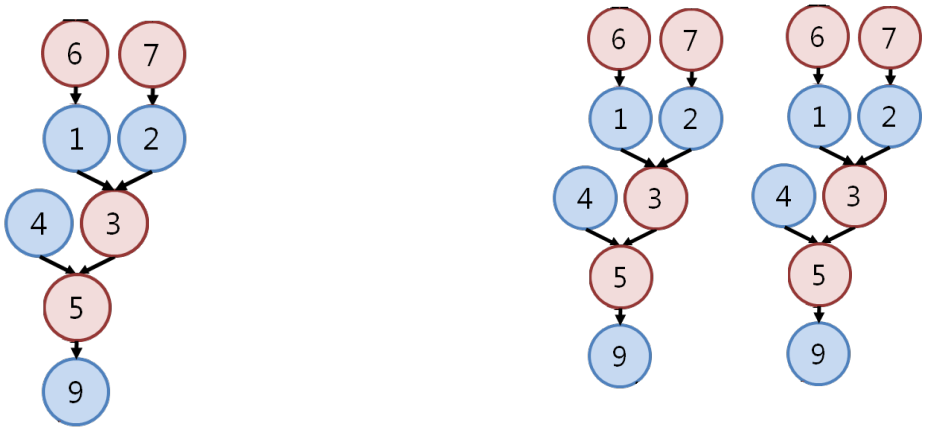


FIGURE 2.2: Generic DFG mapping on CGRA. The DFG nodes are being assigned to the CGRA tiles and the DFG arrows translate into where the result of a tile is being directed.

node from the first layer of a DNN, this layer gets its input from the external world, multiplying it with a weight, adding a certain bias, passing the result through the activation function and sending the input to every node of the next layer to then repeat the process. In this example there are actually 2 kernels: the ReLu and the MAC, that gets repeated for each node, for each layer until the end of the network. This example shows how even a DNN can be deployed on the CGRA. The deployment of a DFG on the CGRA is called *mapping*, where each node of the DFG will be give to a CGRA tile and the arrows that links DFG nodes will be connections between two tiles. An example is represented in figure 2.2.

However it is worth noting that the mapping of a kernel can vary. There are different techniques that can be applied to the kernels that will deliver different DFG and different CGRA maps.
These techniques are used to, in general, increase the performance of kernel itself. As example we will take in consideration the following code, implementing a simple



(A) Basic convolutional kernel with no spatial unrolling, DFG nodes: 6,7 = load, 3 = mul result, 5, add result, 9 store back.

(B) Spatially unrolled convolutional kernel, DFG is replicated because the operation has been doubled

Figure 2.3: Basic Convolutional kernel and spatially unrolled one.

convolutional kernel.

```
1  for (int i = 0; i < CONST_I; i++) {
2    for (int j = 0; k < CONST_J; j++) {
3        C += A[i][j] * B[i][j];
4    }
5  }
```

This kernel can be represented by the DFG shown in figure 2.3a.

One of the technique that can be adopted to increase the performances (reduce the total needed time to compute the kernel) is to *spatially unroll* the kernel. This means to compute, in a single for iteration, the same operation. In this way what we obtain is a reduction of the total number of cycle needed to compute the total kernel. Hardware wise this means to map the same operation twice on the CGRA, having a total number of operation equal to the original times the number of spatial unroll.

```
1  for (int i = 0; i < CONST_I; i++) {
2    parfor (int j = 0; k < CONST_J - 2; j = j + 2) {
3        C += A[i][j] * B[i][j];
4        C += A[i][j+1] * B[i][j+1];
5    }
6  }
```

Where the *parfor* states an actual parallel operation, instead when using a simple for everything inside it, is yet sequential.

The basic concept when dealing with increasing performances in integrated chip design, is to trade area with performances as [34] states, exploiting Moore's law, that states that the number of transistors on an integrated circuit will double every two years with only a minimal increase in cost.

Just from this examples it is pretty visible how CGRA flexibility and computational

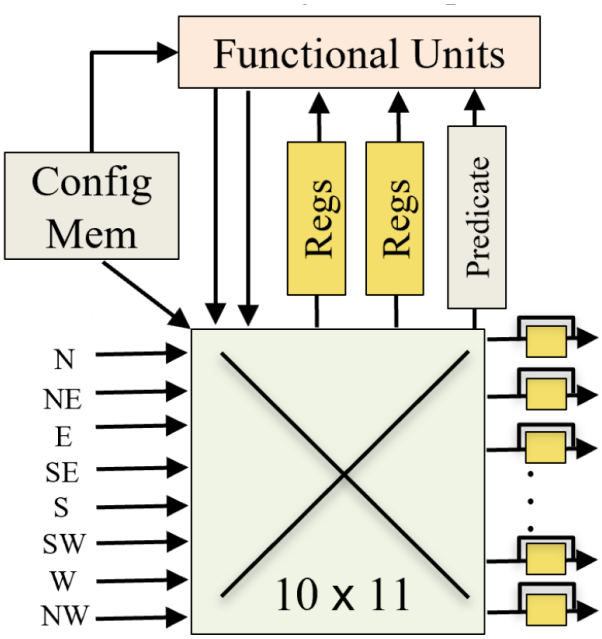


FIGURE 2.4: CGRA tile architecture, the *config mem* is used to control the *crossbar* and functional units. The crossbar is used to direct data within the tile. Image taken from [31]

potential can be exploited. After having dealt with the functional side of the CGRA, let's dive into its architecture, starting from the core of the structure, the Tile.

Tiles

The tile, also known as PE (processing element), is the core of the CGRA, having N of it, the CGRA has N processing unit.
Each tile, generally, is composed by a **memory configuration**, *ALUs* and a module to control the data flow inside the tile, in this case a *crossbar* was included.
The memory configuration is the *Control Unit* (CU) of tile that, at each cycle, defines its internal interconnections and functioning, i.e. which operation the tile has to compute with which input and where to send the result.
The control memory, after being uploaded with the correct memory configurations, loops around until the kernel is finished.
By arranging and connecting tiles in structures, such that input flows from a cell to another, being transformed operation after operation. Each tile will apply its own operator to the data received from previous cell, until the end of the structure is reached. If the structure is considered as a whole, the combination of all the tiles results in the combination of the basic functions in a single complex function.

Architecture

A generic basic CGRA is composed by single processing elements interconnected by a *mesh*, a grid-like structure where the tiles are connected to their respective neighbours, figure 1.3 shows a picture of the CGRA.
The CGRA to get the data has interconnection to an external memory. We can have 2 types of memory, configuration memory and data memory. Configuration memory is from where the configuration of the tiles are taken meanwhile data memory is from where we fetch data. The interconnection to these memory can vary, as its possible to encounter configurations in which the interconnections are connected only to the first row or column (based on where the data memory is placed) or to every tile. The current implemented CGRA configuration is that the connections to the configuration memory are done with every tile, meaning a parallel instruction loading meanwhile for the data memory, only the first row/column are connected. This means that data coming from the memory, has to flow inside multiple tiles to end up in the correct processing element where the data is needed.
The reconfiguration of the CGRA can be done at *configuration* time and *runtime*, namely before or after the execution of each kernel. Configuring the CGRA consists in storing configuration values inside the control memory. These values are a combination of all the setting parameters that can be configured inside the CGRA (this will soon be explained when talking about the control memory). To store the values inside the registers of the control memory a certain amount of cycles are needed. This is an overhead that has to be carefully considered, because a first assumption to use CGRAs is that the configuration time is much smaller then the total cycles needed to finish a process. The specific quantity is not certain as the process can still have a total execution time equal to the configuration, but if the same execution requires more cycles if done on a CPU then CGRAs can still be considered a valuable alternative.
To hide this latency, a double buffering technique could be adopted[33]. This idea consists in using multiple control memory in each tile (at least 2). This will increase the area of each tile (overall area) but could considerably reduce (possibly to 0) the configuration cycles. The idea develops in loading the first control memory and start with the execution of the kernel. The loading of the second control memory happens concurrently to the execution of the kernel, hiding its latency.
Thanks to its flexibile nature and thanks to the possibility of reconfiguring single tiles, the CGRA can host different complementary kernels at once, rebalance its own structure to reduce kernel bottlenecks (DRIPS, 2022[30]) or unrolling operations to exploit its parallel behaviour, increasing the performances.

2.2 OpenCGRA

The OpenCGRA framework is a unified environment for CGRA modeling, testing, and evaluation. Its main properties are the ability to model CGRAs at various levels of abstraction functional, loop, register transfer. Functional is the highest level of abstraction giving almost no insight on how the model behaves except for the

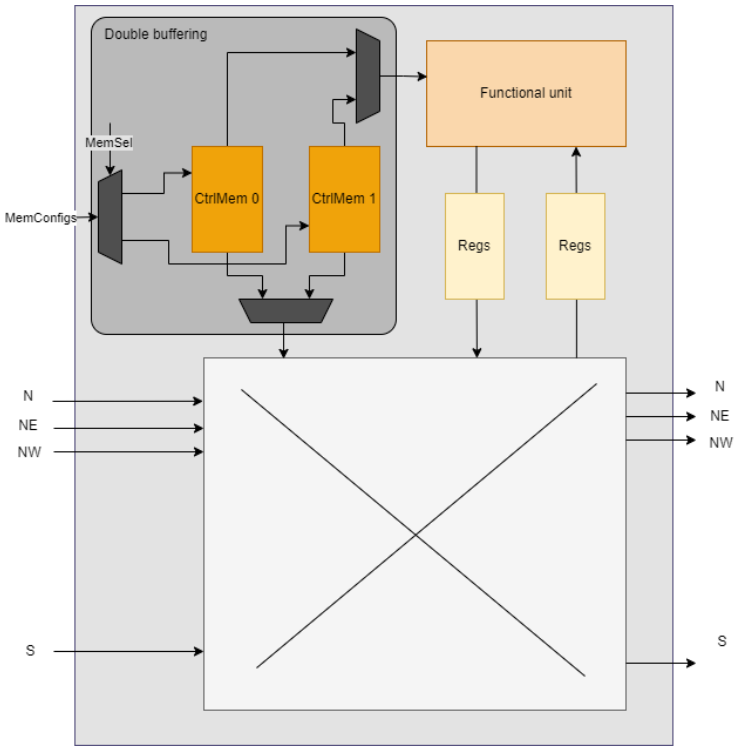


FIGURE 2.5: CGRA Tile with double buffering technique. Muxes are used to decide which control memory to configure and which is leading the tile.

given result. The model can be seen as a black box only knowing the input and out, so basically if the result is correct or wrong. For the second level we can see the functioning on a lower level, in this case within a loop of execution. The last is the register transfer level, which shows all the exchange of data between registers and wires.
Furthermore this framework offsers compiler support for mapping operations to CGRAs and guiding the choice of heterogeneous component design, and the provision of sophisticated test harness for testing CGRA designs modeled at various levels of abstraction.
Mapping refers to translate the C code into CGRA instructions. This process takes some steps, strating from the C code, a specific function is elaborated and initially translated to LLVM, this already grants optimization of the code, that sometimes might be unwanted. Successively the mapping consists in translating these LLVM operations into supported CGRA operations that are strictly related to the LLVM ones.
OpenCGRA allows the simulation of CGRA at several levels of analysis, synthesizable Verilog generation. To ease the usage of the OpenCGRA Framework, a simple and functional GUI (Fig. 2.6) has been developed by its authors. The user can configure the CGRA, enabling the selection of the units to be included in the PE) and the type of interconnections 2.1.

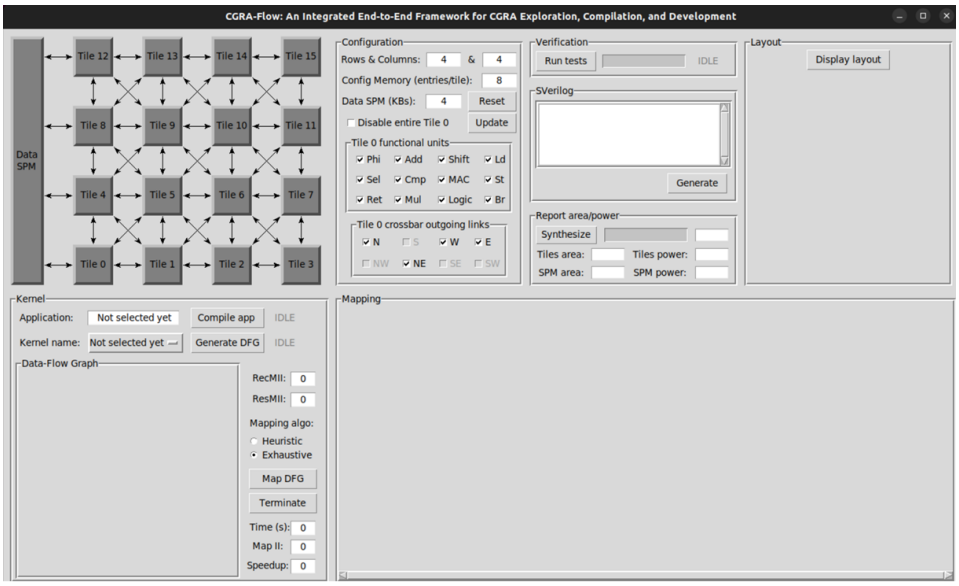


Figure 2.6: OpenCGRA GUI

.

Here is an example taken from CGRABenchmark: GeMM kernel.

```
/* Main computational kernel. GeMM,*/
void kernel(int ni, int nj, int nk)
{
  int x, i, j, k;

//BLAS PARAMS
//TRANSA = 'N'
//TRANSB = 'N'
// => Form C := A*B + C,
//A is NIxNK
//B is NKxNJ
//C is NIxNJ
//#pragma scop
  for (i = 0; i < _PB_NI; i++) {
    for (k = 0; k < _PB_NK; k++) {
       #pragma clang loop vectorize(disable) unroll_count(4)
       // #pragma clang loop vectorize_width(4) unroll_count(4)
       for (j = 0; j < _PB_NJ; j++){
        C[i][j] += A[i][k] * B[k][j];
        }
    }
  }
}
```

OpenCGRA is composed by three packages:
MapperCGRA: It maps the inmost for loop of the selected kernel into CGRA operations (non dimenticarti di spiegare che vuol dire). It generates a binary executable file and converts it to config.json (italic) file containing the CGRA configuration that will be feed to the next package. *OpenCGRA*: it generates the verilog RTL code from

the binary exectuable. It can be paired with a *pytest* file. This type of file is a python file that through the PyMTL libraries tests the generated hardware by loading during compilation certain arrays (constant values, control memory, max cycle simulation length ...). The control memory configuration is taken from a config.json file.
PyMTL, which stands for Python-based Hardware Modeling, Translation, and Hardware Simulation Framework, is a library for the Python language designed to facilitate modeling, translation and simulation of hardware circuits. PyMTL allows to describe, test, and verify hardware systems in Python.
The last module is *CGRABenchmark*: offers different type of kernels supported by the *OpenCGRA*.

2.2.1 MapperCGRA

The MapperCGRA[31] is the portion of OpenCGRA that has the role of mapping the C/C++ kernel into CGRA operations, it uses LLVM 12.0 and CMAKE 3.1.
After selecting a preexisting kernel or writing a custom one, the MapperCGRA translates only the innermost for loop into a binary executable file. The executable file is used later for hardware generation and verification of kernel execution, Along the executable file, a LLVM IR human-readable file is generated. The command to perform such operation is:

```
clang-12 -emit-llvm -O3 -fno-unroll-loops -o kernel.bc -c conv.c
llvm-dis-12 kernel.bc -o kernel.ll
```

It is also possible to generate a PNG file containing a schematic representation of the mapping result (Fig. 2.7) using the command:

```
dot -Tpng _Z6kernelPfS_S_.dot -o kernel.png
```

Let's consider a generic kernel and its corresponding DFG. In figure 2.7 its schematic representation is shown. It is straightforward that such representation can improve the understanding of the underlying mapping of the kernel. The operations reported in the mapping (Fig. 2.7) have names that can be easily associated to common operators (such as *add*) and names that represent more complex operations. Here the list of operations and their functionality reported in Fig. 2.4:

- *Getelementptr*: evaluate the address of the input data, typically defined as a normal addition (there exists another type of addition which is the ADD_CONST in CGRA operations, the name is pretty self explanatory, but it consists in adding the incoming value with a constant determined at compilation time by the user).
- *Cmp*: compare operation. There exists two type of compares, constant and non. The constant compare, compares the input to a constant value that, similar to the ADD_CONST, is loaded into the CGRA local registers during compliation time. For the non constant, two values are compared to each other, uses 2 inputs.

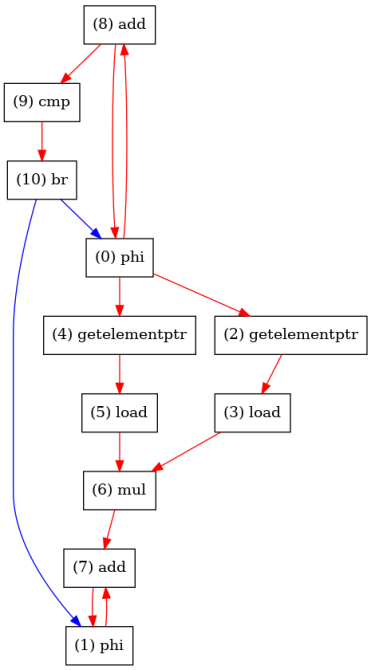


Figure 2.7: FIR DFG

- *Br*: branch operation, used for conditional statement. Typically after a compare node, if the compare returns true then program is sent to node A otherwise node B.
- Phi: depending on the type of the previous node, this operation acquires a certain value. In this case this node is used to feedback to the add(8) node, its previous value.

After the compilation into binary executable file, a new translation into OpenCGRA operations happens, done by:

```
1 opt-12 -load ../build/src/libmapperPass.so -mapperPass ./kernel.bc
```

The translation happens by terms of the **libmapperPass.so -mapperPass** which basically uses all the source code functions to generate the map configuration. By using this command the param.json is called, which contain hardware information about the CGRA and mapping functions. I will not report every field of the param.json, but some of the important ones.

- kernel: denoting the function name that appears in the IR file that was created.
- targetFunction: whether aiming for the loop alone or the complete function. As CGRA focuses mostly on loop acceleration, set it to false.
- doCGRAMapping: stating if the mapping is carried out. This variable can be set to false if all that matters to you are the loop DFG's statistics (number of

nodes/edges, loop-carry dependence length, number of loop-carry dependents, etc.) without any mapping.

- row: number of CGRA rows.
- column: number of columns in the CGRA.
- regConstraint: the quantity of registers that are utilized to store arriving data temporarily before being computed later. By default, set to 8.

