# IMPROVING ENERGY EFFICIENCY FOR CGRA ARCHITECTURES

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF ELECTRICAL ENGINEERING
AND THE COMMITTEE ON GRADUATE STUDIES
OF STANFORD UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

Ankita Nayak
March 2023

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

**Mark Horowitz, Primary Adviser**

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

**Pat Hanrahan**

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

**Priyanka Raina**

Approved for the Stanford University Committee on Graduate Studies.

**Stacey F. Bent, Vice Provost for Graduate Education**

*This signature page was generated electronically upon submission of this dissertation in electronic format.*

# Abstract

The rise of edge computing has resulted in a growing need for executing computationally intensive tasks within strict energy constraints. ASICs are typically used to achieve high performance and energy efficiency, but at the expense of hardware flexibility. Given the fast-paced evolution of edge applications, there is a pressing requirement for flexible yet energy-efficient architectures that can keep up with the latest trends.

Traditionally, reconfigurable computing devices have used processors, where instructions configure the processor in each clock cycle to perform the desired operation. More recently, researchers have explored using Field Programmable Gate Arrays (FPGA), and Coarse-Grained Reconfigurable Architectures (CGRA), which configure the hardware in space (and not time) to provide programmable computing devices. In the space of spatial programmable architectures, CGRAs are a promising alternative to FPGAs due to their higher energy efficiency that comes from operating at a word-level granularity in logic and routing.

This thesis introduces two methods to improve the energy efficiency of CGRAs. First, low-access-cost distributed memories are introduced into the processing elements. While similar to conventional register files, these memories are optimized to work with applications with streaming data, so they "push" the data to the computing elements. These memories help improve the energy efficiency of Deep Neural Network (DNN) applications on the CGRAs. The second method aims to improve the energy efficiency of CGRAs by introducing low-overhead fine-grained power domains to better optimize both active and idle power. Both these techniques have been integrated into a taped out SoC with a CGRA optimized for Deep Learning and Computer Vision applications.

# Acknowledgments

I am deeply grateful to the many individuals who have supported and inspired me throughout my PhD journey, without whom this thesis would not have been possible. First and foremost, I would like to express my sincere gratitude to my advisor, Professor Mark Horowitz, for providing me with this wonderful opportunity. His guidance and mentorship were instrumental in enabling me to complete this thesis. I particularly appreciate the valuable insights he provided, which helped me to refine my ideas, problem formulation, and results, ultimately shaping my research. His breadth of knowledge spanning various research fields and his passion for science and research is inspiring and the lessons I have learned from him will continue to be an integral part of my professional journey.

I am immensely grateful to Professor Priyanka Raina for her close involvement in my research. Throughout my journey, she played a critical role in providing structure, working with me on paper outlines, brainstorming on blocking issues, and helping manage key milestones. I also appreciate the opportunities she provided for delivering guest lectures at her classes, which I thoroughly enjoyed. I would also like to express my appreciation to Professor Pat Hanrahan for serving on my reading committee and reviewing my thesis and Professor Clark Barrett and Professor Benjamin Van Roy for serving on my orals committee. I have learnt a lot from them over our interactions during the AHA project and the reinforcement learning class. Many thanks to Professor Rick Bahr and the rest of the faculty members in the AHA project for the insightful discussions and perspectives that were crucial in shaping my work. Sincere thanks to Professor Sachin Katti, Professor Zain Asgar and Pete Warden for including me in the EE292D teaching staff. I enjoyed preparing for and delivering the lectures I led and interacting with the students on their class projects.

Many thanks to Steve Richardson for always being there to discuss ideas and for spending numerous hours on refining my papers and thesis. Thanks to Mary McDevitt for proofreading the thesis. I am grateful for the outstanding collaborations with my co-authors, as well as the stimulating interactions with the members of the AHA group and mhstudents. My deepest gratitude to my industry managers, leads, mentors, and colleagues for their support and mentorship. It has been a privilege to work with such talented and brilliant minds on cutting-edge research and products. Heartfelt gratitude to all my friends I have met along the way in Mumbai, East Coast, Pacific North West, San Diego and the Bay Area for enriching my life with their friendship.

My grandmother, Late Malini Gandhi, continues to be my inspiration and I hope that I can adhere to her teachings. I am thankful to my extended family for their love and kindness over the years. Sincere thanks to my in-laws (parents, sister, brother) for rooting for my success and to my husband's nieces and nephew, Stuti, Minati and Anay, for the joy they bring to our lives.

I am thankful to my parents, Anita Nayak and Late Professor Anil Nayak, for supporting me in pursuing my goals. My father's passion for science has been inspiring and I cherish the time spent surrounded by mathematics books. My mother's constant encouragement has been an essential motivator, helping me stay focused and determined. I am indebted to my sister, CA Amrita Nayak, for shielding me from tough times so I could focus on my academic and professional pursuits.

Finally, I would like to express my deepest appreciation for my husband, Venkatesh Rao. He has been an invaluable cheerleader throughout as well as a critical voice when necessary. His patience, constant optimism, and sense of humor have helped us navigate the highs and the lows. I am incredibly grateful for his understanding and companionship throughout this journey.

# Contents

# List of Tables

# List of Figures

# Chapter 1

# Introduction

The rise of edge computing has led to a growing need to run computationally demanding applications within strict energy constraints. Application domains such as computer vision and deep learning, which are commonly deployed on the edge, continue to rapidly evolve. Consequently, there is an increasing interest in exploring reconfigurable accelerators [3,6,56,82], as these architectures can support a larger number of changing applications compared to Application-Specific Integrated Circuits (ASICs) [48,51,91]. Creating these programmable solutions requires overcoming two primary challenges: first, developing an efficient hardware base; and second, constructing an application compiler system capable of producing performant mappings of user applications to the hardware. To tackle these challenges, many researchers are investigating the development of spatially programmable accelerators tailored for these new applications.

Edge computing devices are being propelled by machine learning (ML) applications that use deep neural networks (DNNs). DNNs have gained widespread adoption due to their high accuracy compared to classical methods, particularly for classification and detection tasks [30]. DNN applications are composed of two main components: training and inference [64]. During training, the model's parameters, referred to as "weights", are updated with the objective of minimizing a loss function. Training is typically a computationally intensive and iterative process that is performed offline using high-capacity computational resources. Once trained, the models are deployed on the edge devices to perform inference on unseen inputs, a technique also known as on-device machine learning or performing inference on the edge [46]. This approach is preferred for several reasons, such as data privacy, improved latency, and reduced bandwidth [61]. Although inference requires fewer computational resources than training, running these applications on resource-constrained edge devices can still be demanding. Therefore, this work concentrates on the inference component of DNNs, exploring accelerators intended for edge devices.

Field Programmable Gate Arrays (FPGAs) have traditionally been the go-to platform for spatially programmable accelerators, utilizing individual logic cells that can be programmed at the bit

level for functionality and interconnection. These platforms have well-established hardware implementations and a software stack to program their hardware base. Coarse-Grained Reconfigurable Arrays (CGRAs) have recently gained traction as a promising alternative to FPGAs. Unlike FPGAs, CGRAs operate at the word-level granularity in logic and routing, which makes them more suitable for deep learning and computer vision applications. As a result, CGRA-based accelerators for convolutional neural networks (CNNs) have been developed [3, 82], including NeuFlow [20] and Neuro CGRA [37].

This thesis investigates two methods for enhancing the energy efficiency of CGRA accelerators. The first technique modifies the CGRA's memory hierarchy, which has been demonstrated in our prior work [87] to be essential for deep learning applications. The second method involves implementing low-overhead power gating on a CGRA, allowing the use of faster, but more leaky transistors to increase energy efficiency.

## 1.1 Thesis Outline

Chapter 2 of the thesis provides a taxonomy of hardware and uses it to explain why ASICs are preferred for accelerating modern applications and discusses the challenges involved in building ASICs. It highlights the need for programmable architectures that can adapt to rapidly evolving applications, and compares two approaches to programmability - programming in space and programming in time. The chapter then focuses on spatially programmable architectures which are suitable for various deep neural network applications. Given the desire for a programmable spatial accelerator, it describes an FPGA, a commonly used reconfigurable spatial architecture. However, FPGAs have limitations that can be addressed by using CGRAs. It finally describes the architecture of the baseline SoC platform with a CGRA that we developed to introduce the optimization methods discussed in the subsequent chapters.

Chapter 3 discusses how the energy efficiency of a CGRA can be improved for DNN applications by introducing low-access-cost streaming register files that we call *ponds*. Registers are used in almost all modern computing systems and they are in fact so important that a significant part of the instruction word is used to specify which registers the instruction uses. The address generation for these registers can be a challenge in CGRAs, which typically do not use instruction-based address generation. While explicit hardware address generators can be built, their overhead can become large for small register files. To minimize this overhead, this chapter explores DNN algorithms, various types of layers in DNNs, and the commonly used dataflow patterns to execute these layers to understand what complexity is needed for the address generation. It then proposes efficient implementations for generating the addressing hardware.

Chapter 4 discusses how the energy efficiency of a CGRA can be improved through the introduction of low-overhead fine-grained power domains. The chapter begins by reviewing the design

of power domains for ASICs and explaining the importance of electrical isolation between power domain boundaries. It then discusses how the problem of power domains changes in spatially programmable designs, along with the design choices made in introducing power domain boundary protection. The chapter then goes on to detail the framework that was built for introducing power domain circuits to the base CGRA using transformation passes. It covers the power-domain-aware chip design that was implemented, including some of the implementation challenges and choices that were made, and outlines the verification methods that were used to ensure the CGRA with power domains was functioning correctly. Finally, the chapter concludes with a discussion of the power savings achieved for the applications studied.

To summarize, the thesis makes the following contributions:

- Introducing efficient streaming register files in the processing elements to improve the energy efficiency of DNN applications. The thesis evaluates DNN access schedules needed to support mapping of DNNs on the register files. It then demonstrates opportunities to reduce the complexity of the controller logic based on the access patterns required for address and schedule generation, including simplying address generation wherever feasible, sharing controllers by introducing delay blocks and adding timing signals to reduce scheduler complexity. Introducing the register files and appropriate computation blocking results in up to 50% reduction in energy for convolution layers.

- Introducing low-overhead, fine-grained power domains in the CGRA to optimize both active and idle power, with a novel design for the CGRA routing fabric that provides intrinsic boundary protection. This reduces area overhead for boundary protection from 9% to less than 1% and removes delay from isolation cells. The thesis also presents a framework for inserting logic to create power domains using compiler-like passes and formally verifies the output of these passes.

- Implementing both methods in a fabricated SoC with an ARM Cortex M3 processor and a CGRA with 32x16 processing element and memory tiles and 4 MB secondary memory to improve the energy efficiency of the CGRA. The thesis discusses the implementation challenges encountered due to the introduction of fine-grained power domains, including the addressing of the CGRA tiles, the power grid design, well substrate connections and distribution of global signals. This CGRA demonstrates up to 83% reduction in leakage power and 26% reduction in total power compared to an identical CGRA without multiple power domains for various image processing and machine learning applications.

# Chapter 2

# Hardware Accelerators

This chapter provides an overview of various hardware approaches for specific applications and their associated trade-offs. With the growing need for hardware that can adapt to changing applications, there has been a rise in interest in programmable architectures that can load configurations for different applications. This programmability can be achieved through instruction-based programming in time or through the interconnections between computing elements in programming in space. One of the increasingly popular approaches for programmable-in-space is the CGRA. As the subsequent chapters in this thesis focus on improving the efficiency of the CGRA, the chapter concludes with a description of the base CGRA organization.

## 2.1 Need for Acceleration

For decades, Moore's law has guided the development of hardware, which stipulates that the number of transistors on a chip double every two years. Dennard's scaling relates to Moore's law by stating that the computing performance per watt grows exponentially at a similar rate. Essentially, Dennard's scaling postulates that as transistors become smaller, their power density stays constant. As a result, the power use stays in proportion with the area. However, the end of Dennard's law has made it necessary to find new ways to improve the energy efficiency of a system.

One way the research community has attempted to improve energy efficiency is by building specialized hardware for specific applications, also known as hardware accelerators. The datapaths and the access patterns are customized to run the specific application, thus maximally extracting the performance of the system under tight energy constraints. Such a fixed-function hardware allows only a few programmable parameters. Adding new features typically requires new implementation of the hardware. However, building new hardware is time-consuming and expensive. The cost further increases exponentially with newer process nodes [47]. Although this approach of building such energy-efficient but costly fixed function hardware is a feasible option for stable applications that

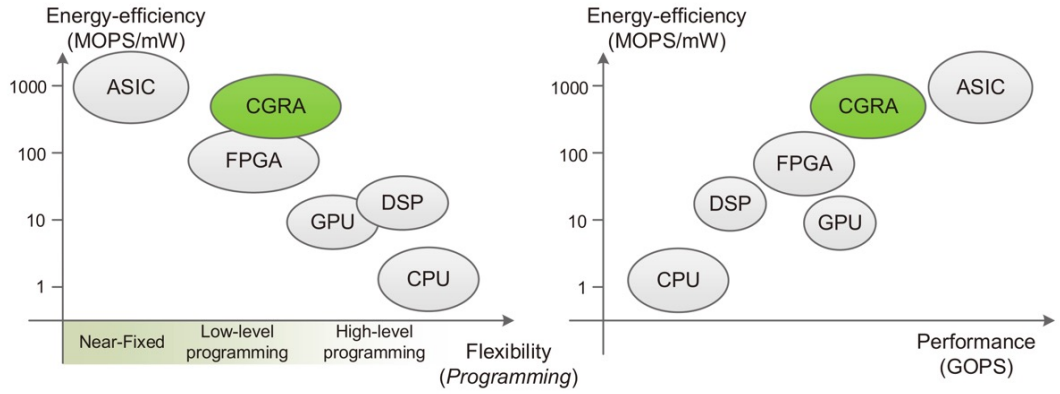


Figure 2.1: Space of programmable machines as described in [53]. Despite their higher energy efficiency and performance compared to FPGAs, CGRAs are unable to bridge the gap with ASICs.

do not evolve constantly over time, not all applications fall into this category.

For example, applications like computer vision and deep learning continue to rapidly evolve, and thus there is a need to efficiently run a broad variety of applications in this space, not just one or a few specific applications. General-purpose compute platforms such as CPUs are better suited to support rapidly evolving workloads, but they lack the performance and energy efficiency required for specific applications. Consequently, as these applications continue to rapidly evolve, the long development cycles for building these custom accelerators can risk the silicon being outdated when it is ready to be deployed. Thus, it is important to explore other ways to improve energy efficiency while providing the necessary flexibility to run a wide range of applications.

## 2.2 Programmable Spatial Accelerators

The need to provide accelerators that are energy efficient while being flexible enough to support the evolving applications motivates the exploration of other types of accelerators. One approach that has recently garnered interest is the evaluation of reconfigurable architectures, which can leverage locality to efficiently support a new class of applications. These reconfigurable architecture platforms consist of a spatial array of processing elements and memories that are interconnected by configurable networks. To map computation kernels onto this spatial array, networks are configured to create the required computational kernels and connect them to the necessary memory buffers. Additionally, memories need to be configured to stream data to the kernels in the correct order. This configuration is sometimes referred to as "programming in space," to distinguish it from the traditional programming-in-time approaches.

Programmable-in-time approaches rely on instructions that change in time to configure the hardware to create the correct results. Arbitrary kernels, i.e., a DAG of operators, can be executed by selecting appropriate instructions to fetch the data from the correct memory locations and perform the needed operation on these operands, before storing the result in another memory location that can be read by a subsequent instructions. Such instruction-based execution suffers from high overhead from the instruction fetch. However, this overhead can be reduced by using accelerators that leverage the SIMD-based approach or vector approach, which can use a single instruction to perform the same operation on multiple data [5, 18]. To further improve performance of this approach, the instruction set can be extended to create longer instructions, such as very long instruction words (VLIW), and by allowing multiple independent operations to be executed simultaneously to maximize resource utilization [21, 34].

Unlike programmable-in-time architectures where programmability is achieved by executing various instructions over time on a single execution unit, a programmable-in-space architecture utilizes many execution units (commonly known as processing elements (PEs)). Each PE executes different operations that typically stay the same over the execution period. Applications are spatially mapped on the array (hence programmed in "space"), and programming is done through static configurations instead of sequence of instructions. Typically, computation kernels accept a stream of input data and create a stream of output data and memory tiles perform the re-ordering of the data [54].

Traditionally, FPGAs have been used as the platform for this class of spatially programmable architectures, also known as reconfigurable architectures. Commonly known FPGA platforms are Stratix 10 FPGA [33] and Xilinx Virtex UltraSCALE+ [59]. FPGAs work with individual logic cells (LUTs), whose functionality and interconnection can be programmed at the bit-level. These LUTs can implement arbitrary logic. More recently, FPGAs have been deployed in datacenters to serve as a general-purpose acceleration platform [11].

However, FPGAs end up paying a high price to support configurability at the bit level. Most recent applications of interest do not need this bit-level granularity since they typically operate at word level. These applications frequently use computations like addition, multiplication, multiply-accumulate, etc. For these applications, CGRAs are a promising alternative to FPGAs, since they operate at word-level granularity in logic and routing. There are many examples of past and present CGRAs. PipeRench [25] is reconfigurable architecture from early 2000. These architectures have spatial array with processing elements and an interconnection network. The PE has ALU that is used to build the dataflow graphs. There are pass registers that implement the pipe stage in between to align data and support variable length FIFOs for delay matching. Other examples include MorphoSys [77], RaPID [17], ADRES [84]. Recently, deep learning applications have inspired CGRA-based accelerators for convolutional neural networks (CNNs), including NeuFlow [20] and Neuro CGRA [3, 37, 82].

In Fig. 2.1, the space of programmable machines is illustrated. As discussed, ASICs offer the

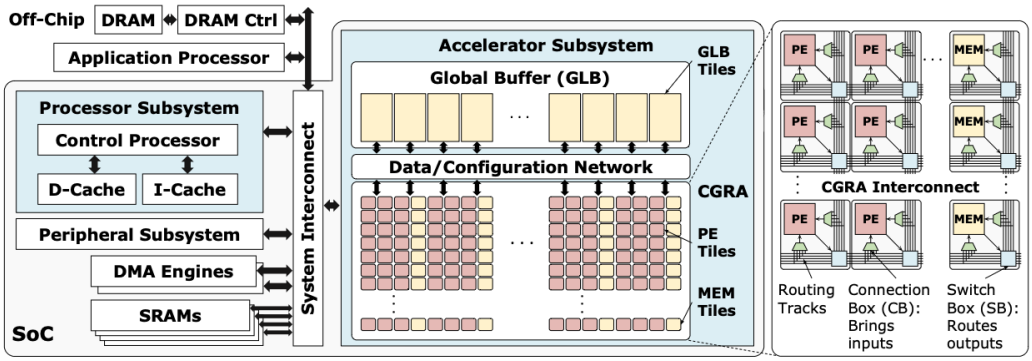


Figure 2.2: SoC platform with ARM Cortex M3 processor and an accelerator sub-system that hosts the CGRA. This platform is used as the base hardware to introduce the optimization methods aimed to improve the energy efficiency of CGRAs.

best energy efficiency and performance, but at the cost of flexibility. While CGRAs are more energy-efficient and perform better than FPGAs, they still fall short of ASICs. Therefore, it is necessary to introduce "ASIC-like" optimization knobs in the CGRA to improve their energy efficiency further. In the upcoming chapters, we explore two methods to improve the energy efficiency of CGRA. The first method focuses on optimizing memory structures for DNN applications by introducing local streaming register files in the PE. The second method targets the improvement of leakage energy by offering fine-grained leakage control. In order to introduce these optimizations, we will describe the SoC platform that incorporates the CGRA we use as our platform.

## 2.3 Baseline SoC Platform with a CGRA

The SoC platform, illustrated in Fig. 2.2, comprises an ARM Cortex M3 processor and a CGRA accelerator subsystem that serves as the base CGRA system. The accelerator subsystem features a global buffer (GLB) with 16 tiles, each equipped with two 128 KB SRAM banks, resulting in a total storage capacity of 4 MB. Each GLB tile includes a load and store unit that streams data to or receives data from the CGRA.

CGRAs have different wiring architectures. One such architecture is an island-style routing network. This architecture involves incorporating the routing resources into tiles that are replicated. Processing elements (PEs) or logic blocks with an ALU for performing computations, such as multiplication and addition, make up these island-style CGRAs. The interconnect is made up of switch-boxes (SBs) and connection-boxes (CBs). CB is a group of switches that connects the logic blocks with the routing tracks, and SB is a group of switches that connects different routing tracks.

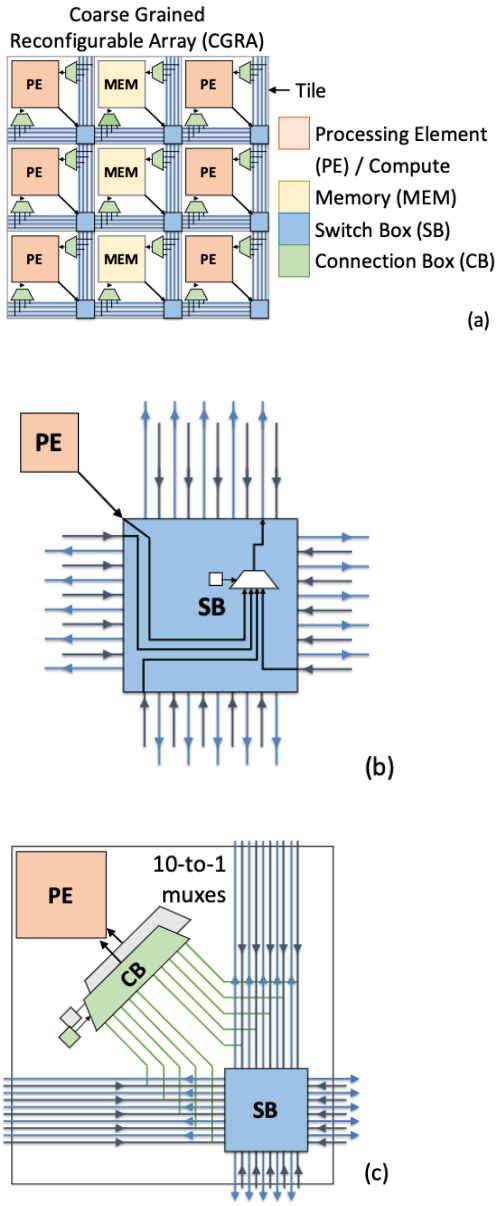


Figure 2.3: (a) Our CGRA consists of a 2D array of processing element (PE) and memory (MEM) tiles and an interconnect with horizontal and vertical routing tracks. (b) Switch boxes (SBs) implement connections between any two tiles. Each SB output (blue arrows) has a mux that selects between the PE (or MEM) output and the (gray) routing tracks coming from the three other sides of the SB. (c) Connection boxes (CBs) select PE/MEM inputs from the routing tracks. Here, the green CB selects from inputs tracks coming from the west and the north, and the gray CB selects from the east and south tracks (not shown).

Our CGRA also follows an island-style organization, as shown in Fig. 2.3.

The CGRA consists of a $32 \times 16$ 2D array of processing element (PE) tiles and memory (MEM) tiles. There are 128 MEM tiles. These tiles consist of PE and MEM cores, respectively, plus a switch box (SB) for each output and a connection box (CB) for each input of the core in the tile. There are 384 PEs that support INT16 and BFloat16 operations. Each of the 128 MEM tiles has a 4 KB SRAM and internal streaming memory controllers that allow diverse memory access patterns. The tiles communicate through a statically configured interconnect with horizontal and vertical routing tracks, connected by SBs and CBs. The SBs facilitate connections between any two tiles, and each SB output on each side includes a multiplexer that selects between the PE/MEM output and the routing tracks from the three other sides of the SB. CBs select PE/MEM inputs from the routing tracks. There is a 16-bit interconnect for data and a separate 1-bit interconnect for control signals.

## 2.4 Summary

With the growing popularity of spatially reconfigurable architectures such as CGRAs as a promising alternative to domain-specific accelerators for edge deployment, improving their energy efficiency becomes increasingly important. In Chapter 3, we propose local streaming register files for the PEs of CGRAs to enhance the energy efficiency of DNN applications, which require complex access patterns. However, managing the area overhead of the register files can be a challenge, given the need to support such patterns. To address this, we evaluate optimizations to reduce the overhead of the addressing logic and review the necessary feature support, including the number of read and write ports and the register file size. In Chapter 4, we introduce fine-grained power domains with minimal area overhead to improve the energy efficiency of CGRAs. Fine-grained power domains incur overhead from isolation cells, which prevent X-propagation from *off* to *on* domains. While this approach can be effective, global signals like `clock`, `reset`, and configuration `address` and `data` flow through every tile in each column from top to bottom, which poses unique design challenges when the top tiles are turned off. Both methods are introduced in the CGRA to improve the energy efficiency of CGRAs.

# Chapter 3

# Energy Efficient CGRA Streaming Memories for DNNs

As the next section reviews, creating and properly leveraging a hierarchy of memory sizes is essential for building fast, efficient computing engines. We use this insight to improve energy efficiency of the CGRA built for flexible acceleration of computer vision and machine learning applications. We introduce low-access-cost distributed local memories closer to the ALU. We call this new memory hierarchy *ponds*. Since the ponds are introduced in every PE tile, the challenge in building ponds is to make them area and energy-efficient while also being flexible enough to map various DNN layers and schedules effectively. This challenge is especially important for convolution layers in CNNs, a type of DNNs, due to their complex access patterns and computational intensity. To address this challenge, we first evaluate systematically the computation loop nests for DNN schedules to determine what feature set and complexity are needed to build efficient streaming register files. These register files are small, so their controller overhead could be large, especially since our CGRA lacks instruction-based methods to embed them. Further, given the complex access patterns needed for DNN applications, both for address generation and schedule generation, we also look at options to optimize the controller effectively, such that it can support the needed access patterns while being area-efficient. Furthermore, we evaluate the number of ports needed for these register files and how their sizes can affect how well different schedules can be mapped. By implementing these local streaming register files closer to the ALU, we can improve the overall energy efficiency of the convolution layers.

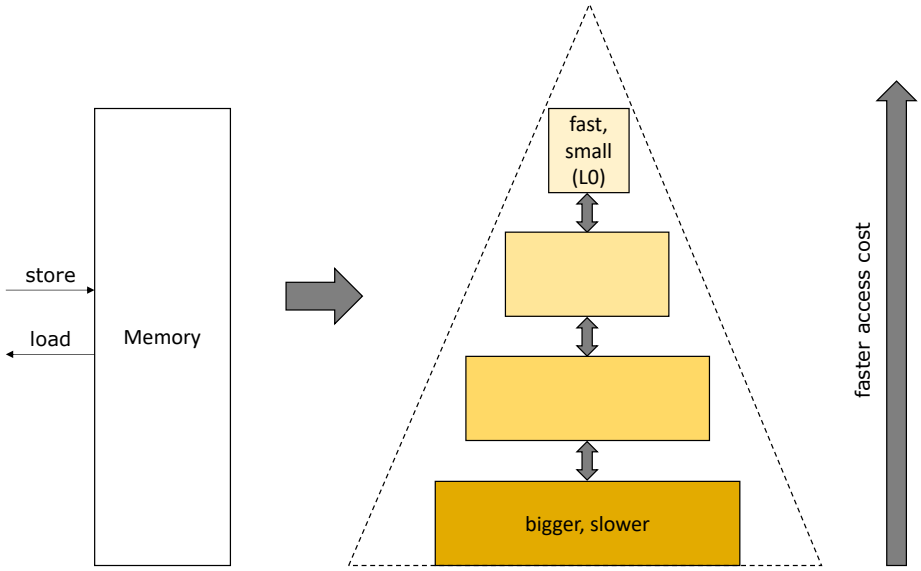


Figure 3.1: Pyramid shape multi-level memory hierarchy system. The levels closer to the processor are small and fast, with low access cost. Further from the processor, the memories become bigger and slower.

## 3.1 Memory Hierarchies

Modern day computer systems have a multi-level memory hierarchy [67, 71] due to the limitations of achieving a single, very large and fast memory next to the processor as assumed by conventional programming languages. In practice, both a very large and fast memory cannot be achieved with a single level of memory since larger memories are slower [31]. Fig. 3.1 illustrates the difference between the programmer's view of memory and how it is actually implemented in the system. The memory is built with multiple levels of storage, where each level holds differently sized data sets, and the levels become progressively bigger and slower further from the processor.

Fig. 3.2 shows the memory cost for 16-bit wide accesses to small register files sized 16B-512B, which reveals that, as the register file size increases, so does the cost to access data from it. The access costs were generated from placing and routing different sized register files which ran at the same clock frequency (900 MHz), and extracting the access energy cost. As can be seen, as the register file size increases, so does the cost to access data from it. Even for large SRAMs of size 32KB-512KB, the access cost increases with the size, but the change is much smaller, on the order of 1.5-2x [39]. Although small memories help contain the access costs, their limited storage results in data being purged and reloaded from larger, more energy-intensive memories many times during the program execution.

To minimize effective access time and energy costs, the sizes of different memories in the hierarchy

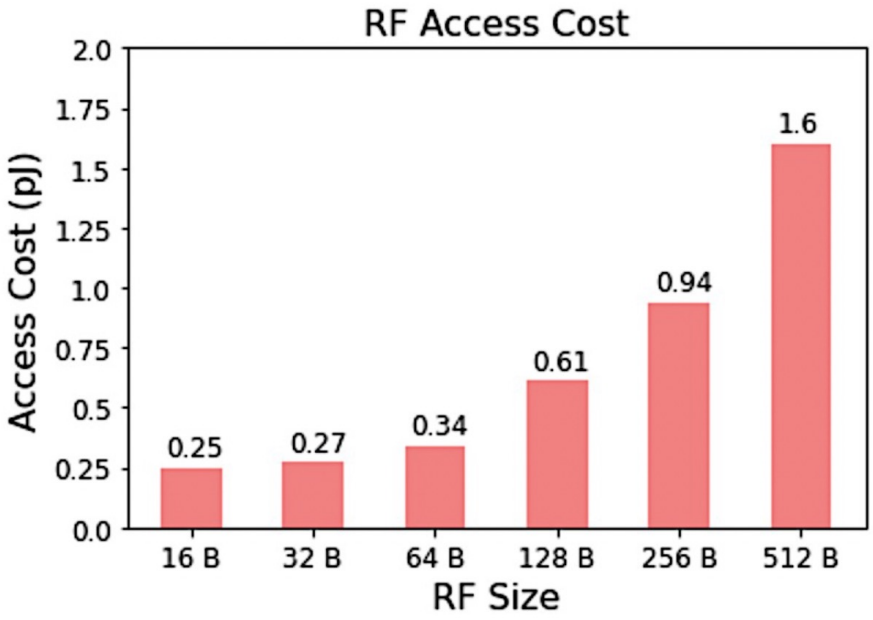


Figure 3.2: Access cost (16-bit) for the register file (16 nm) increases as its size increases [87]. While the use of smaller memories helps lower the access cost, it also results in data being purged out of the memory before it is completely reused, thus increasing the number of accesses to the main memory.

are chosen to balance the energy cost of each access with the overhead caused by each re-fetch [70]. Hardware techniques [38, 79] are used to determine what data will be used and accordingly ensure that most of the data the processor needs is kept in the fastest level closest to the processing element (PE). To reduce the overall energy costs, one would ideally want to schedule the computation such that most fetches by the processing engine are from the nearer, smaller memories. Therefore, memory closest to the processor is typically implemented with small register files (labeled L0 in Fig. 3.1). To optimize the use of these registers, they are an explicit part of a CPU's architecture. Each instruction specifies which register values will be fetched and updated. Load and store instructions are used to send the addresses to the memory. The sequencing of these memory instructions is controlled by instruction sequence instructions like branch and jump. In this way, instructions direct the movement of data from the registers and memory in these machines. Consequently, a considerable amount of the bits in an instruction are dedicated to addressing the register files. Overall, the memory hierarchy in modern computer systems is designed to optimize access time and energy costs by using smaller, faster memories closer to the processor and progressively larger and slower memories further away from it.

In contrast to traditional computing platforms such as CPUs and GPUs, spatial architectures such as CGRAs operate differently. Some CGRAs consist of arrays of simple processors that fetch

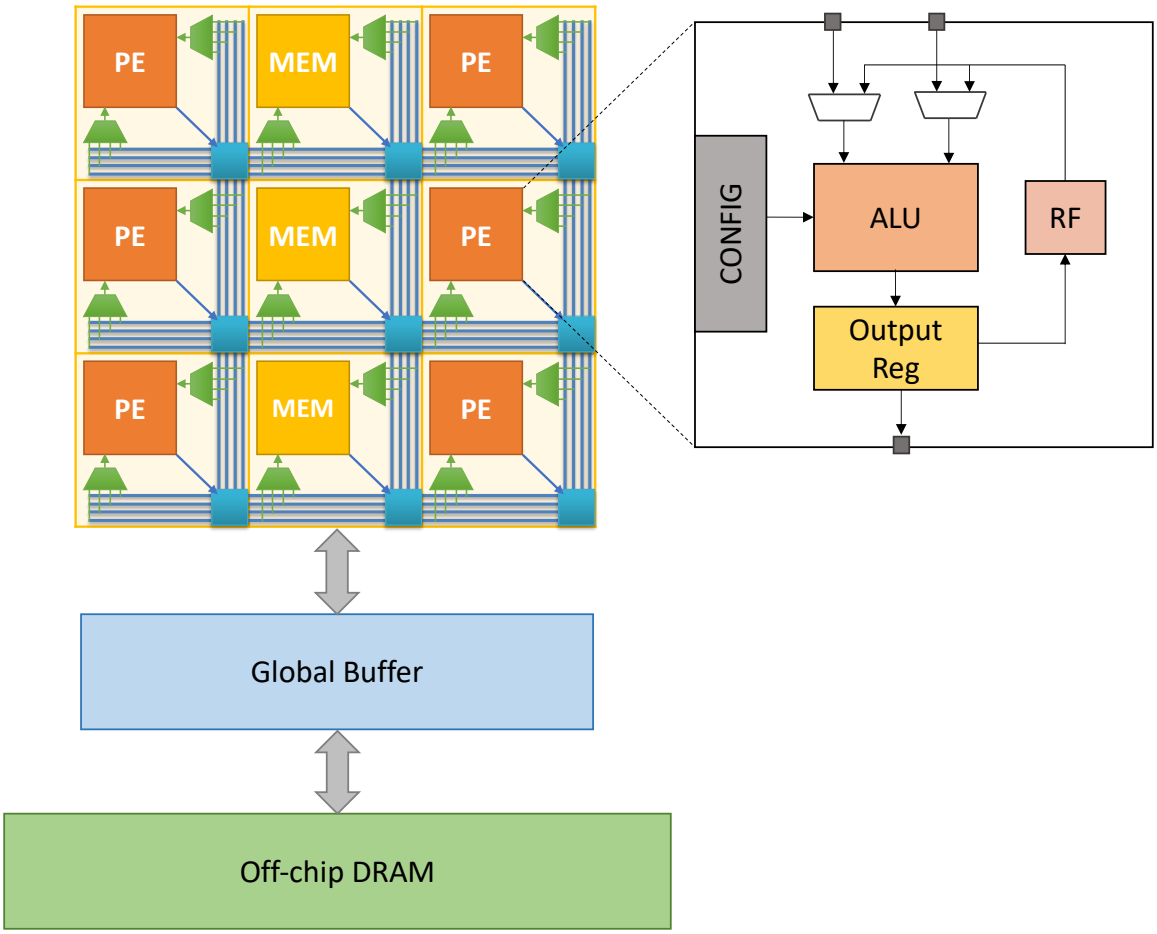


Figure 3.3: Memory hierarchy in a CGRA with a general structure of a register file in a CGRA PE tile.

instructions, while others utilize static configurations instead of instructions. In these systems, programming occurs in space rather than time. Computation kernels are designed to process input data streams and produce output data streams. However, generating input data streams and storing output data streams requires additional hardware to generate the necessary memory addresses. This addressing logic incurs overhead, which becomes problematic for small memories such as register files, the area of which is only a fraction of a PE.

Register files have been a common feature in the processing elements of CGRAs in the past, as documented in various studies [6, 41, 56, 69]. Fig. 3.3 demonstrates an example of how register files can be introduced into the dataflow of a typical CGRA fabric. The ALU in the processing element can fetch data from the register file, and the output of the ALU can be written back to the register file. However, controlling the addressing of the register files is not shown in the figure. As already mentioned, one approach is to continue using instructions. In NeuroCGRA [37], each element is

```
1 for b = 0 : B − 1 do                                              (Batch)
2   for k = 0 : K − 1 do                                            (Output channel)
3     for c = 0 : C − 1 do                                          (Input channel)
4       for y = 0 : Y − 1 do                                        (fmap height)
5         for x = 0 : X − 1 do                                      (fmap width)
6           for f_y = 0 : F_y − 1 do                                (Filter height)
7             for f_x = 0 : F_x − 1 do                              (Filter width)
8               O[b][k][y][x] +=
                    I[b][c][y + f_y][x + f_x] × W[k][c][f_y][f_x]
```

**Algorithm 1:** The seven nested loops of the convolutional (CONV) layer. In this simple example, the loop order is represented by $F_xF_yXYCKB$, from innermost to outermost.

similar to a tiny processor that includes register files, morphable Data Path Units (DPU), Switch Boxes (SB), and sequencers. The register file stores data for the DPUs, which are the functional units for the CGRA. The sequencer controls the addressing of the register file using a sequence of instructions, comparable to the addressing of a CPU.

Without instructions, memory and register file addresses must be created by explicit address generators. This dedicated logic can lead to high area overhead, which depends on its functionality. In a study by Wijerathne et al. [85], using the PEs to create the address generators led to an overhead ranging from 20-80%, depending on the type of memory. They hence introduced a customized address generator, but it accounted for 59% more area. However, building the address generator in the PE is not feasible if one wants to have a useful register file for each PE. Since the storage of these register files tends to be small, the address controller of these memories can dominate the overall area of the register files even when one builds custom hardware for the address generation.

Many CGRA designs support only simple access patterns for register files to minimize the overhead of address generation logic. For instance, Fan's CGRA [19] only supports a fixed step sequential mode in one dimension, where data is read from or written to the start address continuously until the end of the address, with a given step size. However, such simplified access patterns are inadequate for handling the complexity of Deep Neural Network (DNN) algorithms, which we will discuss in the following section.

## 3.2 Deep Neural Networks (DNN) Algorithms

Deep Neural Networks (DNNs) [9] are a subset of Machine Learning used for various applications such as image classification, face recognition, and autonomous driving [12, 27, 32]. In addition to the accuracy of these applications, their implementations must also be energy efficient [81]. We look at Convolutional Neural Networks (CNNs) [2, 65], a type of DNN used for image processing that has more complex sequences. As seen in Fig. 3.4, the convolution layer (CONV), which detects the

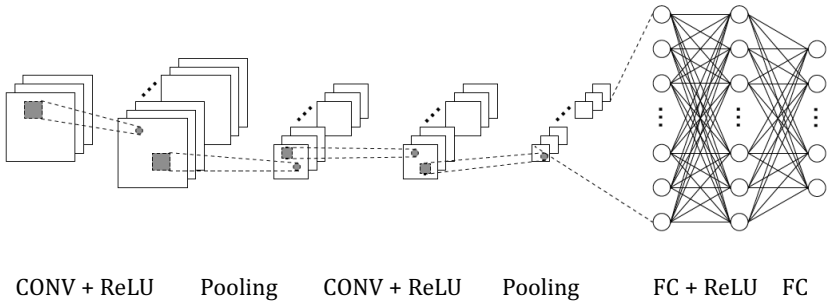


Figure 3.4: Convolutional Neural Networks (CNNs) start with a convolution layer, followed by a non-linear activation function such as ReLU or sigmoid, and then a pooling operation. Typically, a CNN network consists of multiple stacks of convolution, non-linear activation, and pooling layers. The last layer of a neural network is the fully-connected layer, which applies a linear combination to the input values and then possibly an activation function.

presence of a set of features in the input images, is the first layer in CNN. It is followed by a non-linear activation function like ReLU or sigmoid and a pooling operation to reduce dimensionality. Multiple stacks of convolution, non-linear activation, and pooling layers are typically part of the CNN network. The fully-connected layer is the last layer of a neural network, which applies a linear combination and optionally an activation function to the input values received.

In the computation of the convolution layer, a set of input feature maps, which can be thought of as different channels of the image, and a set of filters are used, as seen in Fig. 3.5 which has two sets. Each filter has the same number of channels as the inputs (ifmaps). One can perform a 2D convolution between each input channel and the corresponding filter channel. After accumulation, each filter yields the output for one output channel (ofmap). Multiple ofmaps are generated by mapping all the filters over the ifmaps. To increase efficiency in the fully connected layers, multiple inputs can also be batched together. This enables the computation to be performed simultaneously over multiple inputs which helps amortize the weight accesses by performing the computation on multiple input images simultaneously (Fig. 3.6).

The convolution layer computation in the CNN can be represented as seven levels of nested loops, as seen in Algorithm. 1. These nested loops generate the output feature maps `O`, which have `K` channels of images of size `X` x `Y` by processing the input fmaps `I` of `C` channels. The ifmap data is processed into smaller batches `B` that can increase parallelism and data reuse. The `W` contains the weights for the `K` filters, each of size `C x Fx x Fy`.

To evaluate the complexity of the loop nest, we first focus on the convolution layer. Additionally, we aim to ensure that the controller support is generic and can be extended to other types of convolution layers. To achieve this, we analyze a few cases of convolution:

- **Strided Convolution:** Stride refers to the distance between spatial locations where the

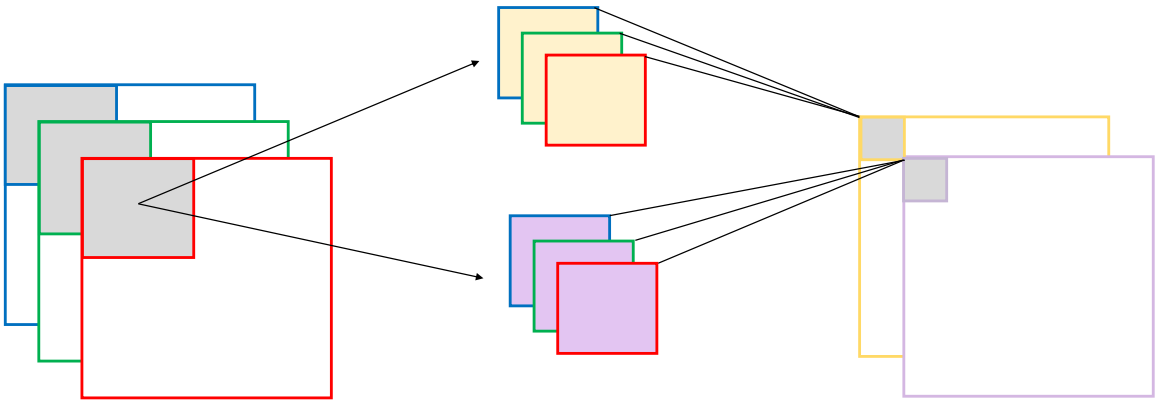

Figure 3.5: In convolution computation, a 2D convolution is performed between each input channel and the corresponding filter channel, which is accumulated to generate the output for one output channel. Multiple outputs can be generated by mapping all the filters over the ifmap. In this figure, the number of input channels(C) is 3, and the number of outputs generated (K) is 2.

convolution filter kernel is applied. In standard convolution, this value is 1, meaning that the kernel is moved one pixel at a time. Strided convolution [43], on the other hand, takes in a stride value greater than 1. This results in downsampling and reduces the amount of information retrieved. However, if the model's accuracy can be maintained, strided convolution can also reduce the computation involved. Supporting strides greater than 1 requires support for strided access patterns, as opposed to contiguous access patterns.

- **Dilated Convolution:** Dilated convolution [89] differs from standard convolution in that the input pixels are not necessarily consumed contiguously. For example, dilation of 1 (standard convolution is with dilation 0), convolves the kernel weights with every other input pixel. Dilation allows spatial information to be merged across the inputs much more aggressively in fewer computations and helps the receptive field grow more quickly. To support dilated convolution, strided accesses of the inputs are needed in both the X and Y dimensions of the convolution inputs, as opposed to contiguous access patterns used in standard convolution.

- **Depthwise Separable Convolution**: To reduce the computational complexity of standard convolutions, depthwise separable convolution [15] splits the computation into two convolutions: depthwise convolution and pointwise convolution. Depthwise convolution convolves the input and filters without accumulation, unlike normal convolution which performs accumulation across input channels. The result of depthwise convolution is an output of size `C x Y x X`, obtained by convolving `C` filters of size `1 x Fy x Fx` with the input. Since this computation is a subset of normal convolution, it doesn't need to be analyzed separately. Pointwise convolution, on the other hand, uses a 1x1 kernel that iterates through every single point, with the same depth as the number of channels in the input. A single pointwise convolution

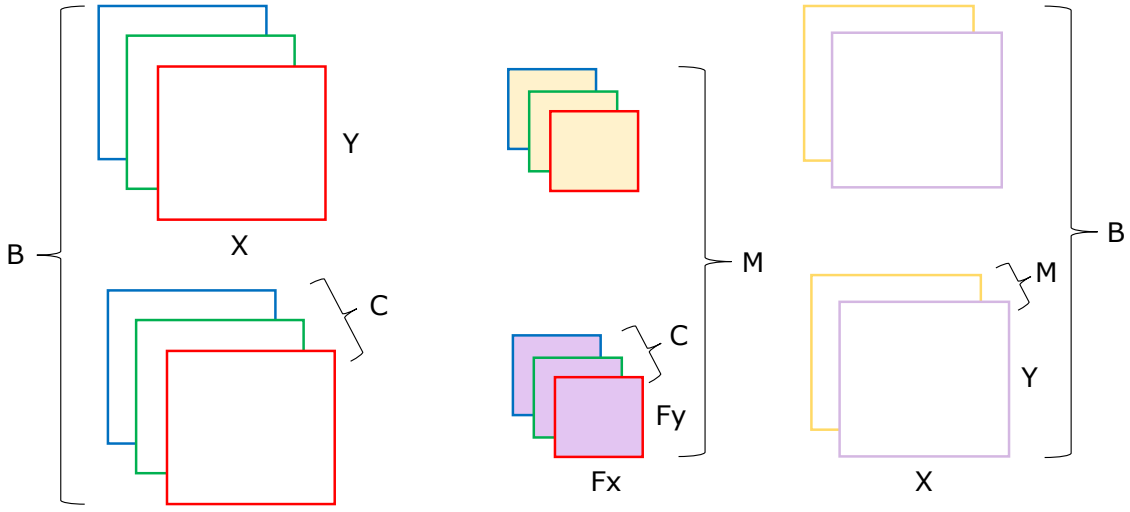


Figure 3.6: To amortize accesses across inputs, multiple inputs can be batched together, allowing for simultaneous computation over all inputs. This approach helps improve computational efficiency when working with multiple input images.

with kernel size `C x 1 x 1` results in an output of size `1 x Y x X`, and multiple such kernels (`K`) will result in an output of size `K x Y x X`. However, the kernel size of `1 x 1` in pointwise convolution limits the locality and reuse in the operation, making it necessary to evaluate the pointwise convolution part of the depthwise separable convolutions.

In addition to the convolution layer, it is important to also consider the fully connected layer, which can be represented as a matrix-vector multiplication and becomes GEMM when the computation is blocked. This multiplication can also be viewed as a convolution layer, as shown in Fig. 3.7.

To evaluate the complexity of supporting convolution inside the ponds, we analyzed important convolution cases and their loop nest complexity. Moreover, it is worth noting that different dataflows may require different access patterns, as dataflow categorizes each schedule based on the type of data block kept stationary for reuse across computations. Hence, we also evaluated several commonly used dataflows:

- **Weight Stationary Dataflow:** The weight stationary dataflow is a technique in which the kernel weights are held stationary in the register files, while the operations are mapped on the PE such that the ones that use the same weights are scheduled together. This approach enables reuse across kernel weights until they need to be purged out for the next set of computations. This dataflow maximizes weight reuse, thus minimizing the memory access costs for fetching the weights. Two implementations are possible for weight stationary dataflow. The first option stores each weight in the register file of each PE, with weights from different filters being stored inside PEs at different columns, while weights from different channels are distributed at

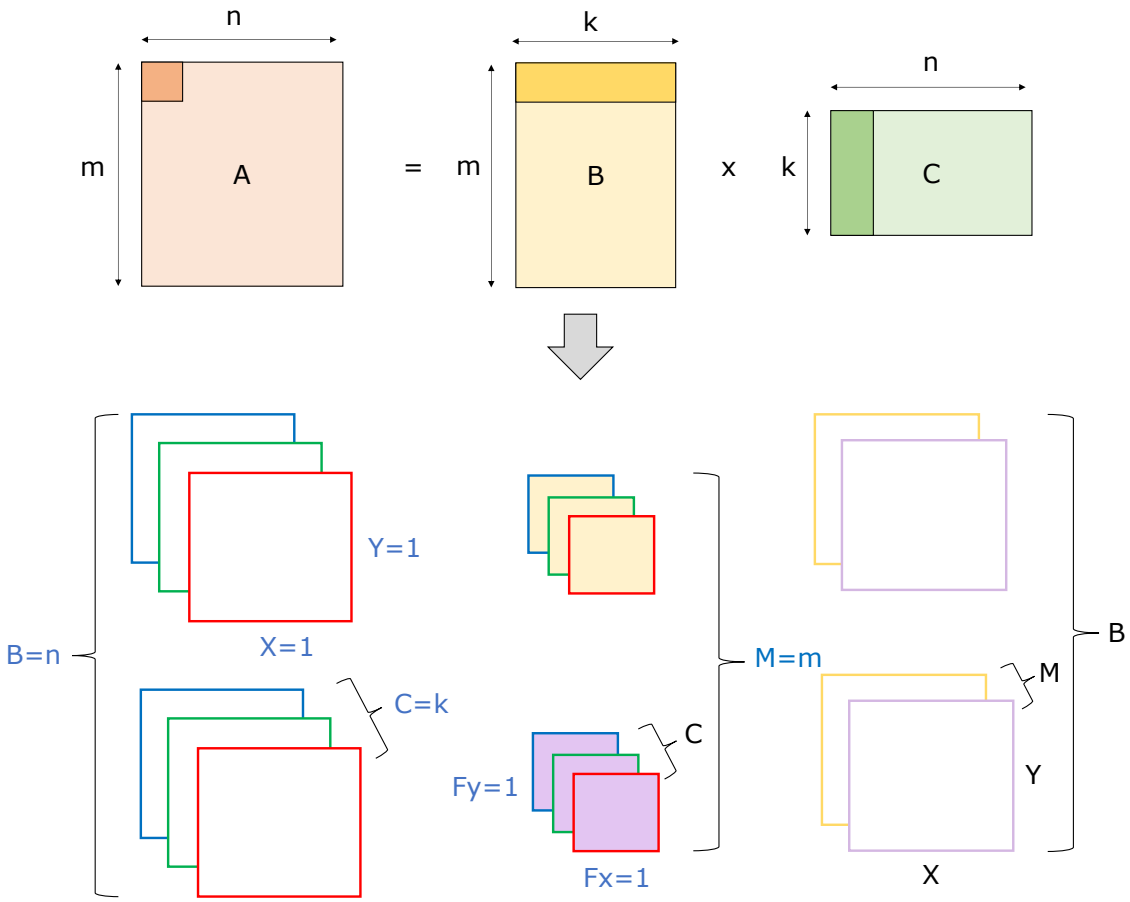


Figure 3.7: GEMM represented as CONV with Fx=Fy=X=Y=1, C=k, B=n and M=m.

different rows, as seen in [40,90]. This particular weight stationary dataflow can be regarded as unrolling and spatially mapping input (C) and output channel (K) dimensions of the filters on the 2D PE array. This weight stationary implementation is commonly used as it is generally flexible enough to support both CONV and FC layers. An alternative is to unroll the filter width and height (Fx and Fy) on the PEs and remain stationary inside the register file, as seen in [72,80]. A new set of corresponding inputs are fetched and mapped on the 2D PE array for each iteration. The inputs are then multiplied with the corresponding weights to accumulate to the output.

- **Output Stationary Dataflow:** Output stationary dataflow is a type of dataflow where the feature map dimensions, width (X) and height (Y), are unrolled spatially. This allows a region of the output to be mapped onto the PE array, and the partial sums are held stationary in each PE's register files for reuse to support local accumulation, as seen in [16,68]. This dataflow

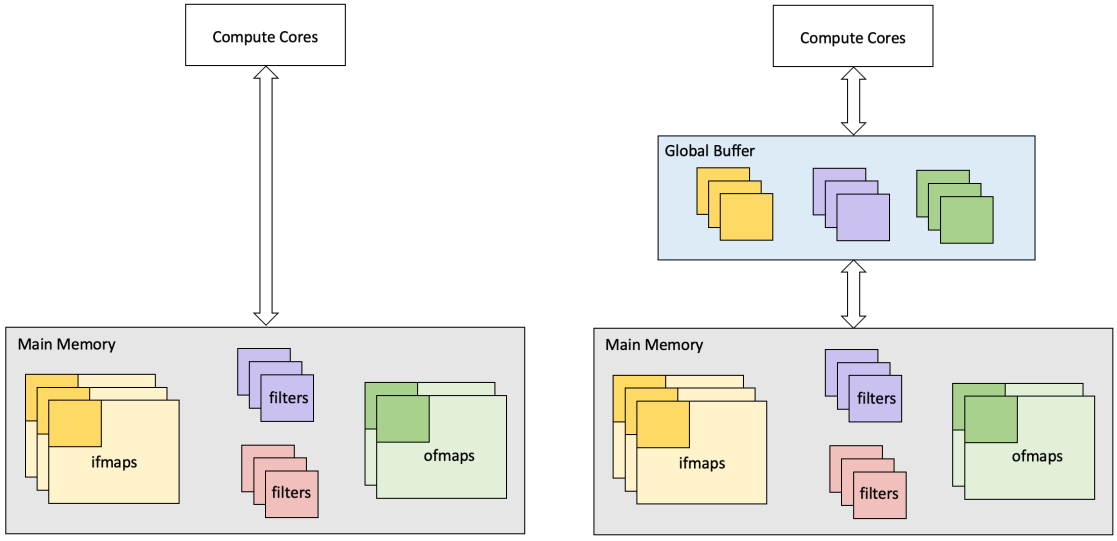


Figure 3.8: Naively, one can fetch data directly from the main expensive large memories. For every CONV computation, it involves two reads and one write, resulting in a large number of memory references and high energy consumption (left). Alternatively, the loops of the computation can be blocked by tiling and re-ordering the loops such that the large data blocks can be broken into many smaller blocks that can be buffered and re-used inside a global buffer (right).

maximizes output reuse, as all operations that contribute to the same output are mapped to the same PE for sequential processing. By minimizing the energy consumption of fetching the partial sums, this dataflow helps reduce memory usage.

To optimize the performance of the CGRA over different convolutions and dataflows, various loop scheduling and memory hierarchies can be explored. This rest of the chapter delves into these optimizations in more detail.

## 3.3 Evaluating DNN Access Schedules

To optimize the performance of CNN computations, it is important to identify the access patterns that arise from optimally scheduled computations. As seen in Fig. 3.8 (left), naively, data can be fetched directly from main memory, which would require two reads and one write for every arithmetic operation in the context of a convolution operation. However, a multi-level memory hierarchy system can be used to reduce memory energy for this operation, similar to how it is done with processors. This is discussed further in Section 3.1.

Loop blocking is a common optimization technique that is used to enhance the re-use and locality in multi-level memory hierarchy systems [10, 60, 86, 88]. Loop blocking is especially important in the

context of DNNs. By properly tiling and re-ordering the loops, the large data blocks can be broken down into smaller chunks that can be buffered in less expensive and smaller streaming register files that are closer to the ALU. This technique minimizes the overall memory overhead. Optimizing the locality for the smaller memories is important for many layers, such as the convolution and fully-connected layers, which have a high computational intensity. The convolution layer has high locality, and the inputs, weights and outputs, are re-used multiple times:

- **Inputs** are re-used since the same inputs are fetched to convolve with multiple output channels.
- **Weights** are re-used as part of the sliding window computation. Once the weights are fetched, they can be reused across the entire input.
- **Outputs** are re-used for accumulation and fetched as input to the next layer.

In addition, due to the significant parallelism involved in the convolution layer, where multiple filters can be run simultaneously over the input to produce the output and multiple inputs can be processed in parallel in batches, an efficient implementation can significantly benefit from proper loop blocking. This can be achieved by tiling and re-ordering the loops, breaking down the large data blocks into smaller ones that can be buffered and reused inside a large SRAM on the accelerator (called the global buffer in the figure). This approach is shown in Fig. 3.8 on the right. By tiling the data, most of the accesses can be served from the less expensive, smaller memory closer to the computation, minimizing the overall memory overhead. Refills to this memory will come from the next larger memory in the hierarchy. To strike a balance between data reuse and energy cost for data accesses, it is essential to block the computation appropriately. The following section will discuss the specifics of how loop nests should be blocked for DNN computation and the number of loop levels that need to be supported.

### 3.3.1 Interstellar Framework

To systematically evaluate the memory design space of DNNs, and choose the best blocking scheme, the Interstelllar framework [87] is utilized. This framework leverages Halide's [1] scheduling language to analyze DNN accelerators, where the Halide algorithm and schedule are compiled to low-level IR to map the applications on FPGA and ASIC designs. Fig. 3.9 illustrates the flow used to evaluate the DNN schedules. The Interstellar framework has an estimation model to evaluate the design space of DNN accelerators, with a schedule optimizer to find optimal schedules for DNNs. The schedule optimizer has two engines: one for cost analysis and the other for schedule generation. The schedule generator iterates over all possible schedule candidates, and the cost analysis engine consumes the iterated schedule to calculate the overall energy of the schedule. This scheme enables an exhaustive search for the most energy-efficient schedule for a given configuration, returning the optimal energy and performance for the given configuration and the optimal schedule. The Interstellar framework operates with a number of input configurations:

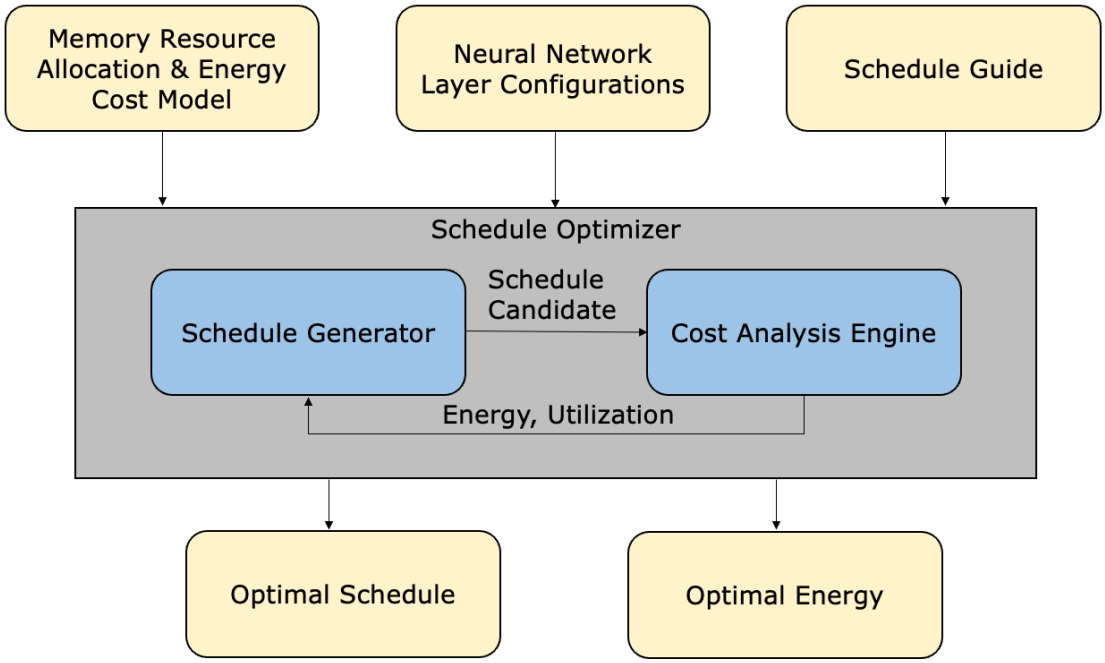


Figure 3.9: The scheduler optimizer reports optimal schedule with its energy and performance. Yellow boxes are the inputs and outputs of the framework. The schedule generator generates schedule candidates and sends them to the cost analysis engine for energy and performance evaluation.

**Memory Resource Allocation & Energy Cost Model:** The memory resource allocation and energy cost model describes the memory resource allocation, including the number of memory levels, their sizes, and the number of parallel compute units available for spatially mapping the application. It also specifies whether a given memory resource is shared across the inputs, weights, and outputs or have dedicated memory resources. For example, a streaming register file in a PE can be used to store all or some of the inputs, weights, and outputs, or each can have their dedicated memory resource. Additionally, energy per access (pJ) is provided for each memory hierarchy level.

**Neural Network (NN) Layer Configurations:** The NN layer configurations are input to the schedule optimizer through a configuration file which includes the dimensions of ifmaps (`fmap_width`, `fmap_height`), input and output channel (`input_fmap_channel`, `output_fmap_channel`), filter sizes (`filter_width, filter_height`), stride (`stride_width`, `stride_height`) and batch (`batch_size`). As discussed in Section 3.2, for the convolution layers, the ifmaps of the `input_fmap_channel` are convolved with the filters, each of size `filter_width` x `filter_height`, to produce ofmaps of `ofmap_fmap_channel`, each of size `fmap_width` x `fmap_height`. The configuration for the convolution layer can also be extended to the fully connected layer of DNN, as seen in Fig. 3.7.

**Schedule Guide:** The schedule optimizer can explore the design space and provide the best schedule and its corresponding energy based on the configurations provided above. However, the search space can become large, which increases the overall runtime of the search. To restrict this search space and make the search more efficient, a schedule guide can be used. The schedule guide indicates the type of dataflow by specifying which dimensions of the convolution layer are unrolled. For example, if the schedule is restricted to be output stationary, the output width and height are unrolled, and the unroll factor is provided as an input. With a PE array of 16x16 processing elements, the schedule lays out a 16x16 output tile, and the register file inside each PE buffers one output pixel. Different schedules, such as weight stationary, can also be evaluated by providing the schedule guide. The next section covers the details of these schedules. The objective is to evaluate the optimal schedules for various CNN layers to determine the different types of schedules that our controller should support.

### 3.3.2 Loop Complexity

To determine the capabilities required for the address controller, this section reviews the loop nests provided by Interstellar to achieve optimal blocking. We evaluate the following configurations, which were discussed in Section 3.2:

- **Neural Network Layer Configuration**
  - Convolution Layer
    - Strided CONV (stride greater than 1)
    - Dilated CONV
    - Pointwise CONV (Fx = Fy = 1)
  - Fully Connected Layer
    - Batch Size = 16 and 32
- **Dataflow**
  - Output Stationary (unroll OX and OY)
  - Weight Stationary (unroll IC and OC or Fx and Fy)

For a representative CONV layer with filter sizes of 5x5, Fig. 3.10 illustrates the pond-level loop blocking for weight stationary dataflow with input and output channels in the innermost loops. In this configuration, each PE handles 4 input channels and generates 8 output channels, assuming a 32-entry pond is used for inputs, weights and outputs each. The computation schedule for CNN loops blocked in the ponds can be seen in the figure. The schedule reuses the weights to generate a new set of outputs when a new block of input is fetched. As seen in Fig. 3.10, in this specific

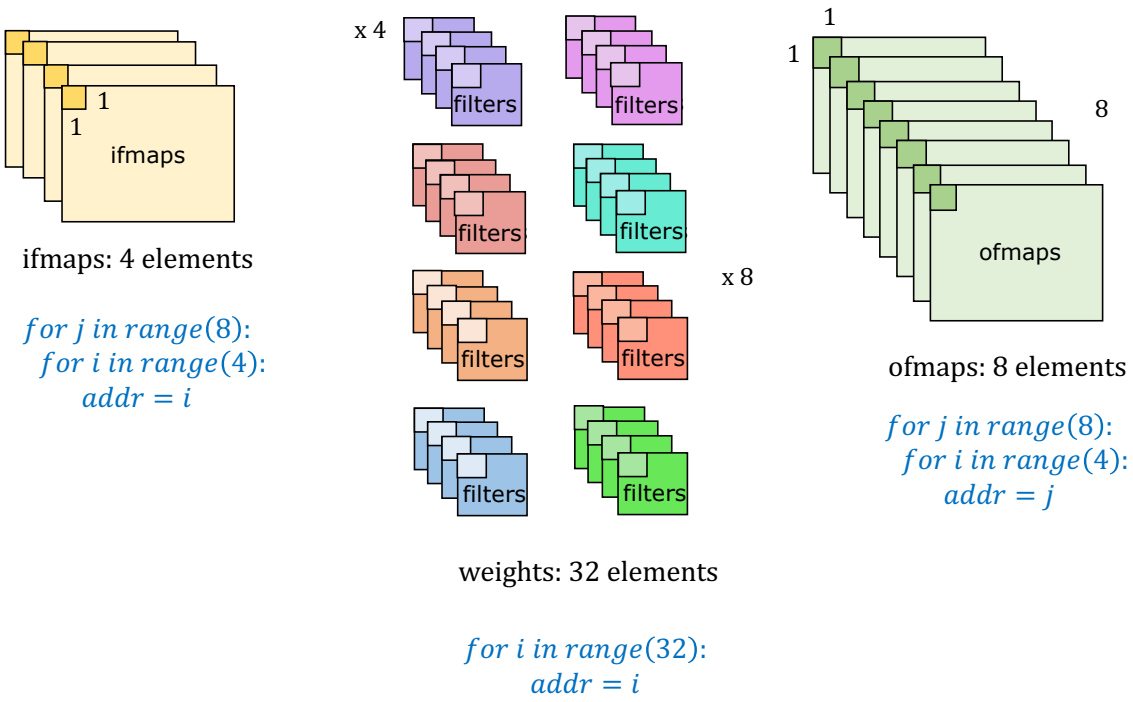


Figure 3.10: Blocked DNN computation schedule for weight stationary dataflow with input and output channels in the innermost loop. Assuming a 32-entry pond is used for each input, weight and output, the weight pond fully utilizes the pond by storing 32 weights. To generate the outputs, each input from 4 channels is fetched 8 times to convolve with the corresponding weights to generate the 8 outputs. The loop nests are as described in the figure.

weight stationary schedule, 4 elements per output kernel are loaded into the weight pond for 8 such output kernels, fully utilizing the weight pond by storing these 32 weight elements. Each input from 4 channels is fetched 8 times to convolve with each weight kernel to generate the 8 outputs. The loop nests for the inputs, weights and outputs are shown in the figure, assuming the starting address is 0. As a result, for the input pond, address 0-3 is fetched 8 times, once for each weight kernel. Notice that we can fuse loops when the resulting address pattern doesn't depend on the loop structure. Thus, although the weight accesses are incremented first over input channels and then output channels, we represent this as a single fused loop: address 0-31 is fetched linearly from the weight pond to perform convolution with the input. Finally, for the output pond, address 0 is written and read 4 times to perform the accumulation, and the same pattern is repeated for each of the other 7 outputs.

Fig. 3.11 shows the pond-level loop blocking for output stationary dataflow for the same CONV layer. In this dataflow, the lower level loops correspond to the 'x' and 'y' image coordinates, followed by the output channel loop. Similar as before, using a 32-entry pond for each input, weight, and

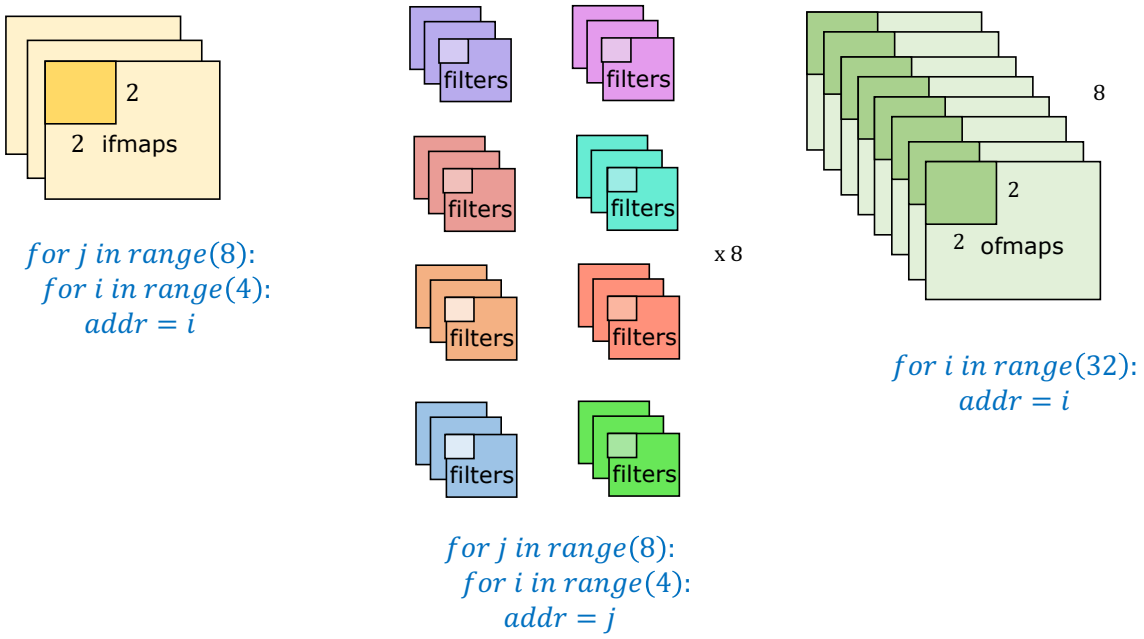


Figure 3.11: Blocked DNN computation schedule for output stationary dataflow. To generate the 32 outputs, each weight value from one channel for 8 such output channels is fetched 4 times, one for each input, to generate the output. The loop nests are as described in the figure.

output, the optimal blocking strategy fully utilizes the output pond by storing 32 outputs. The input pond stores 4 values and the weight pond stores 8. To generate the outputs, a weight value from each of the 8 output kernels is fetched 4 times, one for each input. The 'x' and 'y' image loops are fused in this blocking, and the data is stored sequentially. The weight pond reads one weight address 4 times and reads 8 such addresses, as shown in the figure. This address pattern can also be managed by a two-level loop controller.