During the engineering part of this framework, some of the source functions were changed/added, it will later be explained.
The main instructions of the MapperCGRA that is in our interest is not related to how the mapper searches for configuration related to the kernel, but it is to note that two possible path are possible: exhaustive and heuristic. The exhaustive approach works for the FIR kernel, but not for convolution and GEMM, as the program crashes after few minutes. The high level explanation of how this method works, is that the algorithm tries every possible configuration to then return the cheaper one, the cost is related to a combination on how much the length of the path data has to travel and if there are no conflict.
The heuristic approach will surely return a solution, but few problems may occur: it's not granted that the mapper will find a working solution and the solution found will not be granted to be the optimal one. For the kernel convolution the heuristic approach was the only possible method to follow, giving as final result an approximation of the original kernel. This will be later discussed, where it will be explained the used methods and the outcome of the work.
Regarding the source code of the mapperCGRA, the main focus is brought onto certain set of functions that are strictly hardware related.

```
1 void DFGNode::initType() {
2   if (isLoad()) {
3     m_optType = "OPT_LD";
4     m_fuType = "MemUnit";
5   } else if (isStore()) {
6     m_optType = "OPT_STR";
7     m_fuType = "MemUnit";
8   } else
```

Here are reported few lines of code just to give a view with what we are working with. This function is closely related to the LLVM code, as it is trying to link an hardware module to the LLVM operation. In the specific example of this code, the **m_fuType** is chosen to be a MemUnit in case of load and store operation present in the LLVM file. Together with the FU type the operation type is set through the variable **m_optType**. These string variables, will be later interpreted by different coexisting external files: the *opt_type* and the *map_helper* later used by the OpenCGRA.
These files acts as reference to load the control memory with the correct operations with their corresponding encoded values.

2.2.2 OpenCGRA tools

For this project, a diverse set of tools spanning multiple programming languages and development environments was utilized to achieve the research objectives. These tools facilitated various stages of the design, implementation, and verification processes.

- OpenCGRA, a crucial tool in this project, required Python 3.7 and Verilog/SystemVerilog for its operation. OpenCGRA streamlines the design and configuration of CGRA architectures by automating many steps in the workflow. The AI kernels were initially defined in C/C++ and subsequently translated into binary executables using LLVM 12 [22]. This translation process ensured that the kernels could be efficiently executed on the CGRA, leveraging LLVM's powerful optimization capabilities .

- The SNAX system, integral to this project, required Python 3.10 for various scripting and automation tasks. The system's Register-Transfer Level (RTL) files were written in SystemVerilog, which provided a robust framework for modeling the hardware components of the SNAX system. Programs for the Snitch core were written in C, with Python scripts used to generate configuration data from JSON files. This configuration data was essential for tailoring the system to specific tasks and optimizing its performance.

- During the development of the RTL files, extensive testing and verification were conducted using QuestaSim, a leading simulation tool for hardware verification. QuestaSim enabled rigorous testing of the SystemVerilog designs, ensuring that the hardware components functioned correctly and efficiently. This step was critical for identifying and addressing any issues early in the development process, thereby improving the reliability and performance of the final system.

- All development activities were conducted within a Linux environment, chosen for its robustness, flexibility, and widespread use in academic and professional settings. The Linux environment provided a stable and consistent platform for running the various tools and scripts required throughout the project. It also facilitated the integration of different tools and workflows, streamlining the overall development process .

2.2.3 OpenCGRA capabilities and limitations

The initial work of [31] provides a well rounded architecture that suits the general flexibility needed to handle multiple regular and irregular loops. The capability of handling temporal configuration together allows a wider range of kernel to be integrated compared to the STRELA [37], RipTide [16] or UE-CGRA [36].
Furthermore the possibility to choose the correct functional unit, managing I/O directions, support the system level reconfiguration needed by different workloads of AI, as already discussed in section 1.1.
However the hardware (discussed in section 2.1) is naive, having an average utilization

which is suboptimal, lacking of dedicated DMA controls and peripheral control modules, aspects that later in this work will be explored.

2.3 SNAX system

The SNAX system is a new-born architecture, that bonds the need of having a loose connection with accelerators and a tight connection with memory. With loose it is indicated the amount of information that one system has on the other in order to connect them, and make them communicate, thus, loose means that the accelerator do not have much knowledge on how the system, where it's being placed, behave and controls it. This allows the accelerator designer to do not modify its structure much when placing it into this system, a non intrusive approach.
What is needed are control status registers and an interface translator. Since the system communication protocol is AXI, what the interface translator do, together with the csrs, is to include this protocol in the accelerator structure and communicate with the hosting system. Although, the drawbacks from a loose couple is that to configure the accelerator more cycle are required compared to a direct connection. It's a trade-off between interconnection complexity and computing efficiency. Although, this lack of computing efficiency is not a burden, since this loose coupling is only used to configure the accelerator. What is assumed is that the kernel total size is much bigger then the configuration cycles (100x, 1000x), which is reasonable since AI kernel deal with huge quantities of data.
The tight coupling with the data memory is what make this system stand out from the others. The tight coupling offers a high bit rate transfer, without much latency from a request from the master (accelerator) and the response from the memory.

2.3.1 Motivation

The integration of CGRA into the SNAX system is driven by the need to efficiently process AI and DSP kernels, which typically involve handling massive amounts of data. Achieving high performance in these applications necessitates minimizing data latency and maximizing throughput. The combination of SNAX and CGRA offers several compelling advantages that leverage the strengths of both architectures to meet these stringent performance requirements. AI and DSP workloads demand rapid data movement to maintain high computational throughput. The Tightly-Coupled Data Memory (TCDM) in the SNAX system is highly optimized for this purpose, achieving a minimal latency of 1 cycle from request to response. This low latency is crucial for maintaining the high data transfer rates necessary for efficient kernel execution. Moreover, the SNAX system incorporates Snitch cores, designed to enhance performance through an extended Instruction Set Architecture (ISA) that seamlessly integrates the CPU and Floating Point Unit (FPU).
This integration is particularly advantageous for machine learning and DSP applications, as it optimizes load and store operations by replacing them with register read and write operations. As a result, Snitch cores provide a significant boost in

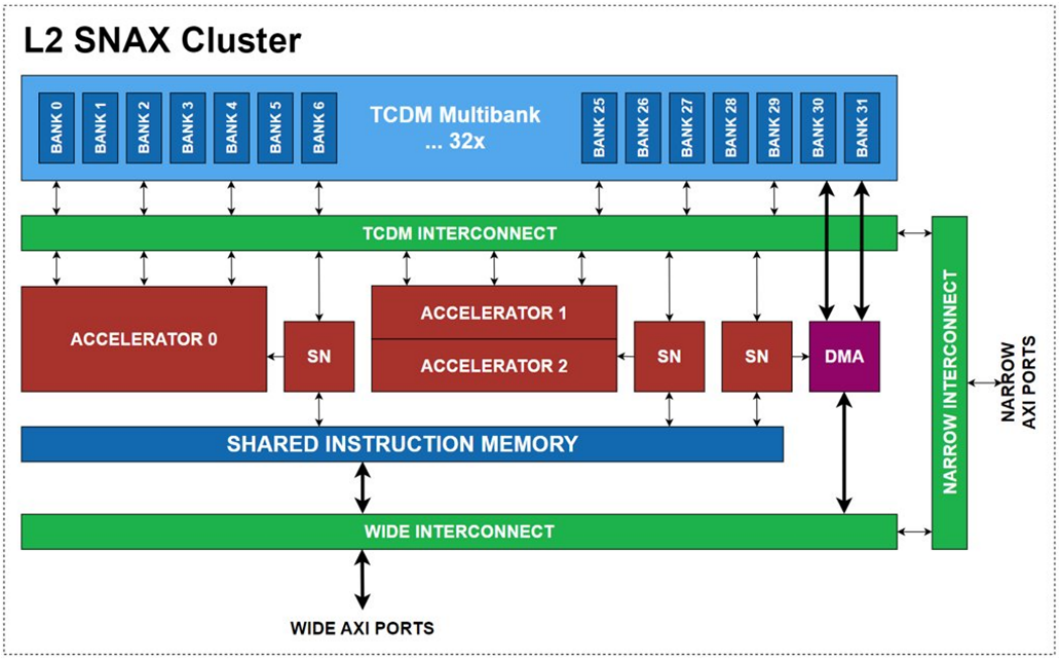


Figure 2.8: SNAX system architecture with multiple accelerators

computational efficiency, achieving a 3.2% area overhead and a 2x increase in energy efficiency.

Adding the CGRA component brings substantial parallel processing capabilities to the system. By configuring the CGRA to handle specific computational tasks, the system can exploit high levels of parallelism, which is essential for accelerating operations such as Fast Fourier Transform (FFT), convolution, and ReLU. The ability to reconfigure the CGRA for different tasks ensures that the system can be optimized for a wide range of AI and DSP workloads. Furthermore, efficient memory management is critical for high-performance computing (HPC) applications. In the SNAX-CGRA system, the CGRA offloads address generation and control logic tasks to external modules like Address Generation Units (AGUs) and Control and Status Registers (CSRs). This offloading frees the CGRA to focus on intensive computational tasks, thereby enhancing overall system performance. Optimizing memory access further enhances performance, with potential improvements of up to 4x compared to existing solutions like STRELA.

The integration of CGRA into SNAX provides a scalable and flexible platform that can adapt to evolving computational needs. The CGRA's ability to be reconfigured at the functional block level allows for continuous optimization and adaptation to new AI models and algorithms, ensuring that the system remains at the cutting edge of performance. Additionally, the combination of Snitch cores and CGRA in the SNAX system results in significant energy efficiency improvements. The architectural optimizations in Snitch cores reduce energy consumption while maintaining high performance, making the SNAX-CGRA system not only powerful but also

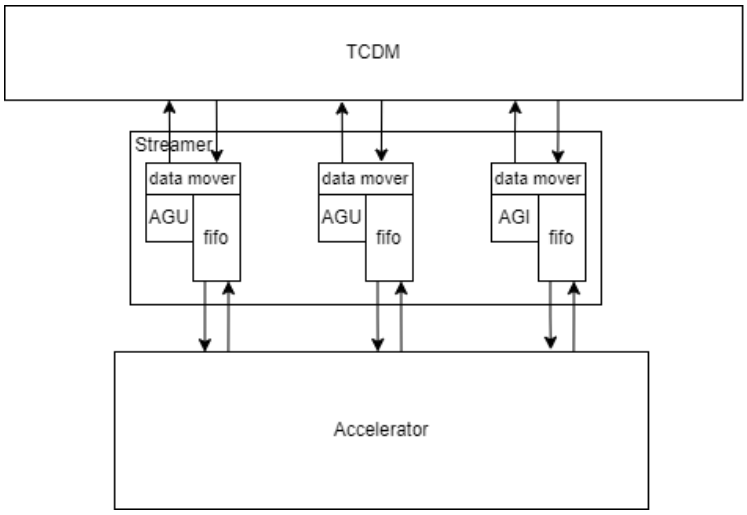


FIGURE 2.9: Streamer architecture of SNAX

energy-efficient, which is essential for large-scale AI and DSP applications.

2.3.2 Structure

The SNAX architecture is show in figure 2.8. The architecture use a modular pattern, different snitch cores can be distinguished all of them sharing the same instruction memory, this is fundamental to maintain coherency inside the control system and ease the programming side.
Following its modular behaviour, accelerators can be placed inside the system, being controlled by a single snitch core. The communication happen through a loose connection, as previously explained. The interface with the data memory (TCDM) is tight and the number of ports can vary depending on the necessity of the accelerator.
The access to the memory can happen in different way, by using or not a streamer. The streamer is placed between the accelerator and the TCDM interconnect. Its role is to hide the latency due to fetching by preloading data into a FIFO and when the accelerator sends a request, the streamer outputs a data, to then refill the FIFO. This is done by implementing inside its structure an address generating unit, that is programmed at compilation time with the set of needed addresses. However, in some cases, it is still possible that, due to parallel requests to the same bank of memory, some latency still may occur, in that case the accelerator need to stall or data inside the banks can be shuffled to remove the conflict.

From these statement it is possible to see how important is to store data in a smart way. Mainly, since AI kernels deal with huge quantities of data, it is fundamental to avoid latencies. This would drastically reduce the time needed, reducing in a drastical way the amount of total cycles. To understand how the memory management works, here is reported an example: Let's say that an accelerator has a total of 3 ports connected to the TCDM. All of them are trying to access the memory on the same

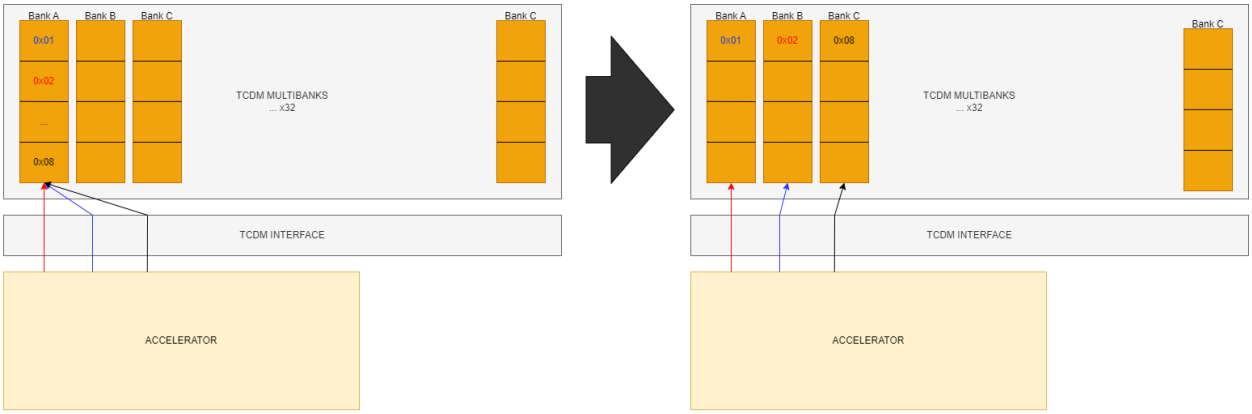


Figure 2.10: TCDM architecture, accessing memory banks. On the left figure the accelerator is accessing the single bank where all data is stored, causing contendecies. Right figure the data is stored in multiple banks, no contendencies are happening.

bank at the same moment, parallel requests. The banks cannot output multiple data at a time thus creating latency for the two other ports that are not granted access immediately. The port that would see the worst latency is the one with the last access granted, in this case a total of 3 cycles.
This would mean that the accelerator has to stall for 3 cycles. Let's take in consideration a kernel with II = 4, and one of the operation is to fetch data from the memory, we would have to stall the accelerator for 3 cycles each loop that is composed by 4, with an increase of II by 3, ending with a II of 7 cycles total, slowing down the process by 42.8%.

2.4 Kernel for benchmarking

2.4.1 Fast Fourier Transform

Compared to the convolution and Relu a deeper explanation is given in order to understand the structure and the data flow of optimized algorithm.
A whole section is given to this topic in order to explain the process and what is happening. This is not done for Convolution and ReLU as they are more straightforward operations, [13] can give more details about the operation and implementation in DNNs.

Introduction

The Fast Fourier Transform (FFT) technique is a fundamental algorithm in both the DSP and AI fields, with enormous implications across a wide range of applications. The FFT is a derivation from the DFT, that exploit redundant calculation of the

DFT, reducing the total computatio to $\frac{N}{2} * log_2(N)$ where N is a power of 2. With such technique we can speed up the process of 205 times in we consider N = 1024, if we increse N to 8192 the speed up is by 1260 [17].
There exists many FFT algorithms, but in this case we will use the radix-2 form, which requires N to be a power of 2. This was first announced by Cooley and Tukey [10] in 1965.
In this section we will give a brief introduction to the DFT to introduce the Cooley Tukey algorithm and it's hardware representation.

From sampling to DFT

When a signal want to be processed in order to get the spectral components, what is being done is to multiply the wanted signal with a train of *Dirac* delta functions. The mathematical representation of this is:

$$x_s(t) = x(t) \cdot \Sigma_{n=-\infty}^{\infty} \delta(t - nT) = x(t) \cdot \delta_p(t) \tag{2.1}$$

After this the Fuorier transform of the signal would be the convolution between the transform of $x(t)$ and $\delta_p(t) = \Sigma_{n=-\infty}^{\infty} \delta(t - nT)$. The single transforms are:

$$\delta_p(t) = \frac{1}{T} \Sigma_{n=-\infty}^{\infty} e^{jnw_0 t} \tag{2.2}$$

Where $w_0 = \frac{2\pi}{T}$ where T is the period of sampling.
Meanwhile for $x(t)$:

$$x_s(t) = X_s(\omega) \tag{2.3}$$

Putting everything together and exploiting the delta function properties we obtain:

$$X_s(\omega) = \frac{1}{T} \Sigma_{n=-\infty}^{\infty} X(\omega - n\omega_0 \tag{2.4}$$

The equation 2.4 can be represented as multiple copies of the spectrum of x(t) for an infinite number of times, representing the Fourier Transfrom.
However in the real world we do not have ifinite computation capabilities and what we are limited to is a *Discrete* version of the transform, so called *Discrete Fourier Transform*. In this version we use a finite number of equally spaced time samples. The final result of the DFT is the finite representation in frequency of the x(t) spectrum that is represented as:

$$X(e^{jwT}) = \Sigma_{n=-\infty}^{\infty} x_p(t) e^{-jwpT} \tag{2.5}$$

Where $x_p(t) = x(pT)$, exploiting the *shifting* property of the δ function to get to the equation 2.5. The differences between the spectrum of a normal Fourier transfor and Discrete one can be seen in figure 2.11. Many aspects can be extended about Fourier Transform but it is not in the scope of the this work and we will go directly to the FFT implementation.

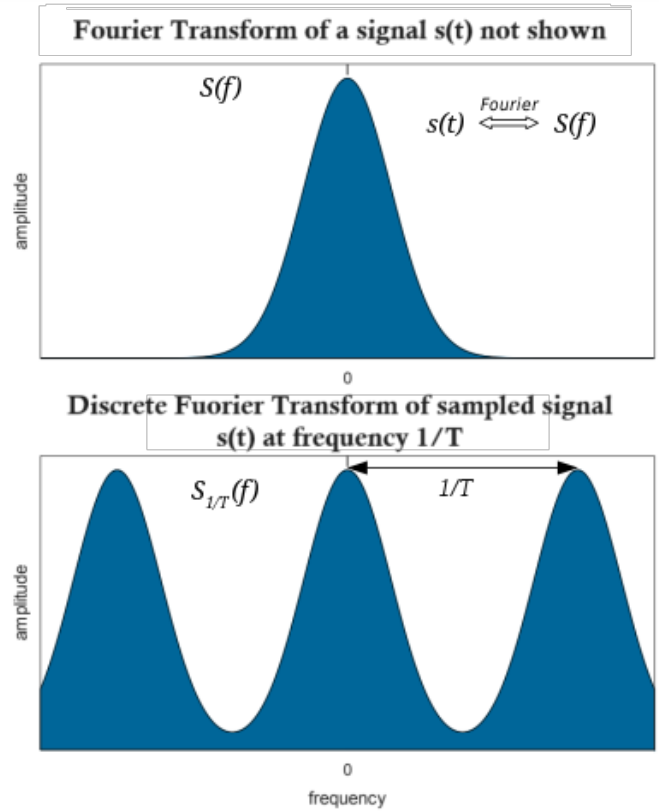


Figure 2.11: Representation of the Fourier Transform (top) and its Discrete representation (bottom).