Finally, Fig. 3.12 shows the pond-level loop blocking for weight stationary dataflow for the same CONV layer with filter width and height as the innermost loops. The kernel width and height are 5 each, and the pond has 32 entries, which can hold the weights and input data to compute a single partial output. To generate one output, 25 elements of the inputs (5 each along the X and Y dimension) stored in the input pond are convolved with the 25 weight elements to generate one output. The loop nests are shown in the figure.[1]

We performed a similar analysis for the other CONV layer variants for different dataflows, as well as explored FC layers. Our findings show that for all of these different layers, a 2-level affine access pattern is both necessary and sufficient for address generation. This two-level affine pattern

[1] As we will see a little later when we talk about how the data is loaded into the ponds, for this loop ordering it is beneficial to leave the x and y image loops distinct rather than fusing them together. This enables loading only the next input column to produce the next output pixel, rather than the complete 5x5 pixel map.

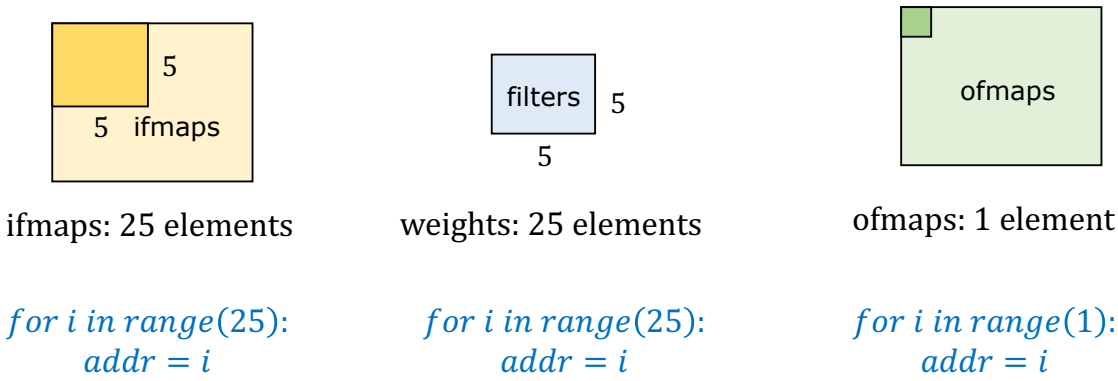


Figure 3.12: Blocked DNN computation schedule for weight stationary dataflow with kernel width and height unrolled. 25 elements of the inputs (5 each along the X and Y dimension) stored in input pond are convolved with the 25 weight elements to generate one output. To generate the next output, only part of the input needs to be re-fetched due to the sliding window pattern of the input. This results in a piece-wise affine access pattern.

$$
\begin{aligned}
&for\ j\ in\ range(range1):\\
&\quad for\ i\ in\ range(range0):\\
&\qquad addr = j\ * stride1 + i\ * stride0 + offset
\end{aligned}
$$

Figure 3.13: Two-level affine access pattern loop nest is needed for address generation.

can be represented as shown in Fig. 3.13, with the addresses loop around each time a new block of data arrives.

### 3.3.3 Loading and Spilling Ponds

As seen from the loop nests in Algorithm 1, the CONV computation involves two inputs: the image input and the kernel weights. If separate register files are used for the inputs, weights, and outputs during the computation phase, it appears that the input and weights register file needs to support a read-only access pattern while the output register file needs to support both read and write access.

In addition to these ports needed for the computation loop, when the application is blocked, the loading of a new memory block while the current block is being executed also needs to be accounted for. As a result, when data is being read for execution, the new block of data needs to be simultaneously loaded in. Thus even though from the computation perspective, the storage for inputs and weights appears to be read-only, it also needs a write port to write the next block of data while the current data is being read for computation. Similarly, the storage for output data which needs a read and write port for computation needs another set of write and read ports to initialize the storage and read the updated block after the computation is completed to write the data to the next level of memory. To remove the need for additional physical ports on the pond, even if one is willing to stall the computation when these load/spill memory operations occur, this

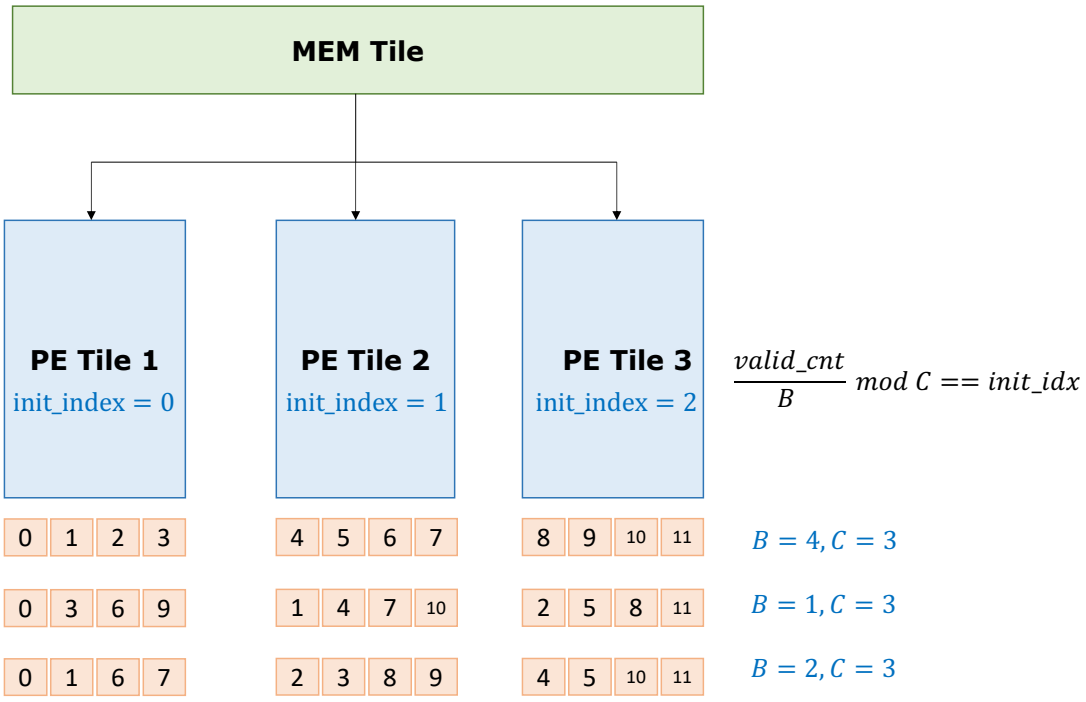


Figure 3.14: Multi-cast support. A memory tile can drive multiple ponds at a time. This memory tile will generate a start signal and a valid signal, common for all the ponds that it is connected to. By appropriately programming the pond count (C), block size (B) and index (init_idx), each pond in the PE tile can identify data corresponding to which valid signals should be captured.

address sequence still needs to be generated. While both sequences are affine with only two or less loop levels, the combination will be piece-wise affine, which requires a more complex controller.

This need to load the register files creates a complication in their control logic, especially when the total number of ponds exceeds the number of memory tiles. As a result, these controllers need to support the situation where one memory tile feeds data to multiple ponds, which is often referred to as multi-casting. As shown in Fig. 3.14, in this situation, the memory tile streams out valid data, and the ponds must be programmed to store the correct data. As seen in the figure, the memory tile streams data0-data11. Sometimes, the first block of data (data0-data3) would be loaded into the "first" pond, the second block going to the second pond, and so on. Alternatively, the data might be interleaved, so that the first pond accepts every $n^{th}$ data (data0,data3,data6,data9). Such patterns can be generated by configuring 3 parameters to generate when the pond should be enabled, as shown in 3.14. It shows the stored data patterns with pond count (C) of 3 and block sizes (B) of 4, 1 and 2, respectively. Depending on the `init_index` programming, each pond in the PE tile would recognize data corresponding to which of the valid signals should be captured.

To efficiently support the analyzed DNN access schedules and the necessary loop nest complexities

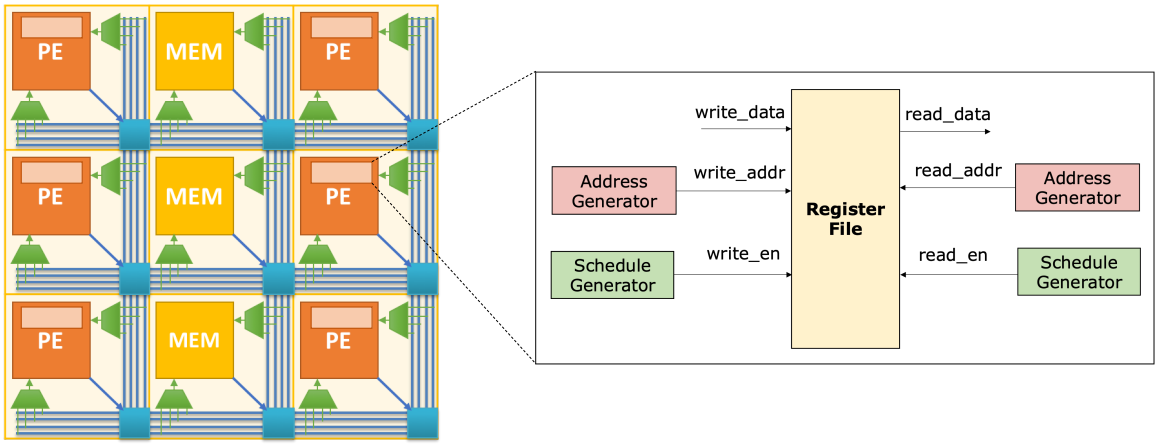


Figure 3.15: A streaming register file, or *pond*, to be introduced in every PE tile. The basic blocks consist of a flop-based storage component along with address and control elements that drive the storage inputs to provide the address access pattern to read to/write from the register file.

for computation, loading, and spilling ponds, the next step is to examine the architecture of the pond controller.

## 3.4 Pond Controller

To meet the requirements of the described applications, we utilize a basic streaming register file architecture, as depicted in Fig. 3.15. The architecture is comprised of two key components:

- **Storage**: The storage component is implemented as a flop-based register file, which is the most efficient implementation for small memories due to its low area and energy requirements. Since these memories are distributed across every PE and require extremely low access cost, flop-based storage is the optimal choice for this new memory hierarchy.
- **Address and Control (Schedule) Elements**: The storage has inputs that drive controls determining where the data is written to/read from (`read/write_addr`) and when the data is written/read (`write/read_en`). The address and schedule generators drive the address and the enable signal to the storage, respectively. Together, the address and schedule generators comprise the controller logic for the register files.

Not shown in Fig. 3.15 are the loop nest counters that inform the address and schedule generators of the current loop position. We refer to these counters as the iteration domain (ID). Fig. 3.16 shows the interaction between the iteration domain, address generator, and the schedule generator for a sample two-level affine function. The domain is defined by the bounds of loops in the loop nest surrounding the statement. The address generator implements the part of the code that maps from

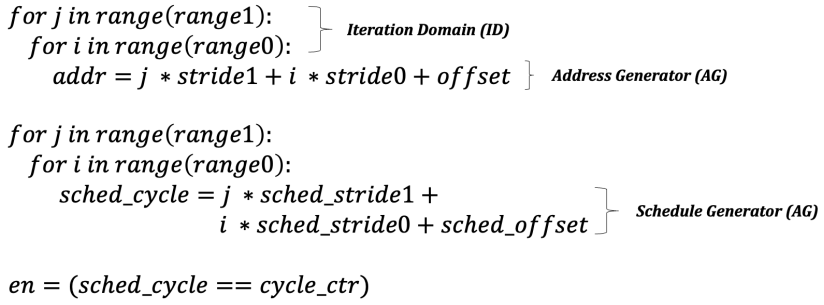


Figure 3.16: Interaction between the Iteration Domain (ID), Address Generator (AG) and Schedule Generator (SG) components. ID refers to the statement instances in the application code that use the port for the read or a write operation. The address generator refers to the part of the code that implements the mapping logic from an ID to a physical address. The schedule generator orchestrates the flow of data. It generates the schedule of all the operations in the iteration domain.

an ID to a physical address. The schedule generator orchestrates the flow of data. Since we compile statically analyzable applications, we use our compiler to determine when each operation occurs. Using this information the schedule generator specifies the number of unstalled cycles between reset and when each operation occurs which is also an affine expression of the ID. Fig. 3.17 shows the mapping of SG, AG and ID to the basic register file shown in Fig. 3.15. While these affine expressions support multiplication by a stride, they can be converted to an iteration expression which can be implemented by a simple adder.

Our first implementation of a pond is shown in Fig. 3.17 and has one read port and one write port each with a two-level affine controller. It can be used for input and weight pond in which data is first loaded and then fetched. Fig. 3.18 shows the relative pond area for a register file with controller supporting affine access pattern loop nests shown in Fig. 3.16 with a single read and write port. The area is evaluated for a 64B pond with different loop dimensions. The controller can have large overhead, especially as the loop structures become more complex (39%-75%) Our analysis of the controller logic area showed that the majority of the area was attributed to the schedule generation block (~60% for a 2-level affine access pattern with 16 bits for range, stride and offset). Even though the loop nest size for the address generation and schedule generation blocks are the same, the bitwidths needed for the schedule generation block are wider, and the schedule generator also needs to maintain a count of the clock cycle to determine when to initiate the operations. Furthermore, in the initial design, the ponds were restarted only when the memory tiles were reloaded which meant that the schedule generator needed enough bits to count all the clock cycles needed to process memory tiles worth of data. For some DNN schedules, 16 bits may not be large enough.

The controller logic for the ofmap pond is shown in Fig. 3.19. It performs local accumulation, which adds some additional computations to the process. There are four controllers involved:

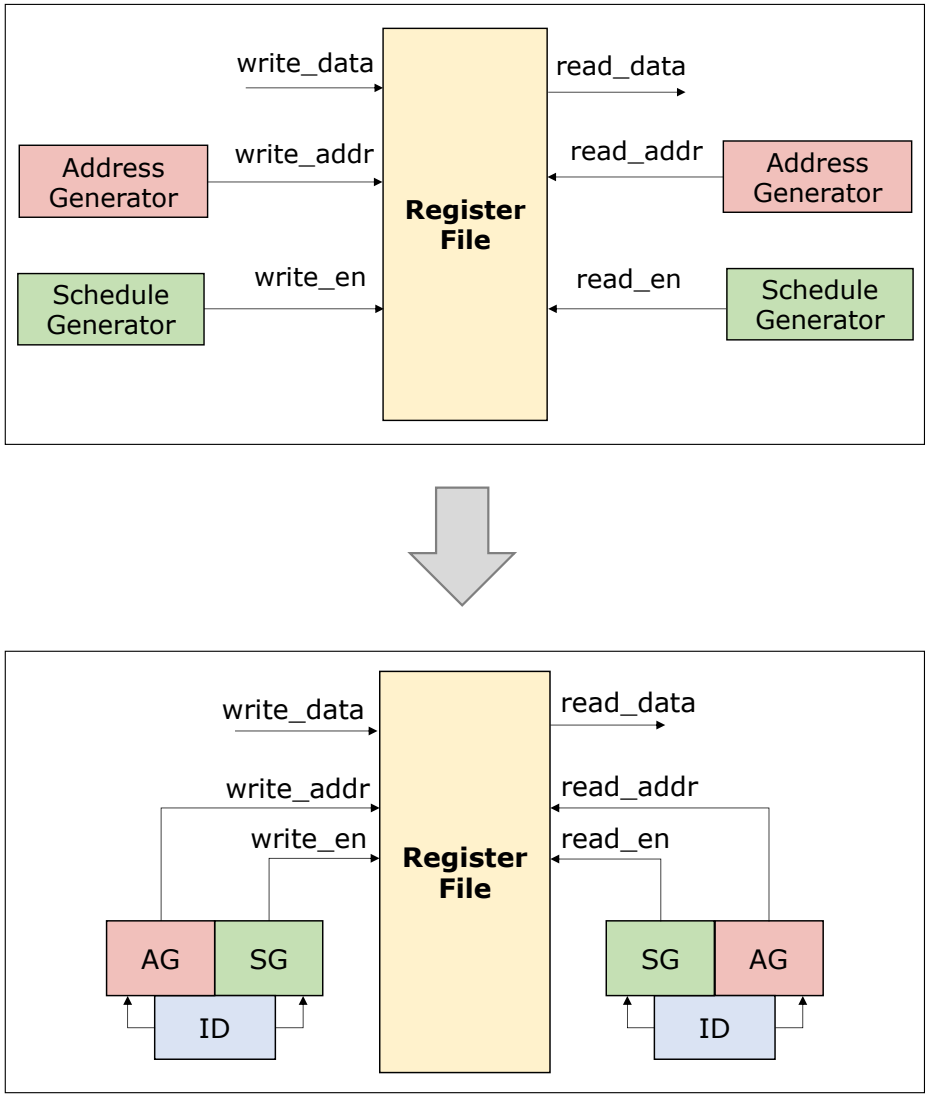


Figure 3.17: Mapping the AG, SG and ID logic to the address generator and the control generator. The three computations in the affine function (as illustrated in Fig. 3.16) can be mapped to three blocks in the address and control generator. The address generator will generate the read/write address for the register file, and the schedule generator will generate the corresponding enable control signals. The iteration domain drives the loop nest of the address generator and the schedule generator.

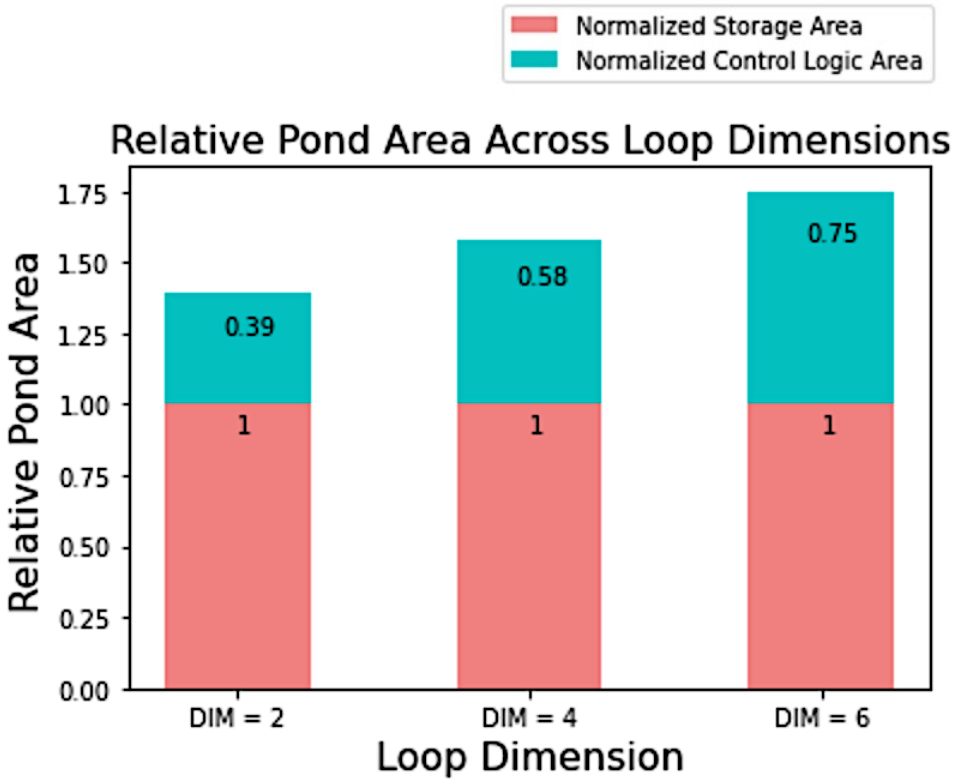


Figure 3.18: Relative pond area analysis for 1R/1W 64B pond for different loop dimension sizes for affine access patterns. The controller can have large overhead, especially as the loop structures get more complex (39% - 75%).

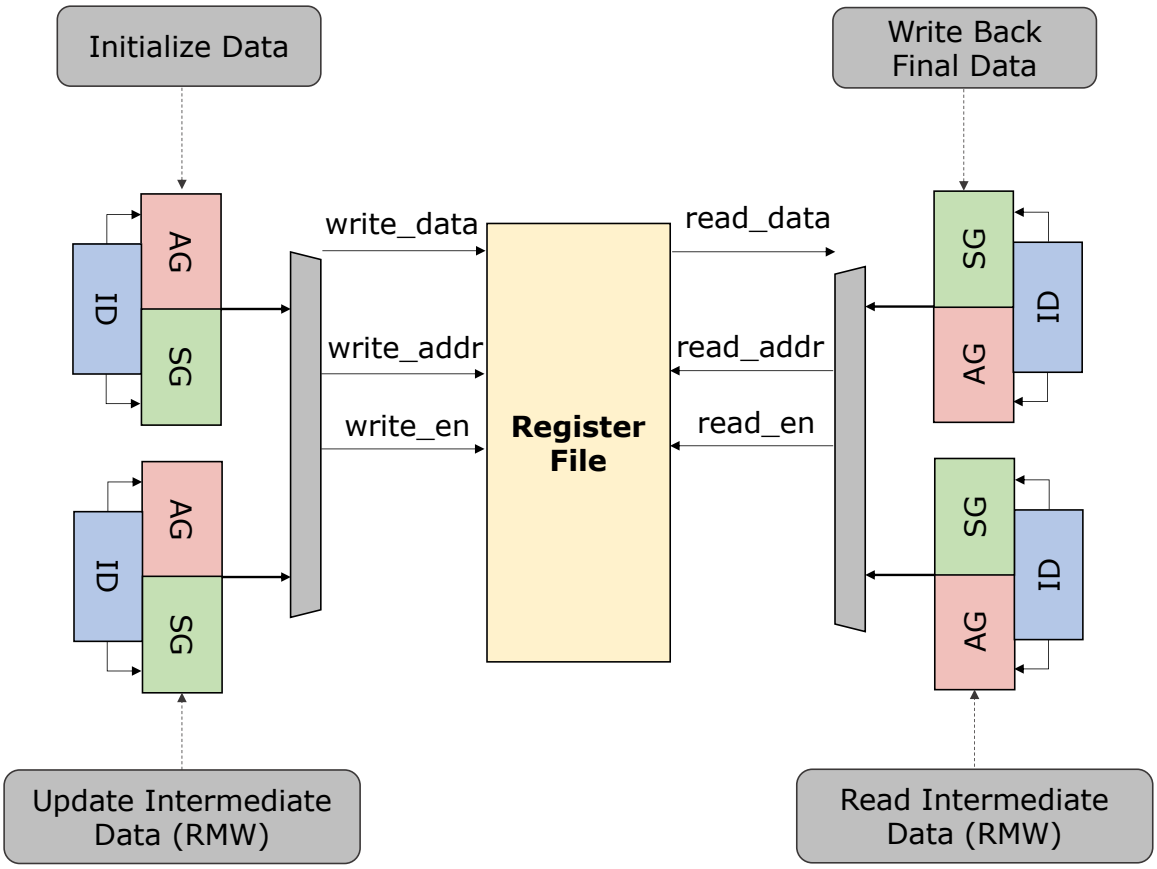


Figure 3.19: Unlike the weight and input ponds, the ofmap pond needs additional controllers for the read and write port to support accumulation for read-modify-write (RMW) operation.

- The "Initialize Data" controller, which is the first write controller, initializes the accumulation value in the register file.
- The "Read Intermediate Data (RMW)" controller, which is the first read controller, orchestrates the intermediate data read for RMW. This controller supports the loop nest needed to program the access patterns for reading data from the pond.
- The "Update Intermediate Data (RMW)" controller, which is the second write controller, updates the intermediate data to the pond, i.e., updating the pond.
- The "Write Back" controller, which is the second write controller, writes back the final output data to the next memory hierarchy.

In order to support the ofmap computation, naively adding two more controllers to the ponds would increase their area and energy consumption. To avoid this, we next explore optimization

opportunities that can reduce the design complexity of the address generators and the schedule generator for each of the four controllers in Fig. 3.19.

## 3.5 Optimizing Ofmap Pond

In order to simplify the overall design complexity of the schedule generators for the four controllers needed to support CNNs in ponds, we add timing signals sent to the ponds. These signals indicate the "start" of the operation and when the input data is valid. The "start" signal is generated each time the pond loop is initiated, which considerably reduces the maximum number of clock cycles that the schedule generator needs to track. To further simplify the schedule generator, the input data contains a valid signal that provides most of the information as to when the data should be written. As described in Section 3.3.3, some additional logic is needed to handle the multi-cast scenarios.

### 3.5.1 Initialization

During multi-cast, since the memory tile is broadcasting to many ponds, the valid signal in this case indicates that the data is valid for some pond, but each pond must still determine which data to capture. As shown in Fig. 3.14, this support must be in the controllers which handle initialization. As described in Table 3.1, `init_index`, `block_factor` and `cyclic_factor` are used to determine multi-cast handling. The need for a schedule generator is eliminated by using `init_start` to indicate the start of a new block of data and `init_valid` as a valid signal generated by the memory tile to indicate when there is valid data to be stored.

For address generation, we observed that the access pattern for data initialization for the pond is in fact simple contiguous writes, i.e., for address generation, the controller only needs the starting address information and the block size of the data to be written to the pond. Therefore, we propose a simple contiguous addressing pattern supported by the initialization controller using `init_offset` and `init_range` parameters, eliminating the need for the two-level loop nest described in Section 3.3.2.

### 3.5.2 Write Back

To simplify the address generation for the write back controller in the CNN pond, we observe a simple contiguous addressing pattern can be used, similar to the initialization block. This is because the write back controller only needs to fetch data in a contiguous pattern to write it back to the memory tile.

However, there is a minor difference in the schedule generation block for write back compared to that of the initialization controller. Unlike the initialization controller which receives an external

valid, the write back controller needs to determine the valid signal for the data that should be pushed to the memory tile. By using a programming interface similar to that of the initialization block, the controller can determine when to send the data back to the top memory. Similar to the initialization block, the schedule generation can be simplified by using a start signal from the memory tile (`wb_start`). The valid generation for the write back controller is handled within the pond.

### 3.5.3 Read Intermediate Data (for RMW)

The purpose of this controller is to retrieve the data from ponds (for input and weight ponds) and fetch accumulation updates for the output pond. Schedule generation for this controller is handled through a loop start input signal (`update_start`) which is used to read the data from the pond. For example, if the weights in a ponds are read 28 x 28 times before they are reloaded, then 28 x 28 start signals must be generated as inputs to the pond. Similar to the write back logic, the valid signals need to be generated internally - i.e., the schedule generator needs to determine when the data should be read. This logic can be achieved through offset handling (`cycle_stride`), which can determine the periodic cycle offsets between each reads (e.g., every one or two clock cycles).

The logic responsible for address generation cannot be optimized and must maintain the complexity described in Section 3.3.2. This is unlike the initialization block, which consisted of simple contiguous writes. To support the needed complexity, an additional parameter called `update_stride` is needed to enable strided access patterns. As discussed in Section 3.3.2, in a weight stationary dataflow with filter width and height as the innermost loops, the same weights can be reused to generate the next output. However, due to the sliding nature of the inputs, $4/5^{\text{th}}$ of the inputs can be reused, and only $1/5^{\text{th}}$ of the inputs need to be fetched for a 5x5 CONV kernel with stride 1. This results in a piece-wise linear affine access pattern for the inputs since we need to fetch 25 inputs the first time, and thereafter we only need to fetch 5 new inputs. Supporting the piece-wise affine access pattern will increase the controller complexity. Hence one can manage the accesses for the inputs so that instead of fetching all 25 inputs in one operation, the fetching of the inputs can be decomposed into 5 operations, each operation fetching 5 inputs each time. Thus the piece-wise affine access pattern can be converted to a standard 2-dimension loop nest where the `x` and `y` input image loops are left distinct instead of fusing them to enable loading the next input column to produce the next output pixel, rather than the complete 5x5 map. This 2-level loop nest is already supported, so no additional support is required.

### 3.5.4 Update Intermediate Data (for RMW)

To support the accumulation for read-modify-logic, in addition to supporting the initialization loop nest, an additional controller is required for writing to the pond. This controller must maintain the same complexity as the loop nest for reading the intermediate data. Hence similar optimizations can

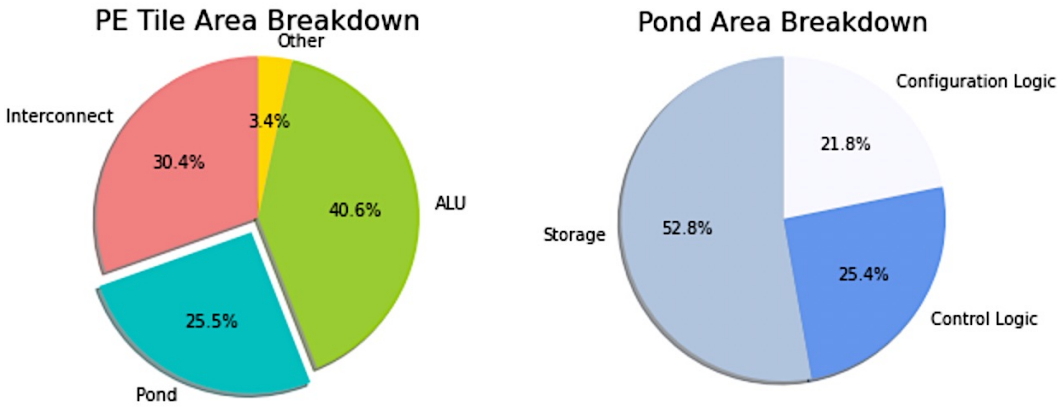


Figure 3.20: PE tile area breakdown. Once we introduced the pond into the PE, the area increased by ~25% (left). ~53% of the pond area is in the storage, and the remainder in control and configuration.

be applied for this controller as well. However, we observe that the programming for all the fields for the intermediate data read and update is the same, except for the delay between the reads and the writes, which depends on the time it takes for the ALU to update the data.

To reduce the number of controllers required, a shared controller could be used for performing intermediate data updates with a delay module between the read and write ports. Therefore, instead of four controllers, three controllers would suffice. This common controller could be shared between the read and write port, along with delay logic between them. This controller responsible for reading and writing the intermediate data can be referred to as an update controller. The delay can be programmed to faithfully capture and write back the data from the ALU when it is ready. `update_acc_en` determines if the RMW loop should be enabled and `update_acc_delay` sets the cycle delay between the read operation and when the valid result is available at the pond inputs.

## 3.6 Introducing Pond in PE

We built an initial version of a 64B 1R/1W pond with 2-level loop dimension for both address and schedule generation (first bar in Fig. 3.18) and introduced it in the CGRA. As seen in Fig. 3.20, introducing the pond in the PE increased the area of the original PE tile by ~25%. Interestingly, after inserting the pond in the PE, only about 53% of the pond area is in the storage, rather than 70% that would be expected from Fig. 3.18. The additional overhead comes from the configuration registers that are needed to hold the pond parameters. These registers are added at the tile level and were not included in the initial graph. They nearly double the controller overhead. Supporting a pond with 2 sets of controllers for the ofmap pond (as shown in Fig. 3.19) doubles the control/configuration overhead of the pond, increasing its area over 1.6x. This change would further increase the PE area

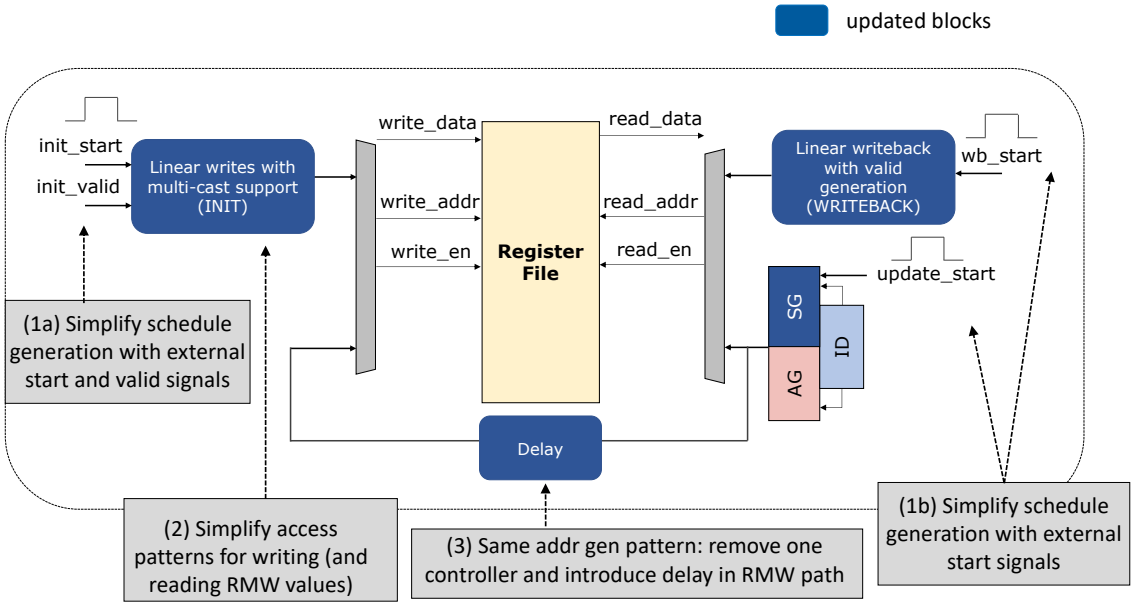


Figure 3.21: The schedule generation is simplified in the optimized ponds with an incoming start signal for all the controllers and a valid signal for initialization controller. The access patterns are simplified for writing to the register file and reading the final accumulated value since they are going to be sequential. The complexity of the address generation for all complex read patterns is maintained as is. Since the read and write controller for RMW have same pattern but with a delay, one of the controllers is replaced with a variable delay block.

by 17%.

Fig. 3.21 shows the optimized pond. Table 3.1 and Table 3.2 present the interface signals and configuration registers for the ponds illustrated in Fig. 3.21. After applying the specific optimizations, the area reduces by ~17% compared to the baseline (see the breakdown on the right side of Fig. 3.22). While the change in the pond area is modest, the area needed for the configuration registers halves from the original 2-port controller. The optimized controller needs 98 registers to hold all the configuration bits, whereas the original pond required 196.

## 3.7 Additional Pond Architecture Considerations

Having set the controller complexity, the next step was to determine the size and number of ports for the pond. Both the size and number of pond ports depend on the area and compute capability of PE. One could build simple PEs that can perform only two-input operations to reduce the PE area and use simpler and smaller ponds. Although such simple, small PEs require smaller ponds, more PEs are typically needed to implement a function. As the PEs become more powerful, the complexity of the pond must increase to provide the required operand bandwidth.

Table 3.1: Configuration registers for the optimized pond

| API Parameter | Bitwidth | Description |
|---|---|---|
| `init_offset` | $\log_2 rf\_size$ | Determines the offset from which data is written to the pond e.g. if data is to be written starting from addr=5, then this field is programmed as 5. |
| `init_index` | 9 | Pond index to determine which of the incoming valid and data signals are associated with the corresponding pond. |
| `init_block_factor` | $\log_2 rf\_size$ | Block size of data that should be consumed by the pond. |
| `init_cyclic_factor` | 9 | Total number of ponds associated with the task. |
| `wb_range` | $\log_2 rf\_size$ | Range of the data to be fetched from the pond. |
| `wb_offset` | $\log_2 rf\_size$ | Determines the offset from which data is fetched to the pond e.g. if data is to be fetched starting from addr=5, then this field is programmed as 5. |
| `wb_index` | 9 | Pond index to determine the writeback sequence used for generating the valid signal internally. |
| `wb_block_factor` | $\log_2 rf\_size$ | Determines the block size of data that should be read from the pond at a time. For e.g. if wb_range = 9 and wb_block_factor = 3, then 3 chunks of data will be sent out from the pond. |
| `wb_cyclic_factor` | 9 | Pond count associated with the writeback task. |
| `update_range` | DIM* $\log_2 rf\_size$ | Range of the data to be fetched from pond. |
| `update_offset` | $\log_2 rf\_size$ | Determines the offset from which data is fetched from the pond e.g. if data is to be fetched starting from addr=5, then this field is programmed as 5. |
| `update_stride` | DIM* $\log_2 rf\_size$ | Determines the stride of the reads. E.g. if data is read from every other address, then this field is programmed as 2. |
| `update_cycle_stride` | 8 | Determines the cycle stride, i.e. when this data is fetched. e.g. If the data is fetched every clock cycle, then this value is programmed as 1. If the data is fetched every other clock cycle, then this value is programmed as 2. |
| `update_acc_en` | 1 | Determines the RMW loop should be enabled (used for accumulation for output). |
| `update_acc_delay` | 3 | Determines the delay for the RMW loop 0: Read and write in same clock cycle 1: Delay of 1 CC 2: Delay of 2 CCs 3: Delay of 3 CCs 4: Delay of 4 CCs |

Table 3.2: Interface signals for the optimized pond

| Interface | Direction | Bitwidth | Description |
|---|---|---|---|
| `init_start` | `in` | 1 | Pulse to determine when the write operation should start. The start is common to all the ponds that are fetching data from the same MemTile. |
| `init_valid` | `in` | 1 | Valid signal to the ponds generated by the MemTile. |
| `wb_start` | `in` | 1 | Pulse to determine when the write operation should start. The start is common to all the ponds that are writing data back to the same MemTile. |
| `update_start` | `in` | 1 | Pulse to determine when the read operation should start. |

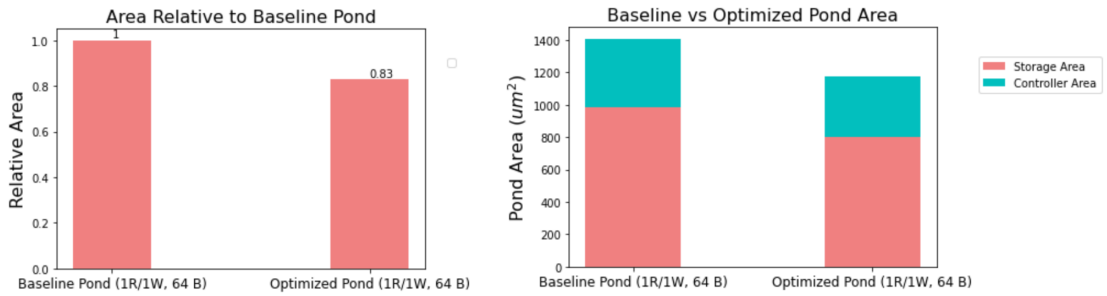


Figure 3.22: New pond area after the added optimizations. The area reduces by 17% while supporting loop nests for both computation and accesses. The right side shows the area breakdown across the storage and controller. The configuration area is not included.

### 3.7.1 Pond Port Configuration

If PEs only support two inputs, then each computation for the multiply/addition required by DNNs will occupy two PEs, resulting in two ponds being available per computation. To address this, one option is to introduce a 1R/1W pond in the PE tile. Each pond in the PE can store either the weights or inputs associated with a given multiply-add operation. As seen in Fig. 3.23 (left), a PE tile can perform the multiplication with the inputs stored in its own pond and the weights stored in the neighboring PE tile (or vice versa). The addition with the accumulated value can occur in the $2^{nd}$ PE tile using the output from the ALU of the $1^{st}$ PE tile and the accumulated value from the memory tile. Alternatively, to avoid fetching the weights or inputs from the neighboring tiles, a 2R/1W pond could be introduced so the weights and inputs associated with a given multiply-add operation can be stored in the same pond. However, since each PE tile can only perform a multiply

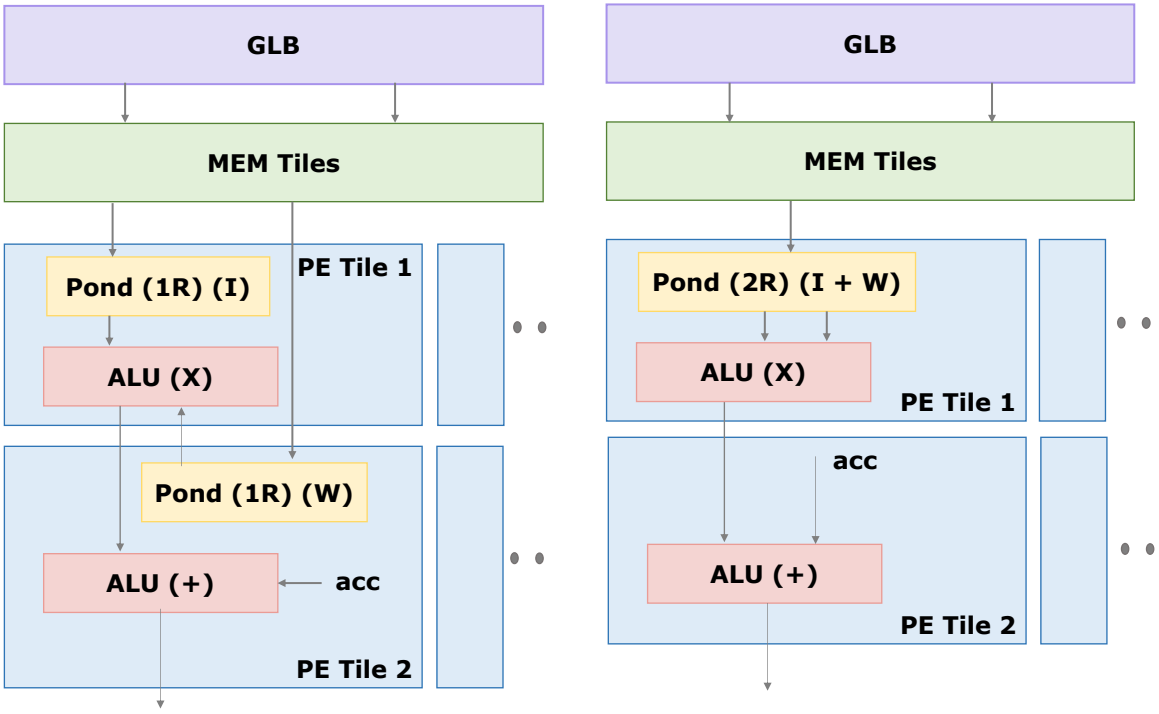


Figure 3.23: PE with multiply or add support in the ALU. Left shows the option with 1R/1W pond and right shows the option with 2R/1W pond. With 1R/1W pond, W or I needs to be fetched from a neighboring tile. With 2R/1W pond, W and I can be stored in same pond but not all ponds will be fully utilized.

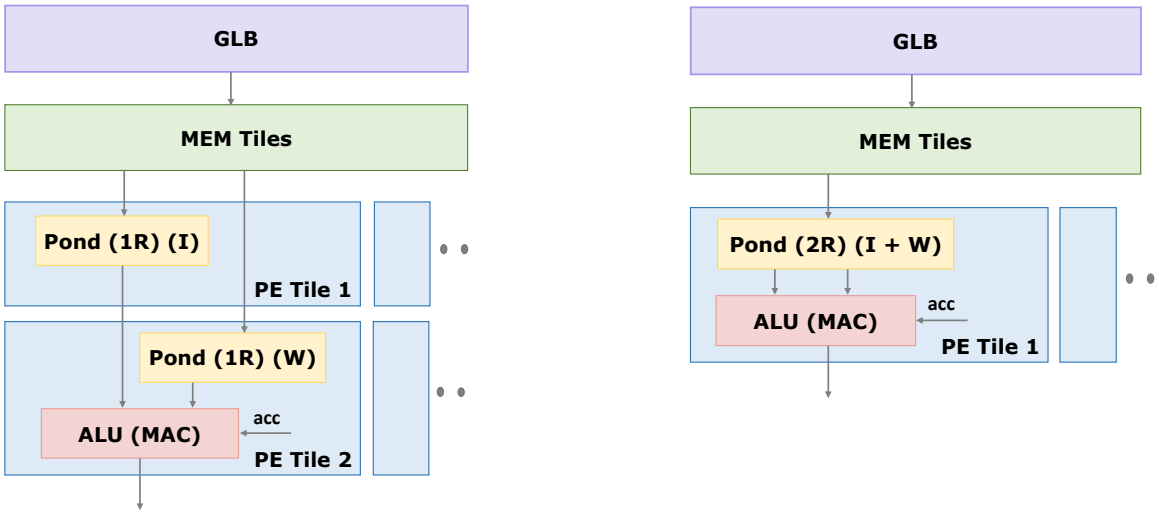


Figure 3.24: The optimized PE supports MAC operation in the ALU. With the MAC capability, the 1R/1W (left) is now underutilized. The 2R/1W pond (right) can now fully utilize the PE. The outputs are still stored in the memory tiles.

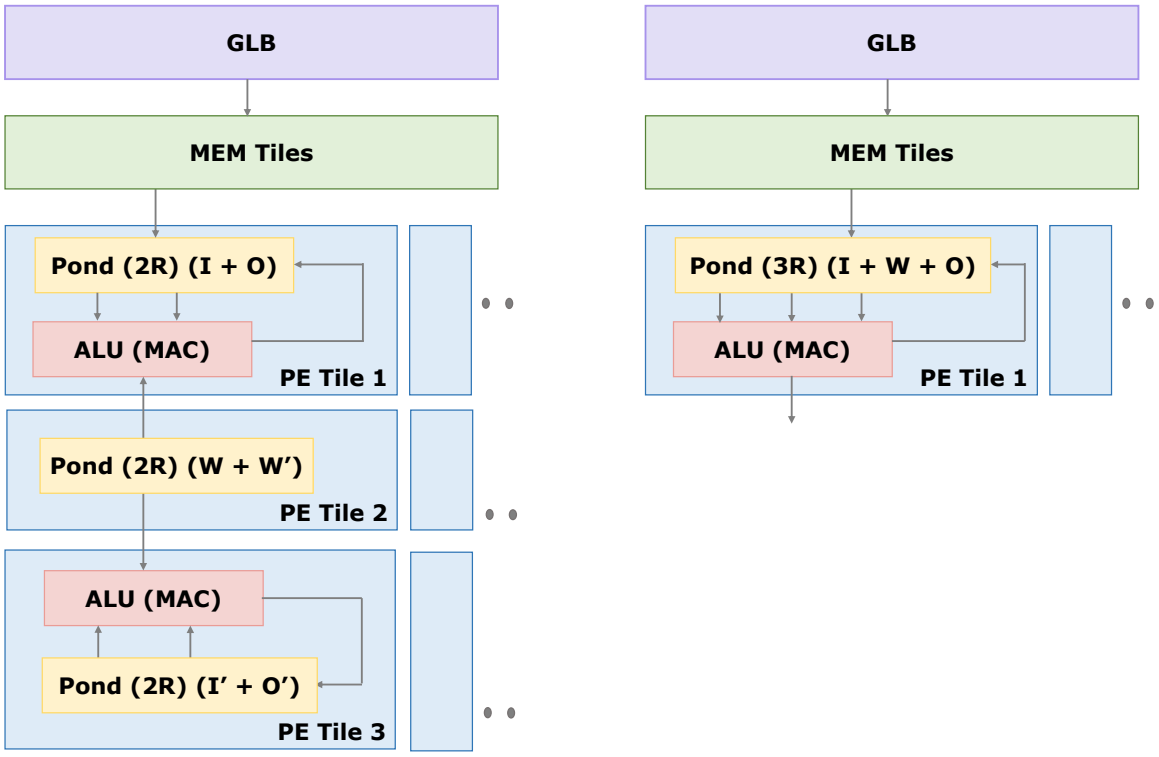


Figure 3.25: For PE with MAC support, two additional configurations are possible. With the optimized ponds and PEs, the 2R/1W ponds can also be utilized to store the output (left). Alternatively, a 3R/1W pond (right) can be used to store I, W and O in the same pond.

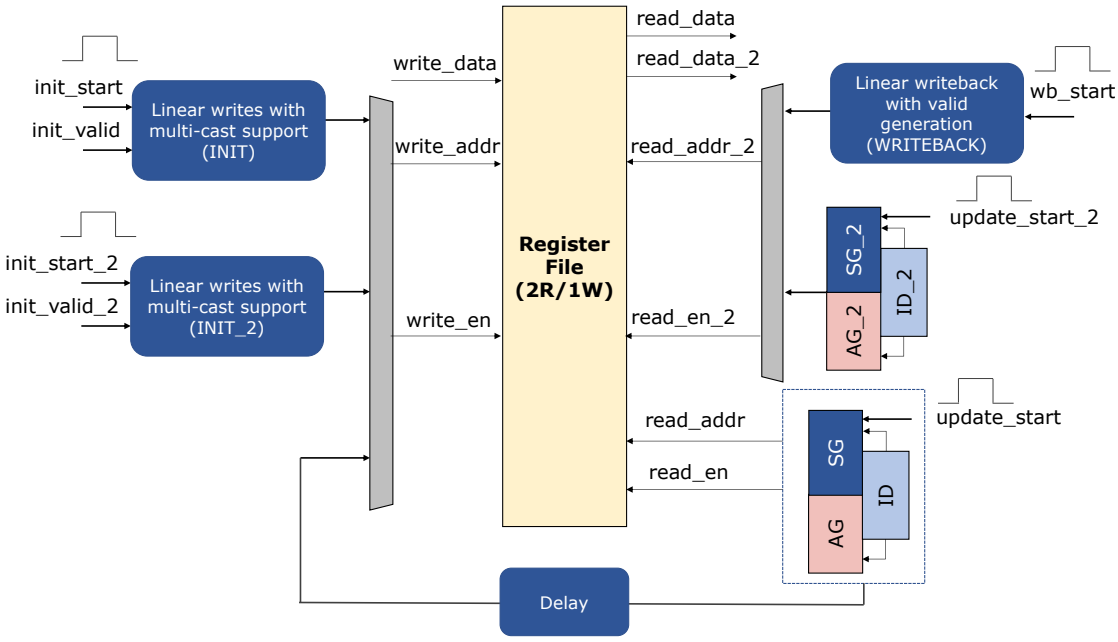


Figure 3.26: A pond with 2R/1W register file that can store two of weights, inputs and outputs. The write control is multiplexed between the two initialization blocks and the delayed accumulation. On the read side, an additional update block is added, which does not need the accumulation support.

or add operation, the pond in the tile that is performing the add could remain unused. The output accumulation can be performed in the memory tile since accumulation will not be supported for such a simple pond. For some schedules, only a single register may be needed for the output pond, such as the WS dataflow with Fx and Fy unrolled. We compared the area of the optimized 1R/1W pond (second bar in 3.28) with that of the 2R/1W pond (third bar). For a 64B pond, the area of the 2R/1W pond was 43% larger than the area of the 1R/1W pond. Therefore, for a design with PEs with only multiply or add support, a 1R/1W pond will be more efficient.

However, as the PE becomes more powerful, more pond ports can be afforded. For instance, for a PE that supports a multiply-accumulate (MAC) operation in the ALU, a 1R/1W pond would result in underutilized MACs. As seen in Fig. 3.24 (left), the ALU in the 2$^{nd}$ PE tile utilizes the ponds from the neighboring PE tile to store inputs and the pond from its own tile to store the weights. However, the ALU in that PE tile cannot be utilized since no ponds are available to store the corresponding inputs and weights. Therefore, if a 2R/1W pond were to be introduced in the PE, as seen in Fig. 3.24 (right), the weights and inputs associated with a given multiply-add operation could be stored in the same pond. This way the inputs or weights from the pond need not be fetched in the neighboring PE tile as the PE would now perform MAC operation, leading to fully utilized ALU. Further, since the ALU and pond could support the MAC and accumulation computation needed for ofmap, the pond could also be used to store the outputs. This support requires the

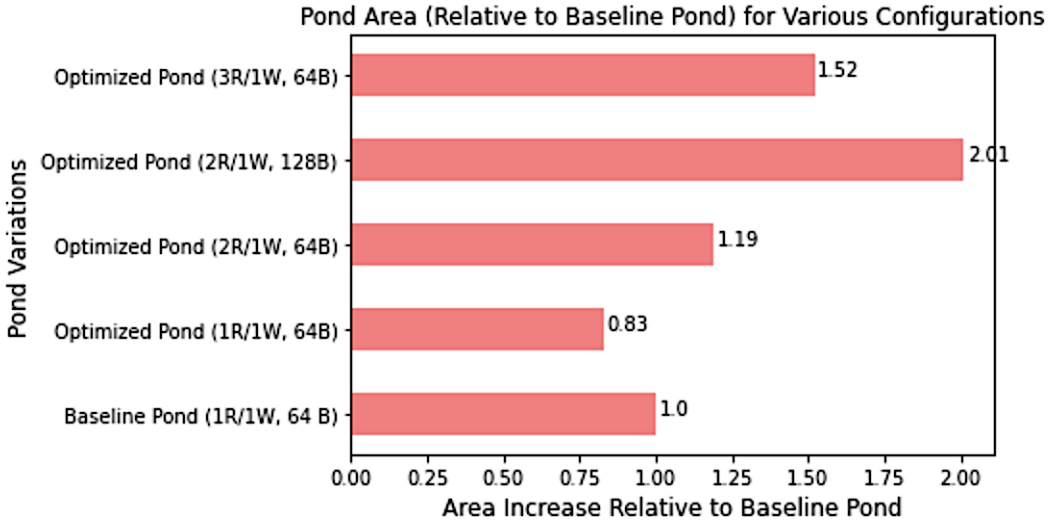


Figure 3.27: Relative area for various pond configurations. While pond with 3R/1W would be more flexible, it is expensive compared to 2R/1W. Doubling the pond also increases the area by 68%. Given the flexibility that a 2R/1W pond provides to map ofmaps either to a register or a register file and lower area compared to a fully flexible 3R/1W pond, the 2R/1W ponds are more efficient for a PE with MAC support.

optimized pond showed in Fig. 3.21. With this support, the pond could store either two of the inputs, weights or outputs, and one of them could be directly consumed from the neighboring tile, as seen in Fig. 3.25 (left). As discussed earlier, our study of various schedules has shown that there are many cases where the best schedule needs only 1 register for storing the output. In those cases, a 2R/1W pond can be configured to operate in the mode as shown in Fig. 3.24 (right), or, when a register file is also needed for ofmap, the configuration in Fig. 3.25 (left) could be used. A final option could also be to introduce a 3R/1W pond, as seen in Fig. 3.25 (right). In this case, the inputs, weights, and the outputs corresponding with the MAC computation could be stored in the same pond itself. Here the ponds as well as the ALUs are fully utilized, and all the weights, inputs and outputs would be contained in the same pond associated with the ALU.

Fig. 3.26 illustrates a pond with 2R/1W register file. The register file now has two independent read ports. As a result, it can store two of the inputs, weights and outputs. Using the 1R/1W pond shown in Fig. 3.21 as a starting point, we add the additional controllers needed for the 2R/1W register file. As seen in Fig. 3.26, on the read side, an extra controller is added, which does not need to support accumulation since the accumulation loop is only needed for storing ofmaps. This controller is multiplexed with the write back controller so that the two update blocks can operate in parallel. On the write side, the additional initialization controller is added, resulting in three multiplexed address generators.

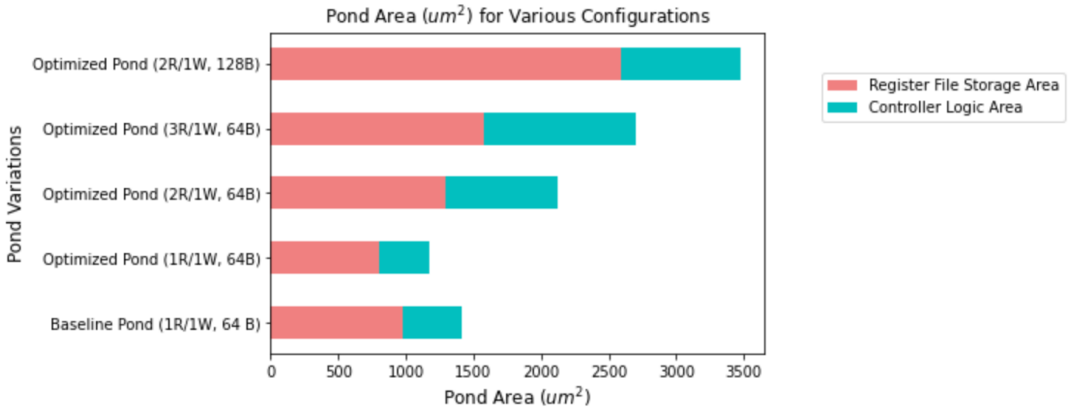


Figure 3.28: Pond area breakdown for various pond configurations. All configurations are synthesized at same frequency.

We compare the area of the 3R/1W pond with a 2R/1W pond and a 1R/1W pond. For a 64B Pond, the area of the 3R/1W pond is 83% more than the area of the 1R/1W pond and 28% more than the area of the 2R/1W pond. As seen in Fig. 3.27, the 3R/1W pond is more area expensive. Fig. 3.28 shows the area breakdown between the storage and the controller logic. If the PE size increases, the number of PEs would need to decrease to fit in the area budget, rendering the 3R/1W pond option infeasible. On the other hand, the 1R/1W pond option becomes infeasible because it does not fully utilize the MAC capability of the ALU in the PE. Therefore, the 2R/1W pond, which provides flexibility to map ofmaps either to a register or a register file and has lower area requirements compared to a fully flexible 3R/1W pond, is more efficient for a PE with MAC support.

Table 3.3: DNN Layers. Naming convention is as followed in Algorithm 1.

| **Layers** | **X** | **Y** | **C** | **K** | **Fx** | **Fy** | **B** |
|---|---|---|---|---|---|---|---|
| conv1 (3x3) | 28 | 28 | 512 | 1024 | 3 | 3 | 1 |
| conv2 (5x5) | 14 | 14 | 1024 | 256 | 5 | 5 | 1 |
| pointwise (1x1) | 56 | 56 | 128 | 128 | 1 | 1 | 1 |
| FC1 | 1 | 1 | 9216 | 4096 | 1 | 1 | 16 |
| FC2 | 1 | 1 | 9216 | 4096 | 1 | 1 | 32 |

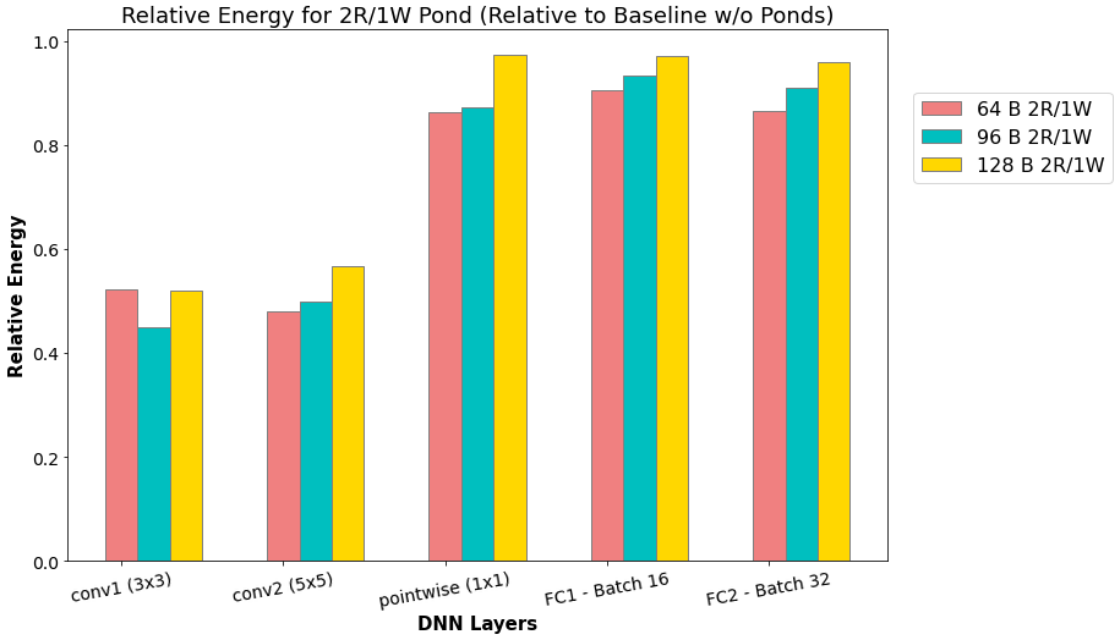


Figure 3.29: Energy analysis for optimized PE with MAC for a 2R/1W pond for different sizes. As pond size increases, the overall energy across application increases. Similar sizes are more efficient and both 32 B and 64B ponds have lower energy.

### 3.7.2 Pond Size

In this section, we assess the impact of pond size on the energy efficiency for various DNN layers for PE with MAC support, listed in Table 3.3. We investigate three pond sizes: 64B, 96B and 128B. We utilize the two configurations discussed in Section 3.7.1 for the 2R/1W port configuration (Fig. 3.24 (right) and 3.25 (left)). When we use the neighboring pond to store either the input, weight and output, we assume half the pond size would be available for it. We provide these configurations to the Interstellar framework and reported the minimum energy from the two configurations.

The result in Fig. 3.29 demonstrated that the energy was maximum for the largest pond size across all the DNN layers. Between 64B and 96B, the smallest pond was generally better for energy. For conv1 3x3, the 96B pond was able to find a schedule with better energy than the 64B pond. Thus smaller ponds are better for energy efficiency compared to larger ponds. We select 64B pond (32-entry) since it is smaller and mostly had the best energy.

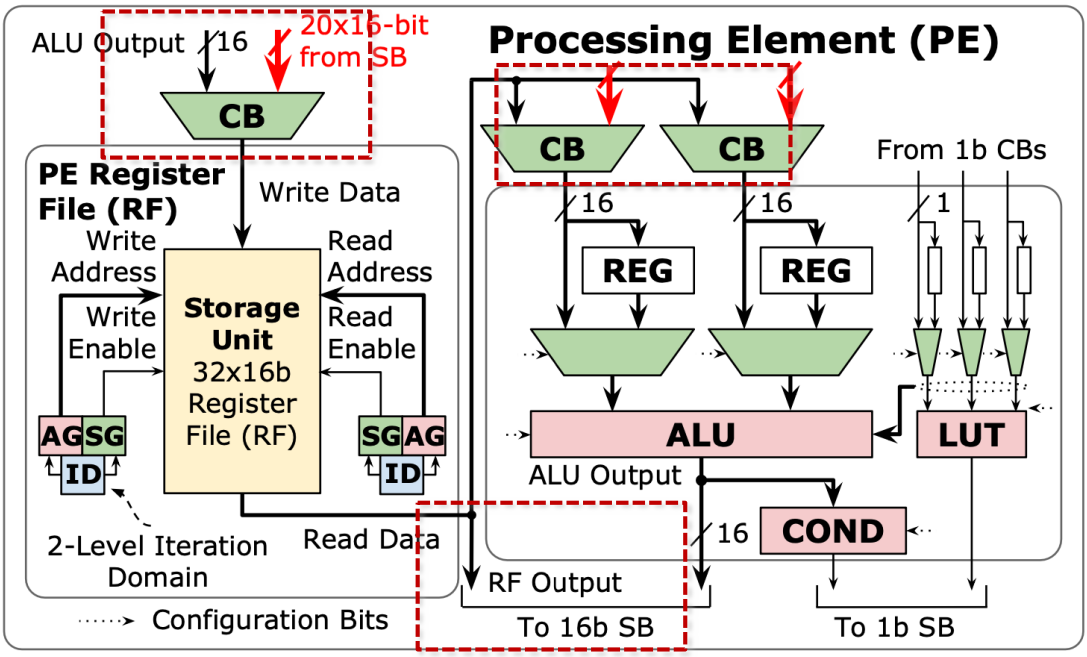


Figure 3.30: Integrating a 1R/1W pond without accumulation support in PEs. The pond output connects the ALU inputs through the ALU CBs. This connection allows the ALU to read data from the ponds. Pond output is also connected to the SB in the PE to drive pond output to neighboring tiles. This enables the ponds from neighboring PE to store either the weights or the inputs. The area overhead to introduce these additional connections to the ALU CBs and PE SBs is insignificant.

Table 3.4: Memory Access Costs

| **Memory** | **Access Cost (16b)** |
|---|---|
| pond 2R1W (64B) | 0.22 pJ |
| pond 2R1W (96B) | 0.29 pJ |
| pond 2R1W (128B) | 0.36 pJ |
| Memory Tile (512 KB) | 5.4 pJ |
| Global Buffer (1 MB) | 26.25 pJ |

## 3.8 Handling Pond Connectivity in the PE

Our next task was to connect the pond to the PE. As Section 3.7 showed, it is useful for the pond to be able to feed data to external PEs in addition to its own PE. The base PE receives all its data via the interconnect, which is connected to the memory tiles. In the rest of the section, we discuss the approach for integrating the pond inputs and outputs with the remaining components of the PE tile.

Fig. 3.30 shows the pond connectivity in the PE for a 1R/1W pond for inputs and weights. A new CB needs to be introduced to drive the pond. Existing CBs for ALUs cannot be used to drive the ponds since the ALU and pond must be able to operate in parallel. Although the additional CBs incur additional area overhead (< 2% of pond area), they provide the flexibility to simultaneously write to pond or ALU through their own CBs. For the optimized pond (Fig. 3.21), an external multiplexer is needed to get the two data streams onto the same input port. A new connection is also needed to drive data input to pond from the ALU output. Specifically, this connection is needed to write the ALU output to the pond for accumulation use cases. The ALU output will drive the CBs of the pond inputs.

A 2R/1W pond, as shown in Fig. 3.26, can use the same input data interface as 1R/1W pond, but the connections for the pond outputs will differ for these two cases. To drive the ALU inputs, all pond outputs must be connected to it. However, there are two options for this connection. The first is to connect the pond output directly to the ALU, which is favorable for timing. However, this approach requires modifying the ALU interface to directly consume the pond data. Alternatively, the pond output can drive the input CBs of the ALU, which is simpler for integration. Nevertheless, it adds CB logic in the timing path between the pond and the ALU. To maintain ease of integration, we choose to connect the pond output to the ALU through the CB as shown in Fig. 3.30.

To make pond data accessible outside its own PE to be used for the ALU of the neighboring

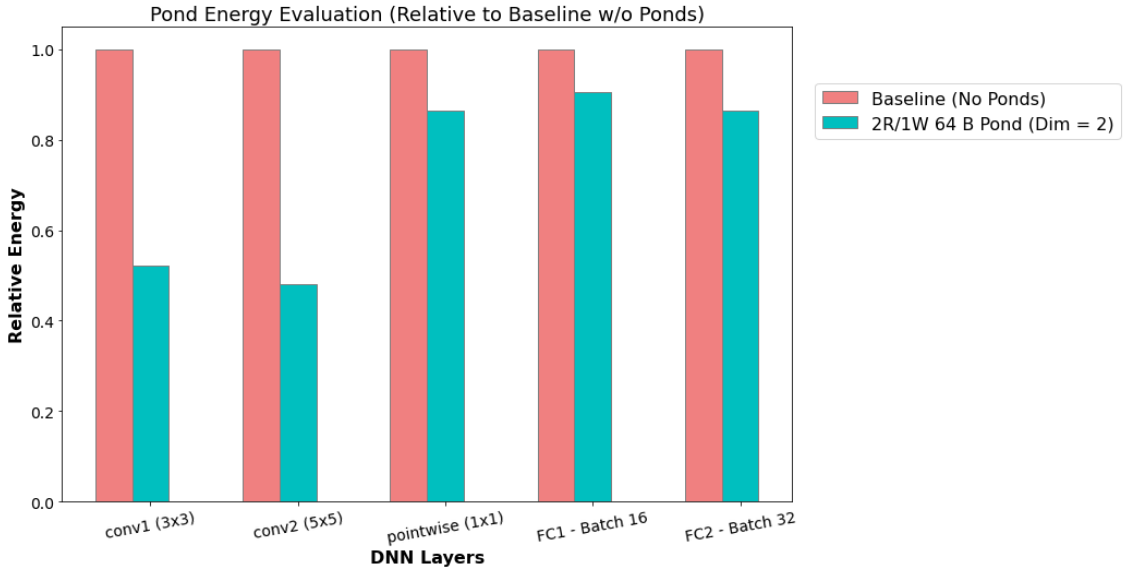


Figure 3.31: Energy result for convolution, pointwise and fully-connected layers with and without ponds. Convolution layers see up to 50% reduction in energy. Pointwise convolution and fully-connected layers with limited locality see up to 13% energy reduction.

ponds, the pond output must also drives the SBs of the PE.

For a 2R/1W pond, both the pond outputs need to drive the CBs of the ALU, as both these outputs are consumed by the ALU. The ofmap data that is read from the pond needs to be written back to the memory tile, and hence the pond outputs needs to be connected to the interconnect fabric through the SB. If the pond is used to store weights/inputs of the neighboring PEs, they too need to be connected to the interconnect network. Hence, both the pond outputs should drive both the ALU inputs and the SBs of the PE.

## 3.9 Pond Measurements

To evaluate the effectiveness of our ponds, we used the Interstellar analytical model described in Section 3.3.1. As described there, the analytical model requires the energy cost model. We hence needed to obtain the access cost for the pond, memory tile and the GLB. To obtain the access costs, we first synthesized the RTL and performed placement followed by gate-level power analysis using PTPX tool and generated the access costs, as shown in Table 3.4. With these access costs in hand, we provided them to the Interstellar model and evaluated the layers listed in Table 3.3 using a weight stationary dataflow with filter input and output channels unrolled.