From FFT to hardware implementation

As stated in the previous section the FFT relies on redundancy of the DFT calculations. From these redundancies a recursive algorithm can be derived, rearranging the problem into two simpler problems, repeatedly. For this reason the algorithm necessitates of a total length of a power of 2 $N = 2^M$. When the problem cannot be simplified more, also called *bottom of the tree*, we have a classical structure named *butterfly* because of its shape, represented in figure 2.12 . Starting from the DFG equation:

$$X_p = \Sigma_{n=0}^{N-1} x_n(t) e^{-j\frac{2\pi}{N}np} \tag{2.6}$$

We can divide it into a summation between two components, corresponding to the odd and for the even n.

$$\begin{aligned} X_p &= \Sigma_{n=0}^{\frac{N}{2}-1} x_{2n} e^{-j\frac{2\pi}{N}2np} + \Sigma_{n=0}^{\frac{N}{2}-1} x_{2n+1} e^{-j\frac{2\pi}{N}(2n+1)p} \\ X_p &= \Sigma_{n=0}^{\frac{N}{2}-1} x_{2n} e^{-j\frac{2\pi}{N/2}np} + e^{-j\frac{2\pi}{N}p} \Sigma_{n=0}^{\frac{N}{2}-1} x_{2n+1} e^{-j\frac{2\pi}{N/2}np} \\ &= A_p + W^p B_p \end{aligned} \tag{2.7}$$

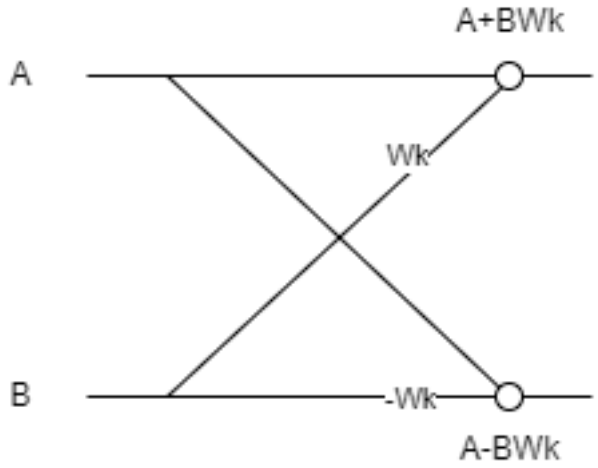


FIGURE 2.12: Butterfly node. Basic block of Cooley Tukey algorithm, performing a radix-2 operation.

Where:

$$\begin{aligned} A_p &= \Sigma_{n=0}^{\frac{N}{2}-1} x_{2n} e^{-j\frac{2\pi}{N/2}np} \\ B_p &= \Sigma_{n=0}^{\frac{N}{2}-1} x_{2n+1} e^{-j\frac{2\pi}{N/2}np} \\ W &= e^{-j\frac{2\pi}{N}p} \end{aligned} \tag{2.8}$$

By looking at the factor A and B it can be seen by the indeces of the summation, that are themselves DFT composed by a total of $N/2$ points, having length as half of the original DFT.
Successively, since we know that DFT is periodic in the frequency domain (of period $N/2$) it is possible to apply further simplifications. Taking equation 2.5 and evaluating it at frequencies $p + N/2$.

$$X_p = \Sigma_{n=0}^{\frac{N}{2}-1} x_{2n} e^{-j\frac{2\pi}{N/2}n(p+N/2)} + e^{-j\frac{2\pi}{N}(p+N/2)} \Sigma_{n=0}^{\frac{N}{2}-1} x_{2n+1} e^{-j\frac{2\pi}{N/2}n(p+N/2)} \tag{2.9}$$

Simplify terms:

$$e^{-j\frac{2\pi}{N/2}n(p+N/2)} = e^{-j\frac{2\pi}{N/2}np}, e^{-j\frac{2\pi}{N}n(p+N/2)} = -e^{-j\frac{2\pi}{N}p} \tag{2.10}$$

Hence,

$$\begin{aligned} &X_{p+N/2} = \Sigma_{n=0}^{\frac{N}{2}-1} x_{2n} e^{-j\frac{2\pi}{N/2}np} - e^{-j\frac{2\pi}{N}p} \Sigma_{n=0}^{\frac{N}{2}-1} x_{2n+1} e^{-j\frac{2\pi}{N/2}np} = \\ &A_p - W^p B_p \end{aligned} \tag{2.11}$$

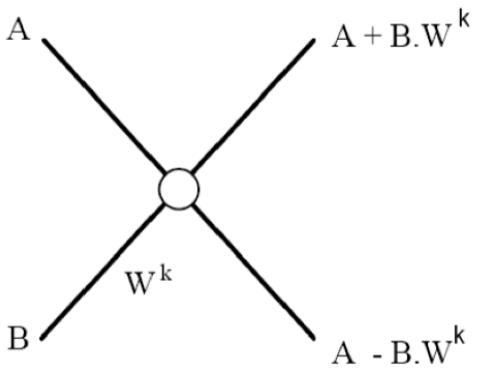


Figure 2.13: Compact representation of a FFT butterfly (radix-2).

With $A_p, W^p and B_p$ as already defined. Comparing the equtaion for $X_{p+N/2} with that for X_p$:

$$X_p = A_p + W^p B_p, X_{p+N/2} = A_p - W^p B_p \tag{2.12}$$

This operations defines the main block (butterfly) structure for the FFT, shown in figure 2.12, where we have X_p on the upper branch and $X_{p+N/2}$ on the lower branch. To get the term $B_p \cdot W^p$ we have to multiply them for the computation of both branch, however we can see how the multiplication needs to be done only once and then we can substract this term to the lower branch.
A compact version of figure 2.12 is depicted in figure 2.13. To give a general understanding of the algorithm for higher order FFT, figure 2.14 shows a 8 point FFT.

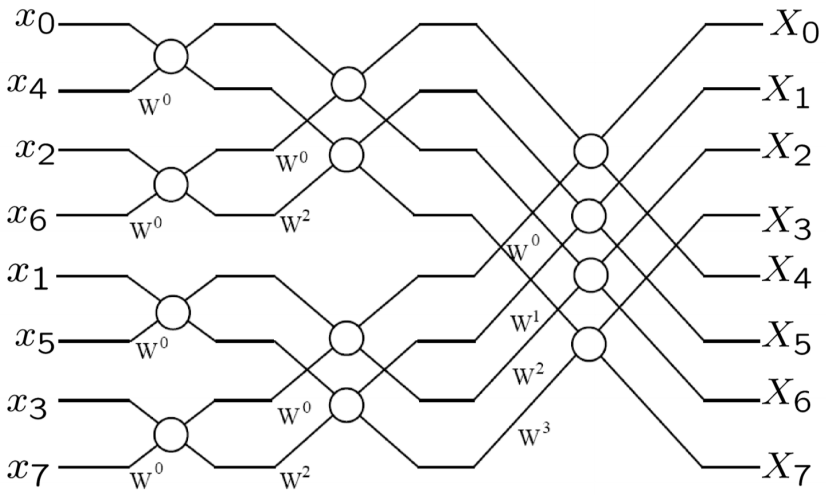


Figure 2.14: 8 point FFT.

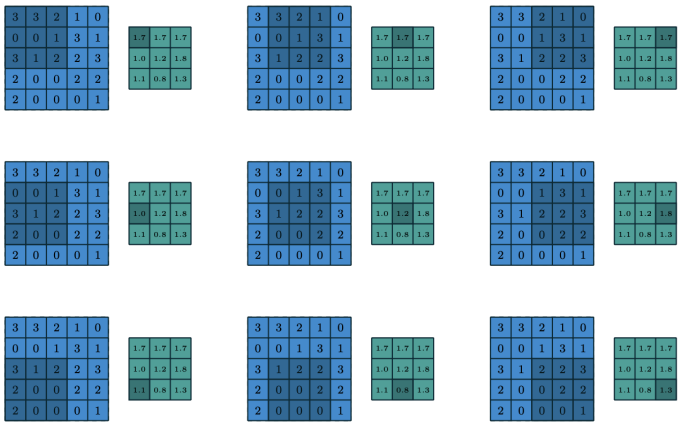


FIGURE 2.15: Image of a convolution adopting a 3x3 weight matrix. Image taken from [12]

2.4.2 Convolution

The convolution kernel is a pillar in the field of AI, serving as a key building block for a variety of applications including image identification, natural language processing, and speech recognition [38]. Convolution is fundamentally the application of a filter or kernel to input data, allowing for the extraction of useful features and patterns. This technique is crucial in allowing machines to perceive, analyze, and respond to complex information, mimicking the cognitive processes of the human brain. Unlike standard linear transformations or mathematical operations, convolutional kernels act on local regions of input data, allowing them to detect subtle patterns and hierarchical structures that would otherwise be concealed. Because of their intrinsic locality and hierarchical processing, convolutional kernels are ideal for tasks like image identification, object detection, and audio processing, which require precise spatial and temporal correlations. The actual name of the implemented kernels that relate to convolutions are: *conv2d*, which representation can be seen in image 2.15, using a 3x3 matrix as weights and *point-wise convolution*.

2.4.3 Rectified Linear Unit

The Rectified Linear Unit (ReLU, figure 2.16) algorithm is a cornerstone in the field of artificial intelligence, particularly in the realm of deep learning. As an activation function, ReLU is essential for introducing non-linearity into neural networks, enabling them to model complex patterns and relationships within data. Despite its fundamental role, ReLU presents unique challenges in terms of hardware acceleration due to its inherent computational pattern. Unlike traditional algorithms that fit well with accelerators designed for MAC operations such as TPUs[21] and specialized accelerators like Eyeriss[8]. ReLU's irregular loop structure does not align well with these conventional accelerators.

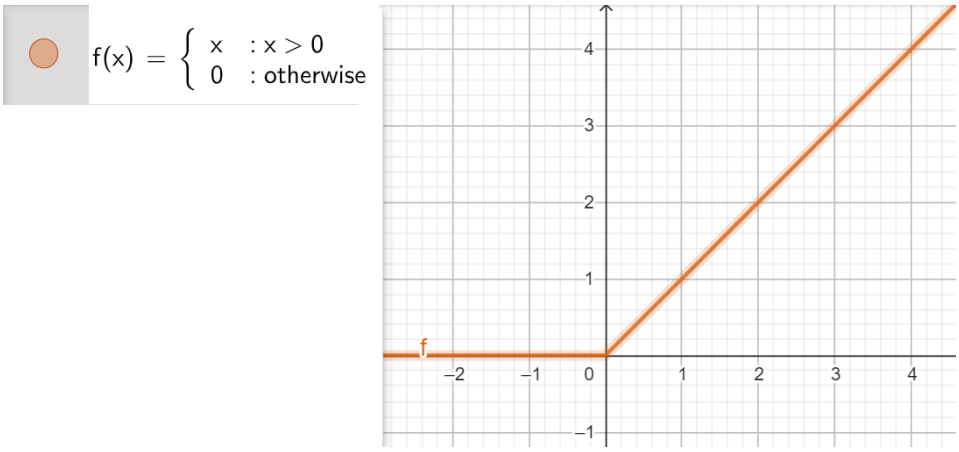


Figure 2.16: ReLU behaviour given a range of inputs.

ReLU's function is straightforward: it outputs the input directly if it is positive, otherwise, it outputs zero. This simplicity belies its computational irregularity, which arises from the need to evaluate each input independently, leading to non-uniform memory access patterns and branching. These characteristics make ReLU less amenable to optimization on hardware accelerators that rely on regular, predictable computation patterns, such as those optimized for dense matrix multiplications.
This limitation highlights a critical need for which the CGRA emerge as a striking solution. In the set of the wide variety of computational patterns that the CGRA can handle, irregular loops is one of those, characteristic of the ReLU algorithm.

Chapter 3

Methodology

In this chapter, we delve into the detailed processes and strategies employed to integrate the SNAX system with a Coarse-Grained Reconfigurable Array (CGRA). The chapter is structured to guide the reader through the key stages of our methodology, highlighting the intricate steps involved in system integration, kernel implementation, and subsequent system optimization.

3.0.1 SNAX-CGRA Integration

The first section of this chapter focuses on the integration of SNAX with CGRA. This process involves detailed planning and execution to ensure that the CGRA's flexibility and performance capabilities are fully utilized within the SNAX system. The integration process is not merely a matter of connecting components but involves significant architectural adjustments and innovations to achieve seamless operation. We discuss the rationale behind the design choices and the specific steps taken to achieve a functional and efficient integration.

3.0.2 Kernel Implementation and Mapping

Following the integration, the next critical phase is the implementation and mapping of kernels onto the CGRA. This section explores the methodologies used to translate high-level algorithmic kernels into low-level instructions that can be executed efficiently by the CGRA. We provide detailed explanations of the mapping process, including the challenges faced and the solutions developed to optimize performance. The implementation covers various kernels, showcasing the versatility and adaptability of the CGRA within the SNAX system.

3.0.3 System Optimization

The final section of this chapter is dedicated to system optimization. Here, we outline the techniques and strategies employed to fine-tune the integrated system for peak performance. Optimization is a crucial step to ensure that the system not only functions correctly but also operates efficiently under various workloads. We discuss

the methods used to minimize latency, maximize throughput, and ensure robust performance across different scenarios. This includes both hardware and software optimizations, providing a comprehensive overview of the steps taken to enhance the system's overall efficiency.

Throughout this chapter, we emphasize the importance of a systematic and thorough approach to integrating advanced reconfigurable architectures with high-performance systems. The methodologies presented serve as a roadmap for similar integrations, offering insights and practical guidance for future research and development in this field.

3.1 SNAX-CGRA integration

To effectively utilize the CGRA accelerator within the SNAX system, it was necessary to make significant modifications to its original structure. In the initial generated architecture, there was an on-chip memory included, which was only needed for testing purposes. However, when integrating the CGRA into the SNAX system, this memory became superfluous. Consequently, all signals originating from the tiles are now redirected to the top module I/O. Directly connecting these signals to the SNAX system is not feasible because the specific protocol for data exchange must be followed to ensure proper communication and functionality. Therefore, to facilitate this integration, an interface is required. This interface is composed of Control and Status Registers (CSRs) and signal management mechanisms, which are essential for managing the data flow and ensuring that the signals conform to the established communication protocol. By implementing this interface, the CGRA can be effectively integrated into the SNAX system, enabling seamless operation and communication between the accelerator and the rest of the system..

3.1.1 Interface

The interface can be divided in multiple sections:

- Wrapper
- Accelerator IO
- CSRs

The Accelerator IO interface behave as a translator for the signals sent by the CPU. Translates the signals coming from the SNAX system into usable data for the accelerator. This is needed because of the nature of the system. The main idea is to have multiple accelerator other than memories, dma core etc... and to communicate with all of them with no issues, offset are included into the address. This offset has to be removed in order to have congruences with the addressing of the CSRs. Furthermore, other than the addressing, the interface decode the instruction of the CPU through the OP_CODE, if it corresponds to CSRW (which is the operation

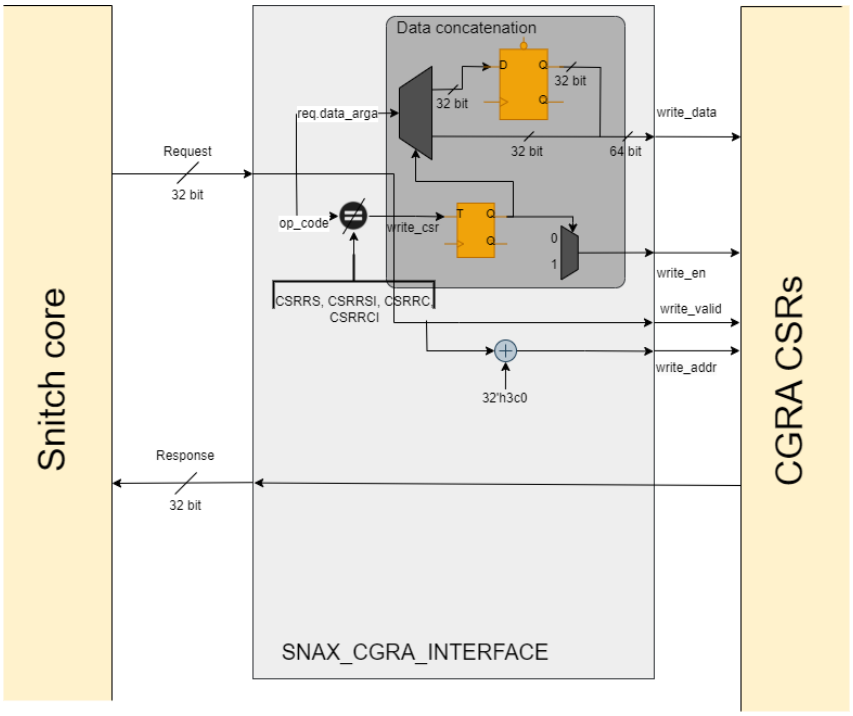


FIGURE 3.1: Snax cgra interface, in dark grey it is shown the concatenation mechanism.

to write inside the CSRs) then it enable the writing for the CSRs. The possible approaches to the CRSs structure are many. A possible implementation for this is could be to instantiate just a single CSR and by using correct addresses load every tile and add to the control memory a signal *enable* that is set to 1 by using a specific address and value. However this method would need to modify the control memory of each tile. To be as generic as possible a non intrusive method was preferred and chosen.

Due to system limitations, the width of the busses are strictly made of 32 bits, this factor is the current bottleneck of CGRA configuration as it has to take 2 cycle per CSR load. This has to be expanded to the total number of control status register and since this number depends on the CGRA and kernel size, if an higher number of tiles is chosen, togther with a higher kernel II, then the configuration overhead would increase.

Because of this limitation the control status register loading takes 2 cycle per register, through a concatenation of data made possible in the interface, this can be seen in figure 3.1. A possible solution for this bottleneck would be to use the TCDM to load the CGRA control status register orchestrating and direct the data through some complex series of interconnection or directly connecting each tile to the TCDM, however this would increase area and shift away from the loose-tight coupling concept on which this system is based.

Sticking to the initial solution the thesis moved towards a system friendly approach of configuring the control status registers, using the limited 32 bitwidth busses.

3.1.2 Control status registers

The adopted CSR structure instantiate 16*4 control status registers. These dimensions depends on the CGRA dimension and kernel's II (i.e. the number of operation executed in loop in each tile). The structure of the CSRs is made in a way it is possible to easily change these parameter by means of parametric values that can be defined when instantiating the module inside the higher module, the CGRA wrapper. In this way it is possible to adapt this structure to every possible CGRA and kernel dimensions.
The control status registers are filled by the data coming from the CPU passing from the interface, after a first elaboration. The data coming from the CPU is being prepared by a C code, translated into assembly instruction, a code example will be given in the testbench section of this chapter. As shown in figure 3.1 the request/response coming from the CPU are actually packages containing multiple information. Here are reported the strcture of the packages:

```
typedef struct packed {
  data_argaddr;
  logic [4:0]  id;
  logic [31:0] data_op;
  data_t       data_arga;
  data_t       data_argb;
  addr_t       data_argc;
} acc_req_t;

typedef struct packed {
  logic [4:0] id;
  logic       error;
  data_t      data;
} acc_resp_t;
```

It can be seen by the names that acc_req_t is the request packet, sent by the CPU (in this case) and the response is the message sent by the control status registers. In general the request is sent by the master and the response is sent by the slave, that is who is receiving the message.
In this structure the data_op corresponds to the instruction done by the CPU, data_arga is the information of the message, the configurations of the tiles, data_argb is the address, which selects the corresponding csr and data_argc is not being used. For the response side, the id is the same as the received one, error is for error detection (feature that is disabled) and data is the received data.

The structure of the CSRs can be views as sets of FIFOs with depth equal to the kernel size and the number equal to the total dimension of the CGRA. In this way it is possible to configure the CGRA in a sparse or ordinate way. In this module, the request/response protocol is followed and implemented.

3.1.3 CGRA wrapper

The wrapper is what bond every main module together: *interface*, *control status register* and *CGRA Rtl*. Interconnections are crucial, but other than that the role of the wrapper is to make communication between internal and external modules efficient and possible. The communication side with the Snitch core has already been taken care of by the snax cgra interface, what is now needed is to operate on the TCDM side.
The packages from the TCDM are different from the snitch core, and even more the protocol differences of the CGRA and the TCDM. To merge these packages a tight connection inside the wrapper was made, creating ad-hoc logic to control and connect these signal. If no other accelerator is present in the system, latency equal to zero is granted as the only access to the memory is done by the CGRA having always granted access. If this is not true and the CGRA is not the only accelerator in the system, stalls may happen. To handle these possible conditions the CGRA has to stall for an arbitrary amount of cycles, until the memory access is granted again. To do so an enable signal was implemented inside the whole snax cgra architecture that could pause all the on going operations and recover when the data is available. The final result was a functioning wrapper including all the modules, inside the SNAX system.
To make the CGRA available to the system, the software side had to be developed together with the hardware side.

3.1.4 Software

In order to control the CGRA, a software was developed. The software included the C code of the implemented kernel, together with ad-hoc functions that could communicate and configure the CGRA. Starting from the convolution, the C code includes the convolution computation, together wit the loading of the TCDM and control status register for the CGRA.
To configure the CGRA we need to configure a total number of control status register equal to $\#Tiles \cdot II$. Doing this by hand each time would result in a cumbersome work and to ease the task some C and Python functions were developed. The main idea is to configure the CGRA through a json file, that consists in a dictionary of parameters for each tile that has to be configured.
Each tile has different degree of freedom such as the *x, y, op_code, output ports configurations, input configurations, predicate, predicate_in* and the *cycle number*.
The x and y are the coordinate of the tile within the CGRA architecture, x and y equal to 0 is the bottom left tile, x increases as we go right and y as we go up. the OP CODE consists in a table of possible operation (a full understanding can be achieved by looking at the opt_type.py file), just to name a few: ADD, MUL, STR, LD. The input output configuration consists in rows of *out_x : "y"* where x represent the output port and y is the input port. This is equal to assign port output x to input port y. X and Y have a limited range that, depending on the type of interconnect configuration, can go from 0 to 3 (if a mesh is adopted) or 0 to 7 (for a

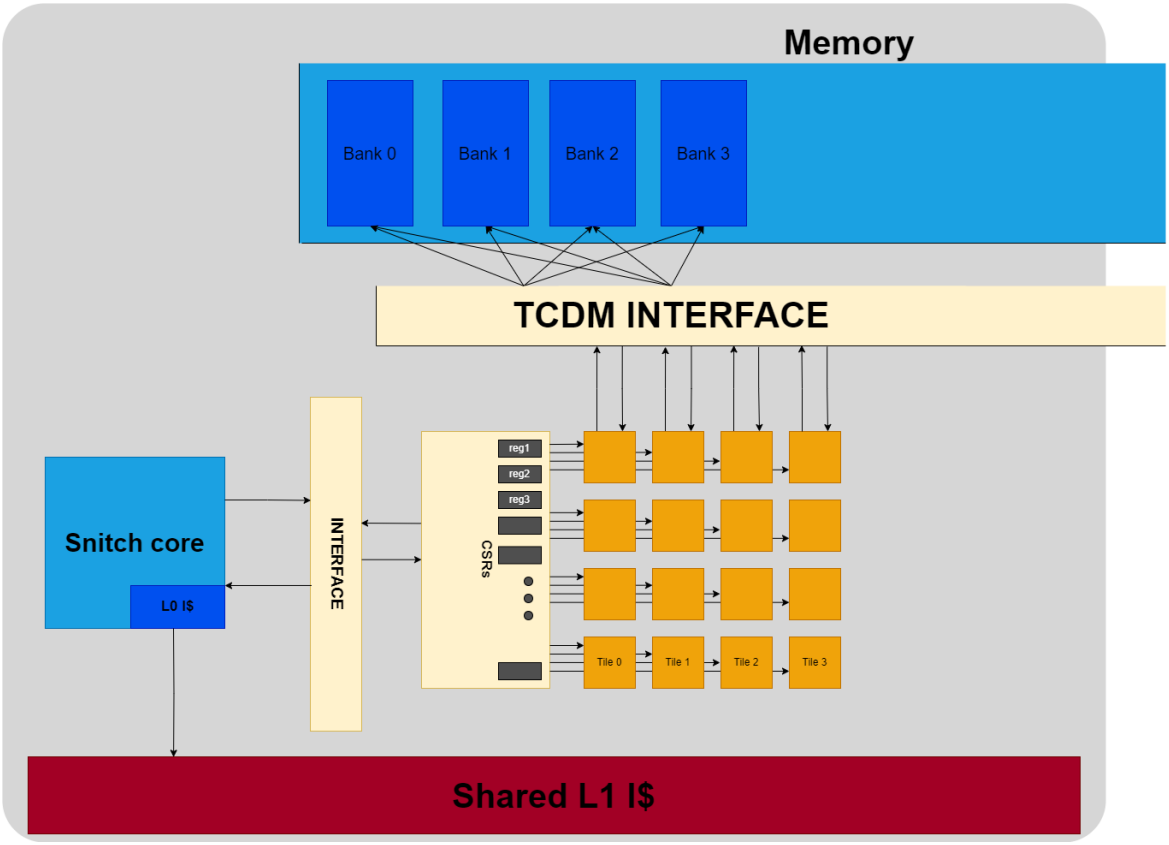


Figure 3.2: SNAX CGRA Implementation, minimized interconnection to give a better view. Loads and stores happen only on the top row of the CGRA.

king mesh).

In this implemented CGRA, a mesh configuration is considered.

To assign the incoming value to the input of the functional unit of the tile we just need to assign the input port from which the data is coming to the wanted input of the functional unit, that corresponds to "output_4 to output_7". Figure 3.3 shows how the directions are assigned to a certain number. The predicate is the conditional state of the tile that, together with the predicate_in indicate what action needs to be taken. This often comes with a PHI opt, that based on the value that is stored inside the predicate_in, will behave in a selective way. It is an if condition.

The json file is taken by a custom python function, that reads and create a **Tile** object with the correct coordinates and cycle, with all the needed configuration. This is passed to method that write all the needed information in data.h file.

For the convolution the data.h contains the *tile_configs* which is a matrix geometrically equal to the CGRA, the input A and B, and the size of the convolution kernel (height and width of the filter). The data.h parameters are then used by some C functions that load the control status registers of the CGRA.

To have a deep understanding everything can be found on the snitch cluster repository.

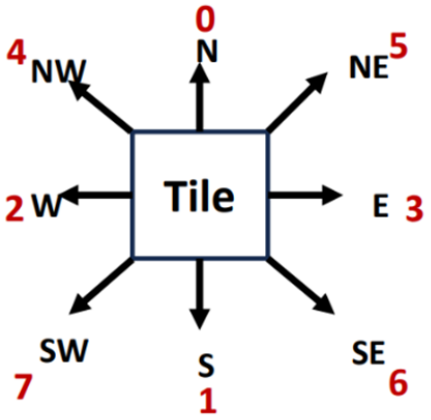


FIGURE 3.3: Tile direction with number assignments used in the CGRA configuration file.

3.2 Kernel implementation and mapping

The OpenCGRA framework was an essential starting point in the approach to developing a functional CGRA system. OpenCGRA, offers crucial tools and insights for navigating the complexity of CGRA design and implementation. Despite its potential, OpenCGRA has difficulties in achieving accurate and efficient mapping, posing substantial obstacles in the development process. One of the key challenges with OpenCGRA was the generation of accurate mapping. Mapping in the context of CGRA designs refers to the act of assigning computational jobs to the CGRA's array of processing elements. However, achieving accurate mapping within the OpenCGRA framework proved to be a difficult endeavor, laden with complexities and nuances.
Despite these challenges, the engineering efforts put into OpenCGRA were critical in increasing the understanding of CGRA structures and their practical use. The iterative process of debugging and optimization provided crucial insights into the intricate details of CGRA design while also laying the framework for future advancements in the discipline.
While OpenCGRA struggled with hardware mapping, its importance in the development of CGRA systems cannot be emphasized. OpenCGRA accelerated CGRA engineering achievements by providing a platform for exploration, experimentation, and cooperation.

3.2.1 Convolution

To start the engineering part and the exploration of the OpenCGRA environment, this work started developing a CGRA kernel and mapping of 2 initial kernel: the Finite Input Response filter (FIR) and a convolutional kernel. As the FIR was just a starting point to deepen the understanding of the framework, in more interest there is the convolution kernel.

C code

The starting point of the framework is making the C Code that represent the kernel behavior. Attention must be taken when dealing with this part. The LLVM translation is not so straightforward because from the C code to the lower level, much more operations need to be taken in order to compute the same function. Because of this, if optimization in the kernel II and optimal operation type want to be reached, it is important how the user structures its code, dealing with simple arithmetic flow like instead of a conditional and nested structure. Furthermore as previously mentioned, the framework itself could only work with the most inner loop of the kernel. Since most kernels operate on nested-loops, loop unrolling was necessary. This means changing the code structure in a way to obtain a single for loop that iterates a total number of time equal to the original one, through some arithmetic statement we can handle the if conditions and operations that are hidden in a for loop. For the convolution it develops like this.
Before unrolling.

```
    for ( int i = 0; i < I_CONST; i++){
        for ( int j = 0; j < J_CONST; j++){
        .
        .
```

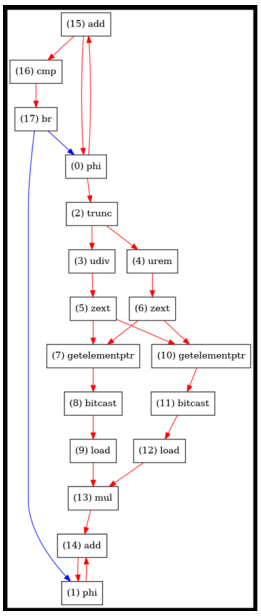
After unrolling

```
    for (x = 0; x < total; x++) {
        i = x / NJ;
        j = x % NJ;
        .
        .
```

This would mean that even the division and modulo operation are computed on the CGRA which are computations that are super expensive hardware wise, since calculator are not optimized to do such operations. Both modulo and division operations involve more complex algorithms compared to basic arithmetic operations like addition, subtraction, and multiplication. These algorithms require multiple steps to compute the result, involving iterative processes like long division or more sophisticated techniques like Newton-Raphson iteration. Due to their complexity, they typically take multiple clock cycles to complete. This is in contrast to the aforementioned operations, which can often be performed in a single clock cycle or a small number of cycles. The ideal is to take this operation and move them outside the CGRA, replacing them with alternatives that later will be explained
The idea with what this work was developed is to only focus on fundamental operations of omplete processes, i.e. in this case the sum and multiplication between the weights and the incoming data. The managing of the whole input matrices are assume to be done externally.
The final implemented kernel resulted in:

```
int kernel(int A[NI][NJ], int B[NI][NJ])
{
  int x = 0, i = 0, j = 0, k = 0;

```



(A) Convolution kernel wrong DFG

```
define dso_local void @kernel([10 x float]* nocapture readonly %0, [10 x float]*
{
  br label %3

3:                                                ; preds = %3, %2
  %4 = phi i64 [ 0, %2 ], [ %19, %3 ]
  %5 = phi <4 x float> [ zeroinitializer, %2 ], [ %18, %3 ]
  %6 = trunc i64 %4 to i8
  %7 = udiv i8 %6, 10
  %8 = zext i8 %7 to i64
  %9 = urem i8 %6, 10
  %10 = zext i8 %9 to i64
  %11 = getelementptr inbounds [10 x float], [10 x float]* %0, i64 %8, i64 %10
  %12 = bitcast float* %11 to <4 x float>*
  %13 = load <4 x float>, <4 x float>* %12, align 4, !tbaa !2
  %14 = getelementptr inbounds [10 x float], [10 x float]* %1, i64 %8, i64 %10
  %15 = bitcast float* %14 to <4 x float>*
  %16 = load <4 x float>, <4 x float>* %15, align 4, !tbaa !2
  %17 = fmul <4 x float> %13, %16
  %18 = fadd <4 x float> %5, %17
  %19 = add i64 %4, 4
  %20 = icmp eq i64 %19, 20
  br i1 %20, label %21, label %3, !llvm.loop !6

21:                                               ; preds = %3
  %22 = call float @llvm.vector.reduce.fadd.v4f32(float -0.000000e+00, <4 x floa
  store float %22, float* getelementptr inbounds ([10 x float], [10 x float]* @0
  ret void
}
```

(B) Convolution kernel wrong LLVM IR

FIGURE 3.4: Convolution DFG & LLVM IR initial generation. Here can be seen different type of unwanted operation, such as extension, truncation, bitcast.

```
5     int total = NI * NJ;
6     int out = 0;
7     for (x = 0; x < total; x++) {
8       i = x / NJ;
9       j = (x % NJ);
10
11      out += A[i][j] * B[i][j];
12    }
13
14    return out;
15  }
```

The next step is to generate a binary execution file from the C code, this will be used for simulation and verification. This translation step is incomprehensible for human, so to understand what is actually being translated, an LLVM file is generated where it can be seen all the LLVM IR operation.
Initially the mapper translated this code into a "wrong" set of instructions, this can be seen in figure 3.4. In the translations there are useless operations for the sake of the convolution, that are: *trunc*, *zext*, *bitcast*. Hardware wise, internally to the accelerator, all the wires/signals will have constant bit width making these operation meaningless. Furthermore, not only these operation are suprfluous but these operation are not supported by OpenCGRA, although the mapper manages to map them, the VectorCGRA does not include them. To overcome this problem it is needed to modify the LLVM IR generated file into a correct set of instruction and

then to translate it to binary executable file.
A new version of the LLVM code was hand made and is reported down below. This final version consists in removing the aforementioned operation to obtain a correct mappable algorithm. It can be seen how the bitcast, trunc and extension operations

```
define dso_local void @kernel([10 x float]* nocapture readonly %0, [10 x float]* nocapture readonly %1)
local_unnamed_addr #0 {
  br label %3

3:                                                ; preds = %2, %3
  %4 = phi float [ 0.000000e+00, %2 ], [ %13, %3 ]
  %5 = phi i64 [ 0, %2 ], [ %14, %3 ]
  %6 = udiv i64 %5, 10
  %7 = urem i64 %5, 10
  %8 = getelementptr inbounds [10 x float], [10 x float]* %0, i64 %6, i64 %7
  %9 = load float, float* %8, align 4, !tbaa !2
  %10 = getelementptr inbounds [10 x float], [10 x float]* %1, i64 %6, i64 %7
  %11 = load float, float* %10, align 4, !tbaa !2
  %12 = fmul float %9, %11
  %13 = fadd float %4, %12
  %14 = add nuw nsw i64 %5, 1
  %15 = icmp eq i64 %14, 20
  br i1 %15, label %16, label %3, !llvm.loop !6

16:                                               ; preds = %3
  store float %13, float* getelementptr inbounds ([10 x float], [10 x float]* @output, i64 0, i64 0), align 16, !
tbaa !2
  ret void
}
```

Figure 3.5: Conv LLVM IR correct code.

are not included anymore, keeping an hardware wise correct flow.

Convolution CGRA map

The openCGRA mapper tries to map the convolution kernel, but unsuccessfully. To understand what went wrong a deep study of the CGRA architecture was done. By deafult each PEs compute its asserted operation by using the first 2 inputs. This was the first issue, the mapper after mapping a result to e.g. input 3/4 of a certain tile, it does not actually tell the same tile to use these inputs. So every input-output match was checked and for each FU set the correct input to use for its operations. This part took a lot of time as at the begin it was uncertain the actual functioning of the framework and every possible condition had to be check, having chats with the author of the works.
Furthermore mapping conflicts could happened. By default if a result or passage of data has to happen between e.g. first row (top one) and second row (the row below) the mapper tries to map it directly by making the data flow first in a downwards way then left or right.
If there are more data that need to have the same path during the same clock cycles, the mapper just refuses to map one of the two. Because of this an entire remapping of the algorithm was carried out.
The new mapping does not have stalls or latency due to dependencies.

Introducing new hardware

The new map required some operations that were not initially supported by the VectorCGRA. These were included in the code by adding:

- New processing elements
- The op-code for each operation
- The interpretation of the op-cope by the mapper

The operations in interest are: *reminder*, *division* and the constant equivalent of these operations, where the FU has as inputs: the data and the constants with which it's doing the operations. This constant is chosen during compilation time and hardwired inside the tile.

3.2.2 Fast Fourier Transform

The second algorithm which was carried out during this is a kernel which is fundamental in DSP and AI domains. Because of this it is shown that CGRAs are not only capable of accelerating AI related kernels, but everyone that has a DFG structure like.

Introduction

The Fast Fourier Transform (FFT) technique is a fundamental algorithm in both the DSP and AI fields, with enormous implications across a wide range of applications. However, incorporating the FFT algorithm into a CGRA design poses distinct obstacles and opportunities. The CGRA's parallel processing capabilities promise to accelerate FFT computations by allowing FFT stages to be executed concurrently. However, achieving this potential, especially considering both addressing generation and data computing performed by the CGRA, necessitates careful consideration of the algorithm's data dependencies, memory access patterns, and computational effort distribution. Mapping the FFT to a CGRA architecture has the opportunity to maximize parallelism while minimizing resource contention and delay. Thanks to the nature of its architecture it is possible to map multiple operations spread across its processing elements, this allow to perform multiple operation in a single cycle. Spatially unrolling the FFT is possible and enanches the overall performances.
However, one of the most difficult aspects of transferring the FFT to a CGRA architecture is controlling memory storage and loading. The repetitive nature of the FFT technique involves frequent access to input data and intermediate outcomes, which places large demands on memory bandwidth and capacity. Efficient memory management algorithms, such as data caching, prefetching, and buffer reordering, are critical for reducing memory access latency and increasing computing performance. That's what SNAX system provides. the SNAX system provides a versatile framework for handling high data dependencies and coordinating complicated computational operations. Furthermore, obtaining peak performance may need offloading specific

processes from the CGRA to other modules or accelerators, as this will be explored in the SNAX domain and later reported.