To choose the best schedule, i.e., loop ordering and block size, we selected the one with the least energy consumption. Fig. 3.31 shows the energy consumption for the DNN layers for both the

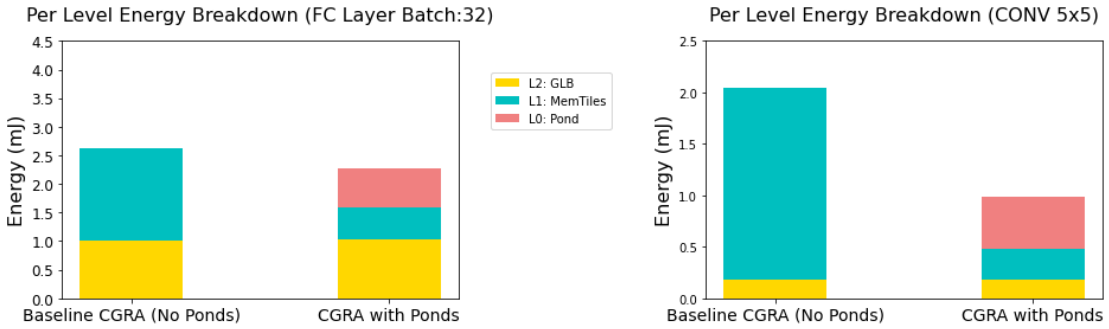


Figure 3.32: Breakdown of energy across the memory hierarchy. For FC and pointwise convolution, energy is dominated in GLB. For the convolution with 3x3 and 5x5 filters, the energy is dominated in the level closest to the ALU i.e. the ponds or the memory tiles.

baseline PE without ponds and the PE with a 2R/1W pond. The energy consumption of the latter is significantly lower (almost half) due to the high re-use and locality captured by the ponds in convolution layers. However, for pointwise and FC layers, where there is limited locality, the energy benefits of introducing ponds are relatively low (less than 15%). The energy consumption for FC layers also improves with an increase in batch size, as multiple inputs can re-use the same weights.

Finally, we analyzed the breakdown of the energy consumption across the memory hierarchy. As shown in Fig. 3.32 (left), for FC (and pointwise) layers, energy consumption is mostly dominated by the GLB, while for convolution layers (right), it is dominated by the level closest to the ALU.

## 3.10 Conclusion

In this chapter, we have discussed how our memory hierarchy optimizations can enhance the energy efficiency of DNN applications. Introducing streaming register files closer to the ALU in the CGRA can help better capture the locality and re-use of convolution layers. However, if the controllers for these register files are not complex enough, it can limit the variety of schedules that can be mapped on the register files. The increased complexity leads to increase in the overall area. Balancing the controller's complexity to support the additional requirements of the ofmap ponds while keeping it area-efficient requires a thorough evaluation of the loop nest complexity and the hardware's size and port complexity. Leveraging analytical models like Interstellar's design space exploration framework can accelerate this process, allowing faster design space exploration to choose the best schedule for a given layer given the hardware cost model. By optimizing the controller logic of the streaming register files, we make the introduction of streaming register files in spatial architectures like CGRA possible.

# Chapter 4

# Energy Efficient CGRA Fine-Grained Power Domains

Many approaches have been explored to improve the energy efficiency of CGRAs. These methods span all the way from circuit-level optimizations including body-biasing [42, 45] and NVM technology [13] to architecture-level optimizations [24, 28, 66]. To be energy-efficient for a wider range of applications, these configurable architectures would benefit from having multiple power domains that can be power gated to turn off unused sections of the chip. These domains allow the use of lower threshold voltage transistors for improving performance, while mitigating their higher leakage power by turning the power off to the units that are not being actively used. Moreover, power gating for saving leakage power can be orthogonal to other power saving methods and can be applied along with other circuit-level and architecture-level optimizations.

Typically, ASICs are partitioned into multiple power domains after careful study of the requirements for a fixed set of applications. Since the applications are pre-determined, these partitions can be coarse-grained. In contrast, to effectively leverage power domains on a reconfigurable fabric for a wide range of applications, the partitions need to be finer-grained. We should be able to program the power gates in these fine-grained domains later in the design cycle, such that unused parts of the fabric can be maximally turned off.

Special isolation cells are needed between power domains to ensure that logic gates driven by an *off* region still see valid logic values because floating nodes can cause short-circuit currents with high power dissipation. Isolation cells clamp the floating nodes to a '0' or a '1,' but adding these cells incurs timing and area penalties, proportional to the granularity of the power domains. If power gates are to be programmable at a fine granularity later in the design cycle, isolation cells are required at the boundary of every power domain. Further, it is typical for CGRAs to have multiple inputs and outputs to support flexible routing, each of which would require an isolation

cell. Isolation cells need a control signal from a controller to determine whether the isolation cells should be turned on or off. Having isolation cells on the output of a fine-grained power domain boundary is area-inefficient since they are placed on the main datapath thus limiting the maximum frequency the design can achieve. In addition, the routing of the enable signals for these isolation cells makes this technique timing-inefficient.

To avoid these overheads, this chapter introduces a CGRA routing fabric that intrinsically provides boundary protection. However, this approach cannot leverage the conventional Unified Power Format (UPF) based flow to introduce the isolation logic. Therefore, our framework incrementally introduces the needed design transformations using compiler-like "passes," and formally verifies them using Satisfiability Modulo Theories (SMT) [78]. Our framework also makes it easy to experiment with different power domain-related design parameters and to generate the necessary collateral for the physical design flow. We also create a place and route (PnR) tool that lets us efficiently map applications on the resulting CGRA with fine-grained power domains, while correctly configuring the re-architected routing fabric so that no X-propagation occurs from an *off* region to an *on* region. Additionally, we address the unique challenges encountered when implementing these fine-grained power domains in CGRAs.

We used this framework to insert power domains into the CGRA of an SoC that we designed and taped out. Our technique reduces the area overhead of boundary protection from 9% to less than 1% and removes the delay from the isolation cells. The resulting CGRA achieves 11 - 89% reduction in leakage power and 0.9 - 79% reduction in total power versus a CGRA without power domains, over a range of image processing and machine learning applications.

## 4.1 Background

A power domain is a collection of instances, all powered in the same way. Properties like supply voltage and power state tables are defined for a domain, so that each power domain on a chip can be individually controlled. With billion-plus-transistor ASICs, power domains with power gating capability can help reduce leakage power for parts of the chip that are not being actively used at a given time [57, 63].

Typically, special power management cells are required for implementing designs with multiple power domains [76]. Since this work focuses on power domain designs with a single primary voltage, with no retention of data when the domain is turned off, we focus here on the power management cells needed to implement such designs:

- *Power gating switches* can shut off supply to standard cells in parts of the design. How they are connected impacts wake-up time and inrush current when the power switches are turned off and on [50]. The details are covered in Section 4.4.3.

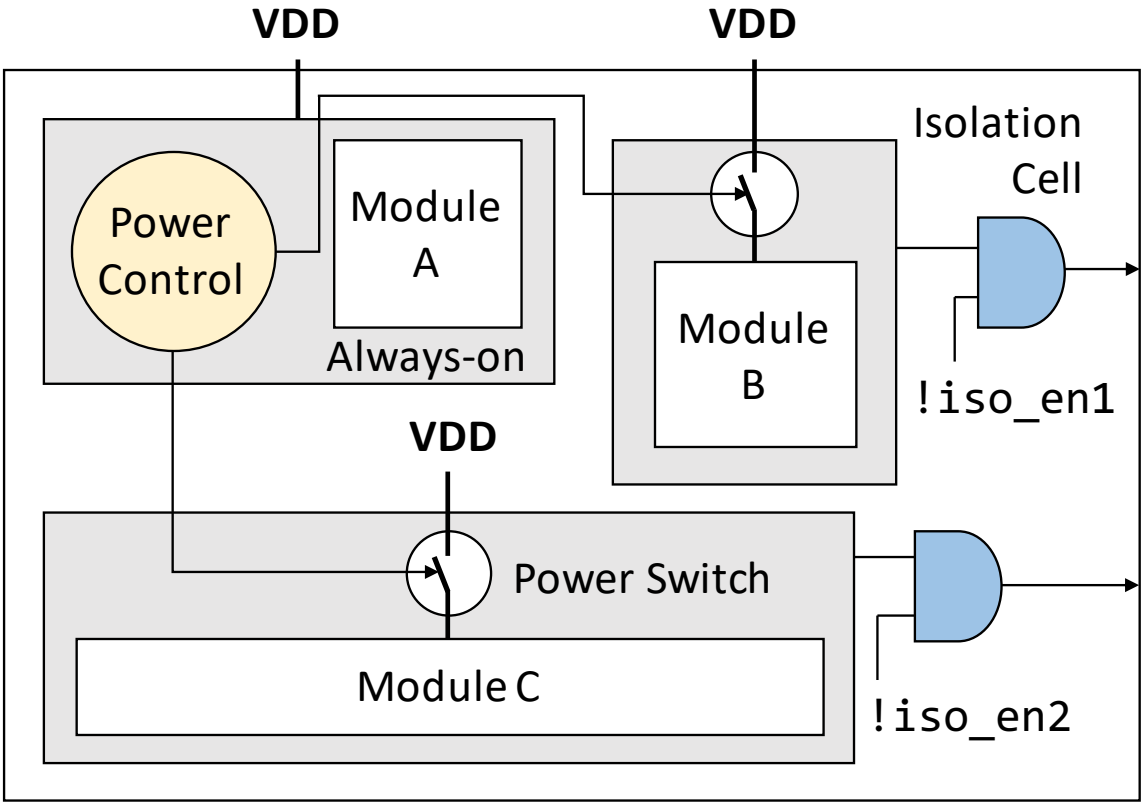


Figure 4.1: This ASIC has three power domains, one that is *always-on*, and two that can be shut down according to signals from the power controller. ASIC power domains tend to be coarse-grained and pre-determined with respect to domain size, count, location, and type and are based on the study of the chip's functional modules and the different use cases of the application that the chip will run.

- *Isolation cells* are special cells required at the interface between *off* domains and *on* domains. They clamp floating signals flowing from the *off* region to the *on* region at a constant value 0 or 1. Section 4.1.1 describes isolation cells in more detail.

- *Always-on (AON) cells* [4] have their own secondary backup power supply, so they can remain *on* even when their domain's primary supply is *off*. They drive signals that need to stay on even when the domain's primary supply is off.

Typically, ASIC design flows follow a *waterfall* approach, in which one or more applications are deeply studied; the hardware and all its features are built around those applications; the software is built for the given application use cases; and the hardware then runs this particular set of applications effectively [75]. The addition of power domains follows a similar approach. Application use cases

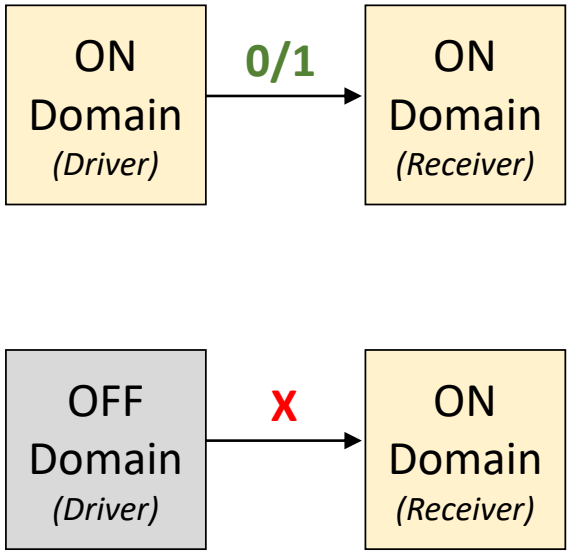


Figure 4.2: With both domains powered up (top), the receiving net sees an unambiguous 0 or 1 from the driver. However, when the driving logic is powered down (bottom), *on* domain inputs may float between 0 and 1.

are used to determine power state transitions. Given the area of the logic blocks on the ASIC and the connectivity between these blocks, power domain sizes and locations are determined. Thus parameters like power domain type, count, size, location, states, and control end up being customized for a given set of applications and their use cases. The intent for the use cases is coded into a power intent file [83], and that is used by the downstream physical design flow to insert the necessary power management cells [22]. Hence, as seen in Fig. 4.1, these power domain partitions are pre-determined and tend to be very coarse-grained based on the different high-level functional modules on the chip.

### 4.1.1 ASIC Power Domain Boundary Protection

Various scenarios arise when a signal driver and its receiver exist in different power domains. When both domains are powered up, the receiving net will see an unambiguous value of 0 or 1 from the driver. However, when the driving logic is powered down, the input to the receiving *on* domain may float between 0 and 1, as in Fig. 4.2.

Two issues result from this behavior:

- Undriven wires can cause unintentional logic values, altering the functionality of the receiving domain and leading to data corruption.
- The floating values can cause crowbar current to flow through the receiving logic, which can

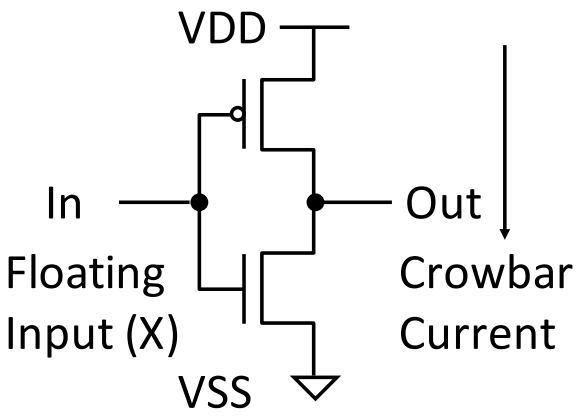


Figure 4.3: If a floating signal between 0 and 1 encounters an inverter, it can cause both the PMOS and NMOS transistors of the inverter to turn on. This will lead to a crowbar current through the inverter.

lead to increased power dissipation and can possibly damage the circuit. For example, as shown in Fig. 4.3, a floating signal to an inverter can cause both the PMOS and NMOS transistors to turn on. This will lead to a crowbar current through the inverter, causing a short circuit between VDD and VSS, which could possibly lead to circuit failure.

A typical solution to this problem is to insert isolation cells at the boundary of the power domains to ensure that the receiving logic always sees an unambiguous 0 or 1, thus avoiding X-propagation. The isolation cell operates in two modes: normal mode, where it acts like a buffer; and isolation mode, where it clamps the output to a fixed value.

For an AND isolation cell (Fig. 4.4), when `!isolation_en` is high, the cell is in normal mode of operation; when the isolation is enabled, the output of the cell is clamped to 0, thus avoiding X-propagation. Similarly, for an OR isolation cell, when `isolation_en` is low, the cell is in normal mode of operation; when the isolation is enabled, the output of the cell is clamped to 1, thus avoiding X-propagation. The nature of AND vs. OR isolation cell means that `isolation_en` will be active high for OR and active low for AND. The choice of AND- or OR-based isolation cells depends on the desired clamping value on the inputs of the *on* domain.

### 4.1.2 Power Domains in Reconfigurable Architectures

Prior work has explored power gating in reconfigurable architectures. Gayasen et al. [23] and Li et al. [49] present placement techniques for mapping a circuit with multiple power domains onto a fine-grained or coarse-grained FPGA; however, they do not focus on the underlying hardware architectures. Bsoul et al. [7] modify an FPGA to enable dynamic power gating, in which logic clusters

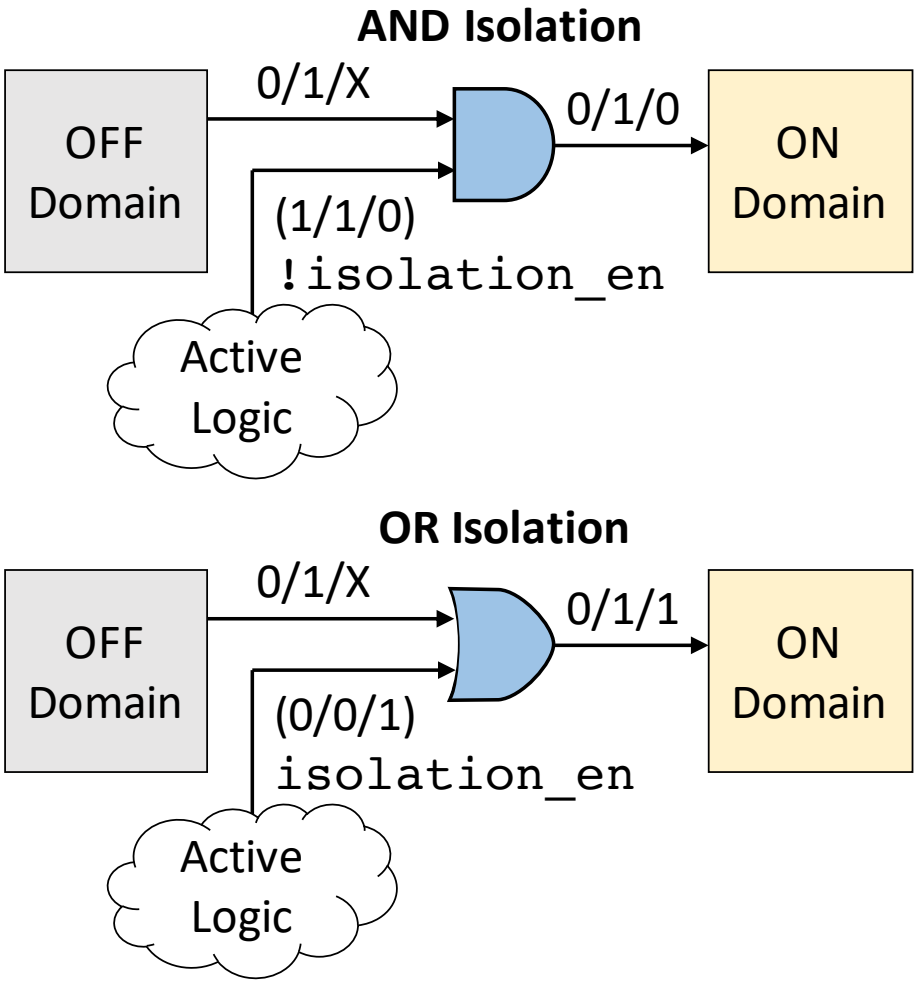


Figure 4.4: For an AND isolation cell, when `!isolation_en` is high, the cell is in normal mode of operation; when the isolation is enabled, the output of the cell is clamped to 0, thus avoiding X-propagation to the *on* domain. For an OR isolation cell, when `isolation_en` is low, the cell is in normal mode of operation; when the isolation is enabled, the output of the cell is clamped to 1.

can be selectively powered down at run-time. More recently, Miniskar et al. [62] explore power gating techniques to turn off the unused coarse functional units in a CGRA-based reconfigurable processor. Lopes et al. [55] present a CGRA for biological signal processing that can be partially turned off by power gating. Korol et al. [44] build a CGRA with four coarse power domains where a power management unit dynamically monitors the utilization of the CGRA and adapts the CGRA's resources to the workload by power gating the unused resources. While these works propose various power gating strategies, they are still geared towards coarse-grained power domains. Other work has explored fine-grained power domains. Lin et al. [52] turn off the unused interconnect switches in the FPGA. Ishihara et al. [35] present an FPGA with lookup-table level fine-grained power gating with small overheads. While their granularity is more fine grained than the other works, they do not explicitly address the electrical concerns of having floating outputs from the *off* region and the overheads resulting from introducing boundary protection logic to clamp those signals. As a consequence, coarse-grained power domains are preferred. This logic is needed at the boundary of every power domain. As the power domain granularity decreases, the number of input and output ports that need isolation logic increase. Furthermore, in spatially programmable designs, it is typical to have a large number of interface signals to support generic routing. Each of these interface signals needs its own isolation logic that will be introduced in the main datapath. In addition, isolation cells need some kind of control mechanism to determine if they should be turned on or off. This approach is thus area- and timing- inefficient for fine-grained power domains, necessitating a new approach to this problem.

## 4.2 CGRA Power Domain Boundary Protection Choices

In this section, we first discuss the different choices for isolating power domain boundaries in a CGRA and their limitations. We then propose a new low-overhead boundary protection mechanism.

### 4.2.1 Boundary Protection with Isolation Cells

Our goal is to have fine-grained power domains on the CGRA. Thus, we first need to determine how "fine-grained" is defined.

One option to introduce power domains is to partition the CGRA into pre-determined power domains, similar to what is done in ASICs (Fig. 4.1). For example, as shown in Fig. 4.5, we could introduce two power domains in the CGRA separated by a vertical boundary. Isolation cells would then need to be inserted along this boundary, wherever this boundary is determined.

When using isolation cells for boundary protection, we have two options, as shown in Fig. 4.5. When a powered-off tile drives a powered-on tile, if the isolation cells are placed in the powered-off region, they will need to be on the output ports of the driving tile (top choice in Fig. 4.5). If the isolation cells are placed in the powered-on tile, they will be on its input ports (bottom choice in

Fig. 4.5).

With the first option, the floating signals are clamped right away, but since these isolation cells reside in a domain that is *off*, they need to be special isolation cells with main power and backup power so that when the tile is *off*, the isolation cells can remain *on*. These cells are typically double height cells and hence are larger than the standard isolation cells. With the second option, the primary rail of the powered-on tile is *on*, and hence regular AND/OR cells can be used for isolation, which saves area compared to the first option.

But there is at least one major drawback to choosing pre-determined power domain boundaries. If there is only one column of isolation cells, the AON-SD boundary is fixed on the chip, which makes the power domains coarse-grained and less flexible. If the application uses fewer columns than what exist in the AON domain, we cannot shut down the excess. Thus, there is only a fixed amount of saving that can be obtained from this scheme. Another, even more concerning problem, arises when a larger application needs even slightly more columns than the AON domain has; in such a case, this scheme would offer no advantage for the mapping of such applications.

Isolation cells could be added at a certain stride, creating multiple domains, but that could increase the area. It will also introduce limitations on the mapping tool w.r.t the nature and size of applications mapped on the chip and its ability to orchestrate mapping of the applications on various parts of the chip. Further, there are many possible variations, such as power domains with horizontal boundaries, with multiple horizontal and vertical boundaries. Fixing these boundaries pre-tapeout will result in rigidity in mapping a large variety of applications, which is a typical use case for CGRAs.

### 4.2.2 Isolation cell-based boundary protection for tile level power domains

Because of their reconfigurable nature, CGRAs can run a wide variety of applications. These applications will have varied sizes, kernel counts and inter-kernel connections, as illustrated in Fig. 4.6. To effectively manage power on a reconfigurable fabric for such a variety of applications, the power domain partitions need to be finer-grained than those discussed in Section 4.1.

It would be ideal to add power switches to every single CGRA tile and make each one its own power domain. However, such a fine-grained support, where one could individually turn each tile on or off, would require us to place isolation cells on all outputs of each tile, as illustrated in Fig. 4.7. Moreover, if each CGRA tile were a separate power domain, the area overhead of this approach could be large, especially for many inputs/outputs per tile. These inserted gates also make timing worse, since the isolation logic is added to every path.

### 4.2.3 Boundary Protection with PEs in "Isolation Mode"

One way of creating a programmable *off-on* boundary could be to add an *isolation mode* to each PE, the function of which would be to safely clamp the incoming X's from the *off* domain [36]. All PEs at the boundary of an *on* domain would be configured in isolation mode to protect the remaining tiles in the interior of the *on* domain. The advantage of this approach is that no additional circuitry is needed for boundary protection, and the domain boundaries can be programmed at runtime. The disadvantage is that all tiles along the boundary would be wasted to provide isolation.

### 4.2.4 Proposed Low-Overhead Boundary Protection

We observe that reconfigurable hardware architectures, like our CGRA, have many multiplexers to allow for flexible routing. Fig. 2.3(b, c) show the multiplexers in switch boxes (SBs) and connection boxes (CBs). Each SB output on each side (blue arrows) has a mux that selects between the PE/MEM output and the routing tracks coming from the three other sides of the SB shown in gray. CB muxes select PE/MEM inputs from among the routing tracks. Here, the green CB selects between input tracks coming from the west and the north, and the gray CB selects from east and south.

To avoid the additional overhead from the isolation cells, we re-architected these existing SBs and CBs to incorporate boundary protection logic. As shown in Fig. 2.3c, all data inputs coming into a tile go only into SBs and CBs. Instead of isolating all signals leaving the *off* domain, we isolate signals coming into the *on* domain, which serves our cause as the newly architected multiplexers ensure that the floating inputs from the *off* regions do not propagate into the *on* regions.

To morph SBs and CBs into isolation cells they must fulfill the following requirements:

- In the normal mode of operation, when the entire CGRA is *on*, the SBs and CBs retain their default functionalities;
- When any SB/CB input is X, the incoming X must not propagate through any of the gates that make up these units; although an input might be X, all gate outputs must be 0 or 1 to ensure that no part of the SB/CB circuit dissipates short-circuit power. A corollary to this rule is that the SB/CB outputs always generate a 0/1 value.
- Lastly, there must be no buffer or inverter in the path to the SB/CB or the X value fed directly into an inverter could cause short-circuit power dissipation.

Fig. 4.8 shows how the SB/CB multiplexers are re-architected to perform normal functionality as well as clamping functionality. Our design breaks down the multiplexer into three stages: one-hot encoding, clamping and an OR tree. The one-hot encoding of the select input (`Si`) makes sure that only one bit of the encoder output (`So`) is high at a time, meaning that all other data inputs (`D`) will be clamped to 0 when not selected. The OR tree processes the output from the clamping logic to

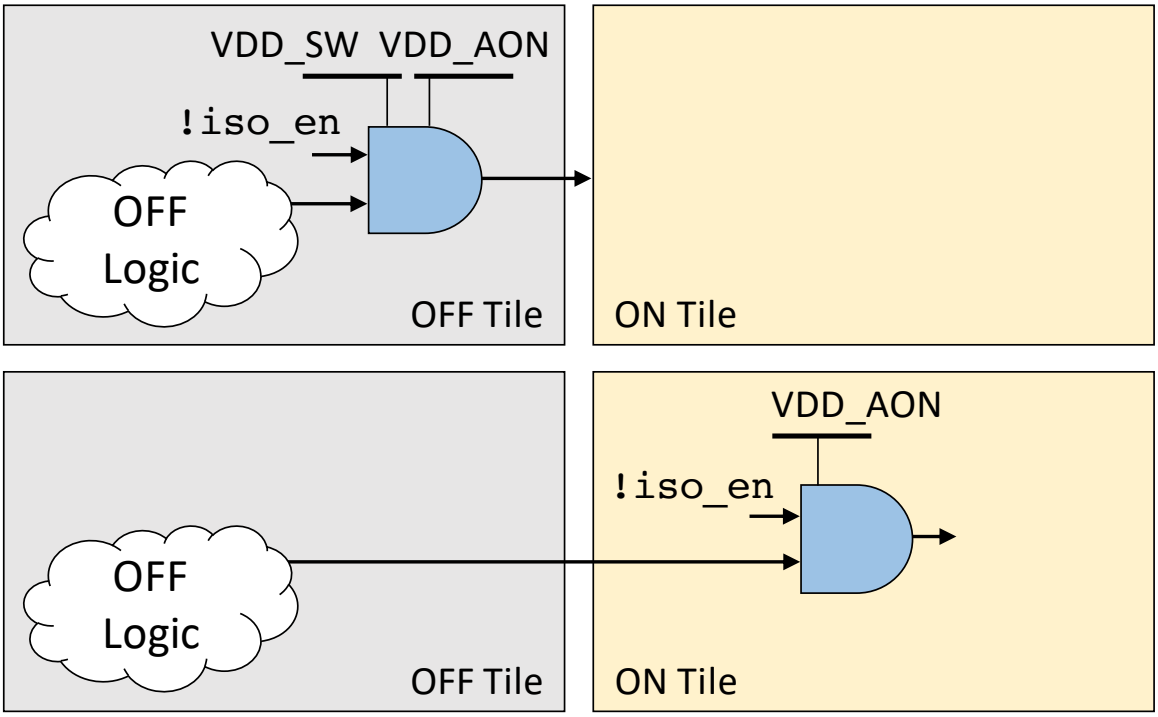


Figure 4.5: The isolation cells can either be placed in a powered-off tile (top) or a powered-on tile (bottom). Isolation cells in the powered-off tile need *always-on* backup power.

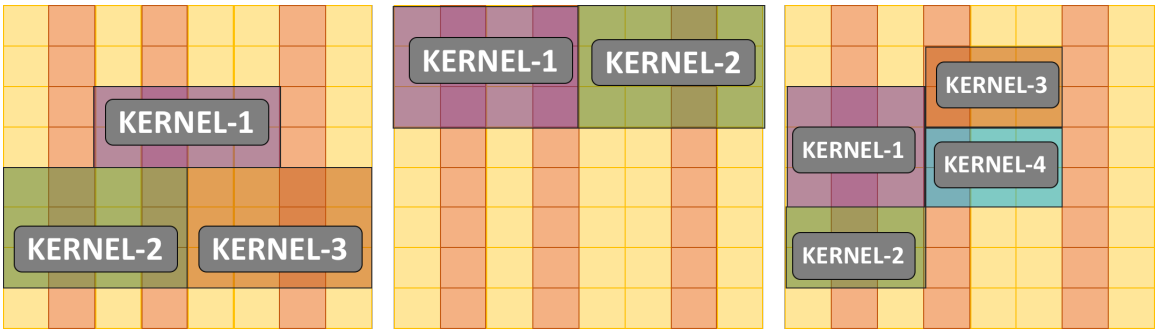


Figure 4.6: CGRAs can run a wide variety of applications with varied sizes, kernel counts and inter-kernel connections.

create the final output. The resulting SB/CB circuit meets all of the requirements outlined in the previous paragraph. Additionally, during implementation, we ensure that there is no buffer on the path to the SB/CB inputs, or a floating input to the buffer could dissipate power.

Fig. 4.9 shows the operation of this new SB/CB in the context of a tile. In this example, tiles to the west (W) and the north (N) of the center tile are *off* and tiles to the east (E) and the south (S) are *on*. The SB multiplexer that sends signals to the south selects between the three other sides - E, W and N and the PE output. Since one input to the SB is always from the PE, even if all the other sides are *off*, the PE can be programmed to drive a 0/1 value. This fixes the SB. For CBs, however, all inputs can be X, as is the case in our example for the green CB on the right. Therefore, for CBs, we add a constant input to the multiplexer (0 in our case).

Thus, because of our re-architecting of the interconnect multiplexers in the CGRA, the isolation logic is naturally embedded in the design. An added benefit of this technique is that the isolation enable signal seen in Fig. 4.5 is not needed anymore, since the select signal of the multiplexer is programmed through the bitstream. This select signal encodes which inputs are being driven by the active tiles. Thus, the need for a controller that determines the isolation enable signal conditions is eliminated, and these additional enable signals do not need to be routed.

Since the isolation cells are embedded in the routing fabric, we cannot leverage the conventional UPF-based flow that is typically used for inserting them. We hence created our own framework for placing these boundary protection circuits, as well as other circuits needed for building the CGRA with tile-level power domains. This framework is described in the next section.

## 4.3 Automated Power Domain Insertion Flow

We created an open-source framework that incrementally makes the design transformations required for introducing power domains into a base design. This framework, starting with the basic CGRA

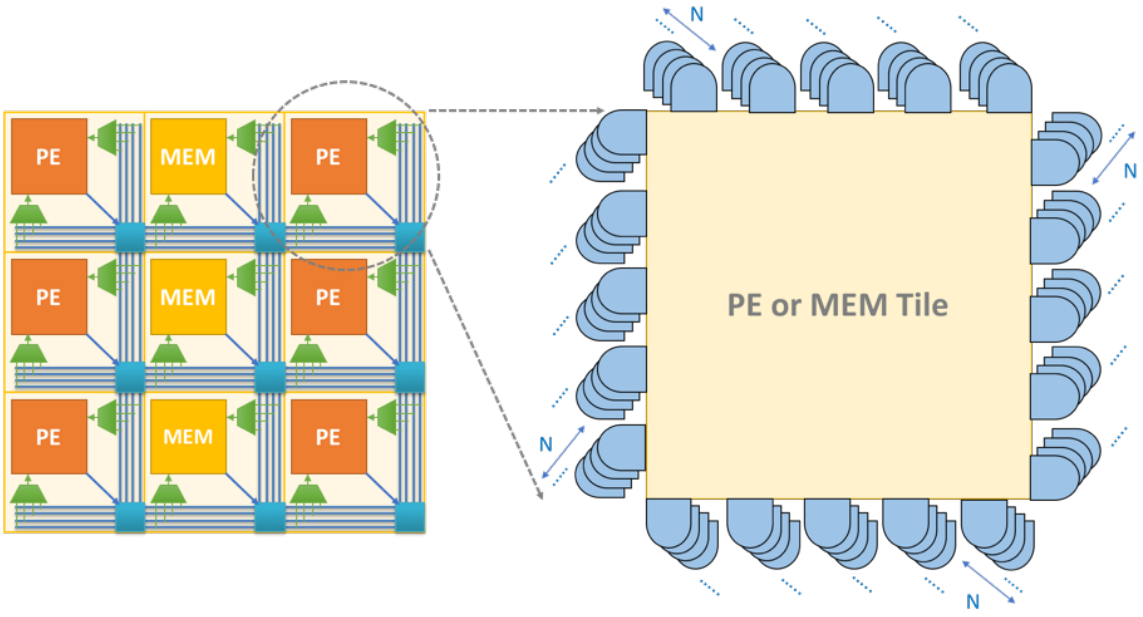


Figure 4.7: To make every tile its own power domains, isolation cells need to be placed on all outputs of each tile.

design, uses compiler-like "passes" to introduce the circuits needed for a power-domain-aware CGRA, as illustrated in Fig. 4.10. We created our framework using **magma** [29]. Magma is an open-source domain specific language (DSL) embedded in Python for describing circuits. It is similar to Chisel [14], which is embedded in Scala. Magma abstracts circuits as Python classes, which can be instanced and wired together in a structural manner.

A dynamic layer on top of magma allows for staged generation of circuits. It provides some primitive functions to modify circuits: `add_port(name, type)` and `remove_port(name)` to add or remove a port to/from a module definition, and `wire(src, dst)` and `unwire(src, dst)` to connect or disconnect ports `src` and `dst`. Together with the ability to replace existing modules and instantiate new modules, these primitive functions allow us to define a "pass," which is any function that traverses the fundamental data structure (in our case a hierarchy of hardware modules), and adds, removes, replaces or modifies its elements.

To create our framework, we wrote passes that perform circuit transformations for adding power domains. This allows us to add power domain-related features to any design in a decoupled fashion, without rewriting its original logical description. Since magma and gemstone are embedded in Python, we can use all of the faculties available in Python to modify circuits. This allows us to express these passes very succinctly with a few lines of code, as shown in Fig. 4.11. We describe the

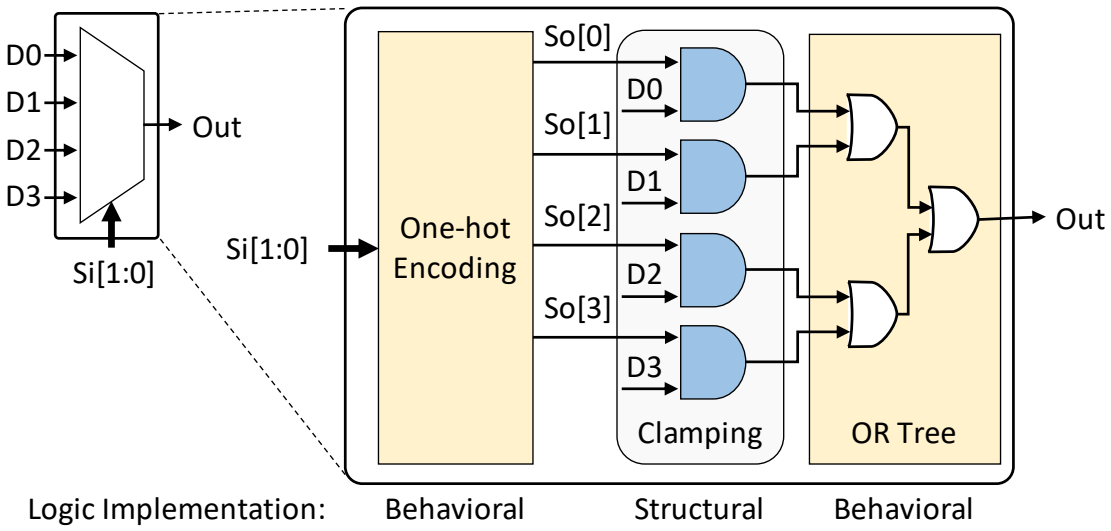


Figure 4.8: Isolation logic is embedded in SBs/CBs by breaking down their multiplexers into three stages: one-hot encoding, clamping and an OR tree. This figure shows the re-architected 4-input mux present in the switch box from Fig. 2.3b.

passes that comprise our framework next.

### 4.3.1 Boundary Protection Pass

The boundary protection pass replaces SB and CB multiplexers in the base design with the re-architected multiplexers from Section 4.2.4, which have inherent boundary protection circuitry. To accomplish this, the pass `unwires` the existing multiplexers and instantiates and `wires` up our modified multiplexers in their place. The code for this pass is shown in Fig. 4.11. It must be noted that if the user desires, this pass can also be used to perform boundary protection using the conventional AND/OR isolation cells described in Fig. 4.4. This would involve `unwire`'-ing all connections from the output port that needs to be isolated, instantiating an AND cell, `wire`'-ing the output port to the AND output, and routing the AND input to all of the original connecting ports.

For the sake of clarity, we discuss further the implementation of the clamping logic inside the re-architected multiplexer. Since the clamping logic is what mimics the isolation cell behavior, the first cell that the select signals and the data inputs encounter in the clamping logic needs to be a 2-input AND cell. One way of ensuring that this design requirement is met is limiting the list of standard cells usable for the clamping logic during the implementation phase (synthesis and place & route) to 2-input AND cells (or AND-OR (AO), or AND-OR-Inverter (AOI)) and then adding some 'dont-use' constraints to this hierarchy to ensure that the desired cells are not swapped with

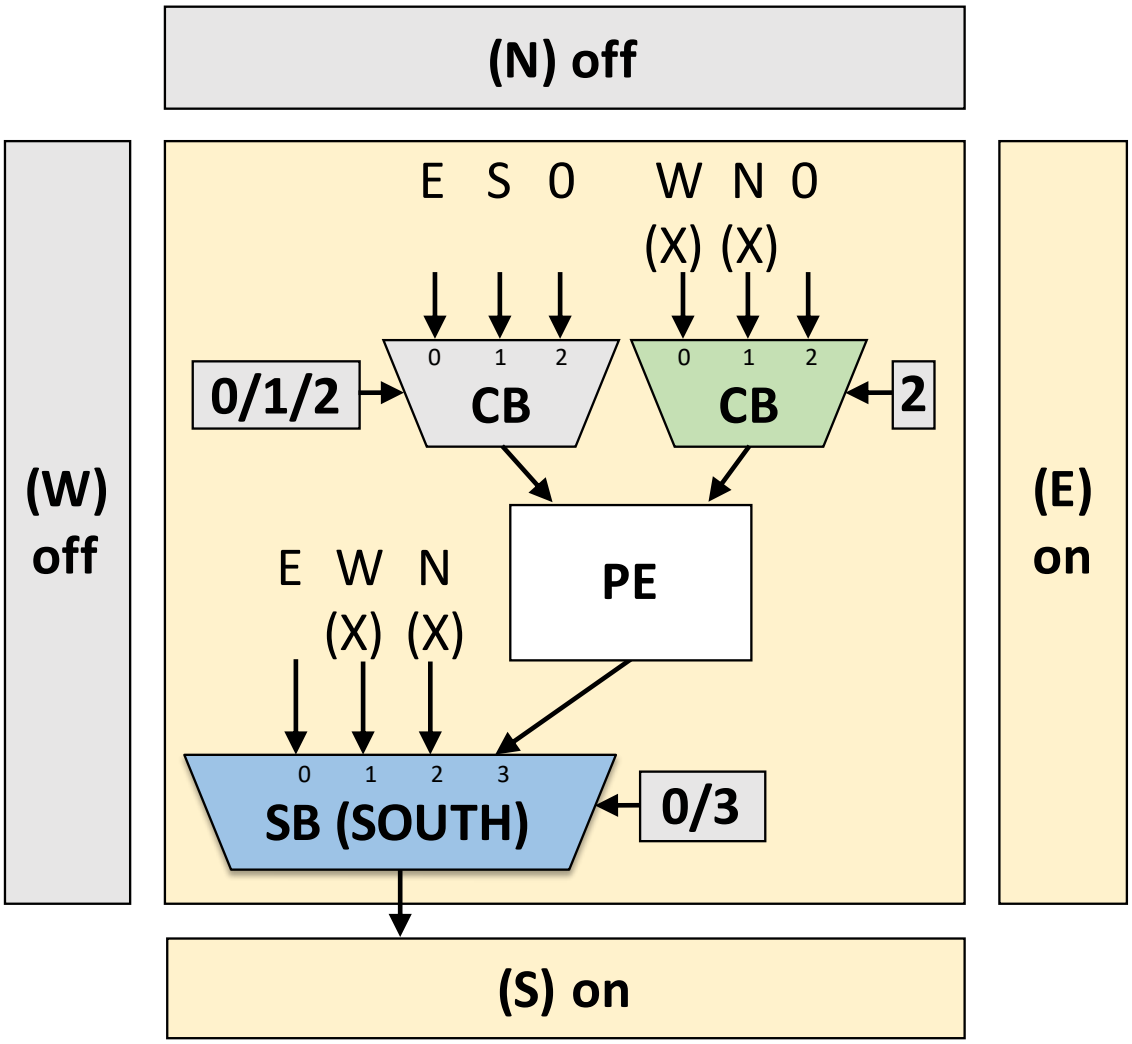


Figure 4.9: PE tile with modified CBs and SBs. Gray boxes indicate which signal can be selected by the muxes to avoid X-propagation. For simplicity of illustration, all inputs coming from a direction are grouped into one arrow for the CB muxes.

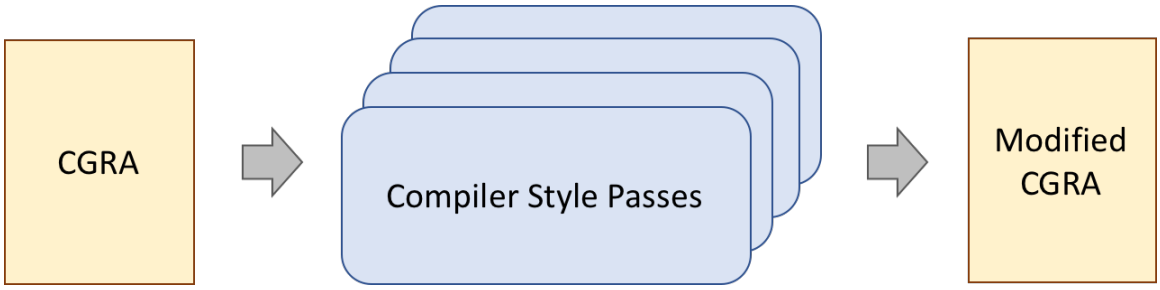


Figure 4.10: Starting with the basic CGRA design, the framework uses compiler-like "passes" to introduce the circuits needed for a power-domain-aware CGRA.

different "functionally equivalent" standard cells. This should allow the synthesis tool to translate the behavioral logic to the desired structural Verilog that is needed for the clamping logic. However, extensive verification of our chip was done at the RTL-level, and only a subset of the tests were run at gate-level, so our pass implements the clamping logic directly as structural logic, while the front-end one-hot encoding and the back-end OR-tree is still maintained as behavioral logic.

As discussed, this structural clamping logic should not be modified by the downstream implementation tools. In the absence of any constraints, the synthesis tool could restructure the clamping logic and implement it by using other functionally equivalent standard cells, i.e., swapping AND cells with INV+NOR cells. However, while the swapped logic would functionally still be correct for SB/CBs, it would break the clamping logic requirements. Hence, once the structural choice of standard cell is made during initial RTL generation, it is important that the same logic be maintained throughout the design flow. The clamping logic can, however, be resized to allow the tool the flexibility to improve timing if needed. For example, if AO cells of 1x drive strengths are inserted in the RTL generated by Gemstone, the downstream tools should be allowed to resize these cells, i.e., use another drive strength or use any other Vt type; note, however, that they must not be decomposed using other standard cells.

Finally, it is important that no buffer be inserted on the path to this clamping logic. The incoming X needs to first see the clamping logic; any other cell insertion on this path, for example a buffer/inverter pair, will cause static power dissipation in these cells, as shown in Fig. 4.3. To address this requirement, we added design constraints on the clamping logic hierarchy in the switch boxes and connection boxes during synthesis and P&R steps. The constraints do not permit transformation but instead allow only resizing of the clamping logic and prevent buffering on the nets between the SB input ports and the clamping logic. These constraints force the synthesis and P&R tool to insert any needed buffers *after* the clamping logic.

Additionally, we introduced checks in the flow at every step in P&R to ensure that this clamping logic is not changed and no buffer or inverter cell is added on the nets between the SB input ports

```
def boundary_protection_pass(interconnect):
  # Iterate through all tiles in the CGRA
  for (x, y) in interconnect.tile_circuits:
    tile = interconnect.tile_circuits[(x, y)]
    # Find the muxes in CB in the tile
    muxes = find_available_muxes(tile)
    for mux in muxes:
        # Create new mux instance
        pd_mux = PowerDomainMux(mux)
        # Unwire and replace the old instance
        # Wire the new instance
        unwire (cb.ports, mux.ports.in)
        unwire (mux.ports.out, pe.ports.in)
        wire (cb.ports, pd_mux.ports.in)
        wire (pd_mux.ports.out, pe.ports.in)
```

Figure 4.11: Since magma and gemstone are embedded in Python, we can express passes succinctly with a few lines of code. This code example shows the boundary protection pass.

and the clamping logic. We do so by reporting out the first level of cells from the fanout of the SB data input ports and ensure that the cell is always an AND-OR cell (or whichever structure was originally chosen during RTL generation), implying that no buffers were inserted on the path to the clamping logic and the clamping logic was maintained. We allow resizing of these clamping cells to help with timing.

We did see diode cells getting inserted on the input for antenna violations. Since diode cells do not have static power, they do not fanout elsewhere and are directly connected to the substrate, these cells were permitted on the input paths. Alternatively, we could fix antenna violations through layer hopping; either of these design choices are acceptable with respect to the clamping logic.

### 4.3.2 Power Switch Insertion Pass

Fig. 4.12 shows the power switch insertion pass, which instantiates and connects power switches to a configuration register that allows the software to control which tiles are powered on. This pass then uses the `wire` primitive function to connect these in different styles, e.g. daisy-chain, all-fanout, or hybrid. The addressing for the configuration register and decode logic is also automatically added by this pass, reducing manual effort from the user. If the user desires to add and connect the power switches during physical design using commercial tools, the user can skip adding the switches and just add the configuration logic. For users interested in RTL-level functional verification, this pass will cleanly add and connect the switches at the RTL level.

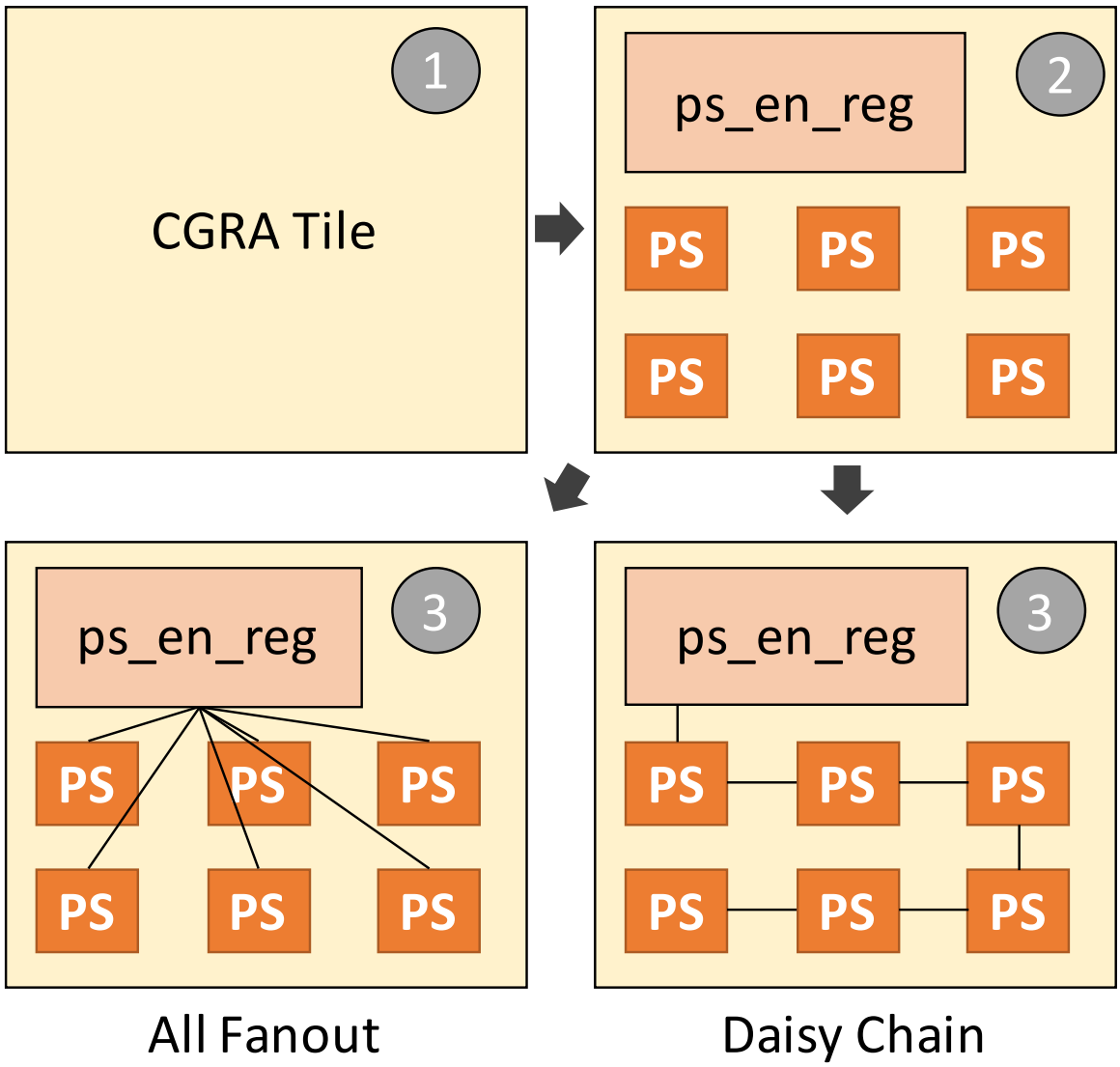


Figure 4.12: Power switch insertion pass adds power switches (PS) and a configuration register `ps_en_reg` that controls if the switches are enabled. These can be connected in different styles, such as all-fanout, or daisy-chain, as described in Section 4.4.3.

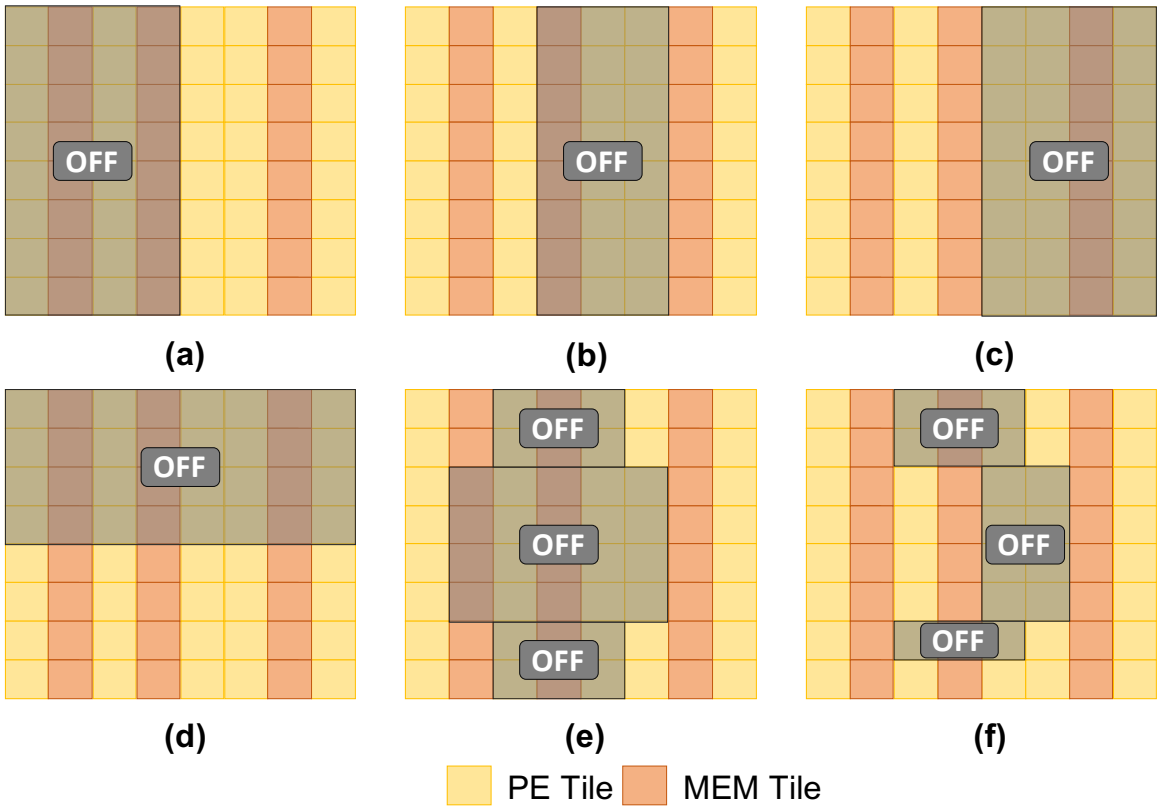


Figure 4.13: Configurable power domains: a sampling of six different ways to configure *on* and *off* tiles in our CGRA.

### 4.3.3 *Always-on* Buffer Insertion Pass

Once we have inserted the boundary protection and the power switches, we now have the capability to independently turn on or off any tile in the fabric. As illustrated in Fig. 4.13, the P&R tool that maps the application on the fabric now has the full freedom to choose any of these or other ON-OFF topologies. But one challenge remains.

In our CGRA, global signals like `clock`, `reset` and configuration `address` and `data` flow from top to bottom *through* the tiles in each column. Thus, if the top tiles in any column were *off*, this would turn off the global signals to the bottom tiles. This is the case illustrated in Fig. 4.13 d - f. To solve this issue, as illustrated in Fig. 4.14, we treat the global signals as *always-on* feed-through nets, so that even when a top tile is *off*, the bottom tiles can receive the global signals. The *always-on* buffer insertion pass inserts *always-on* buffers on these feed-through nets using a process similar to the boundary protection pass. While this takes care of the feedthrough nets, the global signals that have combinational logic need additional handling, which is managed through the debug signal isolation pass.

### 4.3.4 Debug Signal Isolation Pass

There are several test and debug signals in our CGRA that are affected by the introduction of power domains. For example, as shown in Fig. 4.15, the CGRA contains circuitry to read out any configuration register in any tile. Each tile receives a configuration address as described in the previous pass, which it decodes locally to determine if it must put out the value of one of its configuration registers on the `read_config_data` bus. If no register in a tile is selected, the output is zero. All `read_config_data` signals are OR'ed together using an OR chain.

If any tile is *off*, it will send an X into the OR chain, which might corrupt the configuration read-out. The debug signal isolation pass handles this scenario. When the tile is *on*, the transformed circuit on the right behaves as OR logic, but when the tile is *off*, it blocks X-propagation from the *off* tile while allowing the upstream debug signal to propagate.

By successively applying these passes, we were able to introduce the transformations required for configurable power domains in the CGRA. It must be noted that these passes are not specific to our CGRA and can be leveraged by other designs. It is also easy to add design-specific passes in our framework using the basic circuit modification primitives.

### 4.3.5 UPF File Generation Pass

Since configurable cores can be added at any time during hardware generation, we need a flexible, yet reliable, way to generate a UPF file. This UPF file is used for tile-level implementation, the details for which are explained later in Section 4.4. As shown in Fig. 4.16, we use a pass to dynamically introspect the circuit and identify the components that have to be in the *always-on* domain.

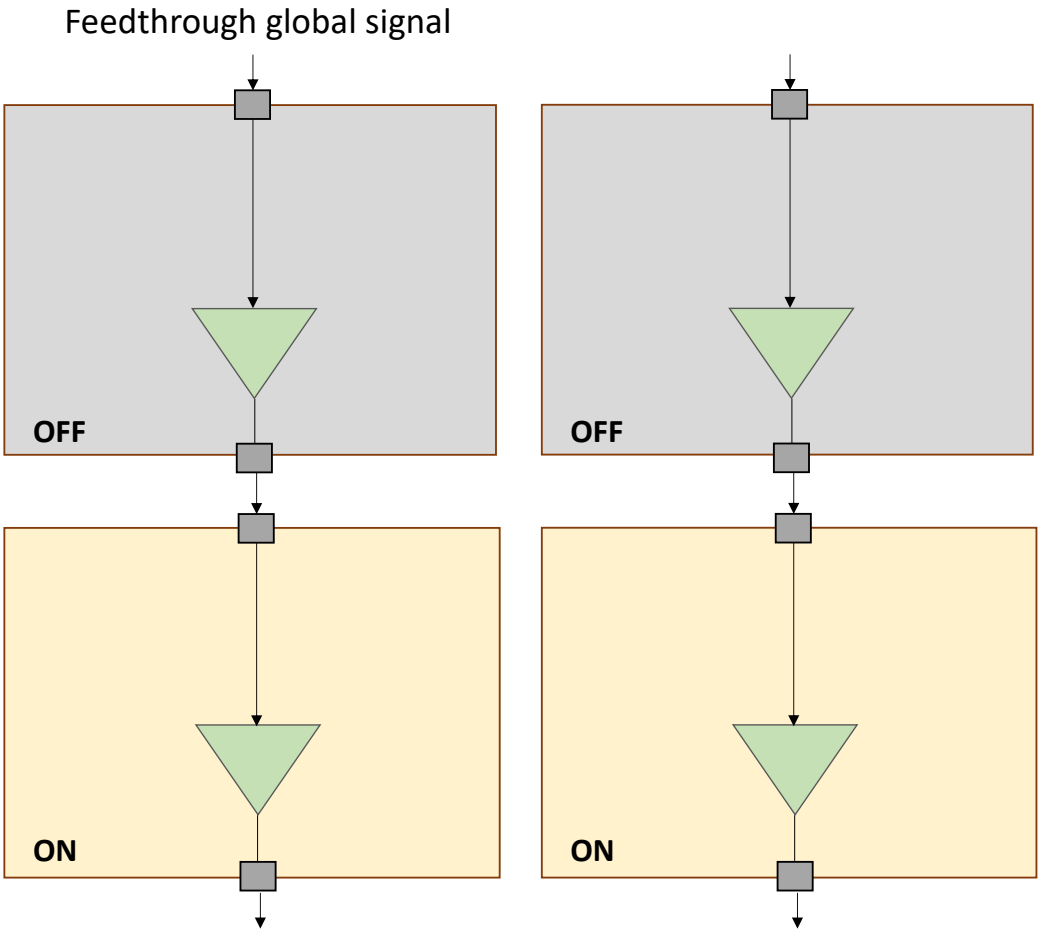


Figure 4.14: Global signals are treated as *always-on* feed-through nets and *always-on* buffers are inserted on these nets.

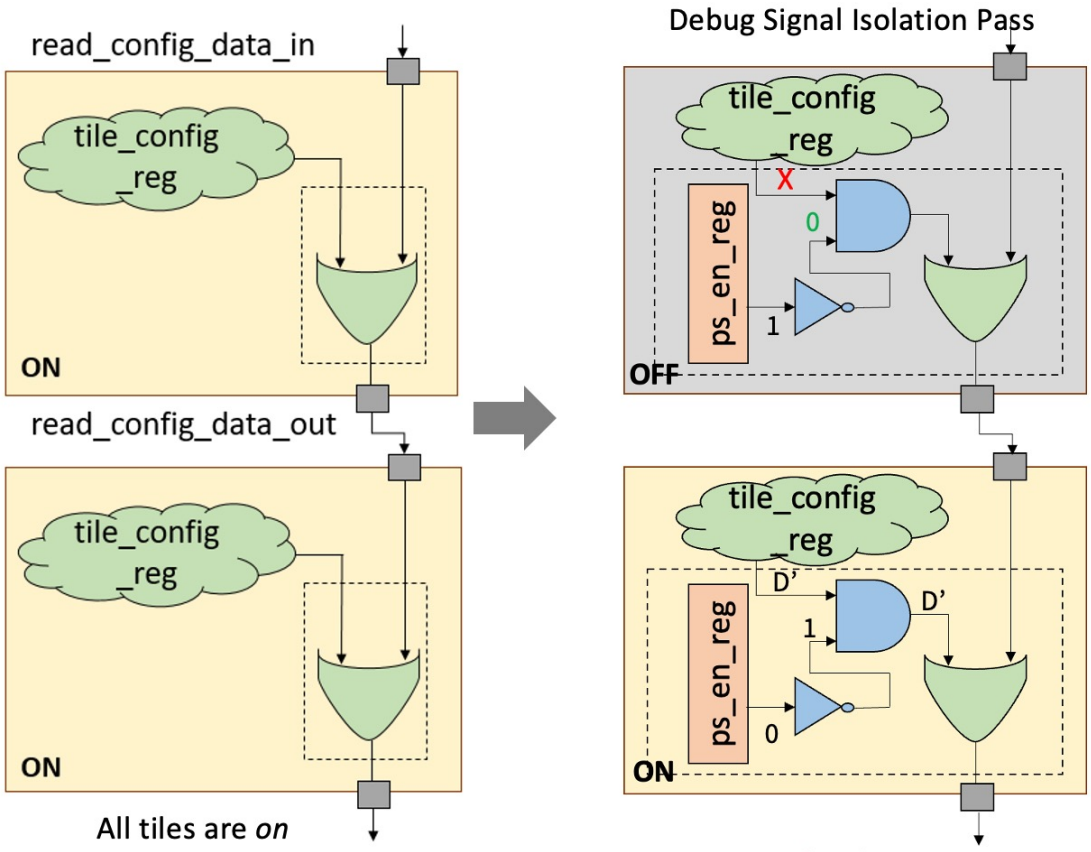


Figure 4.15: Debug signal isolation pass. When the tile is *on* (`ps_en_reg = 0`), the circuit is in normal mode of operation (left column). When the tile is *off* (`ps_en_reg = 1`), the circuit blocks the X-propagation from the *off* tile while allowing configuration data to propagate (right column). (Gates inside dotted box are *always-on*.)

```
def always_on_domain_extraction(interconnect):
  # Iterate through all tiles in the CGRA
  # and extract always-on circuits
  result = []
  for (x, y) in interconnect.tile_circuits:
    tile = interconnect.tile_circuits[(x, y)]
    # Find the power config register
    registers = find_configuration_registers(tile)
    for reg in registers:
        if not isinstance(reg, PowerDomainReg):
            continue
        # Compute connected circuit component
        connected = connected_component(reg)
        result += connected

instances = always_on_domain_extraction(design)
upf_str = create_upf(instances)
```

Figure 4.16: *Always-on* domain cells extraction pass.

During the introspection, we extract configuration-related instances and generate UPF commands automatically. This process ensures the correctness of the UPF file even though the CGRA may change during development.

### 4.3.6 End-to-End Flow

Fig. 4.17 shows our end-to-end flow for adding power domains. We start with a base CGRA built in magma with some user-defined configuration. This configuration includes (1) tile parameters, such as the number and layout of PE and MEM tiles, PE instruction set, and MEM size and addressing modes and (2) interconnect parameters, such as number of routing tracks and the degree of connectivity inside SBs and CBs. We then perform power domain passes on this base CGRA to create a CGRA with fine-grained power domains. The passes produce the modified CGRA RTL, which is now power-domain-aware, *and* the UPF file for the downstream physical design flow of the tiles. This allows us to quickly iterate on different design choices described in the power domain configuration without tedious manual intervention.

In this approach, the design transformations to generate the modified CGRA RTL take just a few minutes to run. We then use commercial EDA tools to perform synthesis, P&R and IR analysis. After P&R, we review the power-performance-area (PPA) and IR results. If the metrics do not meet our PPA and IR budget, we modify the power domain configuration to generate a new instance of the CGRA. By changing the base design and power domain-related configuration, this framework can quickly generate a different CGRA variant.

## 4.4 Power-Domain-Aware Chip Implementation

We used our framework to insert power domains into an SoC with an ARM Cortex M3 processor and a CGRA with $32 \times 16$ PE and memory tiles and 4 MB secondary memory, as shown in Fig. 4.18. In this section, we discuss additional challenges in implementing a CGRA with fine-grained power domains, such as tile addressing, power grid design, well substrate connection and distribution of global signals.

### 4.4.1 CGRA Tile Power Domains

This section describes the various components of the CGRA tile's power intent and the power intent flow we use for tile-level power-domain-aware physical design. We use two power domains *inside* a tile: an *always-on* (AON) domain that holds the minimum amount of logic that must remain on when the tile is turned off, and a *switching* (SW) domain which holds everything else. The *always-on* domain contains: configuration decoding logic and the register that drives the enable pins of the power switches (Fig. 4.12); debug logic needed to read configuration data from the tiles even when the tiles are off (Fig. 4.15); and tie cells for generating tile IDs, to ensure that the tiles can be turned back on through the configuration bus (Section 4.4.5).

The AON domain is physically small, and we place it close to the pins at the top edge of the tile, so that the necessary logic is close to the driving pins. Fig. 4.19 and 4.20 show the layout of the PE and memory tiles respectively with the AON domain, SRAMs and power switches.

As described in Chapter 2, the global signals in our chip flow vertically down in each column. Following global signals are maintained as always-on signals:

- Clock signals: All clock signals are maintained as *always-on*, so that even if the tiles are turned off, the clock signals have *always-on* buffers on their paths so the clocks can stay on.
- Configuration signals: These signals include the configuration address, data, read and write signals for each tile's power switch configuration register `ps_en_reg`. Since these signals flow through the top *shutdown* power domain, they would be turned off when the top domain is off. Hence, these signals are maintained as *always-on* to ensure that the buffers inserted on these signals are *always-on* buffers, such that the tiles can be turned back on using the configuration logic instead of having to turn the tile back on only through reset.
- Debug signals: This includes the signals for reading the configuration data out of the tiles (Fig. 4.15). As described in Sec. 4.4.1, the logic on the configuration read bus is placed in the *always-on* region. This signal is also marked as *always-on* to ensure that buffer insertion, if any, on this path is an *always-on* buffer so that the configuration bus is not broken when it is routed through the *off* tiles in a given vertical column.

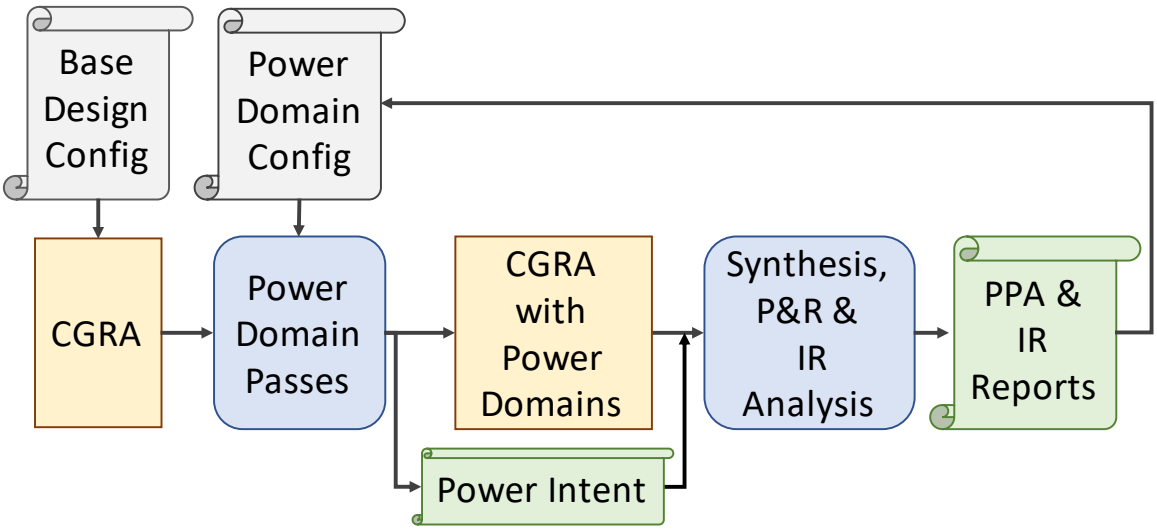


Figure 4.17: Our end-to-end flow for adding power domains.

- Tile-ID associated signals: For similar reasons as configuration and debug signals, the signals associated with the tile ID are also maintained as *always-on*.
- Stall signals: Stall signals stall the entire fabric before loading in new configuration bits.

Our goal is not to substitute what can already be done by commercial tools, so we used the standard UPF-based flow to do the tile-level power-domain-aware physical design. In our tile-level UPF file, we describe the two tile-level power domains (AON and SW), their states and the primary power supplies.

It is important to note that because there are already two power domains within a tile, we need to ensure that the signals from the top *switching* domain that drive the AON domain also have the boundary protection. But these signals do not need any special isolation cells since all the inputs to the AON domain are either the SB inputs that are clamped using the special multiplexers or the global signals that are *always-on*. Hence, X-propagation from the *switching* domain to the AON domain within a tile is not a concern.

### 4.4.2 Power Grid Implementation

Our baseline design without power domains uses a dual-mesh power strategy, with a fine-grained power grid on lower metal layers to help reduce $di/dt$ and a coarse-grained power grid on higher metal layers to reduce IR drop. Re-using this power grid from our baseline design, we had two choices

for the power domains: (1) separate power grids for AON and SW domains, or (2) a homogeneous power grid common to both domains.

When each domain has its own power grid, the SW domain grid requires both VDD and VDD_SW supply nets. The AON domain requires only the VDD supply. However, the AON domain is a significantly small fraction of the overall area, and routing VDD_SW over the AON region has minimal impact on the routing resource utilization, so we decided to use a homogeneous grid, as shown in Fig. 4.21. This also allows for common IR analysis over the two domains. Additionally, as shown in Fig. 4.22, we ensure that the horizontal VDD_SW stripes extend only to the core edge instead of the edge of the tile, so they do not abut the neighboring tile's VDD_SW. Therefore, each tile can be individually turned on or off.