C Code

To properly transfer onto the CGRA architecture the C code for the FFT, it must follow the OpenCGRA framework's limitations. One limitation is the need for a single for loop to provide smooth execution and fast kernel generation within CGRA tiles. Furthermore, the code should reduce exceptions such as if and nested if statements in order to maintain a continuous flow of processing. However, altering the typical FFT method to fit these limits is a challenge. The FFT's natural structure sometimes includes nested loops, which might complicate the process of condensing the calculation into a single loop. Inspired by previous research, notably the work of Riedel et al. [28] and following the [10] algorithm, it is clear that existing FFT implementations may not easily comply to the requirements of the OpenCGRA framework. These implementations frequently have disconnected loops and layered control structures, which limit their applicability for CGRA mapping.

In response, work was done to create a specialized N-point FFT algorithm adapted to the needs of CGRA designs. This required rethinking the traditional FFT technique and developing strategies to obtain a single loop formulation while keeping computing efficiency.

```
    int16_t total = FFTpoint*log2(FFTpoint)/2;
    for (int x = 0; x < total; x++) {
        nlayer = x / cmpVal;
        outerLoop = x % cmpVal;
        k = FFTpoint / divFact;
        cnt = x % k;
        i = cnt + outerLoop / k * n1 * 2;
        l = i + n1;

        upPointer = i + 8 * nlayer;
        downPointer = (i + n1) + 8 * nlayer;

        // upper branch
        tempReal_i = prev_layerReal[i] + cos[upPointer] *
        prev_layerReal[l] + sin[upPointer] * prev_layerImg[l];

        tempImg_i = prev_layerImg[i] + cos[upPointer] *
        prev_layerImg[l] - sin[upPointer] * prev_layerReal[l];

        // lower branch
        tempReal_l = prev_layerReal[i] + cos[downPointer] *
        prev_layerReal[l] + sin[downPointer] * prev_layerImg[l];

        tempImg_l = prev_layerImg[i] + cos[downPointer] *
        prev_layerImg[l] -sin[downPointer] * prev_layerReal[l];

        // Update original arrays
        prev_layerReal[i] = tempReal_i;
        prev_layerImg[i] = tempImg_i;
```

```
30          prev_layerReal[l] = tempReal_l;
31          prev_layerImg[l] = tempImg_l;
32
33          // Use arithmetic to handle the change in divFact and n1
34          divFact -= (divFact >> 1) * (outerLoop / (cmpVal - 1));
35          n1 += ((1  << nlayer) ) * (outerLoop / (cmpVal - 1));
36      }
```

One of the main goals of the proprietary N-point FFT method created for CGRA mapping was to optimize the computation flow in order to make hardware deployment and translation to LLVM easier. The algorithm cyclically coordinates the computation of the upper and lower branches of the Radix-N, to maximize resource efficiency and minimize control overhead. Every control feature of the algorithm is updated using simple arithmetic operations to improve hardware deployment and favourite the translation process. The adopted approach with the coding side simplifies the mapping to LLVM instructions by following elementary arithmetic operations, which paves the way for hardware implementation that runs smoothly in CGRA architectures. It is noted that the code shown here is a simplified representation for clarity and is not the final version. In the final implementation, real and imaginary components are combined into a single array per layer. The imaginary portion is kept in the same array, offset by the amount of FFT points. To reduce computational overhead, it is also assumed that pre-computed values for the sine and cosine functions are kept in memory.
In order to reduce redundancy and enhance memory accessibility, pointers to memory locations are calculated only once and then transferred to subsequent variables.

Translation to openCGRA

Following the development and testing of the N-point FFT method in C, the translation to LLVM was performed, producing a simple LLVM codebase of 75 lines. The LLVM code's relative compactness reflects the efficiency of the underlying C code. Each operation in the C code is fundamental in nature, leaving limited room for additional optimization without sacrificing computational correctness or usefulness.The LLVM code's simplified design permitted kernel extrapolation and subsequent hardware mapping, opening the path for easy inclusion into CGRA architectures. Figure 3.6 shows that the LLVM code generates a DFG with 64 nodes. This full representation captures the whole processing pipeline of the N-point FFT algorithm, including all computational phases and memory addressing complexities associated with the CGRA computation job, ending with a final kernel **II** equal to **5**.

3.3 System optimization

Once the basis are set, the SNAX exploration was carried on focusing the convolution and FFT kernel. The fundamental characteristic of the SNAX system from the accelerator point of view is to have virtually no latency on the access to the memory. This is possible thanks to the memory structure. Being able to have multiple ports

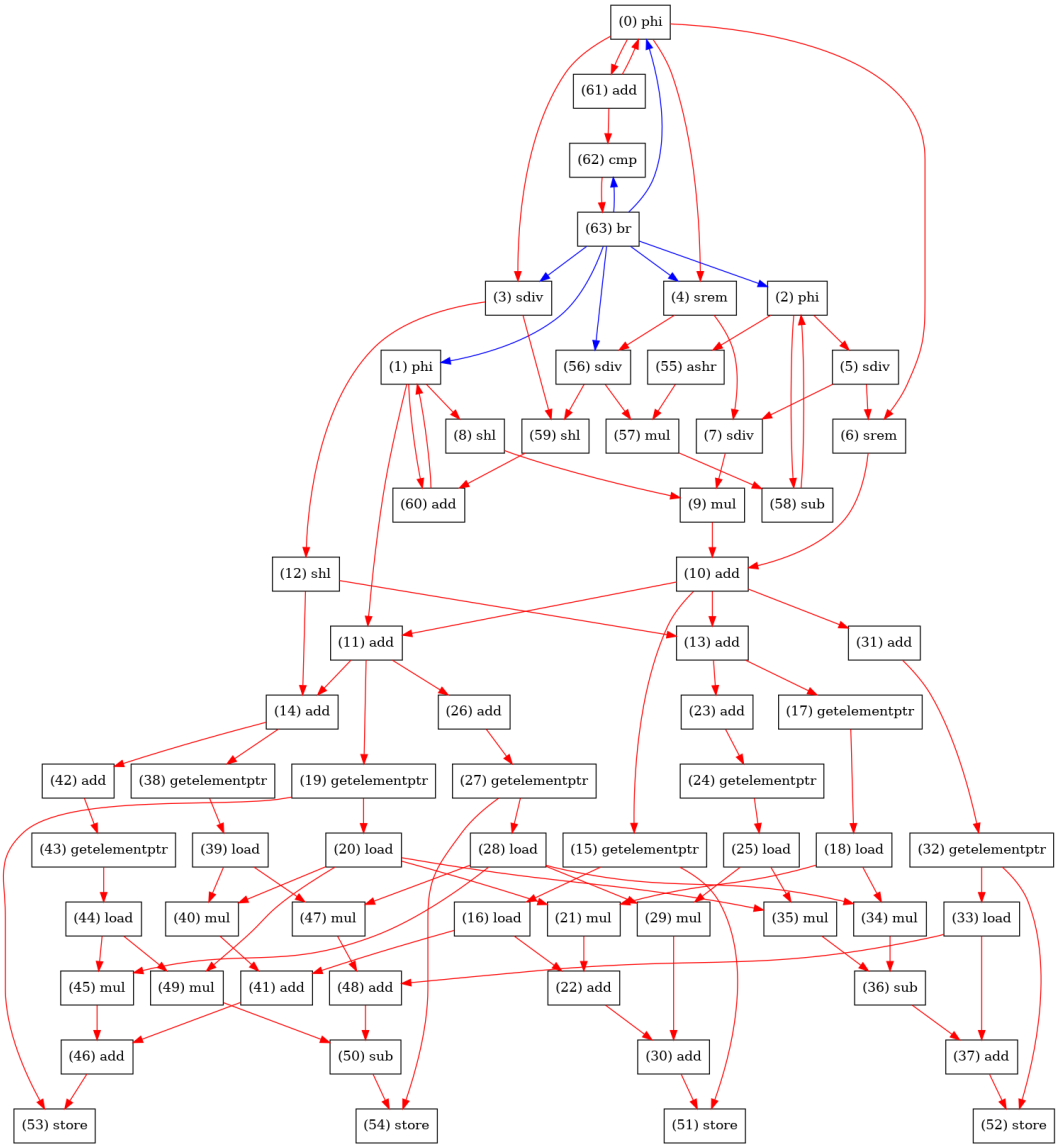


Figure 3.6: FFT DFG.

each one connected to single memory banks gives absolute freedom on how the accelerator has access to the data.
Not only each port has access to each memory bank, but once the port from the accelerator to the TCDM interface is instatiated, it is not fixed to only one memory bank, but it can have access to all the banks, individually, by means of an address offset.
When dealing with AI kernels this is of fundamental importance, since the kernels are heavily data dependent, spatially and temporally.
A suitable example is the Fast Fuorier Transform algorithm.

Until now in this thesis work and in the works of [32] all the operation (including addressing generation) were done internally by the CGRA, this is a waste of hardware resources since when loading the limited hardware with more computation will lead to bigger kernel sizes (II) and thus less total computing efficiency. To ease the workload of the CGRA and focus mainly on the bare computations of the kernels, the addressing generation was offloaded from the CGRA and give this duty to an external module: the AGU (Address generator unit). This aims to reduce kernel sizes starting from the off load of operation from the CGRA and taking in consideration the data dependencies pushing towards a null latency due to loads.

Although the addressing generation and control logic has been removed from the CGRA, these tasks are need to be managed by some other modules. The addresses are managed by the AGU meanwhile the control logic i.e. how many times the current kernel has to be repeated, is taken care by additional logic inside the CSR. The additional logic is just an additional counter that counts the kernel iteration and states when a certain value is reached to then stop the accelerator and the AGUs. This parameter in this way can be easily reconfigured through the Snitch core via the macro *write_csr* and giving it a unique address.

3.3.1 AGU

The address generator unit is the module which duty is to take some of the CGRA workload. It's tasks are strictly related to generating addresses for the memory.
The structure of the AGU is composed by a combinational data path, controlled by an FSM that, depending on the adopted kernel, compute the addresses and the memory access pattern. The relation between the DFG nodes disposition on the CGRA and the address generaton is very tight since the CGRA does not have a continuous homogeneous memory access pattern. In the FFT case study it will be striking on how the memory access is computing intensive and needs to be taken care of.
The AGU has itself some control status registers that can be configured at compilation time with the addresses of the stored data. The number is equal to the maximum bank needed for the implemented kernels, in this case the number corresponds to 5 i.e. for the FFT.

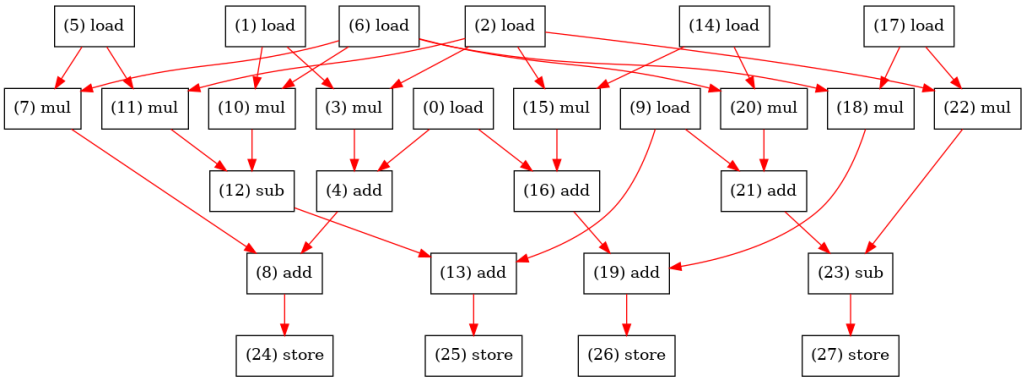


Figure 3.7: FFT offloaded, DFG with no address generation.

3.3.2 Fast Fuorier Transform kernel: OFF-loading address generation and control logic

To successfully map on the CGRA a kernel that does not contain addressing generation, taking the code in the FFT section of this chapter, we move all the directly computation of the FFT inside a function while keeping the addressing generation outside. This will translate in a portion of the LLVM code that only refers to the FFT computation without considering the address generation. However, we still need to tell the CGRA that loads and store needs to happen and to do so we give this function vectors as inputs, but when calling them, we give a constant value for its pointer. The resulting C code will be:

```
1
2  void kernel(int64_t* prev_layerReal,int64_t* prev_layerImg,int64_t*
       prev_layerReal1, int64_t* prev_layerImg1, int64_t* cos, int64_t*
       sin,int64_t* cos1, int64_t* sin1){
3
4          int64_t tempReal_i, tempImg_i,tempReal_l,tempImg_l;
5
6          tempReal_i = prev_layerReal[0] + cos[0] * prev_layerReal1[0] +
        sin[0] * prev_layerImg1[0];
7          tempImg_i = prev_layerImg[0] + cos[0] * prev_layerImg1[0] -
       sin[0] * prev_layerReal1[0];
8
9          // lower branch
10         tempReal_l = prev_layerReal[0] + cos1[0] * prev_layerReal1[0]
       + sin1[0] * prev_layerImg1[0];
11         tempImg_l = prev_layerImg[0] + cos1[0] * prev_layerImg1[0] -
       sin1[0] * prev_layerReal1[0];
12
13         // Update original arrays
14         prev_layerReal[0] = tempReal_i;
15         prev_layerImg[0] = tempImg_i;
16         prev_layerReal1[0] = tempReal_l;
17         prev_layerImg1[0] = tempImg_l;
18 }
```

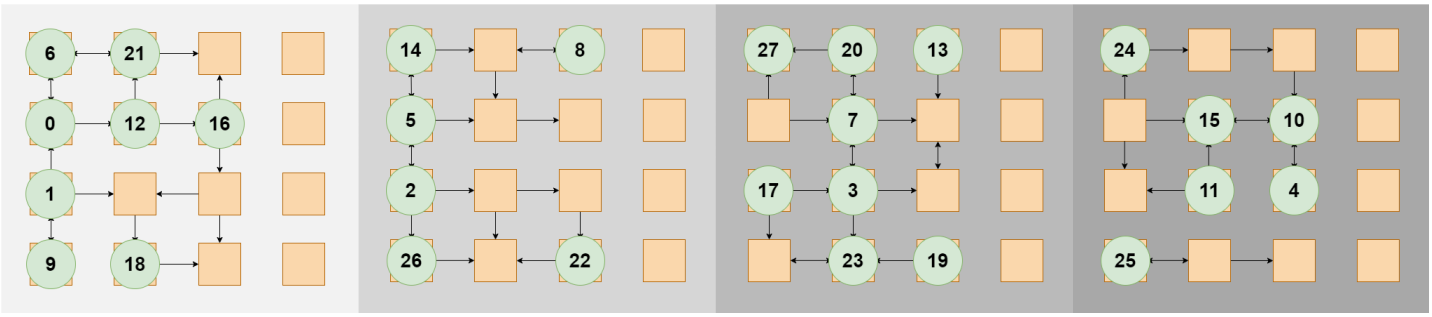


FIGURE 3.8: FFT Map on CGRA, kernel II = 4. In interests are the loads and stores. (0) load real[i], (1) load cos [upPointer], (2) load real[l], (5) load sin[upPointer], (6) load img[l], (9) load Img[i], (14) load cos[downPointer], (17) load sin[downPointer].

```
19
20
21 int main(){
22 .
23 .
24 .
25     for (int64_t x = 0; x < total; x++) {
26     nlayer = x / cmpVal;
27     outerLoop = x % cmpVal;
28     k = FFTpoint / divFact;
29     cnt = x % k;
30     i = cnt + outerLoop / k * n1 * 2;
31     l = i + n1;
32     upPointer = i + 8 * nlayer;
33     downPointer = (i + n1) + 8 * nlayer;
34     //calling kernel function
35     kernel(prev_layerReal,prev_layerImg,prev_layerReal1,prev_layerImg1
       ,cos,sin,cos1,sin1);
36
37     // Use arithmetic to handle the change in divFact and n1
38     divFact -= (divFact >> 1) * (outerLoop / (cmpVal - 1));
39     n1 += ((1  << nlayer) ) * (outerLoop / (cmpVal - 1));
40     }
41 }
```

Due to LLVM optimization it is needed to pass the function different vectors to keep all the needed loads and not merge some of them. Using the same array with different memory access will result in having *getelementptr* operations with the LLVM, which is something that we do not want since will result in an redundant ADD operation in the CGRA. The resulting DFG have a total of 27 nodes which is 3 times less then the containing the address generation. The kernel II is now equal to 4 instead of 5 gaining a 20% in performances.

The FFT is an algorithm heavily dependent on the data disposition spatially and temporally. Since the first iteration, the samples have to be placed in a specific manner inside the memory to then again, on each layer, store them in a very precise

way, as it was shown in the background section.
The loading and storing are not really taken in consideration within the II of the kernel, this means that from an high level point of view can happen at the same time. For example, by looking at the FFT code presented in section 3.2.2, when computing tempReal_i we need to have loaded prev_layerReal[i] and prev_layerReal[l] that are 2 memory address that belongs to the same array. Hardware wise this could be represented by storing the array inside a single memory bank, however multiple accesses to the same memory bank are not possible.
This impose a limitation on the memory bandwith that needs to be taken care of.
By looking at the FFT kernel mapping on the CGRA, it can be seen on how on the first cycle of the kernel (left most square, lightest grey) we have 4 parallel loads. Withing these 4 loads we have a memory bank contention that are DFG node 6 and 9, img[l] and img[i] respectively. This will pose a stall during the kernel, increasing the length of the kernel II from 4 to 5. However, thanks to the SNAX flexibility, it is possible to include a trick with memory accesses.
Instead of storing the array into a single bank of memory, we can split the array into 2 sub-arrays and store them in separate banks, paying attention to the original indeces of the array.
Although, the spatial and temporal data dependencies need to be consider to compute a correct FFT, this means that the loads, from the accelerator side, need to jump between these 2 memory banks in a different way each kernel repetition. In figure 3.9 it is depicted how this is happening.
At the center of the figure (1st Load) the index, so called, **I**, that is being generated by the AGU, only access the memory bank 0. In the second kernel repetition, it access twice the first bank and twice the second bank, accessing memory address 0, 1 and 4, 5. As the kernel goes on the number of times that the index I (L) access each bank contiguously, doubles. This is something related to the FFT algorithm data dependencies. In figure 3.9 it is shown on the left most side the initial structure of the samples.
To start with a simplistic view the array has an increasing number, but following algorithm [10], the memory location stores specific samples i.e. memory location 0 stores sample 0, $1 \rightarrow 4$, $2 \rightarrow 2$, $3 \rightarrow 6$, $4 \rightarrow 1$, $5 \rightarrow 5$, $6 \rightarrow 3$, $7 \rightarrow 7$.
In this way the contendency is solved and latency of 0 is achieved for load and store operations.
In this kernel loads and stores are assumed to happen only on the first column (equal if considered as first row to match figure 3.2). This actually impose a limitation on the %usage of the CGRA, resulting in a completely inutilization of the last column (row).