### 4.4.3 Power Switch Insertion

Power switches within a tile can be connected in various ways—daisy-chain, all-fanout or hybrid (Fig. 4.12). Fig. 4.23 shows the two types of power switches, unbuffered and buffered.

The buffered power switch has an internal buffer that helps skew the enable signal of the power switch by introducing delay as the enable signal traverses the column of power switches. In a daisy-chain connection, the buffered enable signal of one power switch connects to the enable pin of the next power switch, allowing them to turn on or off one after another instead of all at once. This method avoids high inrush current, but the wake-up time is proportional to the number of power switches.

An all-fanout pattern can use unbuffered power switches. In an all-fanout pattern, all the power switches within the tile turn on simultaneously. While this technique improves the wake-up time, the inrush current can be outside the tolerance range of the power grid and the power switches. A good balance between minimizing wake-up time while keeping the inrush current in control is to have multiple columns of daisy chains.

We wanted to ensure that the power switch turn on time is a significantly small fraction of the overall configuration time. At the same time, we wanted to ensure that the power-up spike is not significant when a tile is turned on. Our wake-up time analysis showed that even with a single daisy chain, the wake-up time was indeed a very small fraction of the overall configuration time (see results in Section 4.5.3).

The number of power switches is determined by the IR budget. Based on IR analysis (Section 4.5.3), we chose a checkerboard pattern for the power switch count and insert power switches . We chose to have the power switch insertion pitch and offset in such a way that the power switches overlap with the vertical VDD stripes, and the *always-on* pin of the power switch can be connected to VDD easily through a via stack, as shown in Fig. 4.24. This design ensures that the VDD stripes do not need to be routed in a non-standard manner to be connected to the *always-on* VDD pin, allowing for cleaner connection and reducing the routing congestion. If the lower level metal layers

are fine-grained, then this constraint is not as significant. But if the power grid is coarse-grained, additional handling for alignment with the power pin of the power switch would be required.

### 4.4.4 Well Substrate Connection

The well substrate connection required special handling since our design has two power grids, VDD and VDD_SW. Typically, the substrate connection happens through a column of tap cells inserted at regular intervals determined by DRC rules for the given technology. In the AON domain, we connected the substrate of the standard cells to VDD through these tap cells. The substrate connection in the switching domain also required special handling due to the presence of multiple power grids; thus, the substrate can be connected to either VDD or VDD_SW. Connecting to VDD_SW would be optimal because there would be no contribution to the leakage from the substrate when the tile is *off*. However, there are some constraints in connecting the substrate in the switching domain to VDD_SW.

Power switches are inserted in the *switching* domain, and they connect to both the VDD and VDD_SW supply rails. For a power switch cell with a single well, the well has to be at least as positive as the *always-on* VDD, and this well must hence be connected to the *always-on* VDD, and not VDD_SW. But if the substrate of the rest of the cells in the switching domain were to be connected to VDD_SW, there would have to be well separation between the cells connected to VDD and VDD_SW. This would be done by using split NWELL power management cells with in-built well separation. Since such cells were not available in the standard cell library at the time of our implementation, we connected the substrate of the switching domain to the *always-on* VDD.

This choice provides two options for handling the substrate connection. The first is to insert tap cells in the *switching* region. Typically, in a tap cell, the voltage that drives the power pin drives the substrate connection. However, if the tap cell is inserted in the *switching* region, its power pin needs to be driven by the primary power supply, which is VDD_SW, but the substrate needs to be driven by the *always-on* VDD. The standard cell library provides a tap cell variant where the power and substrate voltage drivers are different, which we can use in our case. The power pin can be connected to primary VDD of the SD domain, i.e., the switching VDD net, but the substrate can be connected to *always-on* VDD. To support the substrate connection, these tap cells need to be placed such that the substrate connection pin overlaps with the *always-on* power grid to avoid routing of power grid as signals

The second option is to connect the substrate through the existing power switches, thus avoiding the need for tap cells. We used this method for our design. Since we inserted the power switches in a daisy-chain fashion, we already had column of power switches inserted in a checkerboard pattern. We allowed the substrate connection to the VDD grid through these column of power switches. The responsibilities of the power switches and the tap cells are separate, and the criteria to determine the number of power switches and the tap cells are also unrelated – power switch count is determined

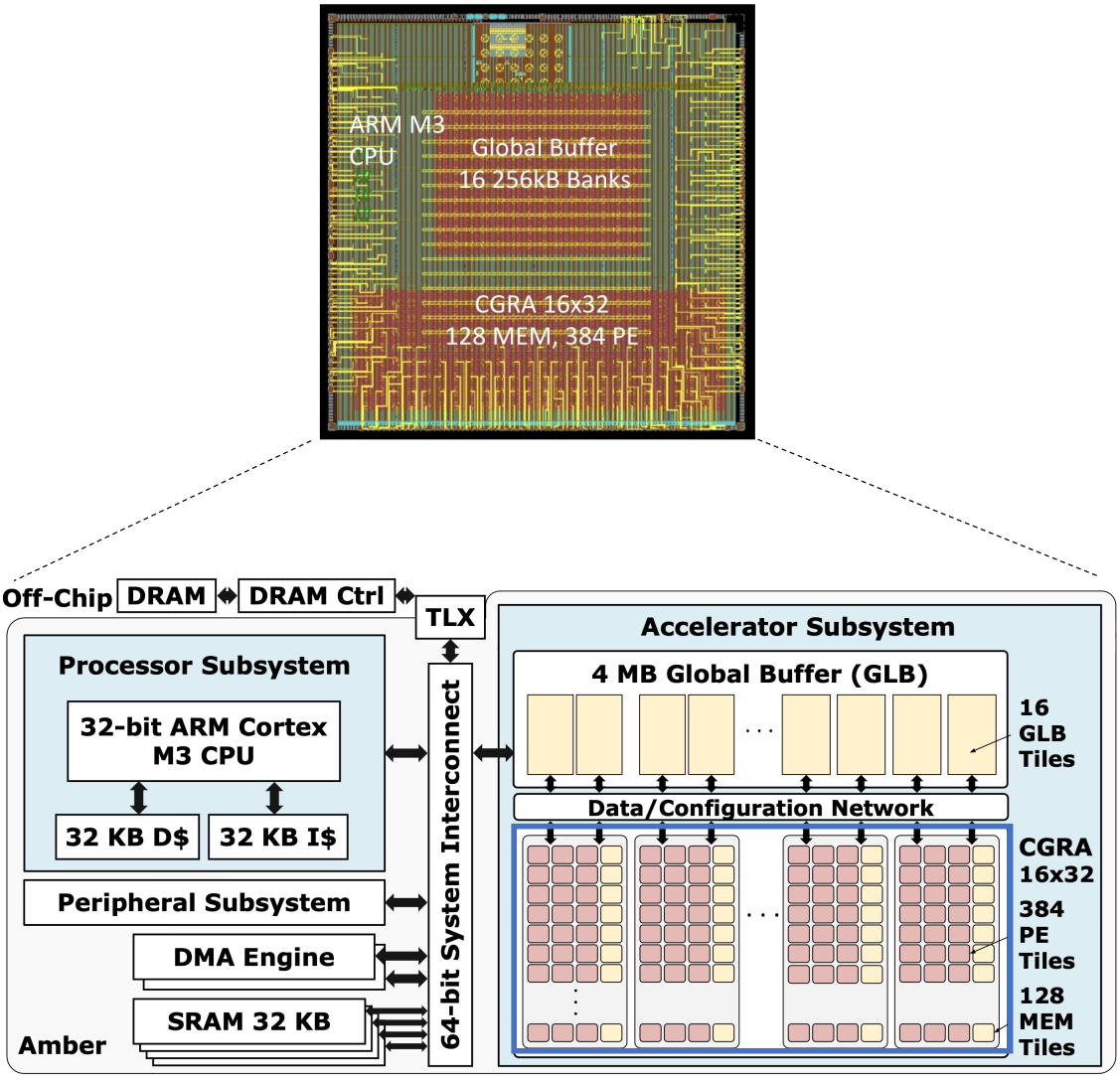


Figure 4.18: Layout of the SoC which includes a processor, secondary memory, and a $32 \times 16$ CGRA with memory tiles and processing elements. The chip is taped out in 16 nm.

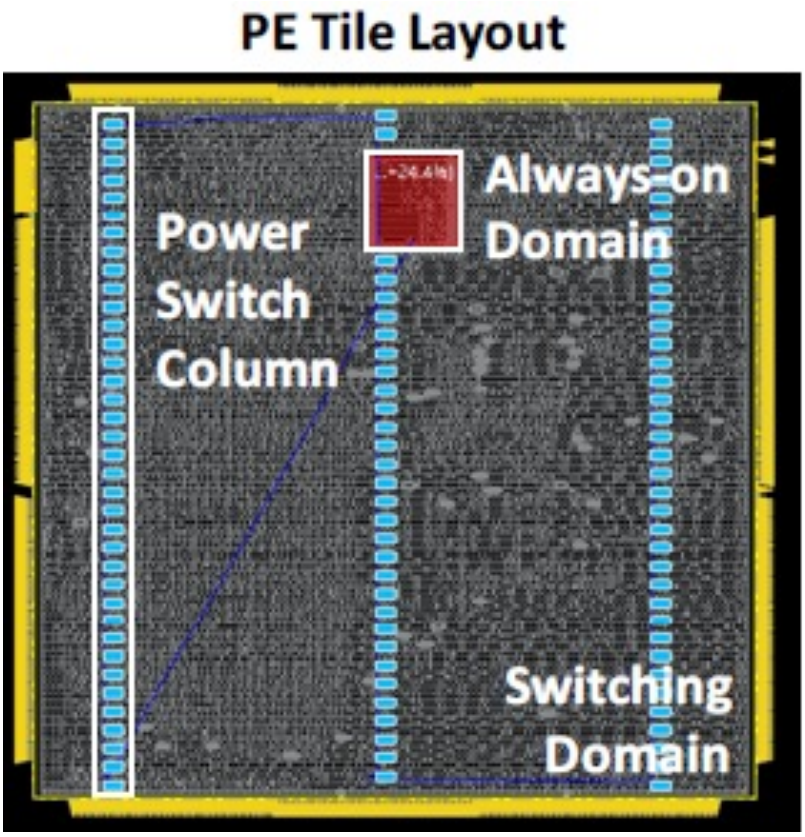


Figure 4.19: Layout of the switchable PE tile with *always-on* domain (red) and columns of power switches (blue). The entire PE tile, with the exception of the *always-on* region, power switches and *always-on* buffers, can be turned off when not in use.

through IR analysis, and tap cell column spacing is determined by the DRC constraint for a given technology. However, the pitch of the power switch column per row was less than the requirement for tap cell insertion to avoid latch up DRCs. We therefore used the power switch itself for substrate connection and did not include an additional column of tap cells to ease routing and congestion. However, the top and bottom boundary of the tiles now had only boundary cells and no logic cells. As a result, power switches were not inserted in those rows. So, as shown in Fig. 4.25, tap cells were inserted only in the top and bottom boundary rows, aligning with the power switch column. The power pins of these tap cells are connected to VDD_SW, and the substrate is connected to the *always-on* VDD. These tap cells are placed such that the substrate connection pin overlaps with the *always-on* power grid to avoid routing of power grid as signals for the substrate connection. The design height is maintained to keep the number of rows even such that no row in the core area of

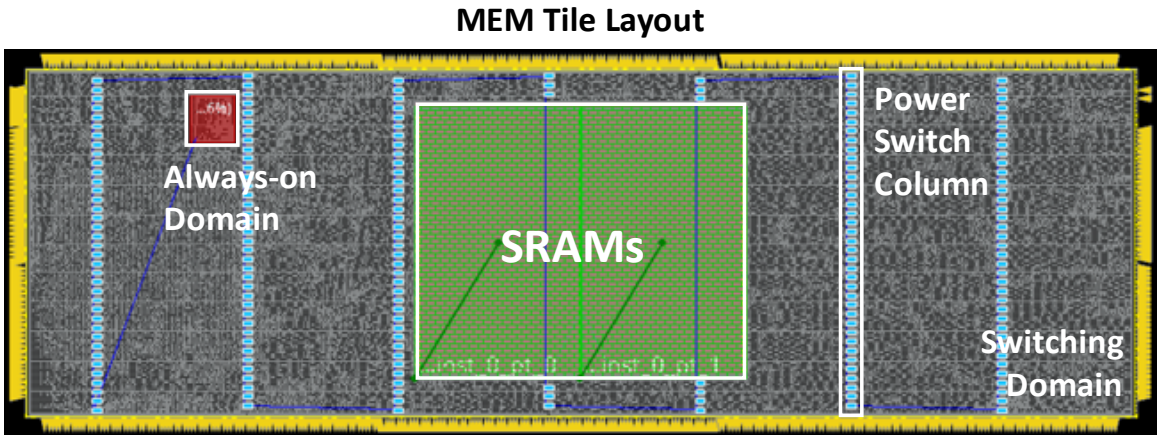


Figure 4.20: Memory tile with SRAMs (green), *always-on* domain (red) and columns of power switches (blue). The entire memory tile, except the AON region, power switches and AON buffers can be turned off when not in use.

the design is without a power switch. (Power switches in our design are double height cells.)

### 4.4.5 Tile Addressing

Each of our tiles has `tile_id` input pins that define that tile's unique address. Each pin has an unconnected tie-hi cell on one side and a tie-lo on the other. For our CGRA, we used hierarchical physical design and hence we aimed to do the physical design for only one tile, per tile type. As a result, the `tile_id` connections needed to be made at the top level, outside the tile hierarchy. But since these tiles abut, it was not possible to insert the tie cells at the top level. Thus, they were included in the tile. At the top level, little strips of metal were laid down to connect each `tile_id` input pin to its adjoining tie-hi or tie-low cell, thus driving that pin with a constant `1` or `0`. The resulting sequence of 1's and 0's is the tile's address.

Each tile has logic that compares (a portion of) the global configuration address bus with its own `tile_id` to determine if the configuration on the bus is intended for it. Therefore, the tie cells controlling the `tile_id` must be always-on or a tile could not be turned on through reconfiguration after it has been turned off. The tie cells, being in the AON region, are slightly further away from the `tile_id` pins, but since these are constant signals ,they do not affect timing.

### 4.4.6 Handling AON Cells

In our design, AON cells are placed on paths that need to be *on* even when the tiles are *off*. While no additional handling is needed, care must be taken to ensure that they do not get placed such

that the *always-on* power supply is inaccessible to the AON cell's *always-on* power pin. This kind of an issue will usually manifest as an LVS error, since the *always-on* power pin of the AON cell will be unconnected and may need manual fixes.

## 4.5 Power-Domain-Aware Chip Verification

### 4.5.1 Formal Verification Using SMT

Our pass-based flow introduces a new concern: how do we verify whether the transformations made by a pass are correct? Conventional approaches insert isolation cells using a UPF-based flow [26], and commercial tools like Conformal Low-Power Verification [8] are used to ensure that signals expected to cross boundaries from *off* to *on* domains are isolated using UPF-based verification [74]. However, since our approach uses the custom boundary protection circuits from Section 4.2.4, we could not use commercial tools for verifying the boundary protection.

To address this problem, we utilized an SMT-based formal hardware verification tool, Pono (formally called CoSA) [58], to prove that the transformations are correct. Unlike simulation-based testing, formal verification is *exhaustive*. Proofs of correctness hold for all possible input sequences to the design, subject to any assumptions provided by the user. Furthermore, SMT-based verification exploits word-level structural information of the design, in contrast to the more traditional satisfiability (SAT) solving approaches [73], which operate on bit-level netlists.

Our goal was to formally verify that no X-propagation occurs in the chip. To achieve this, we were required to encode our condition as a property for CoSA to check. The first step was formalizing the concept of X-propagation. Formal tools do not have a notion of X values, so we could not directly encode the property "there does not exist X-propagation." An X value on an input simply means there are no assumptions on the possible input values in the formal tool. However, there is no easy way to state that X values have not reached the output. Fortunately, our boundary protection drives signals with a known, constant value. Thus, our verification condition is to ensure that when the tile is *off*, then for all possible inputs:

- the top-level outputs have a known constant value;
- internal signals beyond the first level of the SB and CB circuit have been masked to a known constant value; and
- the PE generates some known constant value.

With this formal definition of our desired property, we were able to encode all the verification conditions described in Section 4.2.4 in CoSA's property language. The resulting verification problems were straightforward, and CoSA was able to prove all of them in only a few seconds.

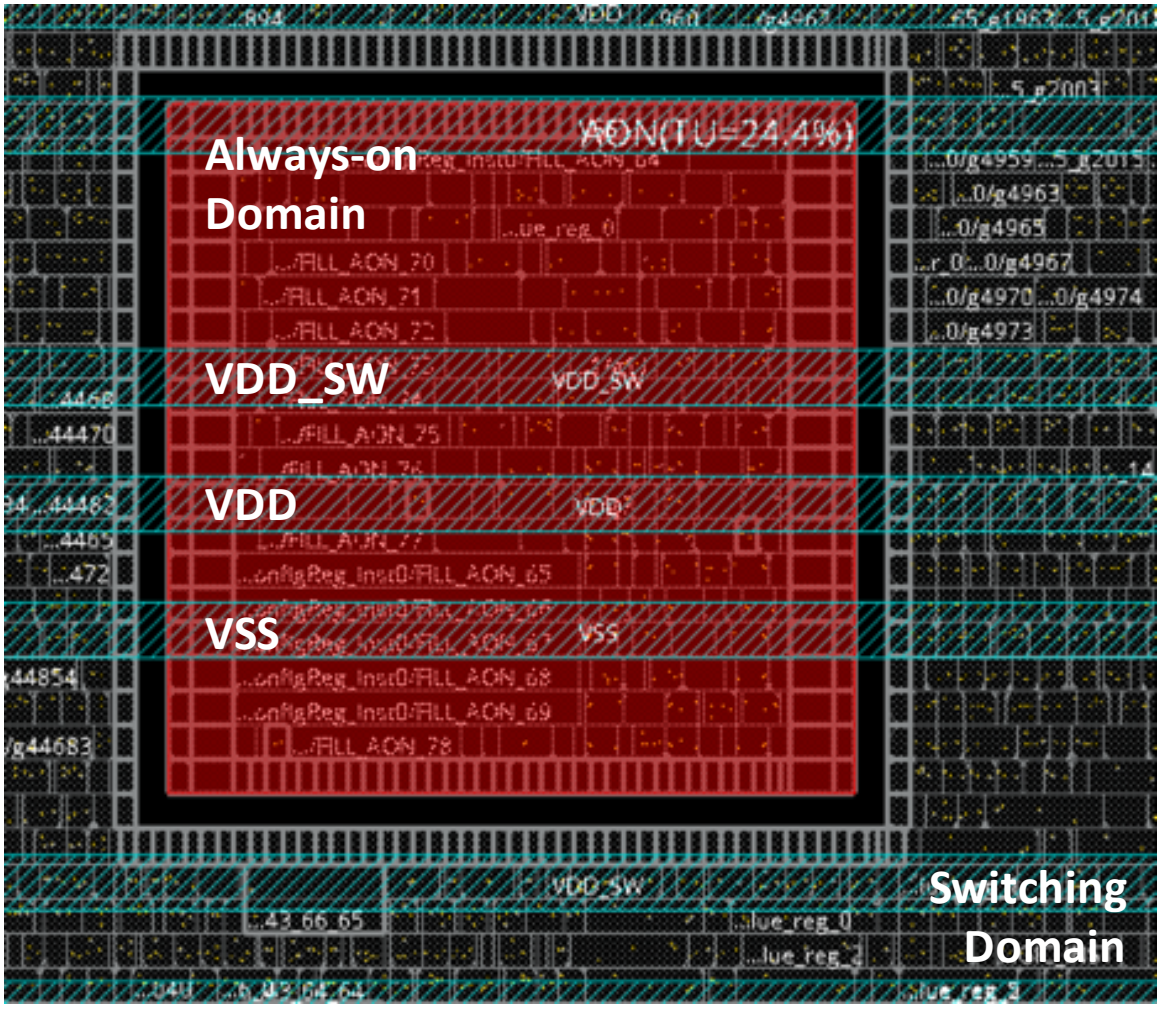


Figure 4.21: Homogeneous power grid over the AON and SW domains.

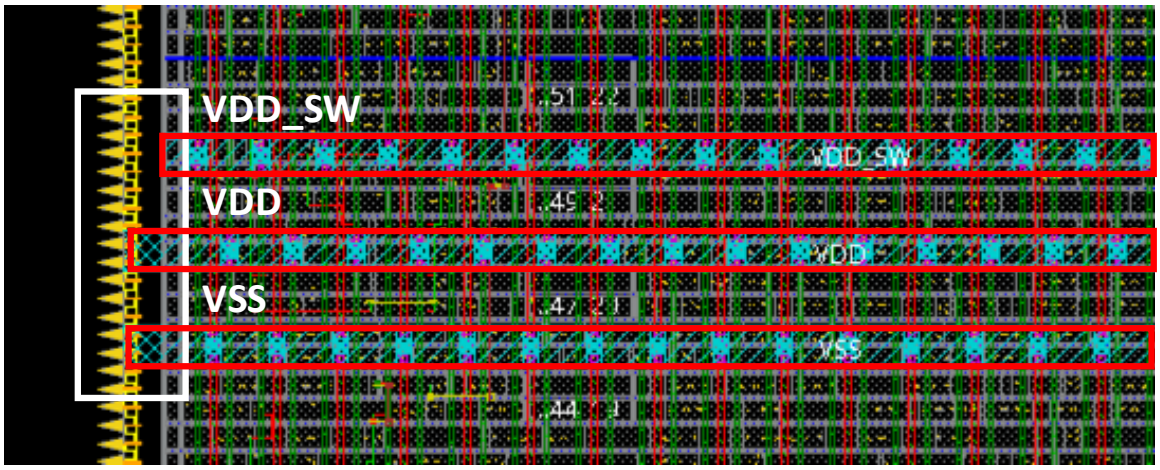


Figure 4.22: Unlike VDD and VSS, switched supply VDD_SW does not continue to the edge of the tile, allowing individual tiles to be turned on or off.

### 4.5.2 Power-Aware Gate-Level Verification

Power-aware gate-level verification is necessary, especially for designs with power domains. The functionality of a multi-VDD design can change if there is any issue or discrepancy between the VDD/VSS connections of the gates. Hence these checks need to be performed explicitly on the fully placed and routed design gate-level design.

We performed the following tests on the fully placed and routed gate-level design:

- *Reset test*: Checks if the tile turns on after reset.
- *Power switch disable*: Checks if the power switch can be disabled, i.e., the tile can be turned off through the configuration register.
- *Tile ID tie-cells*: Ensures that the tie-hi and tie-low cells for the tile ID are *on* even after the tile is turned off, so that the tile can be turned back on through the configuration bus.
- *Power switch enable*: Checks if the power switch can be enabled, i.e., the tile can be turned on through the configuration register. This ensures that all the cells on the power switch configuration logic that drive the power switch control are *on* even when the tiles are *off*.
- *Global signal test*: This check turns off the tiles and checks that the global signals are still on and set to the expected values. It ensures that the global signals flowing through the vertical columns in the CGRA array are not corrupted if any tiles in the columns are turned off.
- *Spurious tile enable*: This test makes sure that no other configuration address turns on the tile, i.e., configuration addressing of the tile is unique.

- *Power switch enable sequencing*: Checks that an *on-off-on* sequence for a tile works as expected.
- *Re-architected CB mux output checks*: Checks if the re-architected CB's final output is zero when the select signal corresponds to the last (constant) input.
- *Re-architected CB mux intermediate signal checks*: Ensures that, in the above case, CB multiplexer's intermediate outputs are also zero, i.e., no X-propagation occurs within the multiplexer.

### 4.5.3 IR Analysis

IR analysis is necessary to ensure an optimal number of power switches are placed in the design such that they satisfy the IR drop and the current density requirements. It is used to check both the operating condition, and the situation when a power domain turns on.

"On" steady-state IR analysis examines how the insertion of power switches impacts the IR drop when the switches are in a steady *on* state. Both static and dynamic power must be used to do this analysis. The power gate transistor is modeled as a linear resistance, the value of which is determined from the device model. The impact of this resistance is modeled during power and rail analysis to ensure that sufficient switches are added in the design to satisfy the current requirement of the switched block. Each switch is checked to see if it is operating in the non-saturated (linear resistance) region.

Equation 4.1 shows a way to calculate the number of power switches based on user-defined IR drop budgets and design parameters. We use typical budgets for IR analysis with $V_{DD} = 0.72$ V. We allow for $\alpha$, which is the total off-chip plus package plus on-chip IR drop budget as a percentage of the supply voltage, to be 10%, i.e. 72 mV. Out of this total IR drop budget of 72 mV, we assume $\beta$, the on-chip IR drop budget for power grid and power switches, to be 30% (assuming 70% for package and off-chip drop), which is 21.6 mV. Finally, out of the on-chip IR budget, we assume $\gamma$ = 50% or 10.8 mV for the IR drop across the power switches (the remaining 50% being assigned to the power grid). The worst IR drop across a power switch for either the PE or the MEM tile is well within this requirement ( ~6 mV).

$$
\begin{aligned}
\textit{Total IR drop budget} &\quad \alpha \\
\textit{On-chip IR drop budget} &\quad \beta \\
\textit{IR drop budget across switches} &\quad \gamma \\
\textit{Acceptable voltage drop across switches} &\quad \gamma\,(\beta\,(\alpha\,V_{DD})) \\
\textit{Power of the block (from P\&R) (mW)} &\quad P \\
\textit{Current through block (mA)} &\quad P/V_{DD} \\
\textit{Acceptable resistance across switches } (\Omega) &\quad \gamma\beta\alpha V_{DD}/(P/V_{DD}) \\
\textit{Resistance of power switch} &\quad R \\
\textit{Number of power switches} &\quad (RP)/(\gamma\beta\alpha {V_{DD}}^2)
\end{aligned}
\tag{4.1}
$$

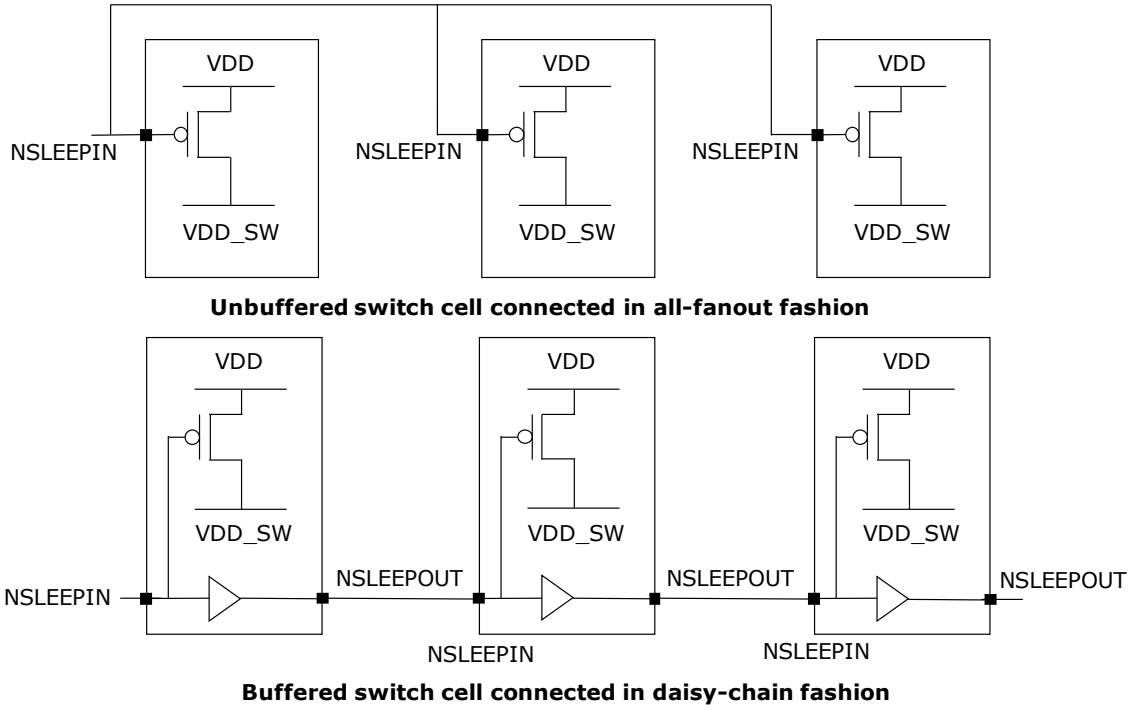


Figure 4.23: An unbuffered power switch cell can be used to connect in all-fanout fashion. In a daisy-chain connection, buffered power switches are used where the NSLEEPOUT of the previous buffered power switch is connected to the NSLEEPIN of the next power switch.

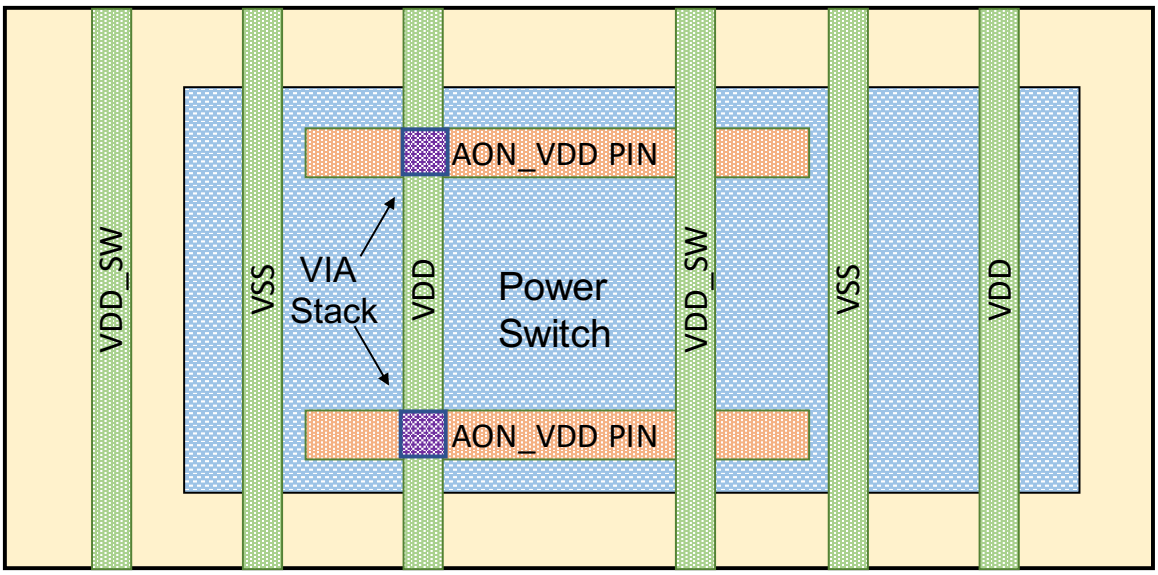


Figure 4.24: The power grid pitch and offset is aligned such that the VDD stripe overlaps with the *always-on* VDD pin.

To analyze the time required to turn on all the power gates, we performed wake-up time evaluation for the power switches using the Cadence Voltus tool. Faster wake-up time is good for timing, but the resulting high inrush current causes higher IR drop. Daisy chaining is good for limiting inrush current, but we needed to verify that the wake-up times were acceptable for the design. Our analysis shows that the wake-up time for the daisy-chained power switches is 17.5 ns for the memory tile and 9 ns for the PE tile. We can use up to a 750 MHz clock to program our configuration registers. So the wake-up time is about 14 clock cycles for the memory tile and 7 clock cycles for the PE tile. We were able to configure one register in 1 clock cycle with parallel configuration. With AXI-Lite, we could configure one register in about 8-10 cycles. There are over 100 configuration registers for memory tile and over 30 configuration registers for the PE tile. As we can see, the wake-up time is an acceptably small fraction ($<0.25$) of the overall configuration time.

In addition to the above verification, we performed various power-aware gate-level checks to thoroughly test the power domain-related functionalities, including enabling and disabling the power switches through the configuration register, to ensure that the global signals are *on* and set to expected values even when the tiles are *off* and turning back on the *off* tiles through the configuration.

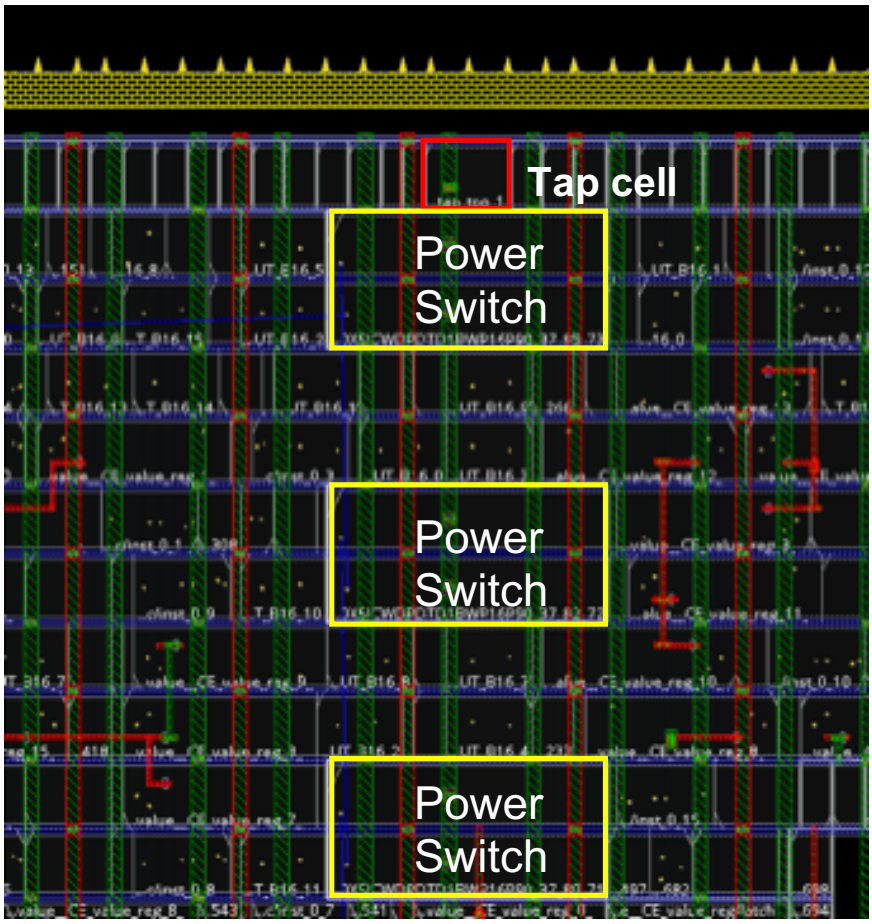


Figure 4.25: Tap cells are aligned with the column of power switches. The VDD vertical rail overlaps with the tap cell's power pin to avoid routing of the power signals.

## 4.6 Results

Table 4.1 shows the area overhead for our CGRA having 1) power domains with conventional isolation cell-based boundary protection; versus 2) power domains with our technique of circuit transformation-based boundary protection; each presented as percent difference from a design with no power domains (no boundary protection). The entire design uses mid-Vt cells except the power management cells, like power switches and always-on buffers, which use high-Vt cells. We avoid using the leakiest variants of the cells to avoid exaggerated leakage power improvements from shutting down the tiles.