3.3.3 Convolution kernel: OFF-loading address generation and control logic

The convolution kernel was subjected to the same procedure of the FFT off-loading. Starting from the C Code, followed by the LLVM translation and the OpenCGRA mapping.

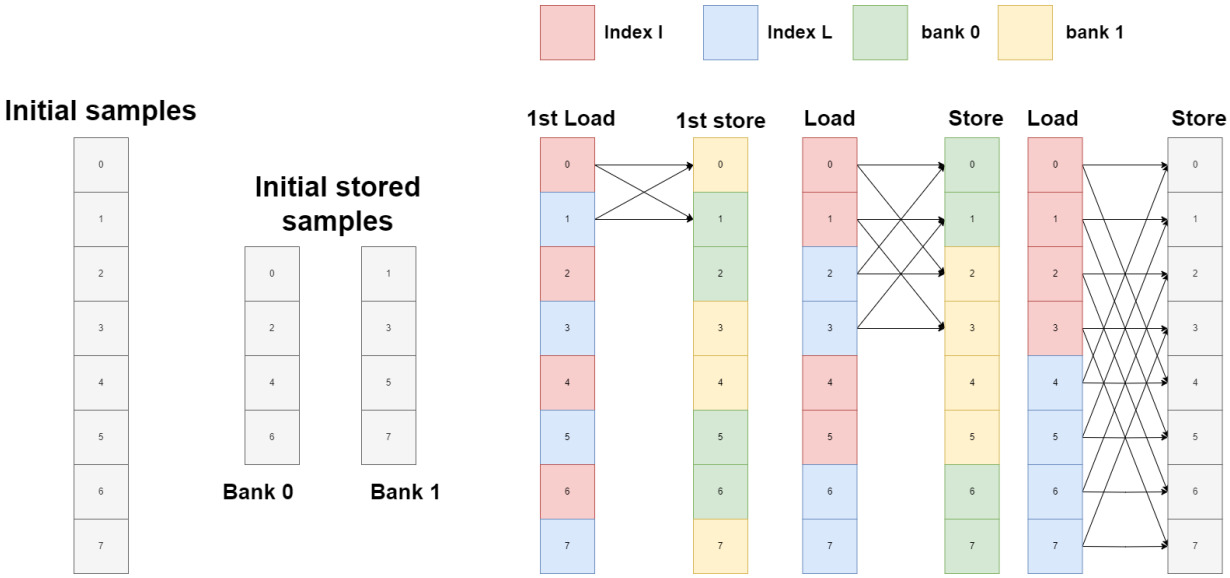


FIGURE 3.9: Fast Fourier Transform algorithm, load and store pattern of overlapped loads. Image flow, startinf from far left: result of sampling of the signal, storing after bit-reversal in different banks. Loads, computation and stores of FFT.

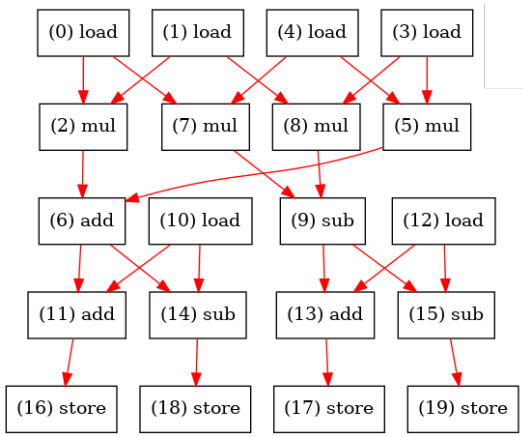
In this case the result was much more striking than the FFT. Off-loading the addressing and the control logic, allowed to reduce the II of the convolution from 4 to 1 gaining a reduction of the II, by 75%. The C code does not requires much explanation, being a MAC operation, and is shown below:

```
1 void kernel ( int A [ NI ][ NJ ] , int B [ NI ][ NJ ], int out[ NI ])
2 {
3     out[0] += A [ 0 ][ 0 ] * B [ 0 ][ 0 ];
4 }
```

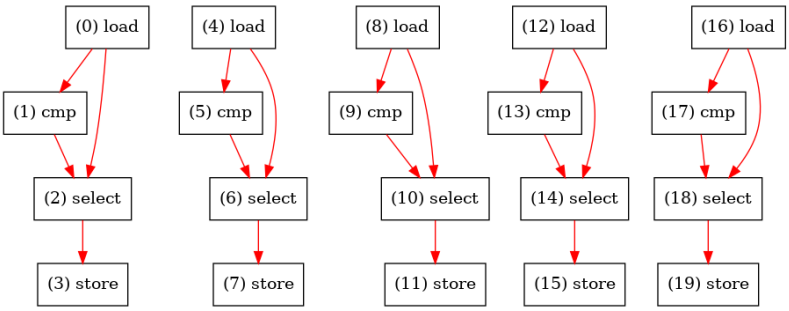
Always because of the LLVM translation and optimizations, already mentioned in section 3.2.2, it is necessary to explicit the addressing and making it equal 0, to avoid *getelementptr* operations.
The resulting DFG consists of 5 nodes and can be seen in figure 3.10c. Here all the loads are happening at the same time and accessing different banks. The only problem with this process is that the store back of the value would happen on the same bank of memory from where the load is trying to fetch. To completely avoid latency due to loads, for the output array it is possible to use 2 banks. The first one contains all the initial data meanwhile the second is used to store the results.

3.3.4 Ld/Str offload, unrolling and FFT properties

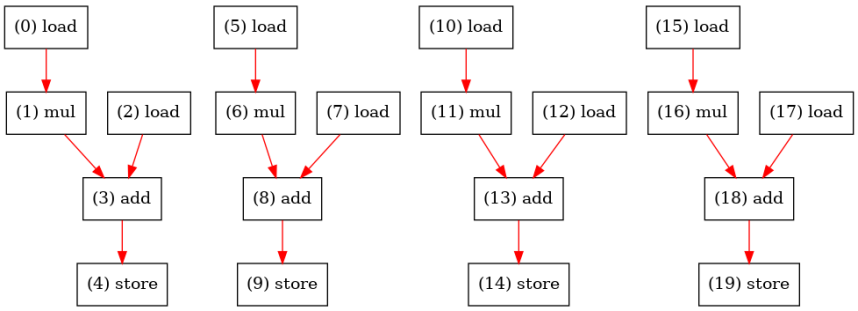
Further improvement can be achieved by entirely removing loads and stores from the CGRA. By refering to section 3.3.2 it can be seen on figure 3.8 e.g. on the first cycle, the DFG nodes 6-0-1-9. These nodes are all loads and stores that take a big part of the CGRA. If these operations are removed and given to an external modules, the



(A) FFT DFG optimized. Structure representing radix-2 structure.



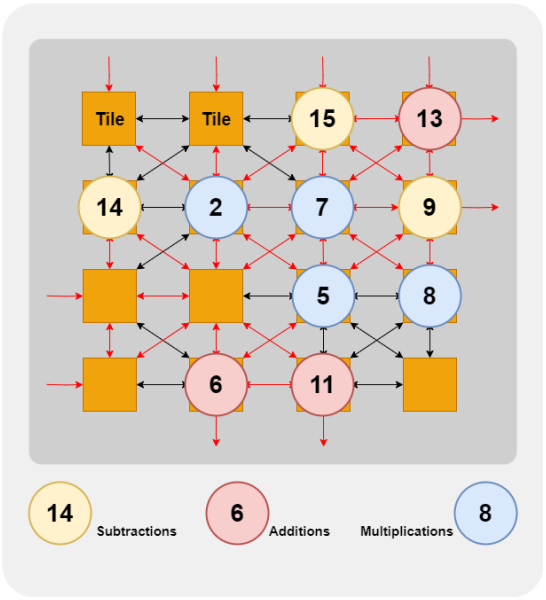
(B) ReLU DFG with unrolling equal to 5.



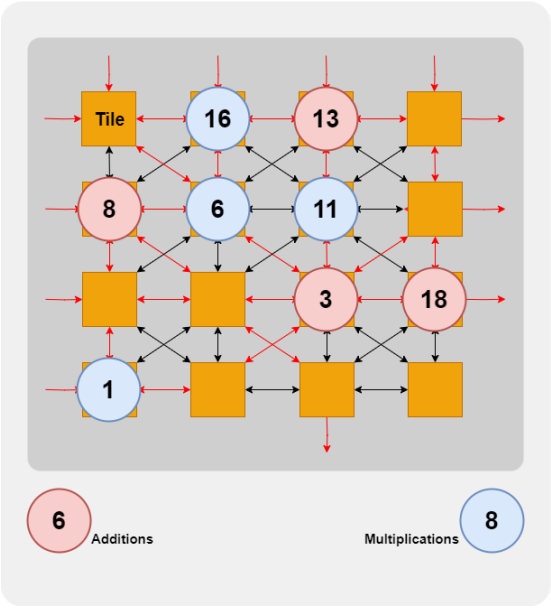
(C) Convolutional DFG optimized with unroll of 4, can exploit multi input channel, point-wise convolution or generic MAC operations with weight stationarity.

Figure 3.10: Convolutional, FFT, ReLU DFG. Kernel adopting all the optimization mentioned in section 3, DFG representing the bare kernel operations, without control DFG nodes. The loads and stores in this case are not mapped on the CGRA but on external modules

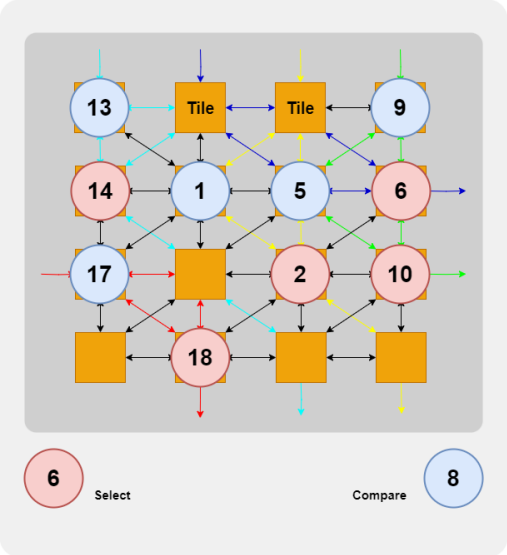
kernel II can reduce and so we can increase the computation efficiency.
The AGU module suits this task as it's already computing addresses, but it was waiting for the CGRA signal to actually fetch or store data.
If now the AGU takes totally control over the data transfer it can operate in symbiosis with CGRA. Furthermore for the FFT kernel implemented in the section 3.3.2 it was noted the relationship between the twiddle factors of *cos[upPointer]* and *cos[downPointer]* being: cos[downPointer] = -cos[upPointer]. The last optimization consists in incrementing the nodes that can load and store. This basically mean that we are increasing the load/store parallelism or increasing the number of ports that are interfacing with the TCDM. By bonding these optimization the kernel from a II of 4 reduced to 1, gaining an increase of 75%. If we compare the current obtained result with the original kernel implementation we get a total gain of 80% in speed.



(A) FFT mapping. In red the connections used.



(B) Convolutional DFG optimized, unroll of 4.



(C) ReLU DFG with unrolling equal to 5. Each interconnection color correspond to a single ReLU.

FIGURE 3.11: Convolutional, FFT, ReLU mapping (II = 1). Kernel adopting all the optimization mentioned in section 3, DFG representing the bare kernel operations, without control DFG nodes. Load and store are the incoming arrows from the external module AGU, section3. The mapping is a mix of manual and automatic operations, the connections between tiles is not made by the compiler.

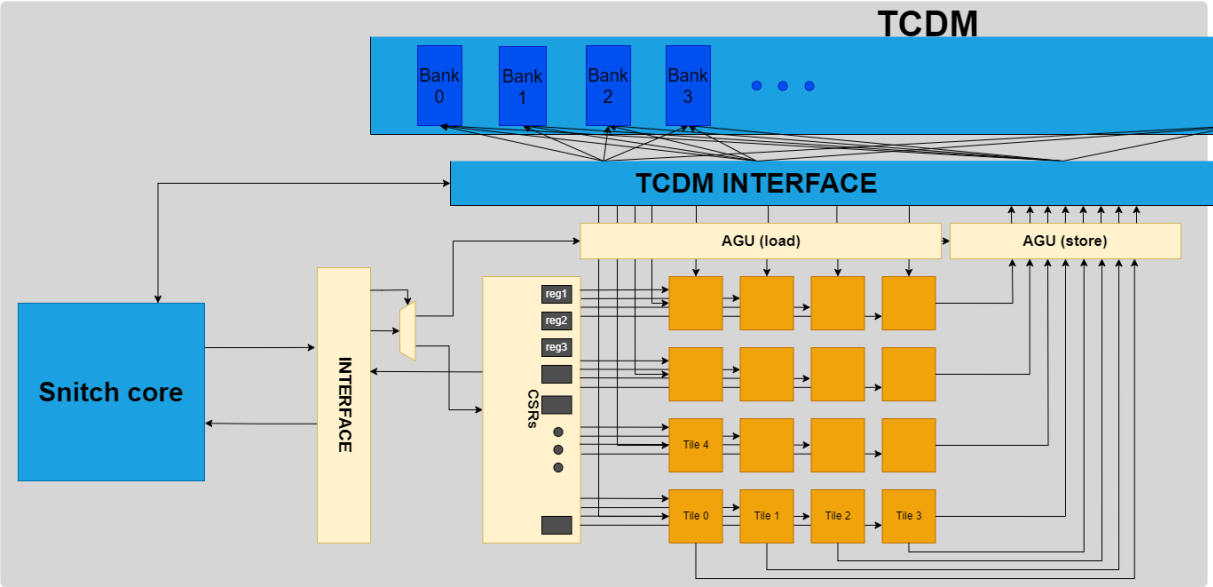


Figure 3.12: Final SNAX-CGRA integration, a MUX is added to guide the data in the architecture, based on the given address.

An execution of this kernel can be seen in figure 3.15
For the convolutional kernel we gain no speed increment, since kernel II is already at its minimum. However if weight stationarity is exploited it is possible to have much more freedom in moving data inside the structure, since the weights are not needed to be loaded and moved to reach the correct tiles, and being able to spatially unroll the MAC operation for a maximum number of 4 times, increasing the overall speed by 4x, this can be exploited for point-wise convolution or for generic MAC operations. Other than this amount of unrolling, the kernel II would increase because of the limited physical size. For real convolution e.g. *conv2d*, however this load and store off-loading can be exploited by now computing the whole 3x3 matrix multiplication on the CGRA. However this consists in having a kernel II = 2.

The ReLU kernel has a final II equal to 1 and, analogically to the convolution, can be spatially unrolled for a total of 5 times, thanks to the fact that it does not consist in a large number of operation.

The final optimized DFG for each kernel are shown in figure 3.10

However these results come with some expenses. Compared to the original kernels where everything (addressing, control logic and computation) is taken care by the CGRA the current solutions consist in the addition of external modules to take care of certain aspect of the kernels procedure. As already stated in section 2.1, we are trading area for performances[34], that in this case consists in speeding up the process.
The final architecture that can support these optimization can be seen in figure 3.12

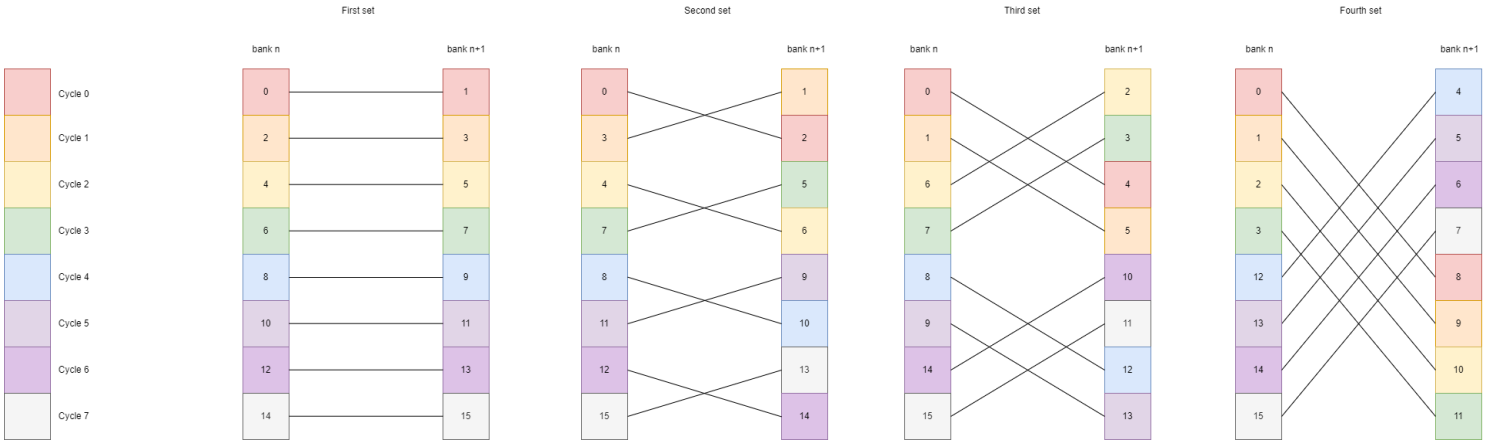


FIGURE 3.13: FFT Memory access pattern. The squares with the same color are happening during the same clock cycle. Important to note how the data is being stored between 2 layers. Looking at the first set, the operation between 0 and 1, will give as output 0 a 1, same pattern for the other operations. Example with 16 FFT points.

3.3.5 Memory access pattern

For the last evaluated kernels, access memory is fundamental if we want to achieve these aforementioned performances. The reason why SNAX was picked as developing environment for these kernels is maunly due to the flexible TCDM. This flexibility will give us improved performances, compared to the mentioned state-of-the-art, around a factor of 4 when dealing with FFT, that is the most data driven kernel among the implemented.
In this section it will be explain how to completely avoid contendency when dealing with highly data driven kernels, that requires a complex access memory pattern.
The possible approach to store data are 2: store data horizontally (Across different banks of memory) or vertically (same bank). Vertical stores will be used in this case. The main idea, when dealing with a combination of multiple parallel loads and kernel II = 1 is to store the wanted data in different banks, in order to avoid memory contendencies. When looking at the FFT kernel (taking in consideration only the real part), we start by storing the initial samples in 2 separate banks (after bit reversal), figure 3.13 in first set, the line couples the data that will be used in the radix-2 operation and its output is shown on the next set by the same number, so first the radix-2 in first set between 0 and 1 will give a certain result that will be stored in the second set in memory node 0 and 1 (refers to figure 3.9).
The procedure to fix this data contendency is explained in section 3.3.2 for the imaginary part,but it can be expanded to the real part of the number, this is needed because now the kernel II is equal to 1 and we have 4 parallel loads (2 loads for real and for the imaginary part). The idea is the same and better showed in figure 3.13

For simplicity, all the enabled FUs are counted as operations in these benchmarks. As the operation count may not be trivial in control-driven applications.

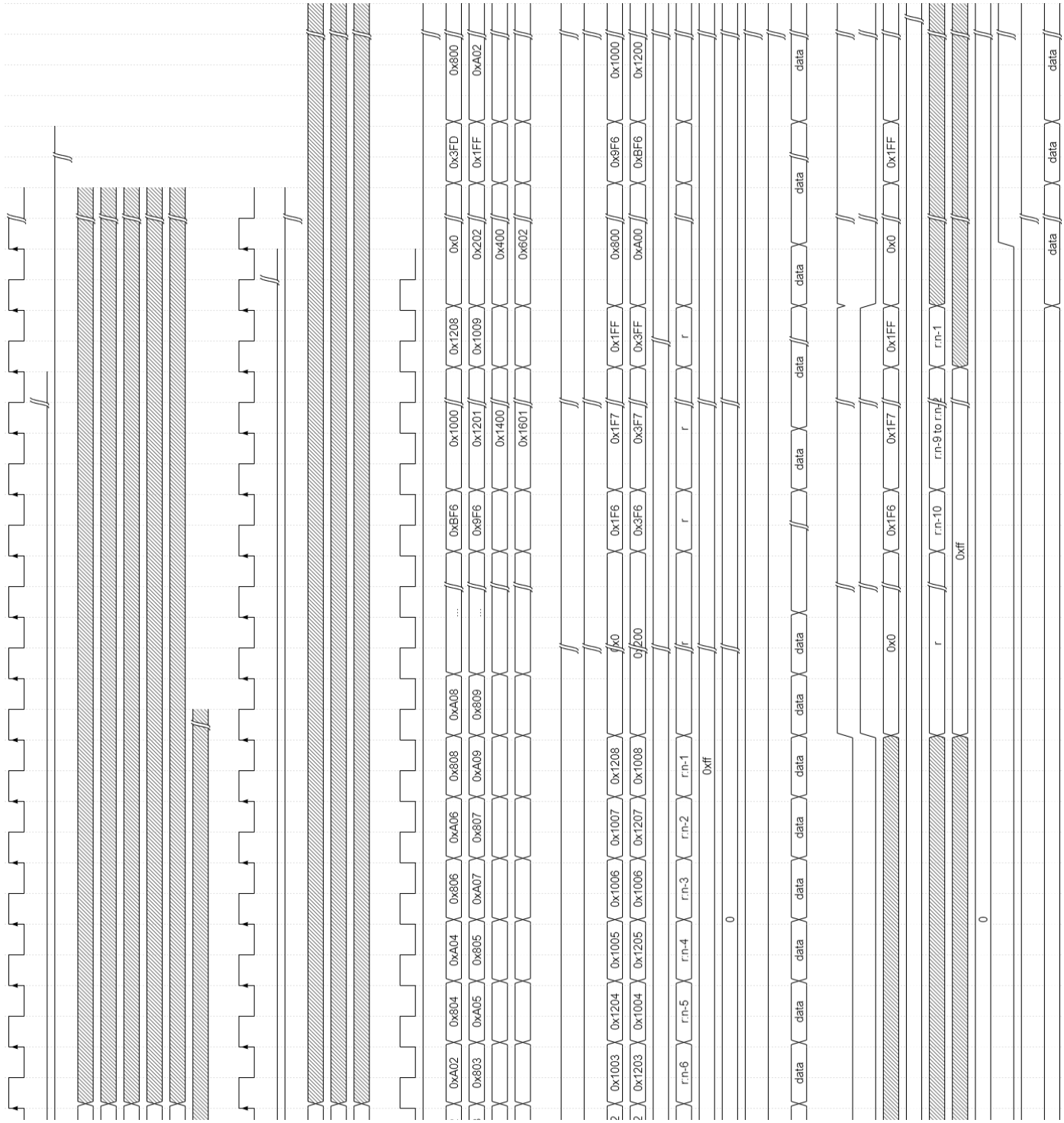


Figure 3.14: Last section of FFT kernel

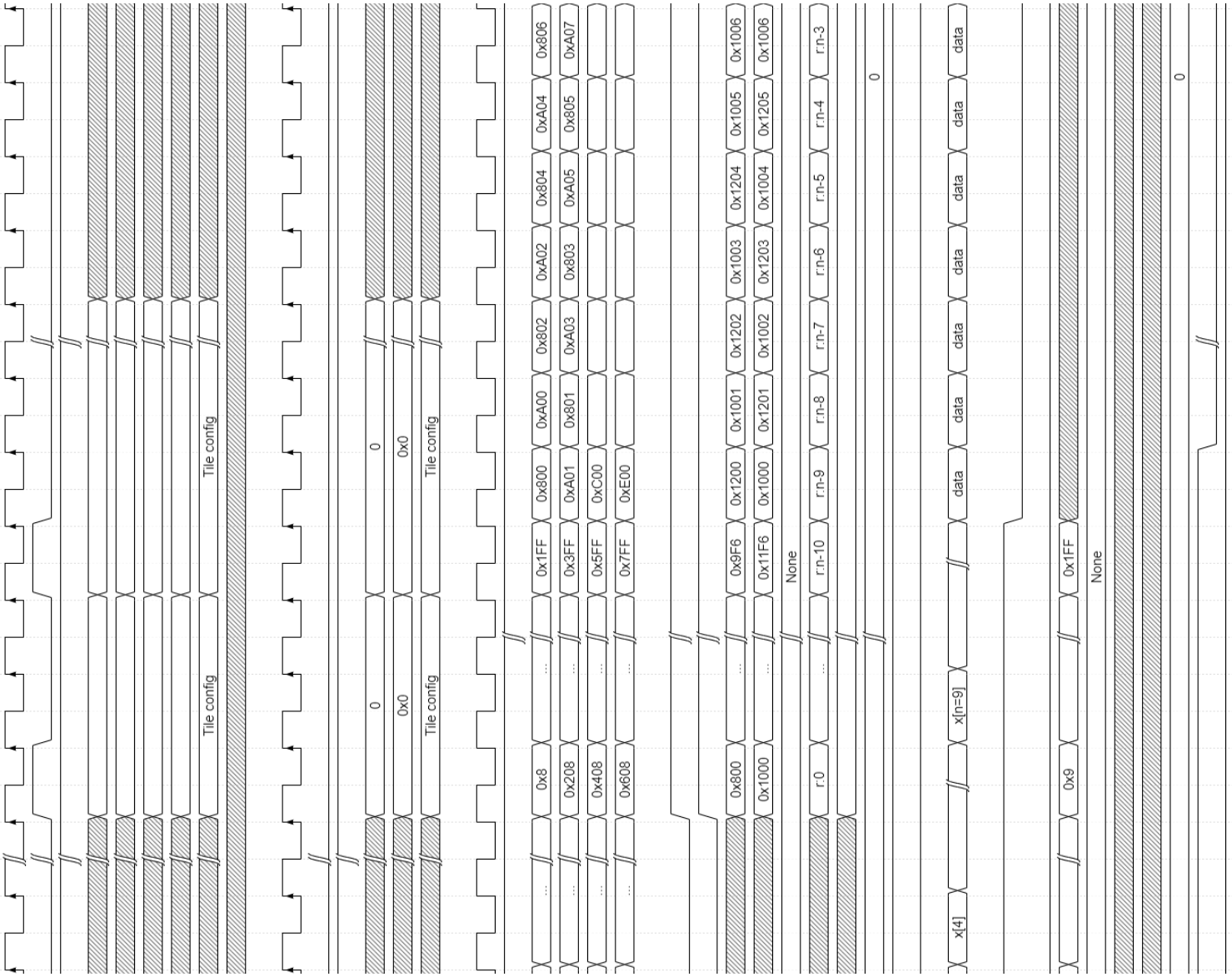


Figure 3.15: Middle part of CGRA computation

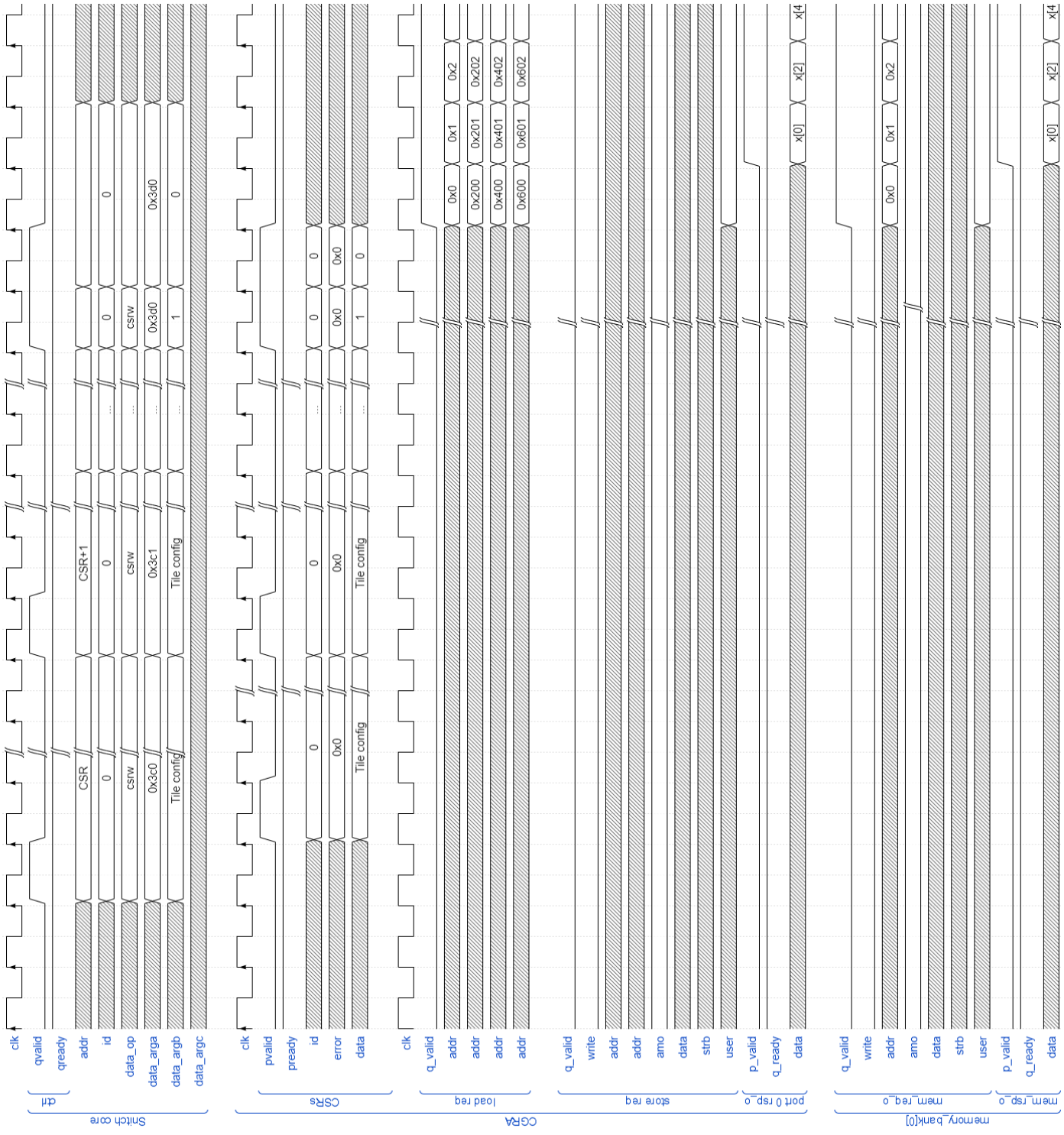


Figure 3.15: CSRs loading and Snitch CPU start command and start of CGRA computation.
Showcasing the FFT kernel implemented on the SNAX-CGRA, showing CSRs loading and memory irregular access.

Chapter 4

Evaluation and Results

In this section we will provide some performances regarding the kernels providing more insights on how the memory contendencies play a big role in developing HPC algorithms, the synthesis results and we will compare our results with the state-of-the-art works mentioned in section 1.3 giving something to compare it with in order to see how this implementation behave.
The final implementation can be seen in figure 3.12, zoomed and updated version of figure 3.2

4.1 SNAX-CGRA system study

Figure 3.15 shows the actual behaviour of the SNAX-CGRA while executing the FFT kernel in section 3.3. The process starts with the Snitch core sending single Tile configuration to the CSRs that due to bus bitwidth limitation, they have to be sent twice, during the first transmission the CSRs receive the first 32 bits and in the second receive the latter 32.
After all the CSRs load, the CPU sends a the start command, a '1', at an unique address *0x3d0* that corresponds to *0x3c0 + CGRA size + 1*. After the start command is asserted the computation CGRA computation starts next cycle.
In the middle figure we can see the store delay of the CGRA, after 10 amounts of clock cycles the CGRA starts to contiuously store into the memory without interrupting thanks to the optimization mentioned in this section.
In the middle and first figure it can be seen how the addresses from the load and store switch at the right time avoiding contendencies between store and load. In the first figure 3.14 it can be seen how the loads start to jump between memory banks as stated in section 3.3.5, avoiding contendencies between loads.

4.1.1 CSRs load cycles and Implicit double buffering

Table 4.3 presents the performance metrics for each kernel. Notably, the reported cycle counts exclude configuration cycles, which reach a maximum of 4 for a kernel iteration interval (II) of 4. However, various methods can be employed to upload

FFT				
		Cycles		
		Snitch core	**Original**	**Optimized**
FFT Nodes	**16**	443	460	45
	32	1163	1132	91
	64	3082	2700	202
	128	7351	6296	458
	256	16732	14348	1034
	1024	100491	71692	5130

Table 4.1: FFT results for CPU, CGRA original and optimized version vs FFT nodes.

the control status registers (CSRs). Initially, the SNAX system does not possess any data in its memory. To acquire data, SNAX accesses external memory, specifically the DRAM. Directly accessing DRAM to fetch data and load it into the CSRs is a time-consuming process, requiring approximately 22 cycles per access. Consequently, loading a total of 64 registers directly from DRAM would necessitate around 2900 cycles, making it a slow procedure. Alternatively, utilizing the Direct Memory Access (DMA) controller to transfer data from the DRAM to the local TCDM before loading the CSRs significantly reduces the cycle count to 907, resulting in a speed-up of 3.2x, taking 7 cycles for configuration sent.
This information might suggest that reconfiguring the CGRA to execute different kernels would consistently require this amount of cycles. However, the CSRs can be employed as double buffering technique, as discussed in Section 2.1. This technique virtually eliminates the latency between kernel executions, reducing the delay to only 4 clock cycles max.

Depending on the kernel the CGRA needs a certain amount of clock cycles to *warm up*, these cycles can vary from 4 to 16 (worst case where the data has to travel the whole CGRA twice) that compared to the whole kernels execution can be neglected.

4.2 Benchmark performance summary and system overhead analysis

The kernels in question relate to section 3, which are: FFT, Convolution and ReLU, in this way we try to cover multiple aspects regarding an AI algorithms that are: data driven application and control driven applications.

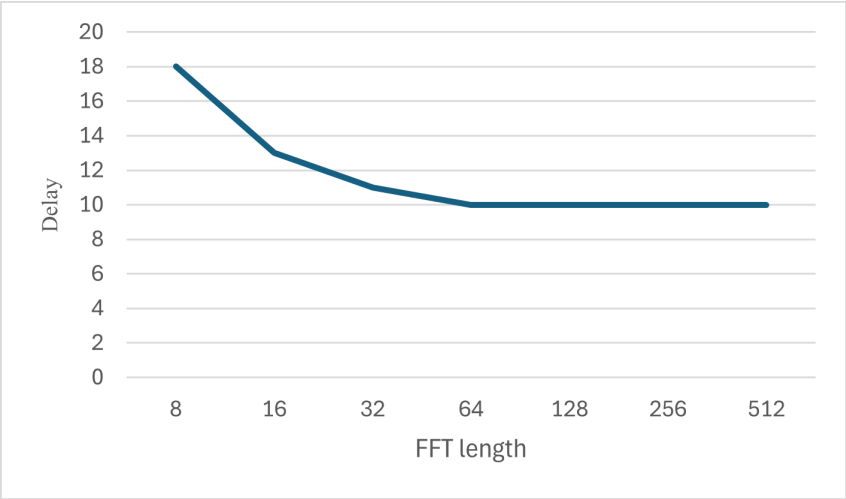


FIGURE 4.1: FFT delay cycles due to data dependency and warm-up cycles compared to ideal FFT cycle length.

In these tests we included multiple sets on the FFT algorithm in order to show the impact of highly dependency on data position temporally and spatially. This behaviour can be seen in figure 3.9.
The number of FFT nodes used range from 16 to 1024, for the convolution it was adopted a point-wise convolution with a 64x64 and conv2d with matrix size of 64x64 and with a weight matrix of 3x3, exploiting weight stationarity for both cases, and lastly for the ReLU a matrix of size 128x128 is used, adopting the optimized version of the kernel. Together with the comparison between the initial kernel without optimization and the optimizied version, it is given a comparison with how many cycle the SNAX Snitch core takes to compute these algorithms. Furthermore the comparison with other state-of-the-art is given. It is worth mentioning that in these reports the cycles refers to the start of the first load to the last store, therefore including warm-up cycles and final stores.
The mappings of the tested kernels are shown in figure 3.11.

Table 4.1 illustrates the clock cycles required to execute the FFT algorithm in its entirety. Notably, there's an interesting observation regarding the FFT size and its influence on computational efficiency. As the FFT size decreases, the deviation from the ideal N log2(N) behaviour becomes more pronounced. However, this deviation stabilizes when the FFT size exceeds a certain threshold, around N = 64, with an offset of 10 clock cycles. This behavior can be attributed to the intrisic warm-up cycles of the CGRA. These warm-up cycles is the result of the time it takes for data to move between processing tiles within the array. In the FFT computation, a delay of 10 clock cycles is observed between the initial data loading into the first store of the results. This delay remains constant regardless of the FFT size. Moreover, this delay propagates through subsequent FFT layers, as they depend on the completion of stores from preceding layers. Thus, in shorter FFT computations,

Kernel performances				
Kernels	**FFT**	**Conv2d**	**Point-wise Convolution**	**ReLU**
Snitch core	100491	216855	53250	163843
CGRA execution cycles				
Original	71692	147456	16384	81920
Optimized	5130	8208	1029	3287
Outputs/cycle	3.99	4.9×10^{-1}	3.98	4.98
Number of operations	51193	73829	8216	32750
Speed-up				
Opt. vs Orig.	13.9x	17.9x	15.9x	24.9x
Opt vs CPU	19.6x	26.4x	51.7x	50x
%CGRA usage	93.75%	93.75%	93.75%	93.75%

Table 4.2: Kernel performances. FFT is using 1024 samples, Convolution and ReLU are using matrices of 128x128. Everything is considering TCDM as memory and not external memory. The reported data is refering to the optimized version of the CGRA where not explicited.

the waiting period for these essential stores becomes relatively longer, leading to increased overall computation time. This relationship is visually depicted in Figure 4.1, emphasizing the impact of FFT size on computational resource distribution and resulting performance characteristics.