As shown in Table 4.1, our power domain boundary protection technique requires less than 1%

Table 4.1: Area overhead of conventional vs. our technique

| **Boundary Protection Technique** | ***PE*** | ***Memory*** |
|---|---|---|
| PDs with Conventional Isolation Technique | +9.2% | +5.7% |
| PDs with Our Technique | +0.7% | +0.6% |

area overhead, while the conventional technique adds an overhead of 9.2% for the PE tile and 5.7% for the memory tile.

It is important to note that the area overhead for the conventional technique is variable and depends on CGRA design parameters such as the tile's I/O count and bit-width of the data buses, since these design parameters determine the number of isolation cells that need to be inserted. Since the boundary protection logic is embedded in our design, it is not impacted by the tile parameters and the overhead is fairly constant for different CGRA parameters. We added just enough power switches (4-6% overhead) to the designs such that the IR budgets are met.

The operating frequency for our CGRA when built with conventional boundary protection is 475 MHz. With our technique, the isolation cells on the critical paths are removed, so we can further push the frequency to over 500 MHz. When a tile is turned *off*, the leakage power savings for the PE and memory tiles are 22× and 46×, respectively. The power switch leakage contributes less than 2% to the PE tile leakage and 1% to the memory tile. Most of the leakage when a tile is off is from buffers added to keep the global signals on.

We ran a range of image processing and machine learning applications on the CGRA. The applications are written in Halide, a domain-specific language embedded in C++, and our compiler automatically transforms them into computation graphs. The official Halide [1] repository uses these applications to benchmark CPU and GPU performance. *Conv 1 × 2* is a 2D convolution that blurs two pixels horizontally. *Conv 3 × 3* is a 2D convolution with a $3 \times 3$ kernel. *Gaussian* is a convolution that blurs an image. *Demosaic* creates an RGB image from raw pixels. *Cascade* has two $3 \times 3$ convolutions performed back-to-back. *DNN conv* is a multi-channel convolution used in convolutional neural networks. *Harris* is a corner detector. *Camera pipe* performs three stages of operations transforming raw images to RGB images. We used our power-domain-aware place and route tool to statically map these applications on the CGRA. The leakage and total power savings for the described applications are shown in Table 4.2. Depending on the number of PE and memory tiles used by the given application, we see an 11 - 82% reduction in the leakage power of the CGRA fabric by turning off unused tiles and 0.8 - 26% reduction in the total power.

The percentage of routing-only tiles changes with the size of the applications. For larger applications, most of the utilized tiles are functional tiles and 20-30% are routing-only tiles. For smaller applications, 60-70% of the utilized tiles are routing-only tiles.

The leakage and total power savings are proportional to the size of the applications. For smaller

Table 4.2: Power savings for applications mapped on a $32 \times 16$ CGRA with 384 PE tiles and 128 memory tiles.

| **Applications** | **% Tiles Utilized** | | **Total Power** | **Leakage Power** |
|---|---|---|---|---|
| | **PE** | **Memory** | **Reduction** | **Reduction** |
| Conv $1 \times 2$ | 17.7% | 9.3% | 26.2% | 82.7% |
| Conv $3 \times 3$ | 31.3% | 29.7% | 12.2% | 66.9% |
| Gaussian | 35.1% | 31.3% | 10.8% | 64.0% |
| Demosiac | 35.9% | 33.6% | 10.2% | 62.8% |
| Cascade | 39.3% | 37.5% | 8.9% | 59.2% |
| DNN conv | 55.2% | 53.0% | 4.9% | 44.0% |
| Harris | 58.8% | 54.7% | 4.4% | 41.3% |
| Camera pipe | 90.0% | 85.0% | 0.8% | 11.5% |

applications, a larger part of the fabric can be turned off, improving the overall power savings. As applications grow and utilize more and more tiles, the power savings become limited. Further, a few additional factors determine the leakage power savings when the tiles are turned off:

- **Number of cells in the *always-on* region:** These cells stay *on* even when the tile is *off*, contributing to the *off* leakage power.
- **Number of *always-on* buffers:** In our case, this count is higher to keep the global signals *on* even when the tiles are *off*. The leakage from these buffers is the majority contributor to the *off* leakage power.
- **Threshold voltage of the *on* cells in an *off* tile:** Using high Vt cells for the *always-on* cells helps reduce leakage in the *off* tile.
- **Substrate connection for the *off* cells:** As discussed in Section 4.4.4, if the substrate of the cells in the *off* region is connected to the *always-on power*, there is substrate leakage even after the tile is turned off. If power switches with NWELL separation are available in the standard cell library, the substrate can be connected to the switching power supply which can help save leakage power.

## 4.7 Power Domain Silicon Measurements

We test the power domain features on the chip. Power domain-related features worked on silicon as expected:

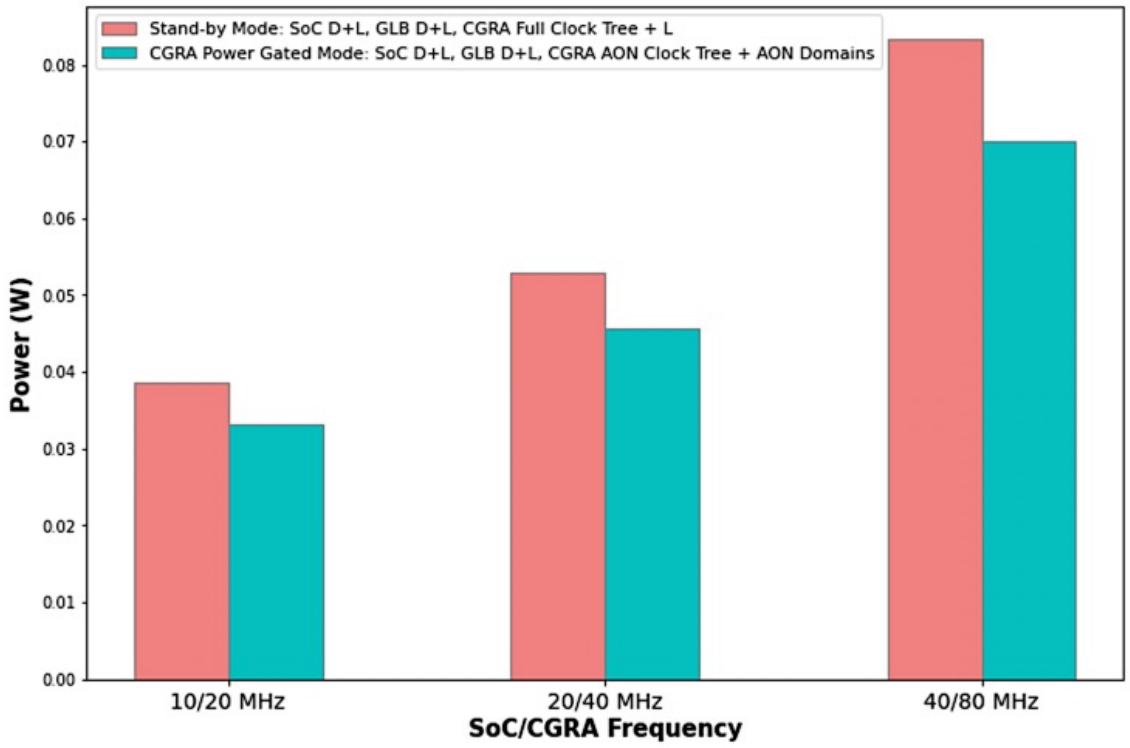


Figure 4.26: Silicon power measurements with and without power domains in stand-by mode.

- Turn on the tiles - Power is delivered as expected to the tiles through the power switches.
- Turning off unused tiles - When the power switch enable is set to high, the tiles are turned off.
- Turning on off tiles through configuration registers - The off tiles can be turned on as expected by programming the configuration registers.
- Global signal integrity was intact when tiles were turned off - Global signals flow through the column of tiles as expected even when some tiles are off.

Silicon power measurements with and without power domains is as shown in Fig. 4.26. The overall leakage contribution in this chip was low due to less usage of low Vt cells. In the next chip, we plan to use more low Vt cells to improve performance and recover power through power gating.

## 4.8 Conclusion

This chapter has described how our approach makes it possible to embed the isolation circuits required for fine-grained power domains into the programmable routing fabric. Although this approach makes the area overhead of fine-grained power domains essentially zero, it cannot completely leverage the conventional UPF power flow. We addressed this issue by using a set of Python-based tools to first add the needed power domains and then modify the routing network. These transformations were done after the "logical" base design was complete. This separation of concerns makes the logical design and power domain transformations more reusable for future designs. The additional validation to ensure power domains are correctly inserted is performed formally using an SMT solver and gate-level verification methods. Creating these tools and addressing the unique implementation challenges makes CGRAs with low-overhead fine-grained power domains possible.

# Chapter 5

# Conclusion

Developing accelerators for edge computing involves balancing efficiency and the ability to accommodate a growing number of use cases as applications continue to evolve. While reconfigurable architectures, such as CGRAs, are promising candidates for such applications, they are still not as efficient as ASICs. The work presented in this thesis provides valuable insights into how some ASIC-specific techniques can be successfully applied to CGRAs, enabling them to better support the evolving demands of edge computing applications such as DNNs.

Our previous investigation into the design space of DNNs revealed that proper blocking schemes can achieve similar energy efficiency and near-optimal performance across many DNN dataflows. However, we found that optimizing resource allocation, particularly the design of the memory hierarchy, has the most impact on energy efficiency. Applications should be scheduled on hardware such that data is fetched from the cheapest memory closest to the ALU, typically registers. Nevertheless, incorporating registers into streaming spatial accelerators presents a challenge in generating the necessary addresses. While some prior CGRAs offer simple address pattern generators, we found these were insufficient due to the complexity of access patterns required not only for supporting computations in convolution layers but also for loading and storing data in the registers files. Our investigations led us to conclude that for read-modify-write support for output feature maps, two different sets of access patterns were needed for both read and write, resulting in significant area overhead.

An initial version of a 64B pond with one set of controller increased the PE area by 25%. Supporting a pond with two sets of controllers for the ofmap pond increased its area by over 1.6x. We hence evaluated each of the access patterns carefully to determine the complexity of the controller responsible for generating addresses and controls for the register and explored opportunities for optimization. Based on these evaluations, we introduced various controller optimizations, such as simplifying address generation patterns wherever possible and adding timing signals to indicate the start of the operation and when the input data is valid to simplify schedule generation. Additionally,

we shared the update controller for read and write with a variable delay logic. These optimizations resulted in a 50% reduction in the area for the configuration registers and improved the PE area by 17%.

To analyze the impact of register file sizes and the number of physical ports on energy consumption, we extended our evaluations. We increased flexibility by allowing the use of a register file from neighboring PE tile, enabling support for different schedules and allowing users to choose whether inputs, outputs, or weights could be maintained in the register files. Our evaluations showed that introducing these register files and appropriate computation blocking resulted in significant energy reduction for convolution layers, up to 50%. However, pointwise and fully-connected layers did not experience as significant an energy improvement due to limited locality and computation reuse, with only up to a 13% reduction in energy consumption.

CGRAs, like ASICs, can benefit from utilizing lower threshold transistors to achieve improved energy efficiency by enabling active compute blocks to operate at lower supply voltages. However, using these devices also necessitates the implementation of a power gating strategy to reduce power consumption when functions blocks are not used. For ASICs, compute units can be easily broken into tightly coupled groups of logic as the interconnection is know during design time. As this is not the case for CGRAs, unless fine-grained power domains are introduced to alleviate the leakage power, the overhead of supporting power domains was not worth the effort.

By controlling the entire system, our proposed method enables the embedding of isolation circuits required for fine-grained power domains in the programmable routing fabric itself with minimal overhead. To achieve this, our P&R software was utilized to configure the fabric such that the *on* tile's multiplexer are programmed to receive only inputs from the neighboring *on* tiles (or receive constant input from the PE). By introducing this method, we successfully embedded isolation logic into the routing fabric. It is worth noting that owning an end-to-end system is advantageous for leveraging such optimization techniques.

In addition, we discovered that introducing tile-level power gating presents other implementation challenges, including routing of global signals through the *off* tiles, power grid implementation, and well substrate connection, all of which require a comprehensive evaluation of their respective trade-offs. It is particularly crucial to ensure the reliability of the debug signals that facilitate the reading of the configuration registers, enabling post-silicon debugging even when certain tiles are inactive. Our findings highlight the potential of application-specific and application-agnostic techniques to enhance the energy efficiency of CGRAs, and we hope that this will inspire further research to identify similar optimization opportunities across both hardware and software stacks.

# Appendix

## PE Tile UPF File

```
####### Create Power Domains #######
# Default Power Domain - Shutdown when tile not used
create_power_domain TOP -include_scope

# AON Domain - Modules that stay ON when tile is OFF
# PS configuration logic and tie cells for hi/lo outputs that drive the tile_id
create_power_domain AON \
-elements { PowerDomainOR DECODE_FEATURE_13 coreir_eq_16_inst0
            and_inst1 FEATURE_AND_13 PowerDomainConfigReg_inst0
            const_511_9 const_0_8}

##### Top-level Connections ######
## VDD
create_supply_port VDD
create_supply_net  VDD -domain TOP
create_supply_net  VDD -domain AON -reuse
connect_supply_net VDD -ports  VDD

## VSS (0.0V)
create_supply_port VSS
create_supply_net  VSS -domain TOP
create_supply_net  VSS -domain AON -reuse
connect_supply_net VSS -ports  VSS

####### TOP SD Domain Power Connections #######
create_supply_net VDD_SW -domain TOP

####### Establish Connections #######
set_domain_supply_net AON \
-primary_power_net VDD \
-primary_ground_net VSS \

set_domain_supply_net TOP \
-primary_power_net VDD_SW \
-primary_ground_net VSS

####### Set all Global Signals as AON #######
set_related_supply_net \
```

```
-object_list {tile_id hi lo clk clk_pass_through reset
              config_config_addr config_config_data
              config_read config_write
              read_config_data_in stall flush}  \
              -power VDD -ground VSS

set_related_supply_net \
-object_list {clk_out clk_pass_through_out_bot clk_pass_through_out_right reset_out
              config_out_config_addr config_out_config_data
              config_out_read config_out_write
              read_config_data stall_out flush_out} \
              -power VDD -ground VSS

####### Create Power Switch #######
create_power_switch SD_sw \
-domain TOP \
-input_supply_port {in VDD} \
-output_supply_port {out VDD_SW} \
-control_port {SD_sd PowerDomainConfigReg_inst0/read_config_data[0]} \
-on_state {ON_STATE in !SD_sd}

####### Create Power State Table #######
add_port_state VDD \
-state {HighVoltage 0.8}

add_port_state SD_sw/out \
-state {HighVoltage 0.8} \
-state {SD_OFF off}

create_pst lvds_system_pst \
-supplies {VDD VDD_SW}

add_pst_state LOGIC_ON \
-pst lvds_system_pst \
-state {HighVoltage SD_OFF}

add_pst_state ALL_ON \
-pst lvds_system_pst \
-state {HighVoltage HighVoltage}
```

# Bibliography


[1] Andrew Adams. Halide.

[2] Saad Albawi, Tareq Abed Mohammed, and Saad Al-Zawi. Understanding of a convolutional neural network. In *2017 international conference on engineering and technology (ICET)*, pages 1–6. Ieee, 2017.

[3] Kota Ando, Shinya Takamaeda-Yamazaki, Masayuki Ikebe, Tetsuya Asai, and Masato Motomura. A multithreaded CGRA for convolutional neural network processing. *Circuits and Systems*, 8(6):149–170, 2017.

[4] AON. Aon cells.

[5] Olexa Bilaniuk, Ehsan Fazl-Ersi, Robert Laganiere, Christina Xu, Daniel Laroche, and Craig Moulder. Fast lbp face detection on low-power simd architectures. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition Workshops*, pages 616–622, 2014.

[6] Frank Bouwens, Mladen Berekovic, Andreas Kanstein, and Georgi Gaydadjiev. Architectural exploration of the adres coarse-grained reconfigurable array. In *International Workshop on Applied Reconfigurable Computing*, pages 1–13. Springer, 2007.

[7] Assem AM Bsoul and Steven JE Wilton. An FPGA architecture supporting dynamically controlled power gating. In *2010 International Conference on Field-Programmable Technology*, pages 1–8. IEEE, 2010.

[8] Cadence. Conformal low power. https://www.cadence.com/content/cadence-www/global/enUS/home/tools/digital-design-and-signoff/low-power-validation/conformal-low-power.html, 2019.

[9] Alfredo Canziani, Adam Paszke, and Eugenio Culurciello. An analysis of deep neural network models for practical applications. *arXiv preprint arXiv:1605.07678*, 2016.

[10] Steve Carr and Ken Kennedy. Blocking linear algebra codes for memory hierarchies. In *PPSC*, pages 400–405. Citeseer, 1989.

[11] Adrian M Caulfield, Eric S Chung, Andrew Putnam, Hari Angepat, Jeremy Fowers, Michael Haselman, Stephen Heil, Matt Humphrey, Puneet Kaur, Joo-Young Kim, et al. A cloud-scale acceleration architecture. In *2016 49th Annual IEEE/ACM international symposium on microarchitecture (MICRO)*, pages 1–13. IEEE, 2016.

[12] Tsung-Han Chan, Kui Jia, Shenghua Gao, Jiwen Lu, Zinan Zeng, and Yi Ma. Pcanet: A simple deep learning baseline for image classification? *IEEE transactions on image processing*, 24(12):5017–5032, 2015.

[13] Zhengyu Chen, Hai Zhou, and Jie Gu. R-accelerator: An RRAM-based CGRA accelerator with logic contraction. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 27(11):2655–2667, 2019.

[14] Chisel. Chisel.

[15] François Chollet. Xception: Deep learning with depthwise separable convolutions. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1251–1258, 2017.

[16] Zidong Du, Robert Fasthuber, Tianshi Chen, Paolo Ienne, Ling Li, Tao Luo, Xiaobing Feng, Yunji Chen, and Olivier Temam. Shidiannao: Shifting vision processing closer to the sensor. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, pages 92–104, 2015.

[17] Carl Ebeling, Darren C Cronquist, and Paul Franklin. Rapid—reconfigurable pipelined datapath. In *International Workshop on Field Programmable Logic and Applications*, pages 126–135. Springer, 1996.

[18] Alexandre E Eichenberger, Peng Wu, and Kevin O'brien. Vectorization for simd architectures with alignment constraints. *Acm sigplan notices*, 39(6):82–93, 2004.

[19] Xitian Fan, Huimin Li, Wei Cao, and Lingli Wang. Dt-cgra: Dual-track coarse-grained reconfigurable architecture for stream applications. In *2016 26th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–9. IEEE, 2016.

[20] Clément Farabet, Berin Martini, Benoit Corda, Polina Akselrod, Eugenio Culurciello, and Yann LeCun. Neuflow: A runtime reconfigurable dataflow processor for vision. In *CVPR 2011 workshops*, pages 109–116. IEEE, 2011.

[21] Joseph A Fisher, Paolo Faraboschi, and Cliff Young. *Embedded computing: a VLIW approach to architecture, compilers and tools*. Elsevier, 2005.

[22] David Flynn, Rob Aitken, Alan Gibbons, and Kaijian Shi. *Low power methodology manual: for system-on-chip design*. Springer Science & Business Media, 2007.

[23] Aman Gayasen, Y Tsai, Narayanan Vijaykrishnan, Mahmut Kandemir, Mary Jane Irwin, and Tim Tuan. Reducing leakage energy in FPGAs using region-constrained placement. In *Proceedings of the 2004 ACM/SIGDA 12th international symposium on Field programmable gate arrays*, pages 51–58, 2004.

[24] Graham Gobieski, Ahmet Oguz Atli, Kenneth Mai, Brandon Lucia, and Nathan Beckmann. Snafu: an ultra-low-power, energy-minimal cgra-generation framework and architecture. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, pages 1027–1040. IEEE, 2021.

[25] Seth Copen Goldstein, Herman Schmit, Mihai Budiu, Srihari Cadambi, Matthew Moe, and R Reed Taylor. Piperench: A reconfigurable architecture and compiler. *Computer*, 33(4):70–77, 2000.

[26] Venkatesh Gourisetty, Hamid Mahmoodi, Vazgen Melikyan, Eduard Babayan, Rich Goldman, Katie Holcomb, and Troy Wood. Low power design flow based on unified power format and Synopsys tool chain. In *2013 3rd Interdisciplinary Engineering Design Education Conference*, pages 28–31. IEEE, 2013.

[27] Sorin Grigorescu, Bogdan Trasnea, Tiberiu Cocias, and Gigel Macesanu. A survey of deep learning techniques for autonomous driving. *Journal of Field Robotics*, 37(3):362–386, 2020.

[28] Kyuseung Han, Seongsik Park, and Kiyoung Choi. State-based full predication for low power coarse-grained reconfigurable architecture. In *2012 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1367–1372. IEEE, 2012.

[29] Pat Hanrahan. Magma. https://github.com/phanrahan/magma,2019.

[30] Geoffrey E Hinton, Simon Osindero, and Yee-Whye Teh. A fast learning algorithm for deep belief nets. *Neural computation*, 18(7):1527–1554, 2006.

[31] Mark Horowitz. 1.1 computing's energy problem (and what we can do about it). In *2014 IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC)*, pages 10–14. IEEE, 2014.

[32] Guosheng Hu, Yongxin Yang, Dong Yi, Josef Kittler, William Christmas, Stan Z Li, and Timothy Hospedales. When face recognition meets with deep learning: an evaluation of convolutional neural networks for face recognition. In *Proceedings of the IEEE international conference on computer vision workshops*, pages 142–150, 2015.

[33] Intel. Intel® stratix® 10 fpga and soc fpga.

[34] Mircea Horea Ionica and David Gregg. The movidius myriad architecture's potential for scientific computing. *IEEE Micro*, 35(1):6–14, 2015.

[35] Shota Ishihara, Masanori Hariyama, and Michitaka Kameyama. A low-power FPGA based on autonomous fine-grain power gating. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 19(8):1394–1406, 2010.

[36] Syed. M. A. H. Jafri, Ozan Bag, Ahmed Hemani, Nasim Farahini, Kolin Paul, Juha Plosila, and Hannu Tenhunen. Energy-aware coarse-grained reconfigurable architectures using dynamically reconfigurable isolation cells. In *International Symposium on Quality Electronic Design (ISQED)*, pages 104–111, 2013.

[37] Syed MAH Jafri, Tuan Nguyen Gia, Sergei Dytckov, Masoud Daneshtalab, Ahmed Hemani, Juha Plosila, and Hannu Tenhunen. NeuroCGRA: A CGRA with support for neural networks. In *2014 International Conference on High Performance Computing & Simulation (HPCS)*, pages 506–511. IEEE, 2014.

[38] Song Jiang, Xiaoning Ding, Feng Chen, Enhua Tan, and Xiaodong Zhang. Dulo: an effective buffer cache management scheme to exploit both temporal and spatial locality. In *Proceedings of the 4th conference on USENIX Conference on File and Storage Technologies*, volume 4, pages 8–8, 2005.

[39] Norman P Jouppi, Doe Hyun Yoon, Matthew Ashcraft, Mark Gottscho, Thomas B Jablin, George Kurian, James Laudon, Sheng Li, Peter Ma, Xiaoyu Ma, et al. Ten lessons from three generations shaped google's tpuv4i: Industrial product. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, pages 1–14. IEEE, 2021.

[40] Norman P Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, et al. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th annual international symposium on computer architecture*, pages 1–12, 2017.

[41] Manupa Karunaratne, Aditi Kulkarni Mohite, Tulika Mitra, and Li-Shiuan Peh. Hycube: A CGRA with reconfigurable single-cycle multi-hop interconnect. In *Proceedings of the 54th Annual Design Automation Conference 2017*, pages 1–6, 2017.

[42] Takuya Kojima, Nguyen Anh Vu Doan, and Hideharu Amano. Genmap: A genetic algorithmic approach for optimizing spatial mapping of coarse-grained reconfigurable architectures. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 28(11):2383–2396, 2020.

[43] Chen Kong and Simon Lucey. Take it in your stride: Do we need striding in cnns? *arXiv preprint arXiv:1712.02502*, 2017.

[44] Guilherme Korol, Michael Guilherme Jordan, Marcelo Brandalero, Michael Hübner, Mateus Beck Rutzig, and Antonio Carlos Schneider Beck. MCEA: A resource-aware multicore

CGRA architecture for the edge. In *2020 30th International Conference on Field-Programmable Logic and Applications (FPL)*, pages 33–39. IEEE, 2020.

[45] Johannes Maximilian Kühn, Dustin Peterson, Hideharu Amano, Oliver Bringmann, and Wolfgang Rosenstiel. Spatial and temporal granularity limits of body biasing in UTBB-FDSOI. In *2015 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 876–879. IEEE, 2015.

[46] Nicholas D Lane, Sourav Bhattacharya, Akhil Mathur, Petko Georgiev, Claudio Forlivesi, and Fahim Kawsar. Squeezing deep learning into mobile and embedded devices. *IEEE Pervasive Computing*, 16(3):82–88, 2017.

[47] Mark LaPedus. Big trouble at 3nm. *Semiconductor engineering*, 2018.

[48] Yuan Lei, Peng Luo, Chi Hong Chan, Xiao Huo, Yiu Kei Li, and Mei Kei Ieong. Low power AI ASIC design for portable edge computing. In *2020 IEEE 15th International Conference on Solid-State & Integrated Circuit Technology (ICSICT)*, pages 1–4. IEEE, 2020.

[49] Ce Li, Yiping Dong, and Takahiro Watanabe. New power-efficient FPGA design combining with region-constrained placement and multiple power domains. In *2011 IEEE 9th International New Circuits and systems conference*, pages 69–72. IEEE, 2011.

[50] Fei Li and Lei He. Maximum current estimation considering power gating. In *Proceedings of the 2001 international symposium on Physical design*, pages 106–111, 2001.

[51] Yixing Li, Zichuan Liu, Wenye Liu, Yu Jiang, Yongliang Wang, Wang Ling Goh, Hao Yu, and Fengbo Ren. A 34-FPS 698-GOP/s/W binarized deep neural network-based natural scene text interpretation accelerator for mobile edge computing. *IEEE Transactions on Industrial Electronics*, 66(9):7407–7416, 2018.

[52] Yan Lin, Fei Li, and Lei He. Routing track duplication with fine-grained power-gating for fpga interconnect power reduction. In *Proceedings of the 2005 Asia and South Pacific design automation conference*, pages 645–650, 2005.

[53] Leibo Liu, Jianfeng Zhu, Zhaoshi Li, Yanan Lu, Yangdong Deng, Jie Han, Shouyi Yin, and Shaojun Wei. A survey of coarse-grained reconfigurable architecture and design: Taxonomy, challenges, and applications. *ACM Computing Surveys (CSUR)*, 52(6):1–39, 2019.

[54] Qiaoyi Liu, Dillon Huff, Jeff Setter, Maxwell Strange, Kathleen Feng, Kavya Sreedhar, Ziheng Wang, Keyi Zhang, Mark Horowitz, Priyanka Raina, et al. Compiling halide programs to push-memory accelerators. *arXiv preprint arXiv:2105.12858*, 2021.

[55] João Lopes, Diogo Sousa, and João Canas Ferreira. Evaluation of CGRA architecture for real-time processing of biological signals on wearable devices. In *2017 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*, pages 1–7. IEEE, 2017.

[56] João D Lopes, José T de Sousa, Horácio Neto, and Mário Véstias. K-means clustering on cgra. In *2017 27th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–4. IEEE, 2017.

[57] Anmol Mathur and Qi Wang. Power reduction techniques and flows at RTL and system level. In *2009 22nd International Conference on VLSI Design*, pages 28–29, 2009.

[58] Cristian Mattarei, Makai Mann, Clark Barrett, Ross G Daly, Dillon Huff, and Pat Hanrahan. CoSA: Integrated verification for agile hardware design. In *2018 Formal Methods in Computer Aided Design (FMCAD)*, pages 1–5. IEEE, 2018.

[59] Nick Mehta. Xilinx 7 series FPGAs: the logical advantage, 2012.

[60] John Mellor-Crummey, David Whalley, and Ken Kennedy. Improving memory hierarchy performance for irregular applications. In *Proceedings of the 13th international conference on Supercomputing*, pages 425–433, 1999.

[61] Massimo Merenda, Carlo Porcaro, and Demetrio Iero. Edge machine learning for ai-enabled iot devices: A review. *Sensors*, 20(9):2533, 2020.

[62] Narasinga Rao Miniskar, Rahul R Patil, Raj Narayana Gadde, Young-Chul Rams Cho, Sukjin Kim, and Shi Hwa Lee. Intra mode power saving methodology for CGRA-based reconfigurable processor architectures. In *2016 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 714–717. IEEE, 2016.

[63] Fahad Bin Muslim, Affaq Qamar, and Luciano Lavagno. Low power methodology for an ASIC design flow based on high-level synthesis. In *2015 23rd International Conference on Software, Telecommunications and Computer Networks (SoftCOM)*, pages 11–15, 2015.

[64] Nvidia. Training vs inference.

[65] Keiron O'Shea and Ryan Nash. An introduction to convolutional neural networks. *arXiv preprint arXiv:1511.08458*, 2015.

[66] Nobuaki Ozaki, Yoshihiro Yasuda, Mai Izawa, Yoshiki Saito, Daisuke Ikebuchi, Hideharu Amano, Hiroshi Nakamura, Kimiyoshi Usami, Mitaro Namiki, and Masaaki Kondo. Cool mega-arrays: Ultralow-power reconfigurable accelerator chips. *IEEE Micro*, 31(6):6–18, 2011.

[67] Ozcan Ozturk, Mahmut Kandemir, Mary Jane Irwin, and Suleyman Tosun. Multi-level on-chip memory hierarchy design for embedded chip multiprocessors. In *12th International Conference on Parallel and Distributed Systems-(ICPADS'06)*, volume 1, pages 8–pp. IEEE, 2006.

[68] Maurice Peemen, Arnaud AA Setio, Bart Mesman, and Henk Corporaal. Memory-centric accelerator design for convolutional neural networks. In *2013 IEEE 31st international conference on computer design (ICCD)*, pages 13–19. IEEE, 2013.

[69] Raghu Prabhakar, Yaqi Zhang, David Koeplinger, Matt Feldman, Tian Zhao, Stefan Hadjis, Ardavan Pedram, Christos Kozyrakis, and Kunle Olukotun. Plasticine: A reconfigurable architecture for parallel patterns. In *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*, pages 389–402. IEEE, 2017.

[70] Steven Przybylski, Mark Horowitz, and John Hennessy. Characteristics of performance-optimal multi-level cache hierarchies. *ACM SIGARCH Computer Architecture News*, 17(3):114–121, 1989.

[71] Steven A Przybylski. *Cache and memory hierarchy design*. Elsevier, 2014.

[72] Jiantao Qiu, Jie Wang, Song Yao, Kaiyuan Guo, Boxun Li, Erjin Zhou, Jincheng Yu, Tianqi Tang, Ningyi Xu, Sen Song, et al. Going deeper with embedded fpga platform for convolutional neural network. In *Proceedings of the 2016 ACM/SIGDA international symposium on field-programmable gate arrays*, pages 26–35, 2016.

[73] SAT. Boolean satisfiability problem.

[74] Reza Sharafinejad, Bijan Alizadeh, and Masahiro Fujita. UPF-based formal verification of low power techniques in modern processors. In *2015 IEEE 33rd VLSI Test Symposium (VTS)*, pages 1–6. IEEE, 2015.

[75] G.A. Shaw, J.C. Anderson, and V.K. Madisetti. Assessing and improving current practice in the design of application-specific signal processors. In *1995 International Conference on Acoustics, Speech, and Signal Processing*, volume 4, pages 2707–2710 vol.4, 1995.

[76] Youngsoo Shin, Jun Seomun, Kyu-Myung Choi, and Takayasu Sakurai. Power gating: Circuits, design methodologies, and best practice for standard-cell VLSI designs. *ACM Trans. Des. Autom. Electron. Syst.*, 15(4), October 2010.

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

[78] SMT. Satisfiability modulo theories.

[79] Marc Snir and Jing Yu. On the theory of spatial and temporal locality. Technical report, 2005.

[80] Mingcong Song, Jiaqi Zhang, Huixiang Chen, and Tao Li. Towards efficient microarchitectural design for accelerating unsupervised gan-based deep learning. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 66–77. IEEE, 2018.

[81] Vivienne Sze, Yu-Hsin Chen, Tien-Ju Yang, and Joel S Emer. Efficient processing of deep neural networks: A tutorial and survey. *Proceedings of the IEEE*, 105(12):2295–2329, 2017.

[82] Masakazu Tanomoto, Shinya Takamaeda-Yamazaki, Jun Yao, and Yasuhiko Nakashima. A CGRA-based approach for accelerating convolutional neural networks. In *2015 IEEE 9th International Symposium on Embedded Multicore/Many-core Systems-on-Chip*, pages 73–80. IEEE, 2015.

[83] UPF. Power intent standard. https://standards.ieee.org/project/1801.html, 2018.

[84] Francisco-Javier Veredas, Michael Scheppler, Will Moffat, and Bingfeng Mei. Custom implementation of the coarse-grained reconfigurable adres architecture for multimedia purposes. In *International Conference on Field Programmable Logic and Applications, 2005.*, pages 106–111. IEEE, 2005.

[85] Dhananjaya Wijerathne, Zhaoying Li, Manupa Karunarathne, Anuj Pathania, and Tulika Mitra. Cascade: High throughput data streaming via decoupled access-execute cgra. *ACM Transactions on Embedded Computing Systems (TECS)*, 18(5s):1–26, 2019.

[86] Michael Wolfe. More iteration space tiling. In *Proceedings of the 1989 ACM/IEEE conference on Supercomputing*, pages 655–664, 1989.

[87] Xuan Yang, Mingyu Gao, Qiaoyi Liu, Jeff Setter, Jing Pu, Ankita Nayak, Steven Bell, Kaidi Cao, Heonjae Ha, Priyanka Raina, et al. Interstellar: Using halide's scheduling language to analyze dnn accelerators. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 369–383, 2020.

[88] Xuan Yang, Jing Pu, Blaine Burton Rister, Nikhil Bhagdikar, Stephen Richardson, Shahar Kvatinsky, Jonathan Ragan-Kelley, Ardavan Pedram, and Mark Horowitz. A systematic approach to blocking convolutional neural networks. *arXiv preprint arXiv:1606.04209*, 2016.

[89] Fisher Yu and Vladlen Koltun. Multi-scale context aggregation by dilated convolutions. *arXiv preprint arXiv:1511.07122*, 2015.

[90] C Zhang, Z Fang, P Zhou, P Pan, and J Cong. Caffeine: Towards uniformed representation and acceleration for deep convolutional neural networks 2016 ieee. In *ACM International Conference on Computer-Aided Design (ICCAD), ACM*, pages 1–8, 2016.

[91] Xingyu Zhou, Robert Canady, Shunxing Bao, and Aniruddha Gokhale. Cost-effective hardware accelerator recommendation for edge computing. In *3rd {USENIX} Workshop on Hot Topics in Edge Computing (HotEdge 20)*, 2020.