For the conv2d we have a kernel II of 2 without the need of configuring multiple times the CGRA since this architecture can exploit temporal configurations. With the implemented kernel we compute the whole 3x3 matrix multiplication between the local stored weights and the inputs, in this way no intermediate store back is needed, exploiting output stationarity.

In table 4.2 there is a voice called *%CGRA usage* that basically states how many tiles are actually being used (for data transfer or computing). The voice *Number of operations* states how many operation are happening in the who algorithm, counting how many FUs are used per cycle. This number is not simply the total number of cycles needed to compute the algorithm times the number of FU used because we need to consider the warmp-up cycles and the last cycle for the stores.

4.3 Work comparison

n this section, a detailed comparison with the works already mentioned in Section II-C is presented. The comparison begins by generating Table IV, which provides a comprehensive summary of the CGRA (Coarse-Grained Reconfigurable Array) specifications. This table includes key details such as the type of control unit employed and the technology node used in each work. Unlike HyCube, Softbrain, and ADRES, which utilized REVAMP [2], we explored integrating the architecture discussed in

Kernel Performances				
Kernels	**FFT**	**Conv2d**	**Point-wise Convolution**	**ReLU**
SNAX-CGRA				
Configuration cycles	1	2	1	1
execution cycles	5130	8208	1029	3287
Outputs/cycle	3.99	4.9*10^-1	3.98	4.98
Number of operations	51193	73829	8216	32750
%CGRA usage	93.75%	93.75%	93.75%	93.75%
CGRA consumption				
Perf (MOPS)	**2,494.64**	**2,245.7**	**1,996.11**	**2,490.87**
STRELA [37]				
Configuration cycles	84		-	84
Execution cycles	20,980	13,931	-	697
Outputs/cycle	1.95	$2.58*10^{-1}$	-	1.47
Perf (MOPS)	1,223.71	1,172.71	-	734.58
Snitch core	100491	216855	53250	163843
Speed-up Ratio				
SNAX vs STRELA	**4.09**×	**1.69**×	-	**3.4**×
SNAX-CGRA vs Snitch	**19.6**×	**26.4**×	**51.7**×	**50**×

TABLE 4.3: Performance comparison with Snitch CPU and STRELA [37].

[31] with SNAX. This integration allowed us to propose a CGRA configuration that incorporates a low latency memory, significantly impacting overall performance, as illustrated in Table V. Furthermore, our work is unique in proposing a Static Dataflow (SD)/Time Multiplexed (TM) approach, as it is geared towards High-Performance Computing (HPC). In the case of STRELA, DSAGEN, and Softbrain, a portion of the kernel responsible for memory access is decoupled from the CGRA computations, which is a strategy similar to ours. The first row of Table IV indicates whether the CGRA operates with time multiplexing or follows a static dataflow model. Additionally, the table specifies whether a scratchpad memory is implemented and if the CGRA can handle conditional statements, such as those found in irregular loops. Currently, the total memory available in SNAX is 128KB, although this can be adjusted based on specific requirements. Similar to STRELA and IPA, our work implements a 4x4 CGRA configuration. This design choice ensures that the integrated kernels are efficiently sustained, achieving a high percentage of CGRA utilization. This high utilization rate signifies that a substantial number of tiles are actively engaged in computing these kernels.

Metric	SNAX-CGRA	STRELA	RipTide	ADRES	HyCube	Softbrain	UE-CGRA	IPA
Internaldatasynchronization	**SD/TM**	**SD**	**SD**	**TM**	**TM**	**SD**	**SD**	**TM**
Irregularloops	✓	✓	✓	✗	✗	✗	✓	✓
Nouseofscratchpads	✓	✓	✓	✗	✗	✗	✗	✗
TCDM	✓	✗	✗	✗	✗	✗	✗	✓
ControlCPU	Snitch	RV32IMC	RV32EMC	-	-	-	RV32IM	OpenRISC
TotalMemorySize(KB)	128	256	256	64	64	64	64	77
CGRASize	4x4	4x4	6x6	6x6	6x6	6x6	8x8	4x4
Technology(nm)	TSMC45	TSMC65	Intel22	22	22	22	TSMC28	STM28
ClockFrequency(MHz)	250	250	50	100	100	100	750	100
SoCArea(mm2)	-	2.38	0.50	-	-	-	-	0.34
CGRAArea(mm2)	0.264	0.25	0.25	0.20	0.165	0.125	0.28	0.20
PEArea(m2)	12,057	13,243	7,000	-	-	-	4,000	7,031

Table 4.4: Comparison between the state-of-the-art mentioned in section 1 and SNAX-CGRA model.

		Perf. (MOPs)			Power Consumption (mW)			Energy Efficiency (MOPs/mW)		
Work	**Frequency (MHz)**	**fft**	**mm 16x16**	**mm 64x64**	**fft**	**mm 16x16**	**mm 64x64**	**fft**	**mm 16x16**	**mm 64x64**
IPA	100	-	65.98	-	-	0,49	-	-	134,65	-
UE-CGRA	750	625	-	-	14,01	-	-	44,61	-	-
RipTide	100	62	-	164	0,24	-	0,5	258,33	-	328
STRELA	250	1,223.71	163.9	437.8	16,84	3,99	7,46	72,68	41,08	58,66
SNAX-CGRA	250	2,494.64	-	-	6.99	-	-	356.9	-	-

Table 4.5: Quantitative comparison between SNAX-CGRA and state-of-the-art work.

4.3.1 Synthesis results

The comprehensive area analysis of the CGRA, synthesized utilizing 45nm technology, is presented in Table 4.4. Upon examination, it is evident that the overall CGRA area is comparable to that of STRELA, RipTide, and UE-CGRA. However, there is a notable disparity in the area occupied by the Processing Elements (PEs). When comparing the CGRA system configuration to that of STRELA, which employs a closely similar configuration, the observed differences can be attributed to the utilization of a smaller technology node and the implementation of a king-mesh interconnect, as opposed to the conventional mesh employed by STRELA. The adoption of the king-mesh configuration is essential to enhance flexibility in data movement, particularly for accommodating the additional four loads implemented in our design.

Despite incorporating this interconnection configuration, the overall CGRA structure occupies 13.8% of the total area, which is comparable to the 10.7% observed in STRELA's configuration. This comparison is illustrated in Figure II, which uses TSCM 16 technology. In contrast, the area distribution in IPA and UE-CGRA is heavily influenced by scratchpads, which constitute one-third of their total area. Meanwhile, for RipTide, a significant portion of the CGRA area, amounting to 50%,

	Cell Area (um)	MGates-Eq	Ratio
CGRA	49117.7	0.24	13.8%
DMA	12157.8	0.06	3.4%
Data Memory	140620.7	0.68	39.4%
Snitches	13158.4	0.06	3.7%
$ICache	32523.5	0.16	9.1%
Narrow TCDM Inteco	7404.0	0.04	2.1%
Wide TCDM Interco + Mux	8332.2	0.04	2.3%
AXI Interco	44497.0	0.21	12.5%
Peripheral & Others	49085.6	0.24	13.8%
Total	356897.0	1.72	100.0%

TABLE 4.6: Total area of the SNAX-CGRA, in TSMC 16.

is dedicated to the Network-on-Chip (NoC) interconnection structure.

The reduced processing area in our CGRA compared to STRELA is primarily due to the advanced node technology, which nonetheless maintains equivalent functionality. Additionally, our CGRA incorporates a control memory facilitating time multiplexing. This control memory occupies a total area of 2945 μm^2, representing 24.5% of the total processing element area. This allocation could pose a limitation for embedded systems. Detailed area metrics are provided in Table 4.6.

Chapter 5

Conclusion

The main objective of this thesis is to provide an integration and exploration of a Coarse-Grained Reconfigurable Architectures into a real complex system, in order to prove that these architecture together with an AI friendly environment, the Snitch Accelerator eXtension, can deliver good performances, dealing with multiple AI workloads containing regular and irregular loops, something that most state-of-the-art, as already mentioned in section 1.1, cannot deal with.
The current work, propose an integration, in the SNAX system, of a 4x4 CGRA focused towards HPC optimizing bottlenecks of FFT, convolution and ReLU, exploiting high parallelism and low latency. This is done by memory management and off-loading from the CGRA operations related to address generation and control logic, giving such tasks to external modules (AGUs and CSRs), increasing the performances up to 80%. The memory managament is fundamental if HPC is considered, and working on memory access optimization can result in an 4x increase of performances, looking at the comparison with STRELA.

The initial idea from where this work started was to give realistic performances about the implementation of CGRAs, in order to give insights on how realistically this approach can be exported and applied to the research domain and give opportunities to new-born models to be executed and classified correctly, or just to provide a DFG processor where to offload generic computing intensive kernels, reducing overall system bottlenecks.

5.1 Future work

To further explore the potential of the CGRA within the SNAX, we can address some aspects that can be explored.

5.1.1 CGRA exploration

CGRA can be subjected to many more architectural considerations and some further design towards power efficiency can be done.

- **Architectural exploration**: CGRA interconnection networks can be expanded by looking at the trade-off of adopting connections that link the furthest tiles with the total overhead. This would limit the data transfer between tiles, limiting congestion and latency due to data movement.
  Implement more advanced PEs to compute more complex operation on single tiles (e.g. MAC operations), enable variable precision on single unit, allowing to increase the throughput.
  Start using hierarchical memories, reducing the overall access to the TCDM keeping data local.

- **Energy-efficienct design**: Implement technique to reduce overall power consumption: clock-gating in a coarse way the entire CGRA when in idle, clock-gating single tiles during operation is not possible as %CGRA usage tends to be very high during execution. Together with clock gating, power gating and dynamic body biasing is possible during idle, in order to decrease leakage.
  During operation what can be done is to play on the supply voltages of each tile based on the critical path, using multiple supplies sources.

Future research could focus on integrating the SNAX system with advanced AI models, including transformers and graph neural networks, to evaluate performance improvements and identify potential architectural adjustments required for optimal compatibility.

5.1.2 SNAX exploration

Adopting SNAX already improved the performances by a 3-4x (compared to STRELA) and reduced the configuration time thanks to the CSRs acting as a double buffering technique. However some consideration can be done at system level.

- **CSRs overhead**: The current implementation of the CSRs include a total of 64 register giving the possibility to simultaneously load each tile of the CGRA for a maximum II of 4. However area can be reduce by adopting a technique similar to [37] using a single buffer and uploading the tile one by one by means of an ID. However the bith width limitation of the busses interfacing CPU and Accelerator is already limited, so attention needs to be payed in this regard.

- **Multi-CGRA systems**: The exploration of the scalability of the CGRA within the snax can be done, by deploying multiple CGRAs in different cluster. However challenges arise in communication, workload distribution and synchronization mechanisms.

Bibliography

[1] D. P. K. A. Andrew Waterman, Yunsup Lee. The risc-v instruction set manual, volume i: Base user-level isa. In *Technical Report UCB/EECS-2011-62, EECS Department, University of California, Berkeley*, 2011.

[2] T. K. Bandara, D. Wijerathne, T. Mitra, and L.-S. Peh. Revamp: a systematic framework for heterogeneous cgra realization. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '22, page 918932, New York, NY, USA, 2022. Association for Computing Machinery.

[3] P. Bouboulis and S. Theodoridis. The complex gaussian kernel lms algorithm, 2010.

[4] P. Bouboulis, S. Theodoridis, and M. Mavroforakis. The augmented complex kernel lms. *IEEE Transactions on Signal Processing*, 60(9):49624967, Sept. 2012.

[5] Y. Cao, S. Li, Y. Liu, Z. Yan, Y. Dai, P. S. Yu, and L. Sun. A comprehensive survey of ai-generated content (aigc): A history of generative ai from gan to chatgpt, 2023.

[6] G. Charitopoulos, I. Papaefstathiou, and D. N. Pnevmatikatos. Creating customized cgras for scientific applications. *Electronics*, 10(4), 2021.

[7] Y.-H. Chen. Eyeriss v2: A flexible and high-performance accelerator for emerging deep neural networks. In *IEEE Journal of Solid-State Circuits 52.1*, 2018.

[8] Y.-H. Chen, T. Krishna, J. S. Emer, and V. Sze. Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks. *IEEE Journal of Solid-State Circuits*, 52(1):127–138, 2017.

[9] A. Coates, B. Huval, T. Wang, D. Wu, B. Catanzaro, and N. Andrew. Deep learning with cots hpc systems. In S. Dasgupta and D. McAllester, editors, *Proceedings of the 30th International Conference on Machine Learning*, volume 28 of *Proceedings of Machine Learning Research*, pages 1337–1345, Atlanta, Georgia, USA, 17–19 Jun 2013. PMLR.

[10] J. W. Cooley, James W.; Tukey. *An algorithm for the machine calculation of complex Fourier series*. Math. Comput. 19 (90), JSTOR 2003354, 297301. doi:10.2307/2003354, 1965.

[11] S. Das, K. J. M. Martin, D. Rossi, P. Coussy, and L. Benini. An energy-efficient integrated programmable array accelerator and compilation flow for near-sensor ultralow power processing. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 38(6):1095–1108, 2019.

[12] V. Dumoulin and F. Visin. A guide to convolution arithmetic for deep learning, 2018.

[13] V. Dumoulin and F. Visin. A guide to convolution arithmetic for deep learning, 2018.

[14] J. Emer. Hardware for machine learning: Challenges and opportunities. In *2017, 2017 IEEE Custom Integrated Circuits Conference (CICC)*, 2017.

[15] S. GHOSH-DASTIDAR and H. ADELI. Spiking neural networks. In *International Journal of Neural Systems 19:04, 295-308*, 2009.

[16] G. Gobieski, S. Ghosh, M. Heule, T. Mowry, T. Nowatzki, N. Beckmann, and B. Lucia. Riptide: A programmable, energy-minimal dataflow compiler and architecture. In *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 546–564, 2022.

[17] S. Godsill. 3f3 - digital signal processing.

[18] S. Han, X. Liu, H. Mao, J. Pu, A. Pedram, M. A. Horowitz, and W. J. Dally. Eie: Efficient inference engine on compressed deep neural network, 2016.

[19] S. Hooker. The hardware lottery, 2020.

[20] D. T. Huynh, D.-V. Vu, and S.-Y. Xie. Entire holomorphic curves into projective plane intersecting few generic algebraic curves, 2018.

[21] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, R. Boyle, P. luc Cantin, C. Chao, C. Clark, J. Coriell, M. Daley, M. Dau, J. Dean, B. Gelb, T. V. Ghaemmaghami, R. Gottipati, W. Gulland, R. Hagmann, C. R. Ho, D. Hogberg, J. Hu, R. Hundt, D. Hurt, J. Ibarz, A. Jaffey, A. Jaworski, A. Kaplan, H. Khaitan, A. Koch, N. Kumar, S. Lacy, J. Laudon, J. Law, D. Le, C. Leary, Z. Liu, K. Lucke, A. Lundin, G. MacKean, A. Maggiore, M. Mahony, K. Miller, R. Nagarajan, R. Narayanaswami, R. Ni, K. Nix, T. Norrie, M. Omernick, N. Penukonda, A. Phelps, J. Ross, M. Ross, A. Salek, E. Samadiani, C. Severn, G. Sizikov, M. Snelham, J. Souter, D. Steinberg, A. Swing, M. Tan, G. Thorson, B. Tian, H. Toma, E. Tuttle, V. Vasudevan, R. Walter, W. Wang, E. Wilcox, and D. H. Yoon. In-datacenter performance analysis of a tensor processing unit, 2017.

[22] C. Lattner. LLVM: An Infrastructure for Multi-Stage Optimization. Master's thesis, Computer Science Dept., University of Illinois at Urbana-Champaign, Urbana, IL, Dec 2002. *See http://llvm.cs.uiuc.edu.*

[23] Q. V. Le, M. Ranzato, R. Monga, M. Devin, K. Chen, G. S. Corrado, J. Dean, and A. Y. Ng. Building high-level features using large scale unsupervised learning, 2012.

[24] Z. Li, D. Wijerathne, and T. Mitra. *Coarse-Grained Reconfigurable Array (CGRA)*, pages 1–41. 11 2022.

[25] Y. Luo, C. Tan, N. B. Agostini, A. Li, A. Tumeo, N. Dave, and T. Geng. Ml-cgra: An integrated compilation framework to enable efficient machine learning acceleration on cgras. In *2023 60th ACM/IEEE Design Automation Conference (DAC)*, pages 1–6, 2023.

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

[27] NVIDIA. What is high-performance computing (hpc)?, 2024.

[28] S. Riedel, M. Cavalcante, R. Andri, and L. Benini. MemPool: A scalable manycore architecture with a low-latency shared L1 memory. *IEEE Transactions on Computers*, 72(12):3561–3575, 2023.

[29] S. H. Roosta. *Artificial Intelligence and Parallel Processing*, pages 501–534. Springer New York, New York, NY, 2000.

[30] C. Tan, N. B. Agostini, T. Geng, C. Xie, J. Li, A. Li, K. J. Barker, and A. Tumeo. Drips: Dynamic rebalancing of pipelined streaming applications on cgras. In *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 304–316, 2022.

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

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

[33] X. Tan, X.-W. Shen, X.-C. Ye, D. Wang, D.-R. Fan, L. Zhang, W. Li, Z.-M. Zhang, and Z. Tang. A non-stop double buffering mechanism for dataflow architecture. *Journal of Computer Science and Technology*, 33:145–157, 01 2018.

[34] M. B. Taylor. Is dark silicon useful? harnessing the four horsemen of the coming dark silicon apocalypse. In *DAC Design Automation Conference 2012*, pages 1131–1136, 2012.

[35] C. Tirelli, J. Sapriza, R. R. lvarez, L. Ferretti, B. Denkinger, G. Ansaloni, J. M. Calero, D. Atienza, and L. Pozzi. Sat-based exact modulo scheduling mapping for resource-constrained cgras, 2024.

[36] C. Torng, P. Pan, Y. Ou, C. Tan, and C. Batten. Ultra-elastic cgras for irregular loop specialization. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 412–425, 2021.

[37] D. Vazquez, J. Miranda, A. Rodriguez, A. Otero, P. D. Schiavone, and D. Atienza. Strela: Streaming elastic cgra accelerator for embedded systems, 2024.

[38] Viso.ai. Convolution operations: an in-depth 2024 guide. *Viso.ai*, 2024. Accessed: 2024-06-04.

[39] D. Voitsechov, O. Port, and Y. Etsion. Inter-thread communication in multithreaded, reconfigurable coarse-grain arrays. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 42–54, 2018.

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

[41] J. Weng, S. Liu, Z. Wang, V. Dadu, and T. Nowatzki. A hybrid systolic-dataflow architecture for inductive matrix algorithms. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 703–716, 2020.

[42] C. Yin, N. Jing, J. Jiang, Q. Wang, and Z. Mao. A reschedulable dataflow-simd execution for increased utilization in cgra cross-domain acceleration. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 42(3):874–886, 2023.

[43] F. Zaruba, F. Schuiki, T. Hoefler, and L. Benini. Snitch: A tiny pseudo dual-issue processor for area and energy efficient execution of floating-point intensive workloads. *IEEE Transactions on Computers*, 70(11):18451860, Nov. 2021.